OSF™DCE

OSF™DCE Application Development Guide

OPEN SOFTWARE FOUNDATION

OSF™ DCE Application Development Guide

Revision 1.0

Open Software Foundation

P T R-Prentice Hall, Englewood Cliffs, New Jersey 07632

Cover design: **BETH FAGAN**
Cover illustration: **STEVE LEWONTIN**

This book was formatted with troff

Published by Prentice-Hall, Inc.
A Simon & Schuster Company
Englewood Cliffs, New Jersey 07632

The information contained within this document is subject to change without notice.

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein, or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing, performance, or use of this material.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN: 0-13-643826-1

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Part 1. Overview of DCE Application Development

Part 2. DCE Threads

Part 3. DCE Remote Procedure Call

Part 3A. Using Remote Procedure Call

Part 3B. Language Syntax and Usage

Part 3C. Supplemental Information

Part 4. DCE Directory Service

Part 4A. CDS Application Programming

Part 4B. GDS Application Programming

Part 4C. XDS/XOM Supplementary Information

Part 5. DCE Distributed Time Service

Part 6. DCE Security Service

Part 7. DCE Distributed File Service

List of Figures

List of Tables

Preface

The *OSF DCE Application Development Guide* provides information about how to program the Application Programming Interfaces (APIs) provided for each OSF^{TM} Distributed Computing Environment (DCE) component.

Audience

This guide is written for application programmers with UNIX operating system and C language experience who want to develop and write applications to run on DCE.

Applicability

This is Revision 1.0 of this document. It applies to the OSF^{TM} DCE Version 1.0 offering and related updates. See your software license for details.

Purpose

The purpose of this guide is to assist programmers in developing applications using DCE. After reading this guide, you should be able to program the Application Programming Interfaces provided for each DCE component.

Document Usage

This guide is organized into the following seven parts:

- For an overview of DCE application development, see "Part 1. Overview of DCE Application Development."
- For information about the DCE Threads Application Programming Interface, see "Part 2. DCE Threads."
- For information about the DCE Remote Procedure Call Application Programming Interface, see "Part 3. DCE Remote Procedure Call."
- For information about the DCE Directory Service Application Programming Interface, see "Part 4. DCE Directory Service."
- For information about the DCE Distributed Time Service Application Programming Interface, see "Part 5. DCE Distributed Time Service."
- For information about the DCE Security Service Application Programming Interface, see "Part 6. DCE Security Service."
- For information about the DCE Distributed File Service Application Programming Interface, see "Part 7. DCE Distributed File Service."

Related Documents

For additional information about the Distributed Computing Environment, refer to the following documents:

- *Introduction to OSF DCE*
- *OSF DCE User's Guide and Reference*
- *OSF DCE Application Development Reference*
- *OSF DCE Administration Guide*
- *OSF DCE Administration Reference*
- *OSF DCE Porting and Testing Guide*
- *Application Environment Specification (AES)/Distributed Computing*
- *OSF DCE Technical Supplement*
- *OSF DCE Release Notes*

Typographic and Keying Conventions

This document uses the following typographic conventions:

Bold **Bold** words or characters represent system elements that you must use literally, such as commands, options, and pathnames.

Italic *Italic* words or characters represent variable values that you must supply.

`Constant width`
Examples and information that the system displays appear in `constant width` typeface.

[ ] Brackets enclose optional items in format and syntax descriptions.

{ } Braces enclose a list from which you must choose an item in format and syntax descriptions.

| A vertical bar separates items in a list of choices.

< > Angle brackets enclose the name of a key on the keyboard.

... Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.

This document uses the following keying conventions:

<Ctrl-*x***>** or ^*x*
The notation **<Ctrl-***x***>** or ^*x* followed by the name of a key indicates a control character sequence. For example, **<Ctrl-c>** means that you hold down the control key while pressing **<c>**.

<Return> The notation **<Return>** refers to the key on your terminal or workstation that is labeled with the word Return or Enter, or with a left arrow.

Problem Reporting

If you have any problems with the software or documentation, please contact your software vendor's customer service department.

Pathnames of Directories and Files in DCE Documentation

For a list of the pathnames for directories and files referred to in this document, see the *OSF DCE Administration Guide* and the *OSF DCE Release Notes*.

Part 1

Overview of DCE Application Development

Chapter 1

DCE Application Development Steps

This chapter, and the two that follow it, make up the first part of the *OSF DCE Application Development Guide*. Together, these first three chapters offer an introduction to the fundamental aspects of DCE application programming. The reader of this guide is assumed to be an experienced programmer.

1.1 Introduction to DCE Application Development

The majority of this first chapter consists of a fairly detailed overview of each of the separate steps that a developer usually has to perform (or have the application perform) from the beginning of coding to the end of execution of a successful DCE application. Chapter 3 describes a practical example of this process: a complete working DCE example application, **timop**. Chapter 2 consists of detailed discussions of some of the fundamental DCE features and services, including use of the name service, coding an ACL manager, security key management, thread-safe programming practices, and other topics.

Before you begin a serious study of the contents of any part of this guide, or indeed of any other book in the DCE documentation set, you should read the

Introduction to OSF DCE. It contains clear and comprehensive overviews, with illustrations, of all the DCE components and of the integrated DCE as a whole; many concepts and details are explained there that are necessary to a full understanding of what is described here.

If you do not find information about topics you are interested in either in this guide or in the *OSF DCE Application Development Reference*, you should also look in the *OSF DCE Administration Guide* and the *OSF DCE Administration Reference*. For example, the DCE Cell Directory Service is not accessed directly by applications (except through DCE RPC NSI or through DCE XDS) so most of the discussion of CDS as a separate component is found in the administration documentation. Although the DCE Security Service is documented in the development books, certain aspects of it important to application developers (for example, adding new principals to the security registry database) are found only in the administration books.

1.2 Content Overview of Part 1

The following subsections contain additional information about the first three chapters of this guide.

1.2.1 Topics Covered in Part 1

The following DCE components are described and discussed in some detail in the first three chapters of this guide:

- DCE Threads Service
- DCE Cell Directory Service (DCE CDS), accessed through the DCE RPC NSI (Name Service Interface)
- DCE Remote Procedure Call (DCE RPC)
- DCE Security Service

Explicit use of all these components is necessary for most DCE applications; you cannot usually get by without them.

In addition, routines from the DCE Distributed Time Service are used in the **timop** example application described in Chapter 3.

1.2.2 Topics Not Covered in Part 1

The following DCE components are not discussed in these first three chapters:

- DCE Global Directory Service (DCE GDS), including the Global Directory Agent
- DCE Distributed File System (DCE DFS)
- X/Open Directory Service Application Programming Interface (XDS API) and XOM (Object Management) Interface

DCE GDS is used for looking up names that are held in directories not located in the local cell. GDS is automatically utilized by the DCE RPC Name Service Interface (NSI) when a nonlocal name is looked up (unless the cell uses DNS as its global directory service); thus, normal application use of GDS is handled implicitly by NSI.

Similarly, applications use DCE DFS whenever they access DFS files; the application routine calls remain the same on this level, and no special programming techniques are required.

XDS API is a general interface into the DCE Directory Service as a whole, both to CDS and GDS. Applications do not require this interface in order to accomplish client/server rendezvous, since this is taken care of by DCE RPC NSI.

Although the DCE Directory Service is a very versatile database which can be used to store and retrieve all kinds of data, the main use of a directory service for distributed applications is to provide a standard facility by which servers can advertise their location to clients. NSI works with a greatly reduced subset of predefined directory entry types that are tailored to this need. Developers can thus avoid having to contend with functionality they do not need, and concentrate on the details of client/server rendezvous.

Thus, the XDS interface is for developers who need access to the full functionality of the DCE Directory Service as a generic directory service. Some likely uses of XDS may occur in the development of the following:

- Alternative versions of DCE tools, such as the CDS namespace browser (a utility that allows users to view the contents of CDS directories), or other administrative utilities and commands

- Software that needs to access directory services in conformance with the X.500 protocols, such as:
  - — FTAM (File Transfer and Management) applications
  - — X.400 mail system applications
  - — Applications that use X.500 for routing

There is no restriction against your using XDS in any DCE application if you wish, but its typical uses fall outside the bounds of the discussion in these first three chapters. However, XDS does offer a functionality that can be of more immediate interest to DCE application developers. This involves the creation and addition of non-NSI attributes to name service entries, thus allowing · extra application-defined information to be stored in an application's namespace entries. A sample program showing how to do this can be found in Chapter 24 of this guide.

Of course, all of the DCE components are treated in detail in their separate parts in this guide, even though not all of them are discussed in this introductory part of the guide.

1.3 DCE Application Development

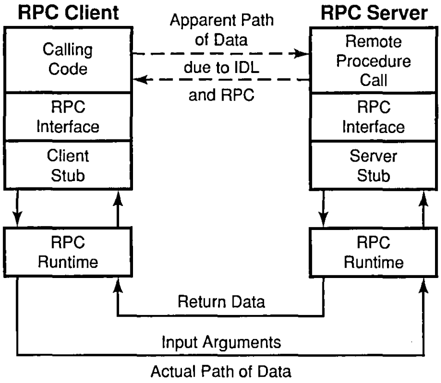
Most of the effort of developing a DCE application usually lies in the familiar steps of planning, writing and compiling the necessary C code, linking the result with the DCE library and other modules, and executing it (perhaps repeatedly). However, there is an important preliminary task that must be performed before you write any other code. Before you can implement the application's client and server, you must write and compile an interface definition file in which you define the application's client/server interface.

This interface, defined in the DCE Interface Definition Language (IDL), consists of a set of ''remote call prototypes'' for the remote procedure calls your client(s) will be requesting your server(s) to execute. After you have written this file, you compile it with the DCE IDL compiler. The final output of IDL compilation is a pair of object files, one for the server module and one for the client, which you must later link with the compiled output of your server and client implementation code. These two IDL output files contain the server and client stub code, where all the details of remote execution, data transfer, and so on, are managed.

The IDL compiler also generates a header file for inclusion in the server and client source files. It contains all the definitions and declarations that result from the IDL file definitions. Among these are, for example, the interface specification data structure, which will be used at runtime to describe the interface being defined here.

Once you have linked the stub files (and the DCE library) to their respective client and server modules, the IDL-generated stubs make the client and server seem to communicate directly through the operation signatures you defined in the original **.idl** file, although in actuality client/server communications pass back and forth through layers of stub and runtime processing, which are necessary to send and receive the data over the network. Figure 1-1 illustrates how the combination of IDL (by means of the stubs it generates) with the RPC runtime routines shields both client and server from the details of network communications.

Figure 1–1. The Combined Effect of IDL and the RPC Runtime



The first sections of Chapter 3 of this guide contain a fuller description of all the input possibilities to IDL, as well as all the different kinds of output it can generate.

Once the work of defining an interface has been completed, the task of implementing the interface (that is, coding the operations, along with the rest of the necessary initialization and management routines, in some programming language) begins. The rest of this chapter consists of detailed explanations of the DCE application development steps from start to finish. For a practical example of the result of such a process, refer to the code for **timop**, which is described in Chapter 3. For more detailed discussions of some of the DCE services and techniques for utilizing them, see Chapter 2.

Each of the DCE components (with the exception of CDS, which is accessed through the RPC-integrated NSI) is discussed in depth in separate parts of this guide. You should also refer often to the *OSF DCE Application Development Reference*, which contains reference pages for all of the DCE library routines mentioned in the following sections.

1.4 Overview of the DCE Application Development Steps

The rest of this chapter consists of a step-by-step checklist of every single one of the decisions that a programmer must make in developing a typical DCE application. Each set of decisions or choices is combined into one step. The combination of all these steps takes you from the initial coding stages into and through the normal course of execution of the application itself. The underlying intention of this arrangement is to give you a useful mental model of the overall code development process.

Figure 1-2 summarizes the organization of the steps.

Figure 1–2. The DCE Steps: The Five Basic Phases Illustrated

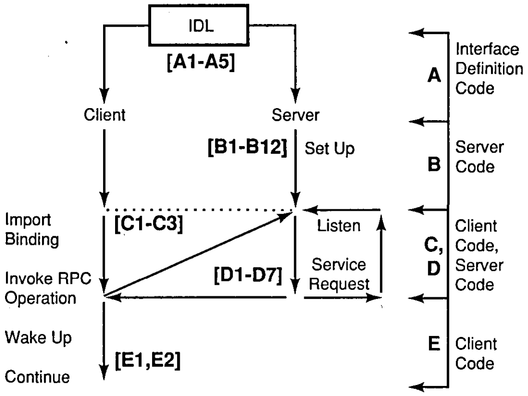


Figure 1-2 is divided into five basic phases, which are identified by the letters **A** through **E** along the right side of the figure. Each of these larger phases consists of a series of steps or decisions that normally occur in the development of a DCE application. The individual steps are indicated by the bold numbers in brackets; each one is described in detail in the following text.

Almost all of the steps in **B** through **E** consist of very specific choices regarding how, or whether, various DCE library routines are to be called. Steps **B2** through **E2** (phases **B** through **E**) can be taken together as a walk-through of the client-side and server-side code of a typical DCE application.

The first phase, **A**, represents a series of things that must occur before anything else in the development process can happen; namely, the IDL file definition and compilation.

Thus, the 5 basic phases of DCE application development are as follows:

A. **CLIENT and SERVER:** Define the RPC/IDL interface [Steps **A1** to **A5**]

B. **SERVER:** Set up and listen [Steps **B1** to **B12**]

C. **CLIENT:** Bind to and invoke the server [Steps **C1** to **C3**]

D. **SERVER:** Service request(s) [Steps **D1** to **D7**]

E. **CLIENT:** Receive the results [Steps **E1** to **E2**]

Following is an overview list of all 29 steps, separated into the 5 main phases previously described. Each step's numeral is followed by a / (slash) and the terms **Client** and/or **Server** to indicate whether it applies to the application's server or client, or both.

A. CLIENT and SERVER: Define the RPC/IDL interface.

A1/Client and Server: Generate the interface UUID.

A2/Client and Server: Determine the interface version number.

A3/Client and Server: Write the **.idl** file.

A4/Client and Server: Write the **.acf** file (optional).

A5/Client and Server: Process the files with the IDL compiler.

B. SERVER: Set up and listen.

B1/Server: Define the manager Entry Point Vectors (EPVs).

B2/Server: Register the object/type UUID associations with RPC runtime.

B3/Server: Register the interface, type UUID, and EPV with RPC runtime.

B4/Server: Specify multithreadedness.

B5/Server: Tell RPC runtime what protocol sequences to use.

B6/Server: Request the bindings from RPC runtime.

B7/Server: Register the authentication information with RPC runtime.

B8/Server: Establish the server principal identity.

B9/Server: Plan what to do when the server terminates.

B10/Server: Register the binding information with the endpoint mapper.

B11/Server: Export the binding information to the namespace (CDS).

B12/Server: Listen for incoming service requests.

C. CLIENT: Bind to and invoke the server.

C1/Client: Import the binding information from the namespace (CDS).

C2/Client: Annotate the binding handle for security.

C3/Client: Invoke an RPC interface operation.

D. SERVER: Service the request.

D1/Server: Wake up in manager routine.

D2/Server: Get the client's Privilege Attribute Certificate (PAC).

D3/Server: Get the object's Access Control List (ACL).

D4/Server: Make the authorization decision.

D5/Server: Service the request.

D6/Server: Return the results to the client.

D7/Server: Continue the listen loop.

E. CLIENT: Receive the results.

E1/Client: Wake up after the RPC call.

E2/Client: Continue.

1.5 The DCE Application Development Steps

The following subsections describe the 29 DCE application development steps.

1.5.1 Step A1/Client and Server: Generate the Interface UUID

The first step in developing any DCE application is to define its interface; these definitions are contained in an **.idl** file, written by the developer. Part of the definition of the interface is its UUID, which is a 128-bit Universal Unique Identifier that identifies it far and wide throughout DCE. Executing the **uuidgen** command with the **-i** option, for example:

uuidgen -i > *your_interface_name*.**idl**

will generate a file containing the skeleton of an interface definition and a newly generated UUID for the interface.

The **uuidgen** command is a general UUID manufacturing utility. It is used (among its other uses) to generate blocks of UUIDs for inclusion in data declarations, and so on. (Refer to the **uuidgen(1rpc)** reference page in the *OSF DCE Application Development Reference*.)

1.5.1.1 The Purpose of UUIDs

UUIDs are used to identify many different things in DCE. These ''things'' can be broadly distinguished into two categories: interfaces and objects. UUIDs that identify interfaces are commonly called ''interface UUIDs,'' and those that identify objects (see the beginning of Chapter 2 for more information about objects) are called, ''object UUIDs.'' However, a UUID in and of itself is neutral data that can be applied to the identification of anything; all UUIDs differ in value, but they are all the same *type* of value.

An interface UUID is the inalienable ''fingerprint'' that a server affixes to the array of operations that it offers; any client that wants to remotely execute any of these operations must present that same interface UUID to the server, thus ensuring that the client gets what it asks for, and nothing but that. This matching of interface UUIDs is done transparently to the application programmer by the server's RPC runtime code, which is located in the server stub. The client's copy of the interface UUID is located in its stub code. Clients must always be linked to a server stub module in order to access that server. The stub modules, as will shortly be discussed, come from the IDL compiler.

1.5.1.2 Summary

To sum up, interface UUIDs are never manipulated by clients. They do not appear in bindings or among the remote call parameters. They are, however, contained in a server's NSI-exported namespace entries so that NSI can make sure that clients import bindings only to servers that offer the same interface that the clients are seeking.

1.5.2 Step A2/Client and Server: Determine the Interface Version Number

The **version** attribute of an interface, specified in the **.idl** file, is used to give a major and minor version number to the interface.

A ''version'' of an interface is the result of compiling and linking some particular version of implementation source code with IDL-processed output, producing an executable version of the application. Thus, there can be more than one existing version of an interface implementation identified by the same UUID, ꞌbut distinguished by version numbers. When the RPC runtime compares the interface in an incoming remote procedure call to that offered by the server (as described in Step C3, Section 1.5.20), it allows the call to proceed only if all of the following are true:

- The UUIDs identifying the interface assumed by the client and the interface exported by the server are the same.

- The interface assumed by the client and the interface exported by the server have the same major version number.

- The interface assumed by the client has a minor version number less than or equal to that of the interface exported by the server.

Thus, correct use of the version number attribute allows an application to have different versions of an interface in existence and yet not have to be concerned about any resulting client/server interface incompatibilities; always, that is, provided that the version differences are accurately assessed by the programmer and expressed in the version numbering. Since the **version** attribute value is determined by the programmer, it is the programmer's responsibility to make sure that interface versions that *seem* to be compatible by version number actually *are* compatible with respect to the implemented operations.

For further information on how to use the **version** attribute, see Chapter 17.

1.5.3 Step A3/Client and Server: Write the .idl File

The **.idl** file is where the set of remote operations that will constitute the interface are defined. Although writing the **.idl** file is listed as a Client and Server step, there is only one **.idl** file (per interface). The default output of its compilation by the IDL compiler will be a pair of stub files, one for the client and one for the server; a header file is also output by IDL (see Section 3.1 in Chapter 3 of this guide). For a detailed discussion of all aspects of the **.idl** file, read Chapter 17 of this guide.

The server implementations of the remote operations are written in C source code, which is compiled and then linked to the stub code output by IDL. The interfaces to these operations are defined and characterized in the **.idl** file, in IDL language. Thus, an **.idl** file's contents are like a set of ''network prototypes'' for a set of operations. The IDL definitions determine not only how the operations ''look'' to client and server (that is, the operations' call signatures, parameter types, and so on), but also what the data looks like when it is transmitted back and forth between clients and servers in a distributed application.

The IDL language is declarative, not procedural. Its look is much like C. Some of the important attributes that it is used to specify are the following:

- For interfaces:

uuid Specifies a string that contains the interface's UUID. (See Step A1, Section 1.5.1.)

version Specifies the interface major and minor version number. (See Step A2, Section 1.5.2.)

endpoint Specifies well-known endpoints (if any) for the interface. (See Step B5, Section 1.5.10.)

- For parameters:

in Signifies a parameter whose value is passed from the client to the server.

out Signifies a parameter whose value is passed from the server to the client.

- For data types:

handle Specifies a customized binding handle. (See Step A4, Section 1.5.4.)

context_handle
A context handle is a pointer to state information used by the server, which is maintained across RPCs; for example, a file pointer. The client is not supposed to do anything with this pointer; it merely passes it to subsequent calls as needed, and it is used internally by the remote calls. This allows applications to have such things as remote calls that handle file operations much as local calls would; that is, a client application can remotely open a file, get back a handle to it, and then perform various other remote operations on it, passing the context handle as an argument to the calls. A context handle can be used across interfaces (where a single server offers the multiple interfaces), but it can *belong* only to the client who caused it to be activated. A context rundown routine can also be declared. This installs a routine into the callee's stub that will automatically be called to reclaim (run down) the pointed-to resource in the event of a communications break between the server and client. For example, in the case of the ''remote file pointer'' just mentioned, the context rundown routine would simply close the file.

transmit_as Allows you to associate specified complex data types with a set of routines (which you must write) that will be implicitly called by the stub code to translate the data into (and back out of) other formats, either to improve the efficiency of transmission or for other reasons.

Operation attributes include specifiers for execution semantics; that is, whether the operation can be safely executed more than once, whether a response is expected, and so on. The default is that operations can be executed at-most-once. Operations parameters (the arguments supplied by the client when it makes the remote call) can be specified as input to the server, output to the client, or both.

For further information on the IDL compiler and the IDL language, see the IDL chapters in Part 3 of this guide, as well as the **idl(1rpc)** reference page in the *OSF DCE Application Development Reference*.

1.5.4 Step A4/Client and Server: Write the .acf File

The Attribute Configuration File (**.acf**) is an optional additional input file to the IDL compiler; if present, it affects IDL's output in various ways.

The difference between the purpose of the **.idl** and an **.acf** file is that while the **.idl** file defines how the network communications between the client and server are handled, the **.acf** file, if one is present, affects only the interaction between the stub code modules and the developer code that they support. In other words, changing the contents of an **.acf** file has no effect on the network communications between the client and server.

Nevertheless, some of the features offered by an **.acf** file are very important, and they cannot be obtained by any other means. The sample DCE application, **timop**, which is described in Chapter 3, has an Attribute Configuration File in order that two attributes (namely, **out_of_line** marshalling and **comm_status**) can be declared. These attributes must be declared in an **.acf** file; they cannot be declared in an **.idl** file. (The **comm_status** attribute allows the status code of a communications failure that occurs in an RPC to be stored as a parameter or returned as a result, rather than being raised to the caller code as an exception.)

Another very important function of the **.acf** file is the specification of a binding method to be used by remote clients of the application. Three methods are available:

- **auto_handle**
- **implicit_handle**
- **explicit_handle** (the default)

Because **explicit_handle** is the default, it is not declared in **timop.acf**, even though **timop** uses the explicit method. All three binding methods are briefly described later in Section 1.5.4.2, and definitive descriptions of them can be found in Chapter 18 of this guide. The binding method you choose determines how much attention your server's clients will have to devote to the upkeep of their binding handles.

1.5.4.1 Binding Handles

A binding handle is simply the data structure that represents the client's current relationship with the server.

This relationship is determined by several items of data. Perhaps the most important is the protocol sequence and network address information necessary to maintain communications between the client and server; however, these are not the only contents of a binding handle. It can contain an object UUID as well. This, if present, is matched (when the client first tries to establish contact with a server) against the object UUIDs registered at the destination host by the resident servers. This allows servers to target their exported bindings unambiguously back to themselves, and not to other servers that may happen to offer the same interface. For further information about UUIDs, see Chapter 2. For further information on the use of object UUIDs, see Steps B2 (Section 1.5.7), B3 (Section 1.5.8), and B10 (Section 1.5.15).

A binding can also contain various kinds of security information. For further information about this possibility, see Step C2 (Section 1.5.19).

There are two types of binding handle: primitive and customized (the latter is sometimes referred to as ''generic'' in the DCE documentation).

- Primitive handles *must* be used when the automatic binding method is employed (see Section 1.5.4.2 for more information about binding methods); they can be used with the other two binding methods as well. The **timop** DCE sample application client (see Chapter 3 of this guide) uses the explicit binding method with primitive binding handles.

Primitive binding handles contain all the information that is needed by a client to bind to a server. They are specified with the predefined **handle_t** type in the **.idl** file; in application code they are declared as **rpc_binding_handle_t**. This is a predefined type that contains (when filled) the bound-to object's network location and, optionally, an object UUID; it does *not* contain an interface UUID.

- Customized handles are application-specific data types designed to meet the special needs of the appplication they are to be used by. They are specified in the **.idl** file with the **handle** attribute; you must supply the type. If your application uses customized handles, you must also supply routines to do the following things:

  — Receive a customized handle, generate its primitive-handle equivalent, and return the primitive handle. (The name of this routine is *custom*_**bind**( ), and it returns an **rpc_binding_handle_t**.)

  This routine is called implicitly by the client stub whenever a remote call is made by the client with a customized handle.

  — Receive a customized handle and its primitive equivalent, and, typically, free the primitive handle and any unneeded resources associated with the customized handle. (The name of this routine is *custom*_**unbind**( ), and it returns **void**.)

  This routine is implicitly called by the client stub after it receives the response to an RPC from the server.

In short, these two routines allow the customized handle to seem to have the same characteristics as a primitive handle in the client code. Note that these two routines are *not* defined or otherwise mentioned in the **.idl** file; they are part of the application source code.

Note also that using either kind of handle makes no difference to the way the incoming calls look to the server; this is a matter for the client's convenience, one way or the other. Servers always export only primitive handles.

1.5.4.2 Binding Methods

Binding methods allow applications (on the client side) to determine how much responsibility they want to assume for the management of their bindings. None of these methods make much difference in what the server has to do, although use of the **auto_handle** method does require that the bindings be exported to the namespace (which they usually are anyway, regardless of the method used).

The following three methods are available. Each is specified by an **.acf** attribute.

- If the **auto_handle** attribute is declared for the interface, the client never even sees the handle: it does not appear as an argument in remote calls, and it is automatically imported from the namespace. The operation definitions coded in the **.idl** file must not contain the binding handle in their argument lists. Also, the server must export its bindings into the namespace; string bindings (see Step B5, Section 1.5.10) will not work.

  A client using an automatic handle will be automatically rebound to a server under some circumstances if a break in communications occurs.

- To use the implicit method, you declare in the **.acf** file a global variable with the **implicit_handle** attribute. The client is responsible for importing a binding itself, or for getting it some other way. It must then assign this binding to the **implicit_handle** global variable. From then on it can make remote calls without specifying the handle as an argument or otherwise bothering with it. The stub code passes the handle to the RPC runtime when remote procedure calls are executed; the handle does not appear among the call arguments.

  An explicitly specified handle in the argument list for an operation overrides the **implicit_handle** attribute for that operation.

  Use of the **auto_handle** and **implicit_handle** methods is mutually exclusive for the same client.

- The **explicit_handle** method is illustrated both in the **timop** code described in Chapter 3, and in the binding steps that follow in the present chapter (namely, Steps C1, C2, and C3, Sections 1.5.18, 1.5.19, 1.5.20, respectively). With the explicit method, the binding handle is passed as an argument explicitly by the client in every remote procedure call it makes. Using this method allows the client to alter the binding in various ways if and when it chooses. For example, explicit handles allow the client to switch type UUIDs in the handle, in order to reach different server type managers (assuming that the server implements multiple type managers). For further information about type UUIDs, which are a kind of object UUID, see Step B3 (Section 1.5.8).

The **timop** application demonstrates another reason why a client might need explicit handles: **timop**'s client code is multithreaded, and each of the threads binds to the server on its own. Neither the automatic nor the implicit binding method would permit such multibinding, since only one **auto_handle** or **implicit_handle** can be used by a single client at a time.

For further information on attribute configuration files, see Chapter 18, which describes **.acf** file syntax. For further information on the IDL compiler, see the **idl(1rpc)** reference page in the *OSF DCE Application Development Reference*.

1.5.5 Step A5/Client and Server: Process the Files with the IDL Compiler

Section 3.1 describes the IDL compilation process in some detail, and explains the part this process plays in the production of an executable distributed application. The present section consists of a summary description of the IDL compiler's input and output.

IDL's input is an *xxx*.**idl** and (optionally) an *xxx*.**acf** file. Its default output is a header (*xxx*.**h**) file that contains definitions and declarations derived from the input for general use in the development source code, and two stub files, one for the client and one for the server, which contain runtime code for marshalling and unmarshalling, message handling, and all the other details of managing network communications. The stub files are output as object code (*xxx*_**cstub.o** and *xxx*_**sstub.o**) suitable for linking with the developer's compiled code. The IDL compiler generates C source code as an intermediate step in the compilation process, and the output of this step can also be saved in a pair of files (*xxx*_**cstub.c** and *xxx*_**sstub.c**).

IDL automatically includes the DCE RPC management interface in the compiled stub code. This allows all DCE applications to use the

rpc_mgmt_...() routines, which consist of various operations that allow callers to learn interesting things about servers, or in some cases to remotely perform useful operations at servers.

In order for a pair of client and server stubs to interoperate, they should be generated from the same interface definition (**.idl**) file, but they do *not* have to be generated with the same Attribute Configuration File (**.acf**). The compatibility rules for interface version numbers also apply (see Step A1, Section 1.5.1).

For further information on the IDL compiler, see the **idl(1rpc)** reference page in the *OSF DCE Application Development Reference*.

1.5.6 Step B1/Server: Define the Manager EPVs

''Manager'' is the DCE term for the part of a server that actually implements a set of interface operations, as distinguished from the more or less generic server code that initializes the server as a whole, obtains and exports its binding information, and so on (see **timop_manager.c** in Section 3.2.6.8 for an example). Manager Entry Point Vectors (EPVs) are the data structures in which are recorded the entry addresses of the application routines that implement the server's operations, as offered through its interface. The server's stub code uses the EPV to dispatch incoming RPCs to the requested operations. A default manager EPV is generated automatically by the IDL compiler and defined (that is, correct addresses are filled into it) in the output header file.

If more than one version of the same interface is to be supported by the same server, one EPV will be needed for each additional interface version. Interface version numbers are specified by the **version** attribute in the **.idl** file. The type manager RPC runtime mechanism, properly utilized, allows a server to declare multiple EPVs under the same interface, and to have the RPC runtime vector the incoming remote calls to the correct implementation code. See Step B3 for further information.

1.5.7 Step B2/Server: Register the Object/Type UUID Associations with the RPC Runtime

The server makes the RPC library call

```
rpc_object_set_type(obj_uuid, type_uuid, &status);
```

repeatedly to associate whatever objects it expects will appear in incoming RPCs with a type UUID. The association is made between each of the expected incoming object UUIDs and the type UUID.

A type UUID is nothing but a special kind of object UUID. ''Type'' in this context refers to a group of ordinary object UUIDs that have all been associated (via a series of calls, as shown here, to **rpc_object_set_type**( )) with another specially generated common object UUID, which can then be used to identify that group of objects collectively.

The type UUIDs in turn will be associated (in Step B3, Section 1.5.8) with the entry points of manager modules in the server. An incoming RPC that contains a ''typed'' object UUID in its binding will be automatically vectored by the server's runtime to the appropriate associated type manager.

The creation of object UUIDs, the determination of what (if anything) constitutes an object for a server application, and the association of these objects' UUIDs into collective types are all application design decisions. If an application makes use of object UUIDs, it makes them accessible to clients by exporting them with its bindings; how this is done can be seen in Step B11 (Section 1.5.16).

1.5.7.1 Object UUIDs

Object UUIDs have a double use in the routing of RPCs, and you may at first find this a bit confusing. One use, which involves typing into object groups, was described in Section 1.5.7.

Object UUIDs are also used in the DCE RPC binding mechanism. The details of RPC binding are explained briefly in Steps B5 (Section 1.5.10) and B10 (Section 1.5.15), and more thoroughly in Section 2.1.5 of Chapter 2. It all comes down to this: clients normally import only *partial* bindings from the namespace. These will carry them only as far as the RPC daemon on the destination server's host; it is the daemon's job to complete the binding with a dynamic endpoint.

This means that some registration of bindings must be done by a server at its host RPC daemon (**rpcd**, also known as an endpoint mapper). Step B10 (Section 1.5.15) describes an example of this. The minimum two items that have to be registered are interface UUIDs and bindings (the latter of which contains the server's dynamically allocated endpoints). With this information available, the endpoint mapper can inspect the incoming RPCs interface UUIDs, select one of the endpoints that was registered under them, and complete the partial bindings. In addition, a server can register its object UUIDs with its endpoint mapper. This allows lookups of endpoints by object UUID rather than interface UUID; the advantage is that object UUIDs are much more specific than interface UUIDs, which may be registered by multiple servers at the same host.

Thus, the type UUIDs and the type manager vectoring mechanism have nothing to do with the use of the object UUIDs themselves as lookups for the host endpoint mapper. The former occurs after the latter happens, at the server. Note also that the latter typically happens only once in an uninterrupted client/server session; after the partial binding is completed, communications proceed directly between the client and server. Type manager vectoring, on the other hand, occurs again and again, every time an incoming RPC contains a typed object UUID.

1.5.7.2 Initializing the Mechanisms that Rely on Object UUIDs

The very different nature of the two mechanisms just discussed (type manager vectoring and endpoint mapping) is somewhat obscured by the order in which they are initialized in the steps in this chapter. The following list shows the relevant server steps, with an indication in each instance of which mechanism they belong to or are related to:

Step B2 Groups of object UUIDs are associated under type UUIDs in the RPC runtime; related to the type vectoring mechanism.

Step B3 Each type UUID is associated with a manager EPV (in the RPC runtime); related to the type vectoring mechanism.

Step B10 Object UUIDs and server endpoints are registered with the server's endpoint mapper; related to the endpoint mapping mechanism.

Step B11 The server bindings (containing the object UUIDs) are exported into the namespace; related to the endpoint mapping mechanism.

See Figure 1-4 for a schematic illustration of how object typing is used by the RPC runtime to vector remote calls to the correct server manager code.

Note that it is not necessary to call **rpc_object_set_type**() at all if you intend to register only one interface (see Step B3, Section 1.5.8) and you do so with a nil *mgr_type_uuid*.

As mentioned previously, a much more detailed discussion of the use of object UUIDs in bindings can be found in Section 2.1.8.

1.5.8 Step B3/Server: Register the Interface, Type UUID, and EPV with RPC Runtime

The server makes the RPC library call

```
rpc_server_register_if(if_handle, mgr_type_uuid, mgr_epv,
                                &status);
```

to register the interface specified by *if_handle* with the RPC runtime. By specifying *mgr_type_uuid* and *mgr_epv*, the server can arrange things so that incoming RPCs whose bindings contain object UUIDs that fall into its (the server's) registered type will be vectored by the RPC runtime to the registered manager.

This option is useful even when only one manager (often called a ''type manager'') is implemented. It allows the server (in conjunction with Step B2) to register categories of object UUIDs with the RPC runtime, and then to export bindings with those UUIDs into the namespace. Incoming calls containing partial bindings that have been imported from the namespace entries that the server exported to will be certain to reach that server, even though other servers on the same host may have exported the same interface. For a fuller explanation of binding, see Section 2.1.5.

1.5.8.1 Type Managers

Generally speaking, type managers are a way of dynamically tailoring interfaces to the various types of object presented to the server for its operations. For example, in the print service application that we have been discussing, the interface could be set up to handle different kinds of printers. Clients would request the same print operations through the same printer interface, but these requests would be dispatched by the RPC runtime to different implementations of the requests, depending on the type of the printer object UUID that is accompanying the incoming RPC. The different implementation modules in the server are called type managers; for each registered type of object there is a different set of routines implementing the interface. When there is only one type of object (the default case), there is only one manager in the server; this is how the term ''manager'' is usually used.

Normally, this runtime dispatching mechanism is triggered by the object UUID (if there is one) in the binding the client uses to make the remote call. The server, when it starts up, makes a series of calls in order to do the following:

- Associate each of the possible incoming object UUIDs with an object type
- Register each object and associated manager module with the RPC runtime

From then on, the dispatching of incoming calls is handled entirely by the RPC runtime. Where there is more than one type manager registered by the server, and an incoming call does not contain an object UUID in its binding, the runtime either chooses a default manager or rejects the call.

Of course, an application can always choose to do its own dispatching of requests among whatever object type handlers it chooses to implement. However, the mechanism provided by the DCE RPC runtime is a very convenient organizational tool and can help keep the interface code clean.

The following subsections describe in outline form the steps that you can follow in coding a print server application with two type managers: one for line printers and one for laser printers.

1.5.8.2 IDL Definitions

The first step is to plan and write the interface definition. First you run the **uuidgen** command

```
uuidgen -i > printer_if.idl
```

to create a skeletal IDL file with the name **printer_if.idl**. The file will contain an interface UUID and version number and nothing else; the rest of the necessary IDL code must be completed by the developer (namely, you).

1.5.8.3 Printer Interface Definition

It is up to the application developer to both define and implement the printer interface operations. Here it will be assumed that this has been done, and that the interface is defined in the file **printer_if.idl**, and implemented in two **.c** files, one for each of the type managers.

The two print manager types will be the following:

L_PR The **L_PR** type manager will manage line printers.

LS_PR The **LS_PR** type manager will manage laser printers.

In order to arrange things for the RPC runtime to dispatch requests between the two managers, some additional initialization code is required.

When the completed **printer_if.idl** file is processed by the IDL compiler, the default output will be a **printer_if.h** file, as well as client stub and server stub files, whose default names will be **printer_if_cstub.c** and **printer_if_sstub.c**, respectively.

The default behavior of IDL is to generate a manager EPV structure in the server stub, consisting of one field for each of the operations defined in the interface definition file. If there is only one server manager type, and if the names of the routines in the manager code are the same as the operations in the interface definition file, then declarations in the server stub code automatically initialize (by name) the EPV structure with the addresses of the appropriate manager routines.

If there is more than one manager type (as is the case in this example), the procedure is different. The interface definition file should be processed with the **-no_mepv** option; this prevents the generation of an initialized management EPV structure in the server stub code. This now becomes the responsibility of the manager code modules, and is done as follows.

Each manager module must declare its own EPV structure variable. The declaration of the structure type is automatically generated by IDL in its header file output and can be found there. In the present example it will be assumed that the type name is **printer_if_v1_0_epv_t**. Since the initialization of these structures is done in the manager code, the managers' implementations of the interface operations can have arbitrary names, although it is a good idea to maintain some correspondence for the sake of clarity.

To show how all this works, the following is a skeletal representation of parts of the two manager code modules. First, the **L_PR** type manager module:

```
#include printer_if.h

static void lpr_op1( <...args...> )
{ <...code...> };

static void lpr_op2( <...args...> )
{ <...code...> };

static void lpr_op3( <...args...> )
{ <...code...> };

        <...>

globaldef printer_if_v1_0_epv_t line_printer_v1_0_manager_epv
        ={lpr_op1, lpr_op2, lpr_op3, <...> };
```

And for the **LS_PR** type manager:

```
#include printer_if.h

static void ls_pr_op1( <...args...> )
{ <...code...> };

static void ls_pr_op2( <...args...> )
```

```
{ <...code...> };

static void ls_pr_op3( <...args...> )
{ <...code...> };

        <...>

globaldef printer_if_v1_0_epv_t ls_printer_v1_0_manager_epv =
        {ls_pr_op1, ls_pr_op2, ls_pr_op3, <...> };
```

Note that the placement of addresses in these vectors is significant; requests are vectored through the interface to operations in one of the two managers, so the addresses in the vector should be the same as their corresponding definition in the **.idl** file.

1.5.8.4 Server Initialization at Runtime

This subsection lists the routine calls that a server with multitype managers would normally have to make in order to initialize itself. Most of these calls will have to be made by any server; they can be seen in context in the **timop_server.c** file, part of the source code for the **timop** DCE sample application described in Chapter 3. Definitive information on all the calls can be found in the *OSF DCE Application Development Reference*. (Note that authentication and key management is ignored here.)

1. Create a UUID for each object to be exported. This is done *only* the first time the server is started. For example:

   ```
   uuid_create((uuid_t *) printer_obj, &status);
   ```

2. Create manager type object UUIDs. This is done *only* the first time the server is started. These object UUIDs should thereafter be stored in a local file and read in by the server whenever it is restarted. If the file cannot be found or read, new UUIDs can be generated. For example:

   ```
   uuid_create((uuid_t *) line_printer_type, &status);

   uuid_create((uuid_t *) ls_printer_type, &status);
   ```

3. Associate object UUID(s) with type UUID(s). This call will be made repeatedly, as many times as is necessary to associate all the objects with their types. For example:

```
rpc_object_set_type((uuid_t *) printer_obj,
                    (uuid_t *) ls_printer_type,
                    &status);
```

4. Register the interface(s), type UUIDs, and EPVs with the RPC runtime. This is where type managers, if any, are registered. This call may be (and probably will be) made more than once. For example:

```
rpc_server_register_if(printer_if_v1_0_s_ifspec,
                       (uuid_t *) line_printer_type,
                       (rpc_mgr_epv_t) line_printer_v1_0_
                       manager_epv,&status);

rpc_server_register_if(printer_if_v1_0_s_ifspec,
                       (uuid_t *) ls_printer_type,
                       (rpc_mgr_epv_t) ls_printer_v1_0_
                       manager_epv,&status);
```

5. Request bindings under the specified protocol sequence from the RPC runtime. For example:

```
rpc_server_use_protseq((unsigned_char_t *) protseq,
                       max_call_requests,
                       &status );
```

6. Get the binding handle(s).

   The bindings retrieved with this call are *full* bindings: they include endpoint information. However, as explained elsewhere in this chapter, endpoint information is *not* exported by the name service into the namespace with the rest of the binding when **rpc_ns_binding_export**() (see step 8) is called. Instead, endpoints are registered locally with the host's endpoint mapper, which will intercept inbound RPCs with partial bindings, insert a registered endpoint, and send it on to the server. For example:

```
rpc_server_inq_bindings((rpc_binding_vector_t **)
                        binding_vector,&status );
```

7. Register the endpoints with the endpoint mapper. Again, this call will be repeated for each object the server intends to separately export into the namespace and register with the RPC runtime. The object's UUID, which was generated earlier by a call to **uuid_create**(), is here associated with the binding that the server intends to export to the object's name entry, and registered with the endpoint mapper.

   The purpose of this call is to associate specific endpoints (server addresses) with specific object UUIDs, so that the host endpoint mapper will be able to complete partial bindings in incoming RPCs and send them to the correct destination server. For example:

   ```
   rpc_ep_register(printer_if_v1_0_s_ifspec,
                   (rpc_binding_vector_t *) binding_vector,
                   (uuid_vector_t *) object_uuid_vector,
                   (unsigned_char_t *) annotation,
                   &status );
   ```

   (Do not confuse this call with the **rpc_server_register_if**() call described in step 4, which is used to register interfaces and manager types with the RPC runtime.)

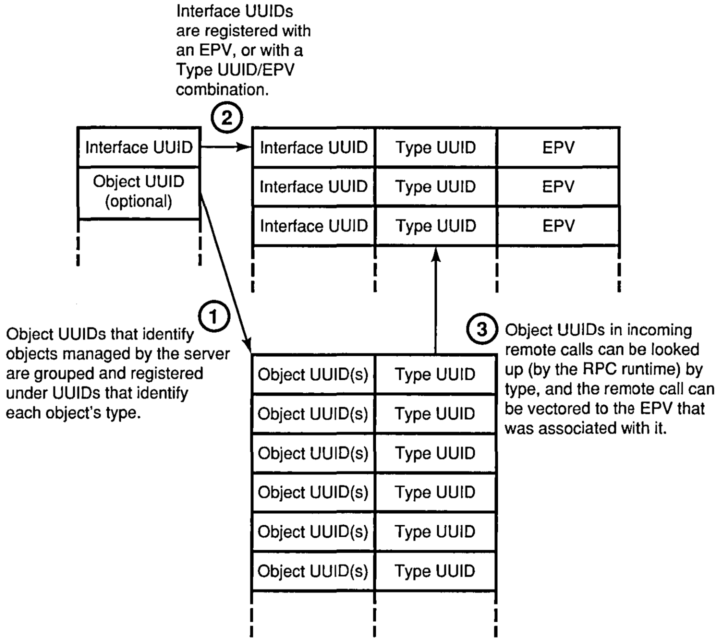
8. Export the object's binding into the namespace. In this example, only a single (partial) binding, rather than a binding vector, will be exported for each specific object into that object's name entry. For example:

   ```
   rpc_ns_binding_export(entry_name_syntax,
                         (unsigned_char_t *) entry_name,
                         printer_if_v1_0_s_ifspec,
                         (rpc_binding_vector_t *) binding_
                         vector,(uuid_vector_t *) object_
                         uuid_vector,&status );
   ```

Figure 1-3 illustrates how (1) object UUIDs are registered under type UUIDs with the RPC runtime, and then (2) how interface UUID/type UUID/EPV combinations are registered separately. The result is that when (3) the server's RPC runtime detects an object UUID in an incoming remote procedure call, it attempts to look up an associated type UUID for that object. If it finds one, it then looks to see whether that type has been associated with the incoming interface UUID; if so, the incoming call is vectored into the EPV associated with that interface/type combination.

The group of object UUIDs identified by (1) in the figure is the information that the server registers with the RPC runtime. This can also be interpreted as the UUID information contained in an RPC incoming from a client. The areas for the storage of the information identified by numbers (2) and (3) are maintained by the server's runtime.

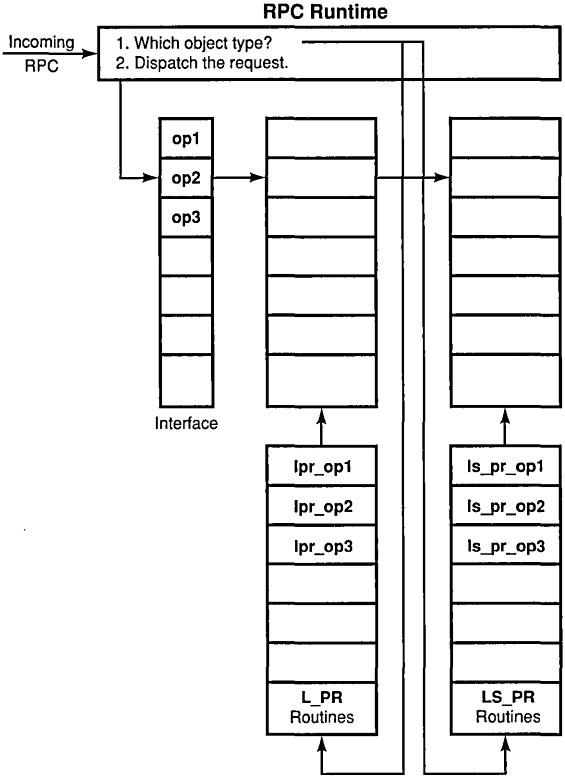
Figure 1–3. Object/Type and Interface/Type/EPV Registration


Interface UUIDs are registered with an EPV, or with a Type UUID/EPV combination.
2
Interface UUID
Object UUID (optional)
Interface UUID
Type UUID
EPV
Interface UUID
Type UUID
EPV
Interface UUID
Type UUID
EPV
1
Object UUIDs that identify objects managed by the server are grouped and registered under UUIDs that identify each object's type.
3
Object UUIDs in incoming remote calls can be looked up (by the RPC runtime) by type, and the remote call can be vectored to the EPV that was associated with it.
Object UUID(s)
Type UUID
Object UUID(s)
Type UUID
Object UUID(s)
Type UUID
Object UUID(s)
Type UUID
Object UUID(s)
Type UUID
Object UUID(s)
Type UUID

Note that type manager UUID and EPV do not have to be registered if there is only one EPV, which is the IDL-generated default. **NULL**s can then be passed for these arguments instead.

Figure 1-4 illustrates how the server's RPC runtime dispatches RPCs to the two different type managers, depending on the object type of the incoming call's object UUID.

Figure 1–4. RPC Server Runtime Dispatching on the Basis of Object Type



When a remote call is received by the server's runtime, the runtime looks for an object UUID in the call's argument list. If one is found, the runtime next determines whether that object UUID has been associated with a type UUID. If it has been, the runtime vectors the request to the manager code that has been registered with that type. Thus, in the figure, if **op2** was requested, and the object UUID contained in the argument list for the incoming call was registered with the **ls_printer_type** UUID, then the request is vectored to the **ls_pr_op2** routine.

Note that object/type associations have nothing to do with endpoint registration (see Step B10, Section 1.5.15).

Note also that the arrows in Figure 1-4 do not necessarily represent activities that occur across the network.

1.5.9 Step B4/Server: Specify Multithreadedness

The *max_calls_exec* parameter to the **rpc_server_listen()** routine specifies how many incoming requests (from different clients) the server is prepared to handle concurrently. In effect, *max_calls_exec* specifies the upper limit for the number of RPC threads that will be spawned by the RPC runtime to handle incoming remote procedure calls. Thus, an important side effect of **rpc_server_listen()**, when the specified concurrency is greater than 1, is to create multiple threads of execution in the server.

The threads are automatically spawned to handle whatever operation is requested by the client. If the maximum number of manager threads is already active and more incoming calls arrive, the RPC runtime buffers them in a call request buffer. In DCE Version 1.0, the capacity of this buffer is a maximum of eight times the value specified in the *max_calls_exec* parameter to **rpc_server_listen()**. As active operations are completed and threads are terminated, new threads are spawned to handle the buffered calls. Incoming calls beyond the call request buffer capacity are rejected (with an error code) by the RPC runtime.

Although the manager threads are automatically spawned and terminated by the RPC runtime, the developer is responsible for coding the manager routines according to thread-safe guidelines so that the threads will execute properly. For further information on thread-safe programming practices, see Section 2.2.

1.5.10 Step B5/Server: Tell RPC Runtime What Protocol Sequences to Use

The server calls the RPC library routine

```
rpc_server_use_protseq(protseq, max_call_requests, &status);
```

or

```
rpc_server_use_all_protseqs(max_call_requests, &status);
```

to obtain a set of endpoints on which to receive incoming calls. This routine only causes the endpoints to be allocated; they are not returned to the server by this routine (see Step B6, Section 1.5.11, for an explanation of how that happens).

In this step, the server begins the process of actually setting up the information that its clients will need in order to bind to it. There are several ways the server can do this. The usual method is to request from the RPC runtime a vector of binding handles; the server then distributes these handles into entries in the namespace, where they can be located and imported by clients.

However, not everything that was in the handles is exported. To reduce wear and tear on the replicated namespace, the dynamic part of the binding information (namely, the host endpoint address) is held back by the RPC runtime. Next the server must register the same bindings again, this time with its host's endpoint mapper (the RPC daemon, **rpcd**, resident on every DCE machine); this time the endpoints are not withheld by the runtime. Later on, partial bindings incoming from clients will each be intercepted by the server's host's endpoint mapper, filled in with a valid endpoint, and sent on to the client's server. Note that the intervention of the endpoint mapper occurs only on a client's first call to a server; after the binding has been filled in with an endpoint, subsequent calls go straight to the server.

1.5.10.1 Well-Known Endpoints

There are, however, other binding possibilities. Two will be mentioned here. The first involves well-known endpoints. A server that uses well-known endpoints does so by declaring them (with the **endpoint** attribute) in its **.idl** file. At the present step it calls the routine

```
rpc_server_use_all_protseqs_if(max_call_requests,
                               if_handle, &status);
```

or

```
rpc_server_use_protseq_if(protseq, max_call_requests,
                          if_handle, &status);
```

to have a vector of binding handles made up for it; but instead of getting a dynamically allocated set of endpoints in the handles, as would happen with a call to **rpc_server_use_all_protseqs( )** or **rpc_server_use_protseq( )**, it receives the endpoint and protocol information that was coded in the interface specification (or in the argument, if the latter call is chosen). The server needs these binding handles in order to be able to put something out in the namespace that clients can import and use to make remote calls with. It exports the handles as described in Step B11 (Section 1.5.16). The binding handles are returned to the server in the following step.

1.5.10.2 String Bindings

A server can also use string bindings. Doing this allows an application to avoid using the namespace and name service altogether. However, in this case, the server assumes the responsibility for making sure that the binding is compatible; that is, that its protocol sequence is supported by both the RPC runtime and the operating system. The **rpc_network_inq_protseqs( )** routine returns a vector of all the supported protocol sequences, and **rpc_network_is_protseq_valid( )** tests whether a given sequence is supported (see Step C1, Section 1.5.18, for further information).

To convert a binding handle (obtained in Step B6, Section 1.5.11) to a string binding, the server calls

```
rpc_binding_to_string_binding(binding_handle, string_binding,
                             &status);
```

The resulting string can be sent by any convenient means to the client, or deposited in some place where the client can pick it up later. Once the client has possession of the string, it calls

```
rpc_binding_from_string_binding(str_binding, binding_handle,
                               &status) ;
```

to transform the string back into a usable handle.

1.5.10.3 Summary

The point of the foregoing discussion is to show that none of the namespace registration and exporting operations is absolutely indispensable under all conditions. A client could receive a string binding from user input, construct a binding handle from it, and initiate RPC operations with its server, all without any intervention on the part of the name service or endpoint mapper (assuming that the server uses well-known endpoints).

The binding and name service routines tend to combine functionality in various ways that can obscure some of the things that actually occur when they are called. You should not be confused by this.

1.5.11 Step B6/Server: Request for Bindings from RPC Runtime

The server calls the RPC library routine

```
rpc_server_inq_bindings(binding_vector, &status);
```

in order to obtain the binding handles that the RPC runtime created for it in the previous step.

1.5.12 Step B7/Server: Register the Authentication Information with RPC Runtime

The server makes the RPC library call

```
rpc_server_register_auth_info(server_princ_name, authn_svc,
                              get_key_fn, arg, &status);
```

to register an authentication service to use for authenticated RPC.

The decision whether or not to use authenticated RPC is something of a cooperative matter between the client and the server. The server registers its preferences in the present call, but when the client calls **rpc_binding_set_auth_info**( ), it registers its preferences about these same things. The client's and server's choices are not required to agree in order for the client to successfully reach the server. If the client's authentication and authorization choices do not agree with what the server expects, it is up to the server to decide whether or not to go ahead with the operations, and how far to cooperate with client requests. See Steps C2 (Section 1.5.19) and D4 (Section 1.5.24) for further details.

This is the essential server-side call to set up authenticated RPC.

1.5.13 Step B8/Server: Establish the Server Principal Identity

When first invoked, a server process uses the login context of the user who invoked it, until it assumes its own identity by accessing its secret key, which is analogous to a user's password, and using it to get its own login context. The first in the series of calls it would have to make to perform this switch-over is the following:

```
sec_login_setup_identity(princ_name, flags, &login_context,
                            &status) ;
```

The server passes its principal name (at this time it is running under the context of whatever principal invoked it); it receives its own login context in return, which it will need in order to validate its identity. It then makes the following call in order to get its secret key (in fact, its password):

```
sec_key_mgmt_get_key(authn_svc, arg, princ_name, key_vno,
                         &keydata, &status);
```

It can now make the following call to validate its own identity; if successful, the server's runtime will receive a ticket-granting ticket from the Security Service's authentication service. Possession of this ticket is a prerequisite for getting tickets to any other service, and these tickets are what authenticated RPC is based on.

```
sec_login_validate_identity(login_context, passwd,
                              reset_passwd, auth_src, &status);
```

If all goes well, the next (and final) call will retrieve the server's own login context, which it can then use in a call to **rpc_binding_set_auth_info( )**.

```
sec_login_get_current_context(login_context, &status);
```

At this point the server has everything it needs (assuming that it knows its principal name) to either register its authentication information with the RPC runtime prior to receiving authenticated requests from clients, or if it wishes, to authenticate itself to some other server.

Of course, it is possible for a server simply to continue using its inherited login context. In that case, all it needs to do is make the last-mentioned call in order to explicitly get its login context. If it does not yet know its principal name, it can first make the following call:

```
sec_login_get_pwent(login_context, &pwent, &status);
```

The (inherited) principal name will be found in *pwent.pw_name*.

When a server has its own identity, it takes on responsibility for the upkeep of its password, which was returned by **sec_key_mgmt_get_key**( ) in the sequence above. What this involves is described in greater detail in Chapter 2 in Section 2.3.

1.5.14 Step B9/Server: Plan What To Do When the Server Terminates

From the point of view of the server, the call to **rpc_server_listen**( ) blocks until one of the server's manager routines calls the **rpc_mgmt_stop_server_listening**( ) routine, or until a client makes a successful remote **rpc_mgmt_stop_server_listening**( ) call to the server.

When either of these things happens, the RPC runtime stops accepting incoming client requests to the server. When all the currently executing operations are completed, the call to **rpc_server_listen**( ) returns.

Server operations can also be terminated by an exception or signal. DCE Threads defines all exceptions as ‘‘terminating,’’ which means that execution must be caught by an exception handler (if one exists) and then be resumed there, or the process will be terminated. Certain signals are defined by DCE Threads as exceptions, which means that these signals have the same general characteristics as exceptions.

For more information on the DCE Threads exception handling interface, see Chapter 7.

1.5.14.1 Management Interface

The **rpc_mgmt_stop_server_listening**( ) routine is part of the DCE RPC management interface, a group of routines that allows clients or servers to find out various facts about the characteristics of a server, and (in some cases) to alter them. All of the management routines have names beginning with **rpc_mgmt_**; details about each of them can be found in the *OSF DCE Application Development Reference*.

One of the management routines, **rpc_mgmt_set_authorization_fn**( ), allows a server to install a monitor routine to intercept clients' remote calls to some of the management routines. When a client calls one of these routines (for example, **rpc_mgmt_stop_server_listening**( )) in order to perform some task on that server, the server's installed routine is automatically called first by the RPC runtime. The routine receives a copy of the calling client's binding, and an argument identifying the management call attempted. The server can now decide what to do with the attempt: returning TRUE from the monitor routine allows the original call to proceed, while returning FALSE causes an error to be returned to the calling client.

1.5.14.2 Server Termination

If (or when) the server terminates execution, it should unregister and unexport any information it previously caused to be placed in the namespace or its host's endpoint mapper. This will prevent future prospective clients from being misled into attempting to reach the server when it does not exist, and also will help to conserve resources in the namespace and the local endpoint database. Note that this namespace housekeeping should be performed in addition to whatever other application-specific cleanup may be required before termination.

Executing the following series of routines will properly clean up after an application that exported partial bindings in the normal fashion. The first call is

```
rpc_ns_binding_unexport(entry_name_syntax, entry_name,
                        if_handles, obj_uuid_vector, &status);
```

This routine removes from the specified namespace entry the server's previously exported binding handles.

```
rpc_ep_unregister(if_handle, binding_vector, obj_uuid_vector,
                  &status);
```

This routine unregisters the server's address information from the local endpoint mapper's database.

```
rpc_server_unregister_if(if_handle, mgr_type_uuid, &status);
```

This routine unregisters from the RPC runtime the (previously registered) association between the *if_handle* and the server's manager EPV. If more than one manager EPV was registered for the server, this routine can be used to unregister one or all of them.

1.5.15 Step B10/Server: Register the Binding Information with the Endpoint Mapper

The server makes the RPC library routine call

```
rpc_ep_register(if_handle, binding_vector, obj_uuid_vector,
                annotation, &status);
```

to register the (dynamically allocated) endpoints that were returned in the binding handles it just acquired. If the server uses well-known endpoints, it does not have to call this routine. However, it does no harm to do so, since prospective clients that happen to possess only a partial binding will not necessarily be able to reach the server otherwise.

1.5.15.1 The Purpose of Registering Endpoints

The purpose of registering endpoints together with object UUIDs is to account for all possible incoming object UUIDs (that is, object UUIDs that could appear in incoming partial bindings arriving at the endpoint mapper), and to associate with each of them one of the server's allocated endpoints. Then the endpoint mapper can simply look up the object UUID, find an endpoint, insert it into the binding, and send the RPC on to its destination.

The server has received (in its binding handles) a certain number of endpoints dynamically allocated on its host machine. However, prospective clients who import this binding information from the namespace will start out with partial bindings when they first try to contact their server, and the partial binding will get them only as far as the server's host's endpoint mapper daemon, **rpcd**. The purpose of **rpc_ep_register( )** is to let the endpoint mapper know what endpoints belong to the server so that it can fill in the partial bindings as they arrive and route the incoming remote calls on their proper ways. Subsequent remote calls executed with the same bindings will go straight to the server, since the bindings are now complete.

An incoming RPC *always* has an interface UUID associated with it; therefore, if a server registers all of its endpoints with the interface it is offering, this will usually be sufficient for the endpoint mapper to send the incoming requests to one of the servers that offer the desired interface, even if there is more than one such server active on the machine. However, if the application is designed in such a way that the binding operation should not be generalized to the interface but must be made more specific (in other words, this server's clients should always bind *to this server and no other*, even if some other server happens to offer the same interface), then object UUIDs must be used to accomplish this.

Of course, the server's interface UUID must also be included in each object UUID/endpoint mapping, since no RPC will pass the endpoint mapper if it does not have a matching interface UUID for its destination server. Therefore, the endpoint mapper takes either two or three types of item to be registered, namely

- Interface UUID
- Endpoints

or

- Interface UUID
- Endpoints
- Object UUIDs

It then generates a cross-product table of all possible combinations of all values of the items. This allows it to find a good endpoint for every possible valid object UUID/interface UUID combination.

1.5.15.2 Summary

The endpoint mapper is the first point of decision for an incoming RPC with a partial binding. The mapper makes its decision *solely* on the basis of the contents of its endpoint map. The object/type and manager EPV registrations that were done in Steps B2 and B3 have no effect on the endpoint mapper. Only *after a client request arrives at the server* does the server's runtime routines vector the request among multiple managers, if type managers have been registered by the server. The endpoint mapper knows nothing about registered object types. (See Step B5, Section 1.5.10, for a further explanation of the role of the endpoint mapper in the binding process.)

Note that the call

```
rpc_ep_register_no_replace(if_handle, binding_vector,
                           obj_uuid_vector, annotation,
                           &status);
```

is used (instead of **rpc_ep_register**()) if multiple instances of the same server will be running on the same host. In other words, calling **rpc_ep_register_no_replace**() a second time with the same interface UUID, object UUID, and protocol sequence will not replace the earlier entries in the endpoint map, but merely add new ones. Obviously, different binding vectors should be passed in the different calls.

1.5.16 Step B11/Server: Export the Binding Information to the Namespace (CDS)

The server makes the RPC library call

```
rpc_ns_binding_export(entry_name_syntax, entry_name,
                      if_handle, binding_vector,
                      obj_uuid_vector, &status);
```

to export the allocated binding handles to the namespace. In the usual case, where the server's endpoints have been dynamically allocated to it, the endpoint information will not be included in the exported handles. Instead, this information will be filled in by the host's endpoint mapper as the partially bound handles arrive at the host in incoming RPCs (see Step B10, Section 1.5.15). However, if the endpoints are well-known, they will be included in the exported binding handles, and clients will thus import fully bound handles.

It is recommended that only one binding handle/object UUID pair be exported to each namespace entry, even though it is possible to export more than one of each per entry. Doing this will ensure that there is a strict determinable mapping from each name entry to each bound-to object.

A client must have a binding handle in order to reach a server, but it does not have to get the handle from the name service. See Step B5 (Section 1.5.10) for an explanation of the use of string bindings.

1.5.17 Step B12/Server: Listen for Incoming Service Requests

The server calls the RPC library routine

```
rpc_server_listen(max_calls_exec, &status);
```

in order to do the following:

- Specify the maximum number of concurrent remote procedure calls to execute

- Begin listening for incoming calls

This call normally begins a ''semi-infinite'' loop, execution of which is terminated only by one of the following events:

- One of the server's manager routines calls **rpc_mgmt_stop_server_listening**( ).
- One of the server's clients makes a remote call to **rpc_mgmt_stop_server_listening**( ). (Note that the server can intercept such a remote call and either allow or prevent it by installing a **rpc_mgmt_set_authorization_fn**( )).
- A signal or exception.

(See also Step B9, Section 1.5.14.)

1.5.18 Step C1/Client: Import the Binding Information from the Namespace (CDS)

The first important thing that the client does is to acquire a binding to the server it wants to request services from. From the client's point of view, there are several binding choices to be made.

The first choice is in regard to the binding method to be used; however, this is determined and implemented in Step A4 (Section 1.5.4) as part of the development coding process (the **.acf** file). The binding method chosen has an effect both on what the client has to do in the present step to acquire bindings, and subsequently on what it must do to maintain them. In this step, it will be assumed that either the explicit or implicit method was chosen. If auto-binding were chosen, there would be no need for a discussion, since the client would then have nothing to do.

1.5.18.1 Getting a Handle

The second choice involves how to get a binding handle. Again, this is a choice that is at least partially dependent on things that have already occurred. The client can always generate a binding handle for itself; the problem is where to get the information that belongs in it. There are two general solutions:

- The client imports from the namespace binding handles that already contain the necessary information, or
- The client receives the information in string form from user input, from a file, from another server, or from any other source. It then converts the string into a binding by calling **rpc_binding_from_string_binding( )**.

The normal way for a server to make its location known to clients is to export its binding information into the namespace. The client can then call the RPC name service library routines

```
rpc_ns_binding_import_begin(entry_name_syntax, entry_name,
                            if_handle, obj_uuid, &import_
                            context, &status);

rpc_ns_binding_import_next(import_context, &binding_handle,
                            &status);

rpc_ns_binding_import_done(import_context, &status);
```

to import one or more bindings from the specified namespace entry. The name service sees to it that only compatible bindings exported under the specified interface, with the optionally specified object UUID, will be returned to the client. (Note that the interface specification is *not* contained in the binding, although it is exported to the namespace entry where it is used by the name service for matching entries to prospective importers.) The object UUID specified by *obj_uuid* is contained in the binding, if it is present. This is the object UUID that was (optionally) registered under a type UUID in Step B2 (Section 1.5.7). Even if *obj_uuid* is not specified in the import call, it will be returned in the binding handle(s) if it was exported by the server in Step B11 (Section 1.5.16).

1.5.18.2 Entry Name

To determine how the client knows the entry name to import from, use the simplest and most flexible mechanism: have the user type it in on the command line. This is the method used by the **timop** client (see **timop_client.c** in Section 3.2.6.5).

1.5.18.3 Binding Compatibility

The protocol sequence used must be supported by both the RPC runtime and the operating system on the client's machine. However, the RPC runtime implicitly takes care of binding compatibility when it returns bindings to importing clients; only compatible bindings are returned.

The routines **rpc_network_inq_protseqs**( ) and **rpc_network_is_protseq_valid**( ) can be used to return all supported protocol sequences and to determine whether a specified protocol is supported, respectively.

To find what protocol sequence is used in a binding handle, make the following series of calls:

```
rpc_binding_to_string_binding(binding_handle, &string_binding,
                              &status);

rpc_string_binding_parse(string_binding, NULL, &protseq, NULL,
                         NULL, NULL, &status);
```

Now all you have to do is compare strings.

Note that in **timop** the client's compatible protocol is hardcoded into the program. The server generates its bindings with the **rpc_server_use_all_protseqs**( ) call so that, on its side, there is no need for any further testing.

1.5.19 Step C2/Client: Annotate the Binding Handle for Security

Now that the client has a binding, it is almost ready to begin RPC operations. One last preliminary task remains; namely, to specify various security-related parameters to the RPC runtime, which will govern the (security) conduct of the ensuing client/server relationship. If the client does not require authentication, it can skip this step completely. The result will be that no authentication will take place between the client and server. It will then be up to the server to decide how far to go with an unauthenticated client (see Step D4, Section 1.5.24).

1.5.19.1 Preparation

What the client usually really wants to do here is call the routine **rpc_binding_set_auth_info**( ) in order to specify all the necessary security parameters. However, when it does this, it should be able to specify its server's principal name so that the server it binds to can be authenticated *to the client*. (The server's principal name is the name by which the server is known to the Security Service.) The client must also supply a handle to its own login context when it calls **rpc_binding_set_auth_info**( ).

There are several ways to determine the server's principal name:

- The server's principal name could be hardcoded in the client. This is not recommended practice for reasons of robustness and flexibility.
- The client can be handed the name as input from the command line when it is invoked.
- The name can be stored in the namespace.
- The principal name can be the same as the name entry (binding information) name.
- The client can query the server's principal name by calling **rpc_mgmt_inq_princ_name**( ). It can then check group membership by calling **sec_rgy_pgo_is_member**( ).

The reason for checking group membership has to do with authorization-related decisions that the client may need to consider. It is not necessarily enough to know that a server has a certain identity; it may also be necessary that it belong to a certain group in order for it to be fully authorized, from the client's point of view, to receive the data that the client will send. In other words, the client may need to make a decision about the server similar in nature to that which the server makes about the client in Step D4 (Section 1.5.24), when it checks the client's authorization, via ACLs, to do the things it wants to do. Security can be just as important for the client as for the server; this is the justification for having to make the extra calls described here.

Getting the client's login context is done with the following Security Service library routine:

```
sec_login_get_current_context(login_context, &status);
```

However, this is not usually necessary. The client can, by passing a **NULL** value to **rpc_binding_set_auth_info**( ) (see Section 1.5.19.2), simply use its default login context.

In any case, note that this login context already exists; the client merely retrieves it. (The client inherited its login context from the user principal who executed it.) The client can now set up for authenticated RPC.

1.5.19.2 Setting Up for Authenticated RPC

The client makes the following call in order to set up the security characteristics of the communications it is about to enter into with the server:

```
rpc_binding_set_auth_info(binding_handle, server_princ_name,
                          protect_level, authn_svc,
                          login_context, authz_svc, &status);
```

The security parameters specified here include *protect_level* for level of protection performed (for example, authenticate only at the beginning of each RPC, or authenticate everything received by the server), *authn_svc* for the authentication service (no authentication at all can be specified here), and *authz_svc* for the type of client authorization information that will be supplied to the server (see Steps D2 to D4, Sections 1.5.22 to 1.5.24 for more details).

The usual practice is to pass **NULL** for *login_context* here, and thus use the default context.

Note that it is the client who chooses whether or not to use authenticated RPC, as well as the level of authentication, and how much authorization information about itself to send. It is then up to the server to accept this arrangement or reject it, or to allow some limited operation with the client, or whatever else it might decide. The server decides which authentication to use (in Step B7, Section 1.5.12). The client also specifies an authentication service (in *authn_svc*), but if this differs from what the server specified, the call to **rpc_binding_set_auth_info**( ) will fail and an error will be returned to the client.

There is an important difference between the rationales of authentication and authorization. Authentication is performed by the RPC runtime and is only indirectly felt by client and server; authorization, however, is for the most part implemented explicitly in the server code if it is implemented at all. This difference is the reason for the larger number of authentication-related arguments that have to be specified in this step. More about this subject can be found in Steps D2 to D4 (Sections 1.5.22 to 1.5.24), where the transactions are seen from the server's point of view.

For further information about authenticated RPC, see Chapters 13 and 42. Chapter 2 contains sections on server key management, which is part of the authenticated RPC mechanism, and on the practical details involved in writing an ACL manager.

1.5.20 Step C3/Client: Invoke an RPC Interface Operation

This step is the culmination of all the foregoing steps; here the client makes its first remote call to the server. This call, which will obviously be application specific (its definition was specified in the application's **.idl** file in Step A3 (Section 1.5.3), and possibly modified by the **.acf** file written in Step A4 (Section 1.5.4)), will look something like the following:

```
my_rpc_op(binding_handle, arg1, arg2, arg3);
```

Note that the presence of the binding handle as a parameter means that explicit binding handles are being used (again, a Step A4 decision).

Note also that after all the preceding talk about interfaces, no interface handle appears in the parameter list. The RPC runtime takes care internally of making sure that the interface offered by the server exactly matches what the client expects. The **my_rpc_op()** routine was (or should have been) defined as part of the application's interface back in Step A3. When the client calls **my_rpc_op()** in the present step, the client stub code (which was generated during the IDL compilation step) will include the correct UUID for the interface the routine is associated with in the data sent out on the network. The RPC runtime uses the interface specification included with each RPC as a ''fingerprint'' to ensure that the operation being requested of a server is in fact implemented by that server. This ensures that interface compatibility is never dependent on the vagaries of application code.

1.5.20.1 The Possibility of Binding Failure

Perhaps the most important thing to mention about this step is that it may not at first succeed. Remember that the client imported a *partial* binding to the server. Completion of the binding, and therefore of the remote call, depends on the endpoint mapper's being able to successfully complete the incoming binding with a good endpoint for either the specified server (if one is specified) or for one of its own choosing. This in turn depends on the up-to-dateness of the host's endpoint database, and that depends on such things as other servers' being conscientious about unregistering themselves when terminating, and so on. Even the target host specified may not be valid when the call is made because of any one of the various network problems that can occur.

In other words, the client should regard an unused binding not as a firm promise that comes directly from the server, but rather as a well-meant expression of intent passed on by the name service and based on circumstances not entirely under anyone's control. This is the reason for the series of binding import calls described in Step C1 (Section 1.5.18).

The prudent thing for a client to do after importing a binding is, therefore, to assume that it will have to perform one or more times a series of steps something like the contents of the following loop:

1. Annotate the binding handle for security.
2. Try it out: attempt a remote call with it.
3. If the call succeeds, discard the binding import context and proceed to step 5 in this loop.
4. Otherwise, if the call fails, import the next binding and return to step 1 in this loop.
5. Proceed with remote operations until finished.

If all imported bindings happen to fail, this could be because the client's cache of bindings has become stale. The client could then try calling **rpc_ns_mgmt_handle_set_exp_age**( ) with a low time-out value, and then retry the above loop. A last resort could be to allow the user to type in a string binding.

Note that if you are using the auto-binding method and the binding becomes unusable for some reason, the RPC runtime will rebind under most conditions.

1.5.20.2 The Result of Successful Binding

If **my_rpc_op**( ) or its equivalent does succeed, the binding will as a result be complete (even if it was partial before), and the information in it can be regarded with much more assurance from then on. Subsequent remote procedure calls by the client to the same server will go straight to the bound-to server.

1.5.21 Step D1/Server: Wake Up in Manager Routine

As explained in Step B4 (Section 1.5.9), server threads are automatically spawned by the RPC runtime in the server manager to handle incoming remote procedure calls from clients. The number of calls that can be concurrently handled depends on the value of the *max_calls_exec* parameter specified in the call to **rpc_server_listen**( ) (see Step B4, Section 1.5.9). The thread is created by the RPC runtime and begins execution in the operation requested. When the operation is completed, the thread is automatically terminated (by the RPC runtime).

For further details on server multithreading, see Step B4 and Section 2.2. See also Part 2 of this guide and the *OSF DCE Application Development Reference* for a comprehensive discussion of DCE Threads.

1.5.22 Step D2/Server: Get the Client's PAC

As mentioned in the previous step, authentication, if it was specified by the client, has already occurred if the client's request is received by the server manager. If the client fails to authenticate itself to the server runtime, its remote procedure call fails.

Authentication, if specified by the client and offered by the server, is performed by the RPC runtime; it is not a responsibility of the application code. However, it is up to the application to formulate its own security policy with regard to the client, based on the following:

- The level at which the client has been authenticated.
- The client's authorization; that is, whether the client should be allowed to access resources it may request.

In order to find out the client's authentication and authorization information, the server calls the following RPC library routine:

```
rpc_binding_inq_auth_client(binding_handle, privileges,
                            server_princ_name, protect_level,
                            authn_svc, authz_svc, &status);
```

The parameters in this call are analogous to the similarly named parameters in the registration routines called in Step C2 (Section 1.5.19). The server can learn what level of authentication, what authentication service, and what server principal name the client specified. Of most interest, however, are the *privileges* and *authz_svc* parameters.

The *privileges* parameter is a pointer to whatever information the client is willing to let the server know about its privilege attributes; *authz_svc* tells what this information is. It could be any one of the following:

- The client's Privilege Attribute Certificate (PAC), containing the client's principal and group UUIDs. These can be used to look up the client's privilege attributes in Access Control Lists, whose entries are keyed by principal and group UUID.
- The client's principal name (a string). This also can be used to look through Access Control Lists, provided that the lists have been annotated with such name strings.
- Nothing. The client chooses not to provide any authorization information.

From now on, it is the server's decision, as implemented by the developer, how to respond to the client's requests for services and resources, depending on the security information the server has learned about it. A non-ACL-based strategy may be implemented using the client's principal name string for lookups. The ACL-based strategy, which is supported by a DCE interface, is described further in the next step.

1.5.23 Step D3/Server: Get the Object's ACL

This step is reached if the client requests access to any object, resource, or service that is managed by the server, to which ACLs are attached. As previously mentioned, the application must implement its own ACL manager if it wants to use ACLs to control access to its resources. For further details on how to go about creating an ACL manager, see Section 2.4.

In order to allow applications to as easily as possible offer an ACL interface that is uniform with that used by the DCE components themselves, the remote ACL interface has been built into the DCE library, and client applications can perform operations on ACLs through another interface, also part of the DCE library, which calls through the remote interface to the appropriate manager. The remote interface, consisting of **rdacl_...()** calls, must be implemented by the server application; clients execute the local **sec_acl_...()** routines, which are linked to every DCE application as part of **libdce**.

For the client, all that is necessary is to possess a binding to the object whose ACL is to be operated on. As long as the application exposes the resources it manages as accessible objects (via the namespace), then the DCE ACL interface provides for a client's being able to bind to the object by calling **sec_acl_bind()**. (In fact, this kind of object-oriented binding model can be very useful, and is discussed in further detail in Chapter 2.) Note that the **sec_acl_...()** routines use an ''ACL handle'' to specify the object whose ACL is to be accessed, so **sec_acl_bind()** must always be called to obtain this handle, even if the client is already bound to the object's server.

There is also a user interface into the ACL operations, embodied in the **acl_edit** command. At the server level, definitions for a local internal management interface, consisting of **sec_acl_mgr_...()** calls, are given in the Security Service section of the *OSF DCE Application Development Reference*, which also contains reference pages for the other two ACL interfaces. This suggested interface is based on one used internally by the Security Service itself. (See the *OSF DCE User's Guide and Reference* for further information on **acl_edit**.)

1.5.24 Step D4/Server: Make the Authorization Decision

In this step, the server's ACL manager inspects the ACL of the resource or object under question, determines whether the client is authorized for the requested access, and takes the appropriate action. The algorithm used to do all this is application dependent.

The application may choose to implement more than one type of ACL (reflecting the different kinds of objects and resources to be protected), thus resulting in several ''type managers.'' For more information on this possibility, see Section 2.4 in Chapter 2.

Although it is up to the application to implement its own ACL storage, testing algorithms and manager types, there are certain DCE-wide design conventions that should be kept in mind and departed from only for good reason. Among these are the following:

- Standard DCE ACL entry types: the kinds of entry that can occur in an Access Control List (for example, **user**, **group**, and so on).
- Standard privilege attributes: the kinds of access that a principal can have to a protected object (for example, read, write, and so on).
- Standard inheritance rules: these rules govern the default characteristics of ACLs created for newly created objects.
- Standard access algorithm: the order in which the contents of a Privilege Attribute Certificate are matched against the various possible entry types.

Information about these topics for application developers designing their own ACL model can be found in the *OSF DCE User's Guide and Reference*, where all the DCE authorization conventions are described in detail.

1.5.25 Step D5/Server: Service the Request

If the client's request is determined to be properly authorized, then the requested operation can proceed.

Note that this step and Steps D3 and D4 are somewhat intertwined. Something like the following could occur:

1. The server wakes up in some routine defined in its manager code. For example, if the client executed the call **my_rpc_op**( ) (see Step C3, Section 1.5.20), then the server will wake up in the routine that implements this remote call.
2. Execution of the **my_rpc_op**( ) routine requires the **insert** privilege for the application's database **my_database**. So **my_rpc_op**( ) begins by checking the client's relevant privilege attribute by making an internal call to the application's ACL manager.

3. If the client is found to have the requisite privilege, **my_rpc_op()** proceeds.

Obviously, other dispatching schemes are possible.

The remote procedure executed in this step is written by the application developer.

1.5.26 Step D6/Server: Return the Results to the Client

At the completion of the operation, the RPC thread that was automatically spawned to execute it is terminated by the RPC runtime. As far as the server is concerned, it is still blocking on the call to **rpc_server_listen()** which was made back in Step B4 (Section 1.5.9). If *max_calls_exec* was specified to be greater than 1 in that call, other threads may still be executing at this time in response to other requests that have been received from other clients. In any case, the call to **rpc_server_listen()** will not return until one of the server's own management routines, or a client, makes a successful call to **rpc_mgmt_stop_server_listening()**. If this happens, the RPC runtime will stop accepting incoming client requests to the server. When all the currently executing operations have been completed, the call to **rpc_server_listen()** will return.

The other way that execution can be thrown out of the **rpc_server_listen()** call is as a result of a signal or exception. For more about this possibility, see Step B9 (Section 1.5.14).

1.5.27 Step D7/Server: Continue the Listen Loop

From the server's point of view, the result of completing the remotely called routine is that it reenters the ''listen'' loop it entered in Step B12 (Section 1.5.17), waiting for further remote calls. The server's runtime handles all the communications details of actually sending any requested data to the client.

1.5.28 Step E1/Client: Wake Up After the RPC Call

From the client's point of view, the server's return at the end of its remotely called routine results in the client's returning from a seemingly locally executed routine.

1.5.29 Step E2/Client: Continue

The client now goes on about its business, which may include performing other remote procedure calls.

Note that there is no housekeeping burden placed on the client with regard to the termination of the relationship with a server. However, a long-lived client might want to make use of the **rpc_binding_free**( ) or **rpc_binding_vector_free**( ) routines to free memory that was allocated for no-longer-used handles. The client could also call **rpc_ns_binding_import_done**( ) to clean up the resources used by the NSI routines. If another binding handle will be needed later on, then **rpc_ns_binding_import_begin**( ) will be recalled.

Chapter 2

Guidelines for Server Writers

This chapter consists of detailed discussions of some of the fundamental DCE services. Use of a DCE service or facility usually has two aspects, depending on whether you consider it from the point of view of a server or of a client. The server-side details are usually more numerous and more complex, and sometimes the client-side aspect disappears entirely. However, there are client-side implications in all of the discussions in this chapter.

2.1 Using the Name Service Interface

Correct use of the DCE RPC Name Service Interface (NSI) is essential to the operation of a distributed application, since NSI is the medium through which the application's distributed parts must find each other. NSI works with named database entries which are hierarchically organized into subdirectories and referenced by the familiar pathname convention.

2.1.1 Introduction to Using NSI

It is important to remember that names and objects are separate things in DCE. Consider, for example, these two DCE names:

/.../tinseltown.org/dce/printers/macmillan

/.../tinseltown.org/dce/employees/goethe

These strings are *not* filenames or file directory names; if you attempt to execute the **ls** command on them, you will only get an error message. They are pathnames that identify entries in the DCE Directory Service, which is DCE's database for storing distributed information. This database is often informally referred to as ''the namespace.''

The most important type of distributed information stored in the namespace is information that enables RPC clients to rendezvous with RPC servers; it is called ''binding information.'' The Directory Service can be used to hold other kinds of data too, but the main subject of the following discussions will be its use as a binding repository.

The set of binding name entries is like a huge data structure of pointers from object names to object locations, and the Directory Service is used mostly as a public DCE locational database, enabling servers to advertise themselves and the objects and resources that they manage, and clients in turn to find and access them. You should never confuse objects with their names; the two are separate things. In particular, the directory service data associated with a name is held in one place (namely, the directory server's database), while the data associated with the object named is held in other place (namely, the object server's database).

How then, you may ask, are filenames represented in DCE? Here are two examples of remote filenames:

/.../tinseltown.org/fs/doc/jones/app.gd/chap2.ps

/.../tinseltown.org/fs/doc/tolstoy/novels/war_and_peace/chap2.ps

As you may have guessed, these too are namespace entries, but the entries in this case refer to remote files, and the entry name as a whole is the remote filename.

What makes these names different from the other two names given earlier is their third element

fs/

which identifies a ''junction'' from the DCE Directory Service's namespace into the DCE Distributed File Service's own, separately maintained, namespace. How junctions work is explained in Section 2.1.3. However, the essence of the matter is that **/.../tinseltown.org/fs** is the DFS file server's DCE namespace entry, and any attempt by a file service client to access a file object whose name begins with **/.../tinseltown.org/fs** will implicitly bind to this server, which will then be responsible for finding, in its own namespace, the file object referred to by **doc/jones/app.gd/chap2.ps** or **doc/tolstoy/novels/war_and_peace/chap2.ps**, and performing the requested operations on it.

2.1.1.1 The UUID

Thus, it is a mistake to suppose that a name is identical to an object. The name merely points in the direction of the object it names. Objects do, however, have identifiers. These are the 128-bit Universal Unique Identifier (UUID) data structures, which are the identities that the DCE components recognize. They are not usually seen by users, although they play a part in the object-finding process.

UUIDs are used within DCE to identify all sorts of things. From the standpoint of the application programmer, they have two main uses: to identify objects and to identify interfaces.

2.1.1.2 Object UUIDs

Although ''object'' is necessarily a rather vague term, a reasonable definition would be the following: an object is any DCE entity that can be accessed by a client, and which can be represented by a namespace entry and identified therein by a UUID. This category can include servers, devices, and other resources. UUIDs that are used in this way are called ''object UUIDs'' in order to distinguish them from the other main use of

UUIDs, namely to identify interfaces (''interface UUIDs''). The difference between these two uses consists only in the way the UUIDs are interpreted by the name service and RPC runtime. Note that it follows from this discussion that an interface is usually *not* an object. Clients do not normally access an interface as such; the interface is rather a description of the rules of access.

As far as the DCE RPC and name service mechanisms are concerned, it is enough if a client is brought into contact with some server, as long as that server offers the service the client is looking for; in other words, as long as the server offers the interface the client wants to use. To accomplish this rendezvous, interface UUIDs are sufficient. They are also mandatory. There cannot be a client/server relationship without an interface, and the entire RPC runtime mechanism is dependent on the concept of interfaces.

Object UUIDs are different. The RPC runtime usually does not care if they are present or not. But if they are present, they activate various runtime mechanisms that allow clients and servers to be much more specific (always within the bounds of a given interface) about what servers are bound to, and/or what resources the servers will use to fulfill the clients' requests. How this works is explained later in this chapter.

2.1.1.3 Interface UUIDs

Every IDL-compiled interface specification has its own UUID associated with it, and the IDL-generated stub routines include this interface UUID with every operation request or return sent over the network by clients and servers. In this way receiving stubs ensure that they and the sending stubs are sharing exactly the same interface. If the interface UUIDs are different, or are not present, then the remote call will not be completed. But interface UUIDs, although they are required, play only a secondary role in a client's finding the interface (that is, finding a server that offers the interface); the main tool for this is NSI, which makes use of the DCE Directory Service, as explained later on in this part of the chapter.

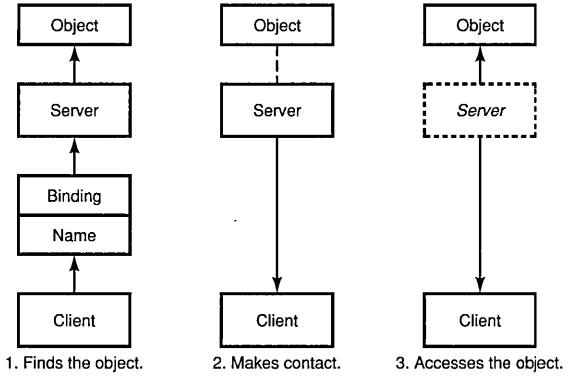
2.1.1.4 Summary: Names and UUIDs

Both names and UUIDs identify objects. But names are separable from the objects they identify, and are only as trustworthy as the binding information their entries contain. UUIDs, on the other hand, are inalienable identifiers. Once the desired binding information for an interface or an interface/object combination has been found and used, the name that was used to retrieve it can be forgotten; it is of no further use. This is not true of either interface or object UUIDs.

Note that names become completely unnecessary only if clients have some other means of obtaining valid binding information for the desired service, such as string bindings. (See Step B5, Section 1.5.10 in Chapter 1, for more information.)

Figure 2-1 illustrates how the information a client finds through a name is turned into network contact with the object named.

Figure 2–1. How a Name Turns into an Object



2.1.2 Binding to an Object

The difference between, for example, reading a local file on a single machine and performing the same read on a remote file in DCE is like the difference between reading information from a phone book yourself and dialing an operator for the same information. The remote operation requires the addition of another active entity that can be requested to perform it, since you cannot. Associated with every piece of remote data available on a network is a remote server to manage that data and make it available. The user may not see the server; even the client may be unaware of it, but it is there.

The DCE documentation often speaks of ''binding to an object.'' In reality, clients can bind only to servers, which then may be requested to perform operations on objects that are under their management. However, it is possible for a server to put bindings into namespace entries that are named for the objects that it manages. Furthermore, these exported bindings can be tagged with object UUIDs in such a way that incoming remote calls from clients can be applied by the server to the object whose name entry the binding was read from (the details of this technique are described later in this chapter). When an application uses this kind of binding model, it is reasonable to say that the client is logically bound to the object, although it is physically always bound to the server that manages the object.

2.1.3 Junctions

Namespace junctions are another example of the ''hidden server'' effect. The following remote filename was discussed earlier:

/.../tinseltown.org/fs/doc/jones/app.gd/chap2.ps

and there it was explained that

doc/jones/app.gd/chap2.ps

is an entry in DCE DFS's own namespace, while

/.../tinseltown.org/fs

is a DCE namespace entry. Suppose a user enters the following:

ls -l /.../tinseltown.org/fs/doc/jones/app.gd

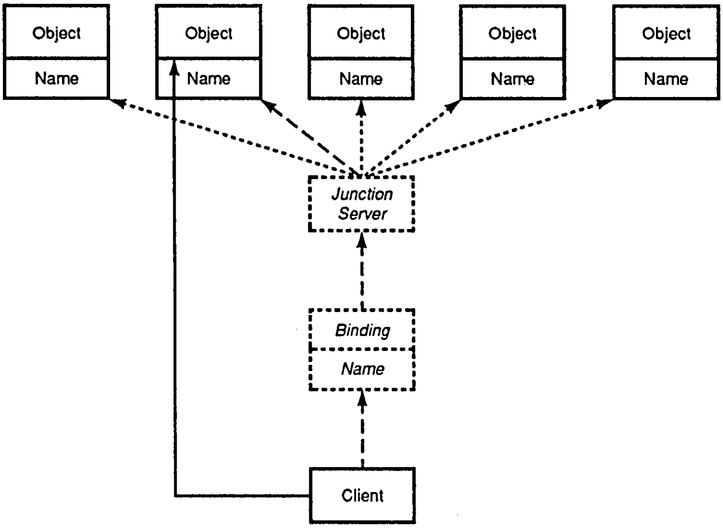
The clerk agent program (called as a result of the user's entering **ls**) will bind to the remote file server via its **/.../tinseltown.org/fs** DCE namespace entry, and pass to it the residual DFS entry name **doc/jones/app.gd**, along with other parameters. The **ls** command behaves this way because the underlying (VFS+ layer) system calls are coded that way. The DFS server then performs the request (note that the details of interaction within DFS are somewhat more complex than implied by this description). The user only types the command line; the rest is done by DCE, and a directory listing appears on the user's screen.

Because the VFS+ system routines, which are used by all possible clients of DFS services (for example, commands like **ls** and **rm**, library routines like **fopen( )** and **fclose( )**), know about the remote file server at **/.../tinseltown.org/fs** and bind to it correctly, the transition from the DCE to the DFS namespace is completely transparent to users. And this is how junctions work. As long as all possible clients behave correctly with a name that includes a junction, the junction will not be perceptible to the clients' users.

2.1.3.1 A Junction Example

Figure 2-2 illustrates the principle of junctions. A junction server, which is reached normally through binding information in the DCE namespace, maintains its own namespace of named objects. The junction server's clients allow users to refer to these objects by actually concatenating the server's entry name and an object's ''internal'' name. The client then in effect breaks this string apart by contacting the server named in the first part of the string, and passing to it the second part, which is a valid name within the server's namespace. The client's user seems to access the object directly.

Figure 2–2. A Namespace Junction



The dashed lines in Figure 2-2 show the progress of the Client's efforts to get access to the desired Object, which involves acquiring a binding to the Junction Server, making contact with it, and passing to it the Object's Name. The solid line shows the apparent direct access to the Object that the Client's user seems to enjoy. The dotted lines show other possible paths of access to the other Objects that the Server manages.

Junction protocol is generally a private matter between an application's clients and servers. However, the **acl_edit** command uses a generalized protocol.

2.1.3.2 Junctions and the ACL Editor

The binding routines that **acl_edit** uses are discriminating enough to detect a junction anywhere in an entry name that is passed to it. This allows a distributed application to have its own namespace for objects with ACLs on them, rather than burdening the DCE namespace by separately exporting binding information for every one of these objects. The separate objects have to be made publicly accessible somehow because entities should be able to access ACLs directly, regardless of whether they happen to already be in contact with the server that manages the ACL'ed object, and indeed regardless of whether or not they happen to be a client of the particular server to which the objects belong.

Suppose, for example, a user enters

acl_edit /.../tinseltown.org/dce/dce_print/cotta

in order to interactively edit the ACL for the printer object **cotta**, where **/.../tinseltown.org/dce/dce_print** is the namespace entry for a print server, and there is no **/.../tinseltown.org/dce/dce_print/cotta** entry in the DCE namespace. The binding routine, **sec_acl_bind**( ), which is called internally by **acl_edit**, receives an error when it tries to bind to the object **cotta**. However, the DCE Directory Service also tells it how much of the name it passed is valid. The **sec_acl_bind**( ) routine then retries the binding operation, this time through the valid entry name (**/.../tinseltown.org/dce/dce_print**), and passes the residual part of the name (**cotta**) as a parameter. Now it is up to the application ACL manager to interpret the residual name correctly and find the requested ACL.

2.1.4 Name Service Terminology

As was mentioned at the beginning of Chapter 1, DCE RPC NSI is an RPC-based interface that uses the DCE Cell Directory Service (CDS) as its database. The NSI routines do not constitute a general interface into CDS as such; they are a set of specialized routines whose purpose is simply to provide ways for RPC servers to advertise themselves to RPC clients, and for clients to find and bind to them.

In fact there is no public general API (Application Programming Interface) to CDS. There is a general CDS interface that is used internally by the DCE components, but applications normally access CDS through NSI. Applications can get full access to CDS, if necessary, by using the XDS interface. For further information on this possibility, see Section 2.1.4.2 later in this chapter, and Parts 4A and 4B of this guide. See also the discussion of Directory Service interfaces in Chapter 3 of the *Introduction to OSF DCE.*

2.1.4.1 CDS Entries

NSI uses a subset of the many possible kinds of CDS entry in order to accomplish its tasks. CDS entries are characterized by the CDS attributes they have; each entry can have one or more such attributes. Each separate attribute defines that entry's ability to contain one or more items of a particular kind of simple or complex information. Section 2.1.4.2 discusses CDS attributes in more detail.

The name service creates and uses CDS entries that use only the following four attributes:

- **binding** The entry has a field that can contain one or more sets of binding information. When the field is read, a binding handle that contains the necessary information from one of these sets is returned, in no particular order.

- **object** The entry has a field that can contain one or more object UUIDs. When the field is read, one of the UUIDs is returned, in no particular order.

- **group** The entry has a field that can contain a pool of one or more references to other (independently existing) NSI entries; each time the field is read, one of these entries is returned. Different entries are returned on successive reads, but the order of return is undefined. Note that the ''other NSI entries'' referred to in the group can themselves be server or group entries. As a result, the act of reading from a group attribute can, depending on the actual API routine called, lead to a series of nested operations. Any nesting is transparent to the client application, however, which seems to perform a simple read and to receive the contents of a single entry in return.

- **profile** The entry has a field that can contain one or more prioritized elements, each of which consists of a reference to another (independently existing) NSI entry. When the field is read, the elements are read in a specified order. The entry referred to in the element may itself be a server or a group or a profile. As a result, any element may in fact, depending on the actual API routine called, resolve on access to a nested path of referred-to entries. As with group entries, this is transparent to the client application.

Although a single entry could contain both group and profile attributes (and for that matter, binding and object attributes as well), it is not a good idea to mix attributes in this way because the results of importing (reading) from such an entry are too indeterminate.

The typical name service entries are as follows:

- **server entry** Contains a binding and an object attribute, making it suitable for containing the necessary binding information for a single server.
- **group entry** Contains a group attribute.
- **profile entry** Contains a profile attribute.

There are no official names for hybrid entries that contain other combinations of attributes, which is perhaps another reason for not creating such entries.

The general name for entries that contain any of these attributes is ''NSI entries,'' since they are a by-product and tool of the NSI DCE RPC library routines.

2.1.4.2 CDS Entry Attributes

Within the DCE Directory Service, entry attributes such as the four previously described attributes are identified by Object Identifiers (OIDs). This is an exception to the general rule that things in DCE are identified by UUID.

OIDs are not seen by applications that restrict themselves to using only the name service routines (**rpc_ns_...( )**), but these identifiers are important for applications that use the X/Open Directory Services (XDS) interface to create new attributes for use with namespace entries.

As was seen in the immediately preceding sections, the name service makes use of only four different entry attributes in various application-specified or administrator-specified combinations. CDS, however, contains definitions for many more than these, and attributes from this supply of already existing ones can be added by applications to NSI entries through the XDS interface. Attributes that already exist are already properly identified, so applications that use these attributes do not have to concern themselves with the OIDs, except to the extent of making sure that they handle them properly.

A further possibility is that an application requires new attributes for use with namespace entries. Such attributes can be created using the XDS interface. When it creates new attributes, the application is responsible for tagging them with new, properly allocated OIDs.

Unlike UUIDs, OIDs are not generated by command or function call. They originate from the International Standards Organization (ISO), which allocates them in hierarchically organized blocks to recipients. Each recipient (typically an organization of some kind) is then responsible for ensuring that the OIDs it received are used uniquely.

For example, the OID

```
1.3.22.1.1.4
```

identifies the NSI profile entry attribute. This number was assigned by the Open Software Foundation out of a block of numbers, beginning with the digits `1.3.22`, which was allocated to it by ISO, and OSF is responsible for making sure that `1.3.22.1.1.4` is not used to identify any other attribute.

When applications have occasion to handle OIDs, they do so directly, since the numbers do not change and should not be reused. However, for users' convenience, CDS also maintains a file (called **/opt/dcelocal/etc/cds_attributes**) that lists string equivalents for all the OIDs in use in a cell, in entries like the following one:

```
1.3.22.1.1.4    RPC_Profile    byte
```

This allows users to see `RPC_Profile` in output, rather than the mysterious `1.3.22.1.1.4`. Further details about the **cds_attributes** file and OIDs can be found in the *OSF DCE Administration Guide*.

Broadly speaking, the procedure you should follow to create new attributes on CDS entries consists therefore of three steps:

1. Request and receive, from your locally designated authority, OIDs for the attributes you intend to create.

2. Update the **cds_attributes** file with the new attributes' OIDs and labels; that is, if you want your application to be able to use string name representations for OIDs in output.

3. Using XDS, write the routines to create, add, and access the attributes.

Non-NSI attributes on NSI entries can be very useful, even though you cannot access the extra attributes through the name service routines but must use XDS instead.

2.1.5 Binding

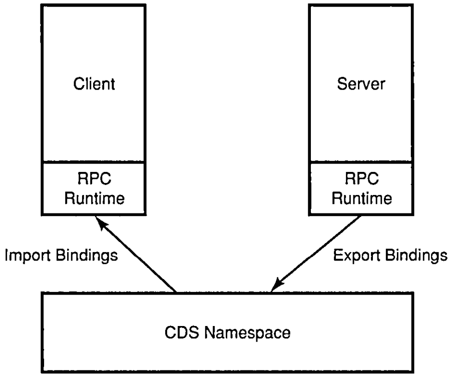
In order to highlight the essentials of name lookup and storage and the management of binding information, many details of DCE RPC operation are either greatly simplified in the following descriptions or omitted altogether. Refer to Part 3 of this guide for the definitive explanations of the mechanics of binding.

A binding is a package of information that describes how a client can contact and communicate with a particular server. Although the underlying protocol that implements the communication can be connectionless or connection-oriented, the relationship itself is still expressed as a binding.

2.1.5.1 Importing and Exporting Bindings

The name service exists to store server binding information into the cell namespace, and to retrieve that information for clients. Using NSI, servers export their binding information to be stored under meaningful names, and clients import these bindings by looking up those names. Thus, the locations of the servers can change, but clients can continue to use the same names to get bindings to the servers. Figure 2-3 shows how client and server use the name service.

Figure 2–3. Client and Server Use of the Name Service



When a prospective client attempts to import binding information from a namespace entry that it looks up by name, the binding is checked by NSI for compatibility with the client. This is done by comparing interface UUIDs. The client presents an interface UUID when it begins the binding import operation; the UUID of the interface being offered is exported to the name entry, but not in the binding handle itself, by the server. If these interface UUIDs match, then the binding handle contained in the entry is considered compatible by the RPC runtime and is returned to the client. If more than one handle is contained in the entry (this is often the case), they are returned one by one on successive imports. NSI also checks for protocol compatibility.

The import routines will return only client-compatible bindings, but a client can sift through the returned bindings and make its own choice as to which ones to use, based on its own criteria. The technique by which this is done consists of converting the bindings into string bindings, and then inspecting (or comparing) the strings. (See Step C1, Section 1.5.18 in this guide.)

Note that binding handles do *not* include an interface UUID. Binding handles do contain a host address, an endpoint, and an optional object UUID, among other things. The interface UUID is associated with the interface's stub code, which inserts it into outgoing RPCs and checks it in incoming ones, thus guaranteeing client/server operational compatibility. This allows binding handles to be used very flexibly: once a client has successfully bound to a server, it can utilize any of the interfaces that server offers, simply by making the desired remote call.

2.1.5.2 Summary

The mapping from name to server that occurs when bindings are imported from the namespace is indirect because binding is a two-step process: first the binding handle is obtained by lookup from a named entry, and then the handle is used to reach a server. The crucial point is that the imported handle will not usually contain a complete binding to a specific server (namely, the one that happened to export it). Completion of the partial binding occurs later, when the client makes its first remote procedure call; the RPC runtime uses UUIDs, not names, to determine how it should complete a binding.

2.1.6 Partial Binding and the Endpoint Mapper

Binding handles imported by clients from the namespace normally contain only partial binding information. The exported binding information is sufficient to locate the RPC daemon on the server's host (the machine the server resides on), but it does not yet include a specific endpoint (UDP or TCP port number) for the desired service on that host.

The reason for omitting dynamic endpoint information in exported binding handles is to avoid unnecessary multiplication of accesses to the namespace. Since dynamically generated endpoints are necessarily reassigned every time a server starts up, entering them into the namespace (and thus forcing CDS to propagate the new information throughout the various directory replicas) would greatly increase namespace housekeeping chores.

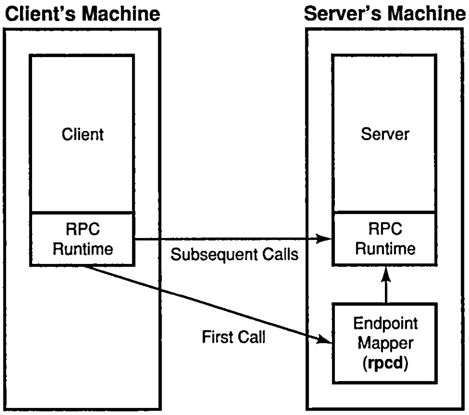
Thus, the last step in the binding process is obtaining an endpoint. The step is performed transparently as far as the client is concerned. It is accomplished by the DCE RPC endpoint mapper daemon, **rpcd**, when the client makes its first call to the partially bound-to server. The **rpcd** daemon manages its own private database of server endpoints for the host on which it is located. The endpoints are registered by the servers as part of their startup routine.

The binding information that accompanies a prospective client's first remote procedure call takes that call to the well-known endpoint of **rpcd** on the exporting server's host machine. The endpoint mapper now takes over. It looks up a valid endpoint for the requested service, copies it into the binding handle, and transfers the call to that endpoint. Subsequent calls from the client, which now has a binding with one of the server's endpoints, will bypass the endpoint mapper.

The endpoint mapper picks an appropriate endpoint for an incoming partial binding by matching interface UUIDs by default. Any endpoint that has been registered under an interface UUID that matches the incoming interface UUID, which identifies the interface requested by the prospective client, is eligible for selection. This mapping process is called ''forwarding'' when it occurs with connectionless protocols, and ''mapping'' when it occurs with connection-oriented protocols.

Figure 2-4 shows the endpoint mapper completing a binding.

Figure 2–4. The Endpoint Mapper Completes a Binding



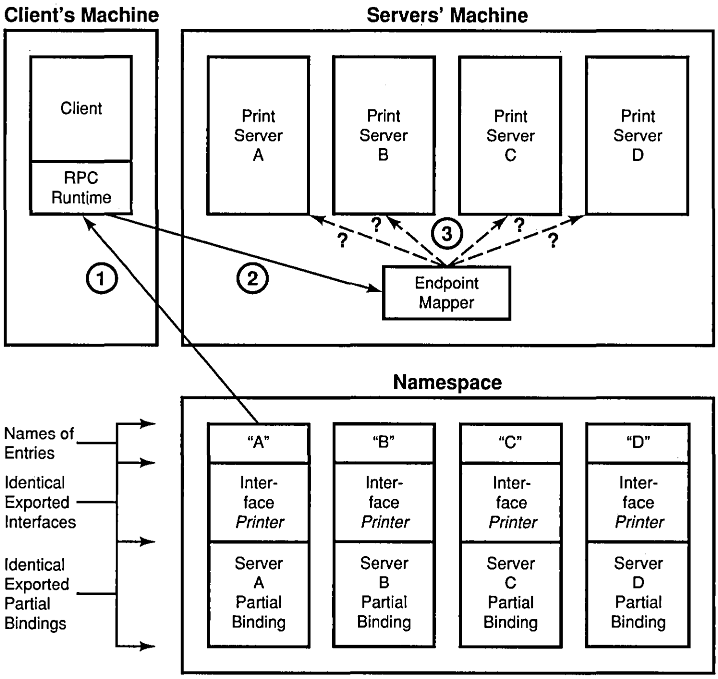
There is an exception to this scheme. Some servers are designed to occupy well-known addresses. The endpoint mapper itself, **rpcd**, is reached in this way, making its accessibility independent of whether or not the namespace is accessible. The endpoint(s) of a well-known address do not change; they are usually specified in the application's interface specification (contained in its **.idl** file; see Step A3 in Chapter 1, Section 1.5.3, of this guide). Bindings to servers that use well-known endpoints are already complete at the time of import; the endpoint mapper never sees these bindings.

2.1.7 Interface Ambiguity and Partial Bindings

The interface UUID, which was generated by the IDL compiler, uniquely identifies the set of operations that the client will access through that interface. In short, it identifies the interface. An interface UUID may also happen to identify *a server* which offers that interface. But if more than one server on the same host offers the same interface (which could easily be the case), the interface UUID alone will not be sufficient to identify a *specific* server. The result is that if a remote call comes in with such an ambiguous interface and a partial binding, the endpoint mapper will have to randomly choose any one of its eligible registered endpoints, complete the binding with it, and send the call on to that server.

Imagine several print servers residing on the same machine (see Figure 2-5). Each server manages a group of printers that share a common physical location. All the printers in room ''A'' are managed by the ''A'' print server, all the printers in room ''B'' by the ''B'' print server, and so on. Now suppose each of these servers has a separate entry in the namespace. (See Figure 2-5 for the sequence of events that occurs.)

Figure 2–5. Print Server Entries in Namespace



The following steps describe the sequence of events shown in Figure 2-5:

1. The Client imports a partial binding to the *Printer* interface from the entry ''A'' in the Namespace.
2. The Client makes its first call with the binding it imported from ''A.''
3. The Endpoint Mapper at Print Server A's host, when it receives the call from the Client, has no way of knowing which of the four Print Servers it should map the call to, since all four servers have registered their endpoints under the same interface. It therefore picks one at random to complete the binding.

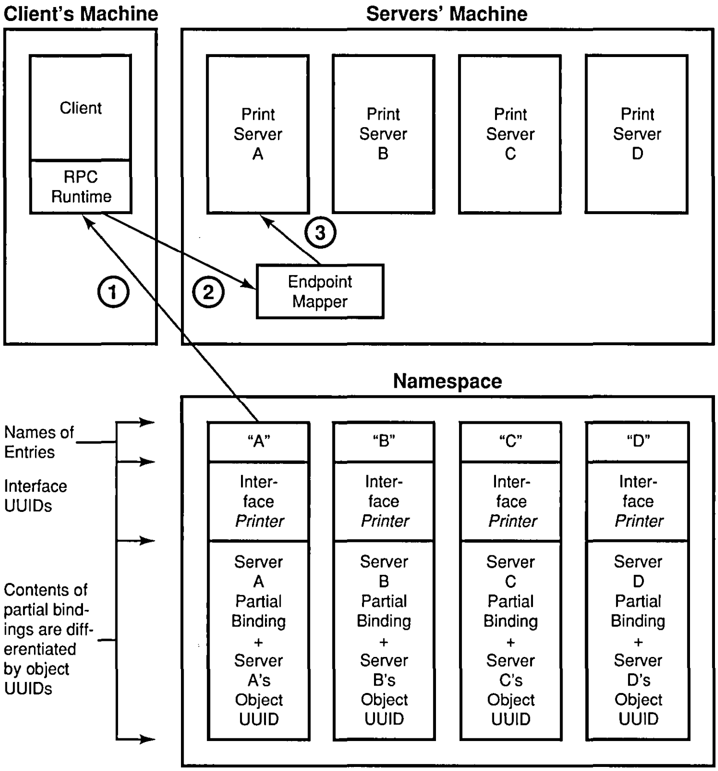
The entry names are different, but the partial binding information contained in the entries is identical, since the servers' host machine is the same. The interface UUID included in the call is no help, since that same interface is offered by all the servers. A client seeking a print server may not care to which server (and thus to which printer) its request goes, but then again, it may care. If it does, there is a way it can specify a server so that the endpoint mapper can select an appropriate endpoint to complete the partial binding.

2.1.8 Using Object UUIDs to Avoid Binding Ambiguity

Binding handles can contain, besides host address and endpoint information, an object UUID as well. The endpoint mapper will try to match an object UUID contained in a binding handle with one of the object UUIDs associated with its map of registered endpoints. This allows even a partial binding to specify a target more precisely than just by host machine. Since object UUIDs are generated by the **uuid_create( )** function call (see the *OSF DCE Application Development Reference*), servers can create as many of them as they need. Steps B2 (Section 1.5.7) and B3 (Section 1.5.8) in Chapter 1 of this guide show how the server sets up the object UUID-mapping mechanism.

For the print server example discussed in the previous section, the namespace entries for the servers could be set up as shown in Figure 2-6.

Figure 2–6. Print Server Name Entries with Object UUIDs


Client's Machine
Servers' Machine
Client
RPC Runtime
Print Server A
Print Server B
Print Server C
Print Server D
Endpoint Mapper
1
2
3
Namespace
Names of Entries
Interface UUIDs
Contents of partial bindings are differentiated by object UUIDs
"A"
"B"
"C"
"D"
Interface *Printer*
Server A Partial Binding + Server A's Object UUID
Server B Partial Binding + Server B's Object UUID
Server C Partial Binding + Server C's Object UUID
Server D Partial Binding + Server D's Object UUID

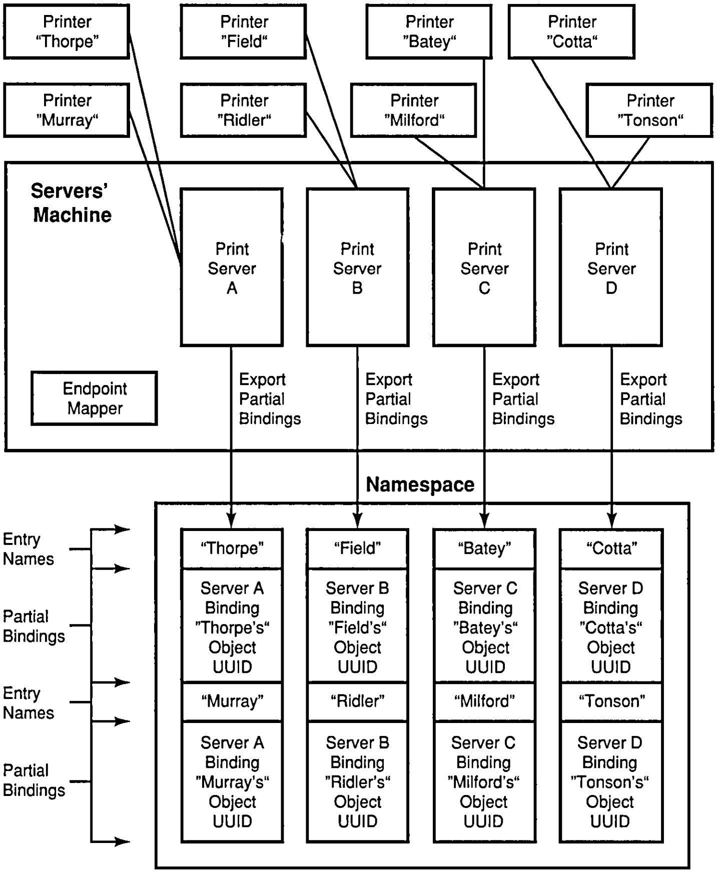
The following steps describe the sequence of events shown in the preceding figure:

1. The Client imports a partial binding to the *Printer* interface from the entry ''A'' in the Namespace.
2. The Client makes its first call with the binding it imported from ''A.''
3. This time the Endpoint Mapper at Print Server A's host is able to match the call with A's registered endpoints, because the endpoints have been registered with both the *Printer* interface and Print Server A's Object UUID, and the incoming call's partial binding also contains Print Server A's Object UUID.

Each server has exported a set of partial bindings that differs from all other servers' by its object UUID (which thus becomes, in effect, a server ID). If, for example, Server A has properly registered its endpoints with the same object UUID as the one it exported its bindings with, the Endpoint Mapper will make sure that a partial binding exported from Server A's name entry will result in a full binding to Server A.

Now suppose that each print server sets up a separate namespace entry for each printer it manages. The printers themselves would, in effect, be identified by their own object UUIDs (see Figure 2-7).

Figure 2–7. Separate Printer Name Entries



Now a client will be able to access a specific printer by importing a binding handle from that printer's name entry. The endpoint mapper at the target host would compare the object UUID in the partial binding with the object UUIDs registered by the print servers, and select an appropriate server. The server in turn would also use the object UUID to select the correct printer for the request, if it managed more than one printer. A namespace set up in this way with a separate entry that contains a unique object UUID for each accessible service resource is called an ''object-oriented'' namespace.

2.1.9 An Object-Oriented Namespace

''Object-specific entries'' are namespace entries that each contain binding information only for one specific object or resource, as demonstrated in the last printer service shown in Figure 2-7. ''Object'' can mean any of several things, depending on what kind of service the application's servers are offering. Table 2-1 shows some examples of objects.

Table 2–1. Some Examples of Objects

Service	Object(s)
Printing	A specific printer
Process Server	A specific server
Queue Service	The print queue,the kill queue, the backup queue

Thus, for a client that wants to have a file printed, it is natural to allow it to specify a printer as a destination. Therefore, the client would bind to the print server through a name entry that specifies a printer. To send something to a different printer, the client would import a binding from the name entry for that other printer. The server may (or may not) be identical, but the object UUID in the binding handle returned would uniquely specify the one printer represented by that entry.

On the other hand, consider an application that returns statistics about the processes currently active on a group of machines. In this case it would be reasonable to regard the server as the object. In the namespace entries for such an application, each entry would uniquely represent one server. A client would import a binding from the name entry for the server it wanted to work with.

In other words, ''object'' is a handy way of saying ''the thing that clients will want to access'' in order to accomplish the task set for the application. If the namespace is organized correctly, clients will be able to import bindings from these objects' entries.

2.1.10 Setting Up an Object-Oriented Namespace

Once you have distinguished the objects your application uses, you must decide on an appropriate set of names for the entries themselves. The entries can be created either by the application (server), if it has the necessary privileges, or by a system administrator using the **rpccp** command interface. (See Section 2.5.1.3 for further information on this step.)

After the entries have been created, each server must do the following:

1. Create an object UUID for each object managed by the server under an interface, insert it into the binding handle(s) for that object, and export the handle(s) for each object to a separate entry in the namespace.

   Note that the object UUID should be generated and exported in general *only once* per created namespace entry, and not each time the server starts up (see the example that follows of how to do this). When a newly restarted server exports its partial bindings, nothing actually happens in the namespace because the partial binding information remains the same (unless the server has moved to a different machine). However, if the object UUIDs are regenerated, then the change in exported information will force needless update activity in CDS, which is where the entries exist.

2. Register with the endpoint mapper the full bindings (including endpoints) obtained for the interface; **rpc_ep_register**( ) performs this operation.

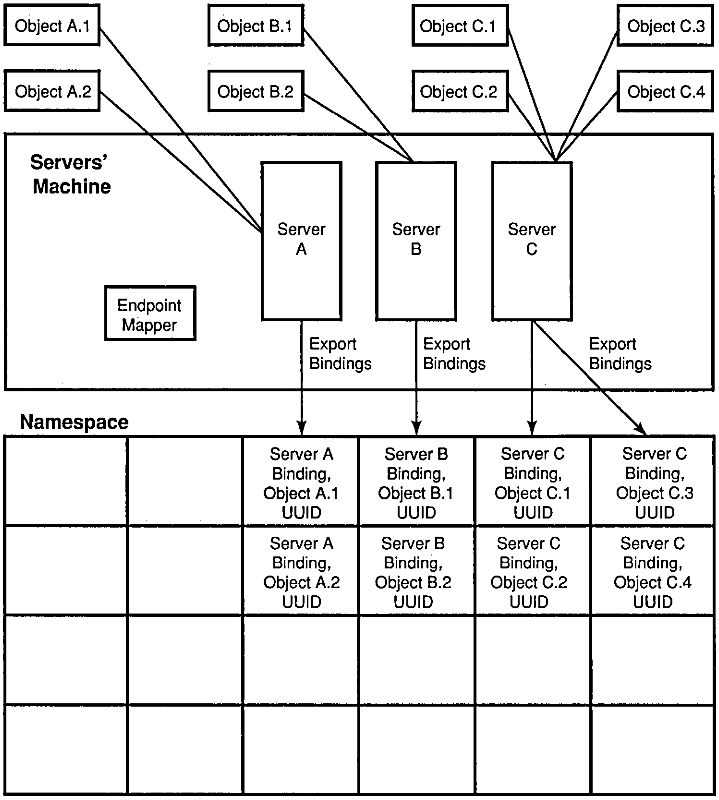
One way of avoiding unnecessary regeneration of object UUIDs would be to have a restarted server check the namespace for the presence of its previously exported object UUIDs, as demonstrated in the following code fragment. Refer to the *OSF DCE Application Development Reference* for further information on the function calls.

```
have_object = false;

/* Create an inquiry context for inspecting the object  */
/*      UUIDs exported to "my_entry_name"...            */
rpc_ns_entry_object_inq_begin(my_entry_name_syntax, my_entry_name,
                              &context, &st);

/* If we successfully created context, look at          */
/*      object UUIDs...                                      */
if (st == rpc_s_ok)
{
    /* Try to get one object UUID from the entry...     */
    rpc_ns_entry_object_inq_next(context, &obj, &st);

    /* If an object UUID is there already, we don't     */
    /*  need to generate another one...                 */
    have_object = (st == rpc_s_ok)

    /* Delete the inquiry context...                    */
    rpc_ns_entry_object_inq_done(&context, &st);
}

/* If there were no object UUIDs in the entry,          */
/*      generate one now...                             */
if (! have_object)
{
    uuid_create(&obj, &st);

    /* Put it in an object UUID vector...               */
    objvec.count = 1;
    objvec.id[0] = &uuid;
}

/* Export bindings. If an object UUID was generated,    */
/*      export it too...                                */
rpc_ns_binding_export( my_entry_name_syntax, my_entry_name,
                       my_interface_spec, my_bindings,
                       have_object ? NULL : &objvec, &st);
```

Whenever you want to offer more than one instance of the same interface on the same host, you *must* distinguish by object UUID the binding information in the name entries exported by the servers, if it is important to distinguish among the servers when binding to them. Otherwise, the endpoint mapper's selection of an endpoint with which to complete the binding from among all the servers on that host that offer the appropriate interface will be random.

Figure 2-8 illustrates what such an object-oriented namespace should look like.

Figure 2–8. Object-Oriented Namespace Organization


Object A.1
Object B.1
Object C.1
Object C.3
Object A.2
Object B.2
Object C.2
Object C.4
Servers' Machine
Server A
Server B
Server C
Endpoint Mapper
Export Bindings
Export Bindings
Export Bindings
Namespace
Server A Binding, Object A.1 UUID
Server B Binding, Object B.1 UUID
Server C Binding, Object C.1 UUID
Server C Binding, Object C.3 UUID
Server A Binding, Object A.2 UUID
Server B Binding, Object B.2 UUID
Server C Binding, Object C.2 UUID
Server C Binding, Object C.4 UUID

Each entry has a name denoting the object represented, although the names are not shown in this figure. (See Section 2.5.1.3 for further discussion on this topic.)

Under this model, clients bind to servers via named objects in the namespace, each of which contains enough specific information in its partial binding to allow the endpoint mapper at the destination host to choose an appropriate endpoint for the incoming RPC.

By setting a namespace up this way, however, you do not necessarily restrict yourself to this one model for accessing binding information. Through the use of two other types of entry, groups and profiles, which can be superimposed on the simple object model, you can set up models where clients bind to abstractions such as services, or directly to the servers themselves. These techniques are described in the next section.

Nevertheless, at this point you have enough information to set up a namespace that consists of an entirely ''flat'' expanse of separate resource entries. Bindings can be imported by clients by looking up specific names. If the client has no specific name to look up, or if the lookup on the name(s) it has fails, it has no alternative way of binding to a server.

2.1.11 Groups and Profiles

Name lookups can be made more flexible with two other types of entry; namely, groups and profiles.

2.1.11.1 Group Entries

A group entry consists essentially of multiple independent other entries whose names are also associated under the group name. These ''other'' entries can be simple (single-name) entries, or they may themselves be group entries. Doing an import from the group entry will return the contents (the binding handles) of its included entries (which are called ''members''), but the selection is made by the DCE RPC runtime, and from the client's point of view is undefined and implementation dependent.

In practice, the way this works with the usual binding import operations is as follows. Clients normally import bindings by first calling **rpc_ns_binding_import_begin**( ) to set up an import context. Once this is done, successive calls to **rpc_ns_binding_import_next**( ) will return binding handles from namespace entries until the handles have all been returned or the client decides to stop; the client decides which handle(s) to use based on its own criteria. When it is finished importing, it calls **rpc_ns_binding_import_done**( ) to free the context. (Several examples of this technique are illustrated later on in this chapter; the client code for **timop**, the DCE sample application described in Chapter 3 of this guide, also contains an example.)

The kind of entry the information is returned from is usually unknown to the client, which needs to know only a name to look up and the interface UUID by which it wants to bind. If the name is that of a simple server entry, then the bindings contained in that entry only will be returned. If the name is of a group entry, then bindings will be returned from members (single entries) of the group, selected (by the RPC runtime) in an undefined order. If one or more members of the group are themselves groups, then the same thing happens recursively whenever these lower-level groups are accessed.

Note that the group entry and its members are separate things. The group entry can be deleted, but its former members will continue to exist as independent entries, unless they too are explicitly deleted. Thus, you can implement a namespace organization where the same bindings can be imported through individual simple entries or through group entries, depending on how the client is coded. (See Chapter 15 for more details on group entries.)

2.1.11.2 Profiles

A profile entry specifies a search path or hierarchy of search paths to be followed through the namespace in order to obtain a binding to a server that offers a specified interface.

When a client imports from an entry that happens to be a profile, successive imports (accomplished by calling **rpc_ns_binding_import_next(**)) return the contents of entries that are read as a result of following the specified path through the namespace. All this is transparent to the client, which sees only the bindings returned. Profiles can be used to set up default paths and groups of paths for users. The **RPC_DEFAULT_ENTRY_NAME** environment variable, which is the default entry name used by the name service in import operations, usually contains the name of a profile.

As with groups, the entries contained in profiles, which are called ''elements,'' exist independently of the profile entry itself.

A very important property of profiles is that they allow clients to know little or nothing about the organization of the namespace itself. Using the default case as an example, consider the following: if the profile at **RPC_DEFAULT_ENTRY_NAME** has been set up with elements containing entries for all possible active servers for a particular application, then clients can simply import from this name and trust the profile mechanism to walk through the various compatible possibilities and return binding handles via successive calls to **rpc_ns_binding_import_next( )**. (Note that a profile entry is not limited to containing entries for just one interface; thus, **RPC_DEFAULT_ENTRY_NAME** could be set up to contain *all* the defaults for a cell.) (See Chapter 15 for a detailed discussion of profiles.)

2.1.11.3 Summary of Namespace Entry Types

Clients access binding information in the namespace by looking up (by name) one of three different kinds of entry:

- A server entry
- A group entry, which contains other entries whose contents are returned to the caller when it reads the group entry
- A profile entry, which specifies a path of entries to be searched whose contents are returned to the caller when it reads the profile entry

Lookups behave differently depending on the kind of entry read. If an entry is a simple server entry, then the search begins and ends right there, whether successful or not. If the entry is a group, then the lookup is more complicated. A binding will be returned from among those that are found to be compatible by the name service, but within that category the selection is undefined. If the entry is a profile, then a specified path of entries is searched. The entries in this path may themselves be other profiles, or groups, or simple entries. The search continues until either a compatible binding is found, or the entire path has been unsuccessfully traversed.

2.1.12 Three Models for Accessing Binding Information

By adding groups and profiles to the object-specific namespace organization originally described, you can implement any or all of the following three basic models for accessing binding information:

- Clients bind to services
- Clients bind to servers

- Clients bind to resources or objects

Each of the three models is described in the following sections.

2.1.12.1 Access By Services

Servers have separate namespace entries; each server distinguishes the bindings it exports with its own identifier; that is, an object UUID that it generates for itself *the first time it starts up*. These separate server entries are also members of group namespace entries, which represent services. The criteria for membership in a service group is that all the servers in it export the interface that identifies that service. (They may happen to export other interfaces as well.)

Clients, in effect, bind to services by importing their binding handles from the group entries. Note, however, that the server-specific entries still exist independently and are accessible to lookup.

This model is appropriate for applications where clients do not care which server they happen to bind to or where that server is located as long as it offers the desired service. The eligible servers are pooled into a group entry from which bindings to one of them are selected in an undefined order and returned whenever a client performs an import operation from the group entry.

2.1.12.2 Access By Servers

In this model, distinct servers have separate and distinct name entries, and clients import bindings directly from the server entries. Hence, an application using this kind of binding model will ''own'' just as many simple entries in the namespace as there are active servers.

Since the client in this model is looking for a specific server, imports will be done directly from the server entries. The only exception to this rule would be where two or more instances of a server were active on the same host, and it was indifferent to the client as to which one it is bound to. The entries for the multiple same-host servers then could be put into a group entry, and binding imports done from the group.

2.1.12.3 Access By Objects

Servers operate on or manage multiple objects. Clients use these objects (via the servers) as resources. For each such resource, the server creates a separate namespace entry and exports its binding information there, distinguishing each object entry with its (the object's) own object UUID.

An example of this model is the printer service that was previously described. Clients will import directly from the name entry of the resource they want to use. For this kind of application, there will generally be more namespace entries than active servers, since each server presumably manages more than one object. If the name entries have been set up correctly and the servers have properly registered the object UUIDs they created, there will be no difficulty in routing any partial binding to the correct server (namely, the server that manages the object or resource specified).

2.1.12.4 Summary of Binding Models

Although the name service allows other approaches, we recommend that whenever possible you use the object-oriented scheme to organize your namespace entries. There are at least two good reasons for doing so. First, it is easy to administer; at the simple entry level, things really are simple. Second, this is the most flexible foundation for building other more complicated access models using group entries and profiles.

The separate name entries in your namespace should contain bindings that will unambiguously resolve to specific server instances. Since interface UUIDs are often offered by more than one server, more information than just an interface UUID is needed in order to give an RPC with a partial binding the required specificity. Object UUIDs provide this extra information. When using object UUIDs to distinguish bindings in this way, servers must take care to preserve their uniqueness across name entries.

Finally, profile entries allow clients to walk through a specified search path of namespace entries and yet be completely ignorant of the actual names themselves. While name independence may not be desirable for an object-based or resource-based distributed application, it can be a powerful mechanism when used with other models.

As you are setting up the namespace organization for your application, remember that there is not a direct exact mapping from names to bound servers. Different names, once imported from, may resolve to identical bindings if the partial bindings were exported on the same interface, from the same host, and not otherwise distinguished from each other by object UUIDs. It is the application developer's responsibility to tailor an application's export and import procedures so that this mapping behaves as intended.

2.1.13 Models Based on Non-CDS Databases

The three models previously described are not mutually exclusive; if the namespace is set up correctly, all three can coexist at the same time. All three of the models are implemented through the functionality of the DCE RPC name service.

Although the emphasis in this discussion has been placed on the storage and retrieval of binding information, the namespace entries can be used to store additional states for objects. In order to do this, an application would have to create additional attributes on the CDS entries it intended to use because the name service recognizes only the four NSI attributes: binding, object, group, and profile.

Such additional entry attributes would be created and accessed through XDS. However, whenever you find yourself contemplating extending the name service in this manner, you should carefully consider whether the name service (and, consequently, CDS) is the best mechanism for doing what you want to do. For some further discussion of what is involved in adding attributes to CDS entries, see Section 2.1.4.2, earlier in this chapter.

In the preceding example, where an object-oriented namespace containing separate entries for individual printers was described, only the identifier for the printer (the object UUID) and the binding for the server that managed it were stored in the CDS entry. Other information, such as what jobs are currently queued for the printer, who owns the jobs, and so on, was maintained by the server. This data could be stored in CDS only by creating new attributes to put it in, but it would be changing too quickly for CDS to efficiently keep up with it anyway. The performance of both the application and CDS would suffer from such an arrangement.

It is possible to imagine distributed applications whose resources (the objects they are managing) are of such a nature that they could be more efficiently managed through a private application-implemented database. Suppose the number of managed objects is very large, or that the state of the objects is volatile. It would certainly be a bad idea to try to use CDS to store this kind of information, which would be changing much more rapidly than CDS's ability to propagate the updates.

2.1.13.1 Example of a Privately Managed Database

As an example of such a privately managed database, consider a print service where jobs are submitted not to individual printers, but rather to a generic printer service. The client, **lpr**, binds (probably through a group entry) to some certain print server, and sends the job to be printed to that server, which then, after some thought, sends the job to one of the printers that it manages.

Consider, for example, what happens if a user invokes the client **cancel** sometime later to stop a job. If, for example, the original command was

lpr War_and_Peace.ps

and the subsequent request to cancel is

cancel War_and_Peace.ps

then how does the server that **cancel** binds to find the right job to delete? There is no guarantee that **cancel** will bind to the same server that happened to receive the original print request, so having each print server keep track of its own jobs would not be the answer.

One way to keep track of jobs queued would be to have a dedicated ''job location server'' as part of the application. Each time a print server queued a job to a printer it would record the fact (with all the pertinent details) with the location server. Whenever a job completed, the server would again notify the location server to remove its record of that job from its database. A client **cancel** then binds first to the location service, where it receives the name of the print server associated with the job it wants to cancel. It then looks up that name, binds to the right print server, and sends the cancel request. In effect, the location server has become a name service for **cancel**.

This method of organizing activity results in a split-model database. The print servers' binding information is managed through CDS, as usual, and the location server manages other more volatile information associated with those same servers.

Another way a server could maintain its own database of named objects would be by implementing a junction. (See Section 2.1.3 earlier in this chapter.)

2.1.13.2 Combining Models

In designing a binding access model for an application, consider also whether it may be appropriate to combine some of the models previously discussed. In the print service application, it may be desirable for servers to also offer a management interface to specific servers rather than to specific objects; for example, **lpr**, **lpq**, and **lprm** are generic application clients, so it is appropriate for them to bind to printer objects, but if **lpr_mgmt** is supposed to manage characteristics of a whole service, then it should bind to servers.

2.1.14 An Object-Oriented Model with Grouped Binding Information

The following variation on the object-oriented binding model shows how the group attribute can be used in object entries. In this model, each of the object entries contains, as before, an object UUID that will uniquely identify (either to the endpoint mapper on the exporting server's machine, and/or to the server itself) the object referred to by that entry. However, the object entries do not contain any binding information. Instead, a group attribute in each object entry refers clients' import operations back to the server's own separate entry, which contains the binding information for that server.

The namespace ingredients of this model are the following:

- A single namespace entry for the server, which contains a binding attribute and, possibly, an object attribute. Thus, this entry contains all the binding information that is exported to the namespace by the server.

- One namespace entry for each object that the server offers. Each entry contains an object attribute that contains that object's UUID, and a group attribute that refers back to the exporting server's namespace entry.

Note that the object entries consist of a combination of attributes not encountered before (object and group). Although unorthodox combinations of attributes are not generally recommended, they can sometimes be useful, as in this example.

The advantage of this scheme is that it greatly reduces the amount of server-provoked export activity into the namespace. When the server is first activated it creates all the namespace entries, exports the objects' UUIDs into the object entries, and initializes the group attributes to refer to the server entry. It exports its binding information into the server entry only. From then on, whenever it is restarted, all the server needs to do is re-export its binding information into the single server entry. Everything else remains the same; that is, the objects' UUIDs have not changed, nor has the name of the server entry to which the object entries' group attributes refer. Thus, instead of exporting bindings to every one of its object entries on subsequent startups, the server exports to only one entry.

Of course, if the system were restarted or the namespace reinitialized, then the original start-up process would have to be repeated.

The slight disadvantage of this scheme occurs on the client side, where the import process becomes somewhat more complicated than it would be if all necessary information (both binding and object UUID) could be read in from the same entry.

2.1.15 Server and Client Steps

The following subsections describe in detail, from both the server's and the client's side, how this model works.

2.1.15.1 Server Export

This section lists the steps that the server must perform to set up and initialize its namespace. Each step consists of the NSI function that must be called to perform the operation.

1. **uuid_create( )**

   To create an object UUID for each object that the server intends to export.

2. **rpc_server_register_if( )**

   To register interface(s) and EPVs with the RPC runtime. (This is also where manager types, if any, are registered.)

3. **rpc_server_use_all_protseqs( )**

   To request bindings from the RPC runtime for each object.

4. **rpc_server_inq_bindings( )**

   To get the binding handles for each object.

5. **rpc_ns_binding_export( )**

   To export the binding information of the objects' common server to its own separate name entry. This step is performed *only once* for each collection of objects managed by the same server.

The final three steps set up the grouped collection of service objects:

6. **rpc_ns_binding_export( )**

   To export each object's object UUID to its own simple name entry. A **NULL** is passed as the *binding_vec* parameter to specify that only an object UUID, and no bindings are being exported.

   Note that each object UUID must be exported to *both* the object name entry and the server entry; therefore, this call will be made twice.

7. **rpc_ns_group_mbr_add( )**

   To add the server's name entry (created in the first step) as the sole member of an NSI group attribute in each of the separate objects' name entries created in the second step.

8. **rpc_ep_register()**

   To register each object's UUID with the server's host machine's endpoint mapper.

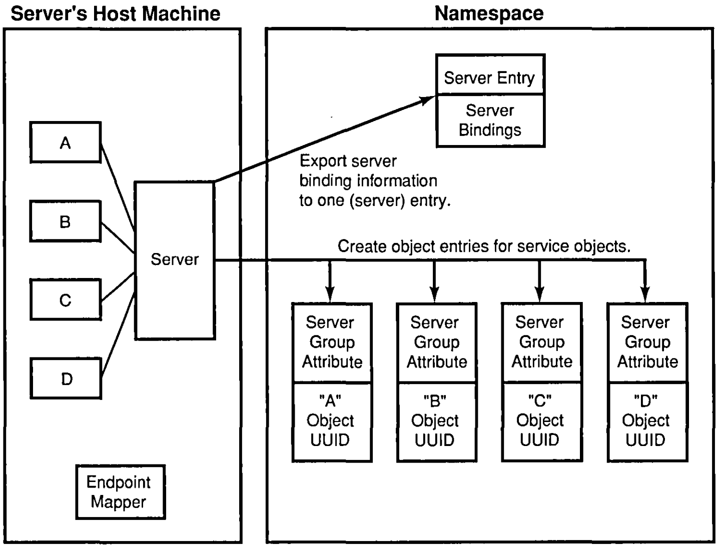
   Unlike the object-oriented model originally discussed, where there was a set of binding handles in each object entry, and where each object's set of handles was registered with that object's UUID in this step, there is only one set of binding handles in the grouped model. Therefore, when registering object UUIDs with the endpoint mapper, an application that uses the grouped model should reregister the same set of handles with each object UUID. The point of this step is to make sure that, when presented with an object UUID in an incoming RPC, the endpoint mapper can look that UUID up in its database and find an endpoint that has been registered with it. Registering the server's bindings (that is, endpoints) with all object UUIDs will accomplish this.

Step 6 is made necessary by the way the ACL editor's binding mechanism works. (Applications gain access to the ACLs that an application maintains on its objects through the client agent **acl_edit**, which uses a standard DCE-wide interface for ACL operations.) The **acl_edit** mechanism contains code that allows it to bind to the server that implements the ACL manager responsible for the object whose ACL is desired. However, these generalized binding routines necessarily conform to certain fixed ways of doing things. If the **acl_edit** binding mechanism obtains an exported object's object UUID from the object entry, it will use that object UUID in its subsequent import through the group attribute.

Thus, the object UUID will be contained in the handle structure that the client presents to the **rpc_ns_binding_import_next()** call, expecting it to be filled in with binding information. However, the RPC runtime always tries to match such an input object UUID with a UUID contained in the entry that the caller is trying to import from. If no matching object UUID is found, no binding information will be returned. Thus, *all* the single object UUIDs separately exported to the object entries must be exported to the server entry as well, if the exported objects are to have ACLs accessible through the **acl_edit** mechanism.

Figure 2-9 illustrates the resulting namespace arrangement.

Figure 2–9. The Export Operation in a Model with Grouped Bindings



This generic server manages four objects, called simply ''A,'' ''B,'' ''C,'' and ''D.'' One entry is created for each of these objects, and a separate entry is created for the server itself, where the binding information is held.

The result of all this is that there is now one more namespace entry for a given service instance than there would have been with the object-oriented model discussed earlier. The group attribute in each entry is a level of indirection that allows the server to dispense with exporting many copies of the same thing.

If a directory with the proper permissions has been set up for it in the namespace by the system administrator, a server should be able to create the object entries simply by making the calls described here.

2.1.15.2 Client Import

To bind to an object managed by the server as previously described, a client performs the following series of library calls:

1. **rpc_ns_entry_object_inq_begin( )**

   To set up an object inquiry context.

2. **rpc_ns_entry_object_inq_next( )**

   To return the object UUID that the server exported to the object's entry.

   This UUID will allow the server host's endpoint mapper to accurately map the incoming remote procedure call to the server that exported this entry. The UUID may also be used by the server itself to determine which object the client wants to access. Note that although this set of library routines is designed to accommodate schemes in which multiple object UUIDs have been exported to the same entry, the model described here requires that *only one* object UUID (the unique identifier of the object to bind to) be exported.

3. **rpc_ns_entry_object_inq_done( )**

   To delete the object inquiry context.

4. **rpc_ns_binding_import_begin( )**

   To set up a binding import context.

   An alternative to using the binding import routines would be to use the group member inquiry (**rpc_ns_group_mbr_inq_...( )**) routines to learn the name of the entry referred to in the group attribute, and then to do a direct import from that entry.

   The reason for using the **rpc_ns_group_mbr_inq_...( )** routines, rather than the normal import functions (**rpc_ns_binding_...( )**), would be to make sure that the group (and not some other) attribute in the entry is read. The **rpc_ns_binding_import_next( )** routine is defined to successively exhaust the contents of an entry's

   - **binding** attribute
   - **group** attribute
   - **profile** attribute

Since the model described here employs object entries with only group attributes and no binding or profile attributes, using the normal import routine should work fine.

5. **rpc_ns_binding_import_next( )**

   To read the entry's group attribute.

   The name service's access to (and return of the binding handle from) the entry's group attribute is transparent and unerring because there is only one set of binding information associated with a given entry in this scheme, and that information is found only in the group attribute. Note that if there had been more than one member in the group, which in fact is generally the case when group attributes are used, then the order of return would be random. Or if there had been binding information associated with *both* attributes, then here also the order in which binding handles would be returned would be random; that is, the caller may get a handle from the simple name attribute first, and then the handles exported to the group members, or it may get one or more of the group's member's handles, then one or more of the simple entry's handles, and so on.

6. **rpc_ns_binding_import_done( )**

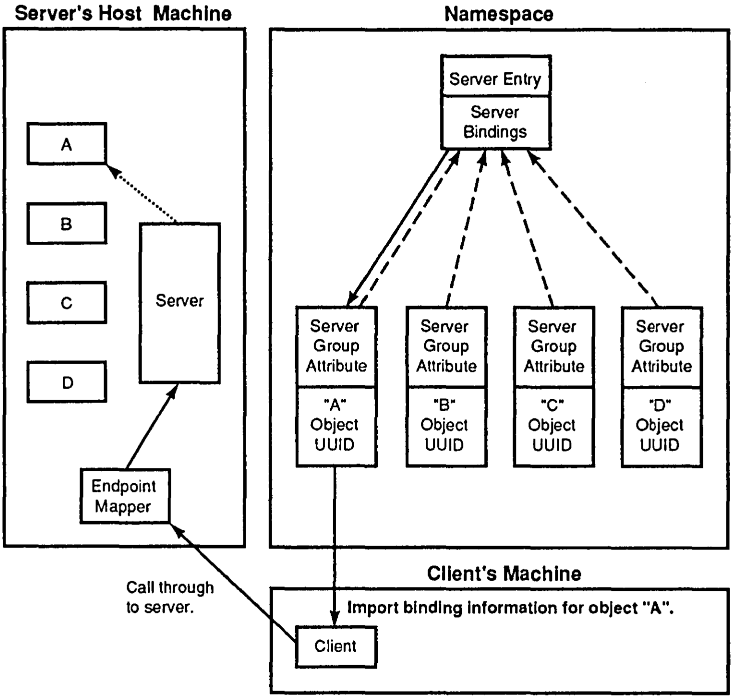
   To delete the binding import context.

7. **rpc_binding_set_object( )**

   To insert the object's object UUID into the imported binding handle.

Figure 2-10 illustrates this activity.

Figure 2–10. Importing from a Model That Uses Grouped Bindings



The client shown in the figure imports a binding for object ‘‘A.’’ This becomes (through the group attribute) a referral back to the server’s entry where the bindings are held, and a binding is indirectly imported from the server entry. The object UUID for ‘‘A’’ is read, in a separate operation, directly from the object’s entry. With this information in its binding handle, the client makes its first remote call through the server’s interface. The call finds its way to the endpoint mapper via the partial binding information, and the endpoint mapper completes the binding by looking up the object UUID, which was registered there by the server.

2.1.16 Global Organization of the Namespace

Since DCE is designed to support very large namespaces, it uses a hierarchical service for binding. The global scale is separated into cells whose boundaries are administratively defined. For example, a company using DCE may have a cell containing its employees and local services. The cell namespace administrator could decide to put all the service entries in a single directory if the cell were small.

Both the import and export name service operations support default values derived from environment variables; for example, **RPC_DEFAULT_ENTRY_NAME**. The environment variables can be set by start-up files to the name of a well-known directory within the cell. The only remaining decision then will be how to name the actual entries within the directory. One easy method is to use mnemonic names, or names of interfaces such as **binop**, **spm_library**, and so on. If these entries are only being accessed by clients through profiles, their names will not be directly visible to the client anyway.

But now imagine a larger organization. The administrator will want to define some naming hierarchy based on geography, organization, or other criteria. Somewhere within this hierarchy some writable directories (or parent directories) would be created, which could contain server entries, profiles, and so on. If clients are using only profiles to access bindings, then this organization will still be transparent to them. If clients want to bind to specific servers or objects, then more attention must be paid to the names given the servers' or objects' entries. The names should in some way reflect the organization, geography, or other relevant aspects of the server or object.

In summary, the important points to keep in mind are the following:

- The model should be appropriate for the organization and permit efficient administration of the namespace.
- There should be simple guidelines for naming objects and services so that users have a good chance of guessing the right answer.

2.2 Thread-Safe Programming

The following subsections describe thread-safe programming.

2.2.1 Introduction to Thread-Safe Programming

DCE contains a user-space threads package both for use in client applications and in order to allow concurrent request handling in servers. The DCE Threads functionality is made available through a pthreads interface. If the underlying operating system has its own (kernel) threads package, the DCE pthreads interface becomes an interface into the native threads implementation.

It is possible to write DCE applications without explicit multithreading, although RPC always employs multithreading on its own. However, it is worth your while to use DCE Threads, particularly in server applications. When developing a new multithreaded application or converting a single-threaded application into a multithreaded application, you must employ some special coding practices to ensure that the concurrent threads do not interfere with each other in various ways during execution. There are two kinds of code to which this principle applies:

- The multithreaded code itself, which must be made safe.
- The code in libraries used by threaded applications.

The second case further subdivides into two scenarios. If you are developing a multithreaded application that uses nonthreaded libraries, you must access the routines in such libraries in a thread-safe way. On the other hand, if you are writing a library, you should bear in mind that thread-safeness is desirable even if the library routines themselves are not intended to be multithreaded, since the routines may be called by multithreaded applications. The following subsection offers some guidelines on how to ensure thread-safeness in your code.

One important feature of DCE Threads, namely the exception handling interface, is not discussed in this section. In general, only topics that relate directly to thread-safe programming are discussed in this section. For a comprehensive discussion of DCE Threads, including an example program, refer to Part 2 of this guide, and to Chapter 1 of the *OSF DCE Application Development Reference*.

2.2.2 What Thread-Safe Means

The only resources private to a thread, as opposed to a process, are the following:

- A program counter value; that is, the address of the instruction the thread is about to execute.
- A stack pointer value; that is, a certain amount of memory allocated to the thread at its creation, in which its local variables are stored.

Thread-safeness is mostly a measure of the integrity of memory, both local and global, when code is executed by more than one thread. Suppose that the stacksize allocated for a group of threads is adequate; in other words, there is enough space on the stack to accommodate all the local variables created as the result of the deepest possible nesting of subroutine calls during the threads' lives. Making the code thread-safe then becomes a matter mainly of making sure that all operations by threads on global data are atomic; that is, not interrupted by other operations on the same global data (or other instantiations of the same operations) being executed by other threads that are executing the same code. The local variables are taken care of by the threads' local storage allocated to them at their creation. Consider a subroutine called **navigate** in which the following statement appears:

```
longitude = longitude + 1;
```

where **longitude** is a global variable. An application cannot just spawn a group of threads and release them on this statement. The unrestricted execution of the multiple threads of the machine instructions compiled from this ''simple'' statement on the *same* (because it is global) variable, all interfering with each other in a completely indeterminate way, will result in **longitude** containing a useless value when the threads have finished with it.

2.2.2.1 Locks

The remedy for this difficulty is to make sure, by means of special coding practices, that only one thread has access to the global variable for the duration of its operation(s) on the variable. The data structures associated with these coding practices are usually called ''locks,'' but you should not be mislead by this term; there are no ''built-in'' barriers to access to any data. All locking mechanisms depend for their efficacy on certain routines being explicitly called at the right times, both before and after access to the sensitive data. In the remainder of this section, the terms ''lock'' and ''unlock'' will occasionally be used as handy abbreviations for the two halves of this process, which is described in more detail in Section 2.2.6.

Locks can also be used to regulate threads' access to executable code. This is the ''brute-force'' way to make a section of code thread-safe, at the cost of temporarily losing most of the benefits of multiple threading. For example, consider the following code fragment from an imaginary threaded application:

```
latitude = current_latitude();
date = get_date();
speed = current_speed();
```

where the routine **get_date**() belongs to an unthreaded library. The call to **get_date**() can be made thread-safe simply by surrounding it with a global lock:

```
latitude = current_latitude();

/* acquire global lock before calling unthreaded routine */
pthread_lock_global_np();
date = get_date();
/* unlock access for other thread(s) */
pthread_unlock_global_np();

speed = current_speed();
```

The result is that access to the routine **get_date**() is serialized; that is, it can be called and executed by only one thread at a time. Further discussion of global locking can be found later in this chapter.

Note that there is nothing intrinsically unsafe in the statement

```
longitude = longitude + 1;
```

as long as **longitude** is a local variable.

2.2.2.2 Summary

In summary, code can be thread-safe in either of two basic ways:

- By being thread reentrant; in other words, the code uses only local storage, and can be safely executed by multiple threads of execution as is. Thread-reentrant code actually uses threads, but does so safely.
- By being made accessible to only one thread at a time; that is, each currently executing thread locks the code from access by any other threads, which wait in turn for their chance to lock and execute the code. This is the ''brute-force'' approach, which results in safe execution at the cost of the advantages that would be derived from multithreading. In other words, the code is made thread-safe by not using threads.

Note: ''Code'' is used in this context to mean both executable statements and data storage space.

The two approaches are not mutually exclusive within an application. Thread-reentrant code will often have to make use of locking mechanisms in order to serialize access to global data or other critical sections of code.

Threads are a sort of trick played on the operating system; that is, they allow a process to temporarily multiply itself over a certain section of iterated code and thus execute the totality of the iterations faster. However, none of the operating system's housekeeping mechanisms are aware that the original process has suddenly become several threads of execution. So it is up to the process itself to provide the protection for its address space that would otherwise (if the separate threads were separate processes) have been provided by the operating system.

2.2.3 Making Code Thread-Safe

The following two subsections describe how to make sure that multithreaded code will execute safely and correctly, and how you can safely use nonthreaded code with a multithreaded application.

2.2.3.1 Thread-Reentrant Code

Truly threaded (as opposed to merely thread-safe) code is thread reentrant, which means that the code can be safely subjected to execution by multiple threads. Locking mechanisms are used to access global variables (and any unthreaded code), but for the most part the threads execute concurrently, subject to the scheduling and priority policy, until they are terminated.

Following is a list of the guidelines you should follow in order to produce thread-reentrant code:

- Use the proper locking mechanisms to access global variables.
- Use the global locking mechanism to access unthreaded code.
- Make sure that the threads' stacksize attribute is adequate to accommodate the deepest possible nesting of subroutine calls that can occur during the threads' lives.
- If for some reason it is awkward or not possible to use the stack for local storage, use the **pthread_keycreate()** mechanism to set up a private static storage space for the separate threads.
- Make sure that your compiler generates thread-reentrant code.
- Document the code as being thread reentrant.

2.2.3.2 Using Nonthreaded Code As Is

If your threaded application calls routines from libraries that you either know or suspect to be nonthreaded, you will have to surround *all* ‘‘unsafe’’ library calls with the global locking mechanism. This is the only way to ensure that only one thread of execution is active in the library at a time.

There is only one global lock. It is acquired by a successful call to **pthread_lock_global_np**( ), and released by calling **pthread_unlock_global_np**( ). From within your application code, you can make calls to nonthreaded library routines safe in the following way:

```
/* Since my own routines are presumably correctly coded for */
/*   threading, no other precautions are necessary...       */
my_own_routine(num);

/* However, only one thread at a time can be allowed to     */
/*   access the code in the next call, since it's in a      */
/*   nonthreaded library...                                  */
pthread_lock_global_np( );
nonthreaded_routine( );
pthread_unlock_global_np( );

/* Back in my own --presumably thread-safe-- code, I can    */
/*   now continue as before...                               */
another_of_my_routines( );
```

As with any threaded routine, you should imagine this code being executed simultaneously by several threads at runtime; each of them is at some indeterminate point in the code, using its own separate copies of local variables, but only one copy for each global variable. However, at the call to **pthread_lock_global_np**( ) each one must pause (if another thread already holds the lock), queue up, and execute **nonthreaded_routine**( ) separately, one at a time.

If you have access to the unsafe code, you can position the lock operations on the other side of the subroutine calls at the beginnings and ends of the subroutines themselves.

Note that it is essential to use the one global lock to serialize all of an application's accesses to unthreaded libraries, even though it may appear from the application code that the accesses can be safely synchronized with multiple local locks. The reason for this is that you cannot be sure what is going on inside these libraries, where there could be various unsafe interactions that would escape any local locking scheme.

2.2.4 How Code Becomes Multithreaded

Multithreading occurs in a DCE application in either of two ways.

The first is by explicit calls to **pthread_create**( ). The application code passes to **pthread_create**( ), among other things, the address of a routine that the thread, once created, will execute. The thread lasts until it either returns from this routine or is explicitly terminated by a call to **pthread_cancel**( ), **pthread_detach**( ), or **pthread_exit**( ). Calls to other routines can occur before this happens. Note that this is the usual way threads are created and terminated.

Multithreading can also occur implicitly in DCE server applications, when the server begins listening for incoming client requests by calling the routine **rpc_server_listen**( ). If the *max_call_requests* parameter, which specifies the maximum number of incoming calls the server is willing to concurrently handle, is greater than 1, then the RPC runtime will spawn up to that number of threads for the server's manager routine as the calls come in. Note that implicit multithreading in the server is the server side of the RPC thread concept. This means that you should follow thread-safe programming practices in coding server manager routines, even though you are not explicitly creating the threads.

2.2.5 Memory Management in Threads

You can adjust the stacksize attribute for server manager threads, *before* they are created, by calling the **rpc_mgmt_set_server_stack_size**( ) routine. There are also DCE RPC routines that allow you to perform memory management specially tailored for the requirements of manager threads. These routines are part of the RPC ''stub support'' interface, so called because these routines are also used in the IDL-generated stub code to perform various memory management tasks. All of the routines have names beginning with **rpc_ss_**, and can be looked up in the *OSF DCE Application Development Reference*.

You should refer to the memory management information in Chapter 17 of this guide for further information about the use of the stub support routines in the server manager. However, one important detail will be mentioned here. When extra memory has to be dynamically allocated within some server manager thread for the purpose of performing a client-requested operation, it is very important that this memory be deallocated when the server-side operation completes execution. This ensures that the server will not continue to accumulate dead memory from operation to operation, growing bigger and bigger until catastrophe occurs. The amount of memory occupied by a server should remain constant across RPC operations.

If you use the **rpc_ss_allocate**( ) routine to allocate extra memory required within a manager thread, the memory will be automatically deallocated, along with any memory that was allocated within the server stub (for marshalling, unmarshalling, and so on), by the RPC runtime when the server-side operation completes execution. To use this mechanism within threads that you explicitly spawn from a manager thread, you should first call the **rpc_ss_get_thread_handle**( ) routine in order to get the handle of the manager thread. You should then pass this handle to the threads that were newly created by **pthread_create**( ). If any of these threads call **rpc_ss_allocate**( ), they must first call **rpc_ss_set_thread_handle**( ) in order to associate the manager's thread handle with any allocation requests made by calling **rpc_ss_allocate**( ) in this thread.

The details of this technique are described in the reference page for **rpc_ss_get_thread_handle**( ) in the *OSF DCE Application Development Reference*, and in Chapter 17 of this guide.

2.2.6 Mutexes

As seen in the previous section, there are times when you must restrict access to some data or area of code to only one thread at a time. DCE Threads provides several mechanisms for accomplishing this. The global lock, which was demonstrated in Section 2.2.3.2, is one of these. The global lock, however, is an extreme measure that is not usually necessary. The more usual locking mechanism is the ''mutex'' (*mut*ual *ex*clusion object).

Mutexes, in contrast to the global lock, are embodied in data structures that you declare and optionally initialize. There can be any number of mutexes in an application; the idea is that each mutex is dedicated to serializing thread access to one particular data structure or block of code. However, their use is similar to that of the global lock. A call to **pthread_mutex_lock()** or **pthread_mutex_trylock()** is placed before the statements that access the sensitive code. The first statement executed after completion of the sensitive statements is a call to **pthread_mutex_unlock()**.

Any thread that approaches code locked by a mutex has to execute the call to **pthread_mutex_lock()** or **pthread_mutex_trylock()** first. If no other thread is currently holding the mutex, the result of either call is that the calling thread acquires the mutex and continues on through the following code. While this is going on, access is blocked to any other thread (or threads) that tries to do the same thing; that is, with **pthread_mutex_lock()** the thread will simply block on the call, while with **pthread_mutex_trylock()** a value is returned (without blocking), indicating that the mutex was not acquired.

At the end of the stretch of protected code, the first thread encounters the call to **pthread_mutex_unlock()**; once this call is executed, the mutex is released and can be acquired either by a waiting thread or by the next thread that happens to reach the mutex acquisition routine. Suppose the acquisition routine is **pthread_mutex_lock()**, which means that the second thread has so far been blocking on the call. This call in the second thread now returns with that thread holding the mutex; the thread now proceeds to execute the subsequent code.

Note again that the only thing that prevents the threads from ''barging'' into the locked code is the call to **pthread_mutex_lock()** or **pthread_mutex_trylock()** that precedes it.

There are three kinds of mutexes: fast, recursive, and nonrecursive. The advantage of the second type is that it can be locked more than once by the same thread without having been unlocked first. Doing this with a fast mutex will result in a deadlock, a condition where the thread will never return from the second attempt to acquire the mutex. Nonrecursive mutexes cannot be relocked like this; however, they cause an error to be returned if it is attempted, rather than deadlocking. Once a mutex has been declared, you can initialize it as the kind you want by associating a properly filled-in attribute's object with it in a call to **pthread_mutex_init()**. The global lock is a recursive mutex.

A typical use for mutexes is to serialize multithreaded access to an application's global variables (where a different mutex is associated with each variable), or to other shared data or code that is known to be accessed only within the application. On the other hand, the global lock is called for in situations where you do not know what the ramifications of certain accesses (typically into other libraries) are, and thus you cannot be sure that what look like separate accesses into different libraries do not actually clash because of hidden interdependencies between the two. In such cases, the only safe procedure is to serialize accesses to all of the libraries with the same lock.

2.2.6.1 When Signaling a Condition Variable Results in Its Deletion

Consider the following code fragment executed by a ''releasing'' thread:

```
pthread_mutex_lock (m);

        <...>

/* Change shared variables to allow some other thread to proceed */

pthread_mutex_unlock (m);
                            <---- Point A
pthread_cond_signal (cv);   <---- Statement 1
```

Now consider the following code fragment executed by a ''potentially blocking'' thread:

```
pthread_mutex_lock (m);
while (!predicate ...
      pthread_cond_wait (cv, m);

pthread_mutex_unlock (m);
```

Note that it is possible for a potentially blocking thread to execute at *Point A*, find the predicate **TRUE**, and therefore not be blocked on the condition variable. In general, this does not cause a problem, but there is one exceptional set of circumstances. This arises when the released thread is the owner of the condition variable and is free to delete it without any further synchronization with the releasing thread. The released thread may thus delete the condition variable at *Point A* before the **pthread_cond_signal( )** is executed by the releasing thread. This will result in an attempt to signal a nonexistant condition variable. The error is only *optionally* detected in POSIX or in the DCE Threads architecture.

The situation described can occur when the releasing thread is a ''dependent'' thread and the waiting thread is the ''master'' thread; and in a code sequence such as the one just illustrated, the last dependent thread tells the master that it is safe to deallocate the variables shared by master and dependent.

In situations where the very act of signaling a condition variable may cause the condition variable to become deleted, it is best to signal or broadcast *with the mutex held.* For example:

```
pthread_mutex_lock (m);

        <...>

/* Change shared variables to allow some other thread to proceed */

pthread_cond_signal (cv);    <---- Statement 1
pthread_mutex_unlock (m);
```

Of course, there are many ways to code races with threads, but the situation described above is a particularly important one to beware of.

2.2.6.2 Using pthread_cancel( ) to Terminate a Thread

The **pthread_cancel( )** routine allows a thread to cancel itself or another thread. The routine is fully described in the *OSF DCE Application Development Reference* and in Part 2 of this guide. Its use is straightforward, but if you use it to cancel a thread that makes use of mutexes or condition variables, you should keep in mind the following aspect of its operation.

The canceled thread receives the cancel in the form of an exception. If the thread has not disabled its cancelability by a call to **pthread_setcancel( )**, its effect is to immediately terminate the thread. However, if the thread happens to have acquired a mutex (including the global lock) when it is canceled, the mutex will remain in its locked state and no other thread will be able to acquire it. Moreover, the data that was protected by the mutex may be in an inconsistent state as a result of the thread's having been canceled in the middle of its operation on the data.

The easiest way to prevent this is simply to disable cancels before entering code for which access has been restricted by a mutex. If this is undesirable, you can explicitly handle a cancel by coding an exception-handling block. The DCE Threads exception handling interface is described in Chapter 7 of this guide.

This same possibility exists with condition variables, since the variable is protected by a mutex. An example of handling a cancel (or any other exception) while using a condition variable follows. It is substantially the same example that appears in Part 2 of this guide.

```
#include <pthread_exc.h>

    <...>

/* First, lock the mutex that protects the condition variable  */
/*      and the predicate...                                    */
pthread_mutex_lock(some_object.mutex);

/* Add this thread to the total number of threads waiting for  */
/*      the condition...                                        */
some_object.num_waiters = some_object.num_waiters + 1;

/* Enter the exception handling block...                        */
TRY

    /* Test the predicate condition...                          */
    while (! some_object.data_available)

        /* If the desired condition is not yet true, wait for  */
        /*    it to become true. This next call also auto-      */
        /*    matically releases the mutex...                   */
        pthread_cond_wait(some_object.condition, some_object.mutex);

    /* Code to access data_available goes here */

    <...>
```

```
/* If a "cancel" exception occurs during the call to          */
/*      pthread_cond_wait(), the thread will resume execution  */
/*      in the FINALLY block following...                      */
FINALLY

    /* Remove this thread from the total number of threads     */
    /*    waiting for the condition...                         */
    some_object.num_waiters = some_object.num_waiters - 1;

    /* Release the mutex, and then continue with the           */
    /*    exception --i.e., cancel...                          */
    pthread_mutex_unlock(some_object.mutex);
ENDTRY
```

Note that in order to handle the cancel as an exception, you must **#include** the **pthread_exc.h** header file rather than **pthread.h**; this allows you to use the DCE Threads exception interface.

Further information on mutexes can be found in Part 2 of this guide, and in the *OSF DCE Application Development Reference*.

2.2.7 Methods for Synchronizing Threads

There are a couple of ways that cooperation among the threads can be synchronized at critical points in the code: by using condition variables or by calling the **pthread_join**( ) routine.

2.2.7.1 Condition Variables

A condition variable causes threads to wait at a certain point in the code until a specified condition attains a specified state. The mechanism actually requires three objects:

- A global variable, called the ''predicate,'' which contains the present state of the condition
- The condition variable, which DCE Threads uses to maintain a queue of all the threads currently waiting on the condition

- A mutex, which regulates access both to the predicate and to the condition variable

Using the condition variable mechanism is a multistep process. First, the thread acquires the mutex, then it reads the predicate. If the condition is already satisfied by the current state of the predicate, the thread does not have to wait; it releases the mutex and continues on. If the predicate is not yet in the desired state, then instead of releasing the mutex the thread calls the **pthread_cond_wait( )** routine with the condition variable and the mutex as parameters. The thread blocks on this call, and at the same time the mutex is automatically unlocked so that

- The predicate can be read by other threads
- Other threads can be queued on to the condition variable
- The predicate itself can be updated

Meanwhile, another thread elsewhere should be either monitoring or performing some activity whose progress will eventually require the predicate to be updated to the waited-for state. When that happens, this second thread will

1. Acquire the condition variable's mutex
2. Update the contents of the predicate
3. Signal one of the waiting threads to wake up by calling **pthread_cond_signal( )**, or signal all of the waiting threads to wake up by calling **pthread_cond_broadcast( )**
4. Release the mutex

The thread was able to quickly acquire the mutex because, as each of the waiting threads acquired it and then called **pthread_cond_wait( )**, the mutex was automatically released again. Each of the waiting threads went to sleep thinking that it possessed the mutex, although in fact none of the threads did so at that time.

When the waiting thread(s) called **pthread_cond_wait( )**, they did so in a **while** loop whose continuation condition depended on the state of the global predicate variable. When a thread wakes up, it returns from the wait call and automatically reacquires the condition variable mutex. This time, the predicate's new state drops the thread out of the **while** loop; the mutex is explicitly released, and the thread continues on through the code. (Putting the wait call in a loop guards against spurious wakeups: if the predicate has not changed when the thread is awakened, the thread will stay in the loop and recall **pthread_cond_wait( )**.)

If **pthread_cond_timedwait( )** is used instead of **pthread_cond_wait( )**, the waiting thread will wait only a specified amount of time for the specified condition to change to the desired state. At the expiration of this interval, the thread will wake up (return from the call) just as if it had been signaled to do so. This type of wait should be combined with a compound condition in order to specify different subsequent actions for the thread, depending on why it woke up.

2.2.7.2 Explicitly Joining Threads

Another way to synchronize thread activity is by using the **pthread_join( )** routine. The calling thread passes the identifier of the thread it wants to ''join'' with; the result is that the calling thread blocks until the specified thread terminates.

Further information on both of these techniques, together with an example program, can be found in Part 2 of this guide, and in the *OSF DCE Application Development Reference*.

2.2.8 Thread-Specific Storage

As was mentioned at the beginning of this section, threads depend for their local storage on a certain amount of memory allocated to them from the stack when they are created. Once a thread has been created, this ''stacksize'' attribute cannot be altered.

Since most compilers do not check for stack overflow, you should ensure that your thread stack is big enough to accommodate the deepest possible nesting of calls that could occur in the thread. DCE Threads has a routine (**pthread_attr_getstacksize( )**) that allows you to check the stacksize. You can also change the stacksize attribute of a thread *before* creating the thread, by calling

1. The **pthread_attr_create( )** routine to create the attribute object

2. The **pthread_attr_setstacksize()** routine to set the desired stacksize
3. The **pthread_create()** routine, passing the attribute object as a parameter to create the thread

For situations where, for whatever reason, it is not feasible to use the stack for local storage, DCE Threads provides a mechanism for allocating thread-specific static storage. The following steps should be performed:

1. Call the **pthread_keycreate()** routine to generate a key that will be used by the threads to reference the static storage. Note that this step should be performed *before* the threads are created, and it should be performed *only once*; there is only one key, which is shared by the threads.
2. Create the threads.
3. Within the threads, if static storage is required, allocate the memory, and then associate the memory with the key by calling **pthread_setspecific()**.
4. At any time thereafter, call **pthread_getspecific()** to retrieve the address of the thread's static storage.

This technique is handy for avoiding having to pass data explicitly down through many layers of function calls within a thread.

Further information about all of these topics can be found both in Part 2 of this guide, and in the *OSF DCE Application Development Reference*.

2.2.9 Other Programming Considerations

The following subsections describe various other safety-related aspects of multithread programming.

2.2.9.1 Forking in a Threaded Application

The **fork( )** system call causes the creation of an exact clone of the caller's address space, resulting in the execution by two address spaces of the same code. In order to avoid the problems that would arise in a threaded environment when one thread, possibly without the others' knowledge, executes a **fork( )**, the POSIX model defines **fork( )** to result in the propagation only of the calling thread. Any other active threads are immediately terminated without notice.

The abrupt destruction of the other threads means that any mutexes they may have been holding at the time of the **fork( )** will persist in the locked (and therefore unacquirable) state. On the other hand, assuming that the call to **fork( )** is followed by a call to **exec( )**, then the outstanding mutexes will remain so only until **exec( )** is called, when the new process space will be reinitialized.

Thus, ''out-of-state'' mutexes are a problem for the forked thread only in the interval between the **fork( )** and the **exec( )**. Even so, as long as no calls occur here to routines outside the application, you can determine whether the thread is going to encounter any mutexes that could have been locked by the destroyed threads. However, it is impossible to be sure of this if calls into other libraries, which may have hidden interdependencies, occur in this interval.

Aside from these considerations, there is also the question of what happens when **exec( )** completes and execution returns to the original forking (and now lone) thread, which is left with an address space that may contain out-of-state mutexes (as well as an inconsistent state in the data protected by the mutexes) as a result of the **fork( )**.

For cases where forking in the presence of threads is felt to be necessary, DCE Threads provides a mechanism, the **atfork( )** call, which allows you to install ''fork handler'' routines for an application or a library. These routines will be automatically run as follows:

- A routine that will be run just prior to the fork in the parent process; that is, just before all of the other threads are terminated
- A routine that will be run in the child process just after the fork occurs; that is, just after all the other threads are terminated
- A routine that will be run in the parent process just after the fork occurs; that is, just before the parent (forking) thread resumes execution

Further information about **atfork( )** can be found in Part 2 of this guide, and in the *OSF DCE Application Development Reference*.

2.2.9.2 Restrictions on Software Interrupts and Exceptions

From a portable point of view, it is unspecified in which thread (on which stack) a software interrupt handler will run. It is also unspecified what happens if an exception propagates out of a software interrupt handler.

As a consequence, a software interrupt handler must not allow an exception to propagate out of it. The reason is that the exception could be caught by some random exception handler in some thread and result in strange behavior.

Thus, it is best to avoid complicated coding in a software interrupt routine. If you must write a software interrupt handler, ideally you should just release a waiting thread using the previously mentioned signal or enqueue functions. Note that this has the advantage of minimizing the code in the software interrupt, which benefits the application by reducing the latency and increasing the throughput for such interrupts.

2.2.10 DCE Threads and DCE RPC

DCE RPC internally uses a vendor-provided threading facility, POSIX pthreads. There is wide variation in the completeness and/or transparency of the various pthread implementations provided by vendors. The limitations of a given pthread implementation are inherited by any application that uses DCE RPC, including applications that unknowingly use libraries that internally happen to use DCE RPC.

The DCE RPC runtime has internal threads that need to run in a timely fashion; correct operation of the runtime depends on this. Typically, this means that the application or pthreads implementation must neither perform nor allow operations that block the entire process.

Refer to the platform's or vendor's pthread release notes to determine what limitations the implementation has. If you are developing a library that uses RPC, you should instruct users of this library to refer to the pthreads release notes. Limitations may include, but are not limited to, the necessity of using thread-safe libraries, and compliance with POSIX nonprocess-blocking call behavior for system and library calls.

2.3 Managing the Server's Authentication Key

The following subsections describe how you manage the server's authentication key.

2.3.1 Introduction to Authentication

The essence of authenticated RPC is that a client attempting to access a server must present a ''ticket'' to that server in order to prove its identity at each remote procedure call before the call can proceed any further. This server-specific ticket was previously acquired by the client's RPC runtime from the authentication service; it encrypted the client's Privilege Attribute Certificate (PAC) using a secret key *known only to the server and the authentication service*, and then it sent this ticket back to the client's runtime for presentation to the server's runtime. The PAC contains the client's UUID. If the server's runtime can decrypt the ticket using the server key, that means that this client (identified by its UUID in the PAC inside the ticket) got a validly encrypted ticket to this server (whose key was used to encrypt the ticket) from the authentication service, which is the only entity aside from the server that has access to the server key. This constitutes the ''authentication'' of the client. The authentication service was satisfied with the client's representation of itself, and the ticket the client presents is the proof of this. The server can now use the client's PAC to determine the client's authorization.

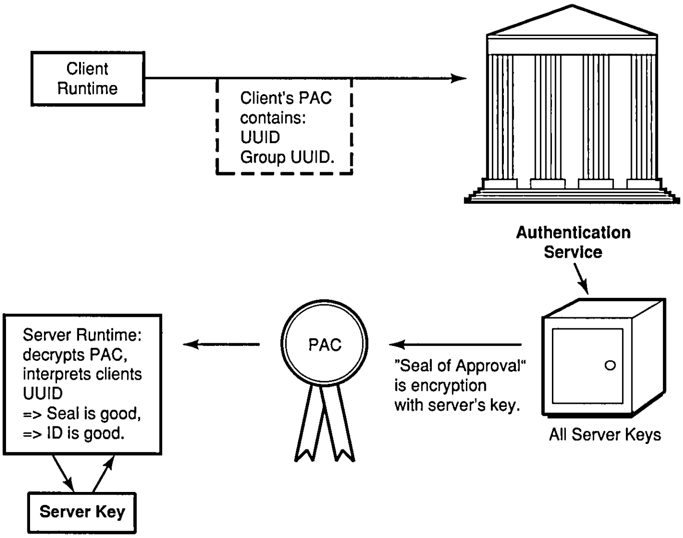
Note that all this back-and-forth ticket manipulation is performed by the RPC runtime; it is not the responsibility of applications. The runtime is also not responsible for the keys used to encrypt and decrypt the tickets; these must be supplied by the entities that intend to use them.

The server's key has a second use. If the server has to perform remote procedure calls to some other server (in other words, needs to become a client itself), the server's key is used as the basis of a login-like authentication process that produces a privilege ticket-granting ticket that the server's runtime can use to get tickets to servers.

In fact the ''server key'' is the server's encrypted password. The server sees a plaintext string that is exactly analogous to a user's (human principal's) password. From this a key appropriate to the designated enciphering mechanism is generated, as needed, by the authentication service or the server's runtime. Server key management arises from the need to provide servers with the means to remember, change, and manipulate their keys as human users are able to do.

Figure 2-11 illustrates the client/server authentication process, and also includes some details about server key management that are discussed later in this section.

Figure 2–11. Authenticated RPC and the Server Key


Client Runtime
Client's PAC contains: UUID Group UUID.
Authentication Service
Server Runtime: decrypts PAC, interprets clients UUID => Seal is good, => ID is good.
PAC
"Seal of Approval" is encryption with server's key.
All Server Keys
Server Key

The process by which the client convinces the authentication service of its identity is not illustrated here. Nor does this figure show all the separate steps required for the client's runtime to get the ticket from the authentication service, then present the ticket to the server, and so forth.

This figure is intended to illustrate only the concept of ticket encryption and decryption as the basis of authenticated RPC.

There are numerous details of the authentication process that are not pertinent here and therefore ignored; a lengthy description of authentication can be found in Chapter 40. The rest of this section is a short discussion of the server's secret key, explaining how it is generated and stored, and how (and why) it is managed.

Note: The term ''key'' is used loosely throughout this section. Although it properly means only the key derived from the password, it is often used to describe the plaintext string as well.

2.3.2 Server Key Storage and Retrieval

The current server key is actually stored in two places:

- In a local key data file, by the server

  A default local file is created by the Security Service when the server key itself is first created by a system administrator running the **rgy_edit** command. This file is owned by root, and in order to access it with the key management routines, the server itself must also be running as root. However, the server can also specify its own local file as an argument to any of the key management routines.

  This copy is used by the server runtime routines to decrypt incoming client tickets, and is also used when the server needs to acquire a login context.

- In the Security Service registry, by the Security Service

  This copy is used by the authentication service to encrypt tickets, for clients, to the server.

The key itself, which is in plaintext, is sent over the network as infrequently as possible.

The key management routines mainly affect the server's local copy, but some of the routines have an indirect effect on the registry copy in that they provide for updating of the registry copy when the local copy is changed.

Server key files are often referred to as ‘‘keytab’’ files elsewhere in the documentation.

2.3.3 Setting Up the Server Key File

In order to possess a password, a server must be a principal; that is, it must have an account in the Security Service registry. There are two ways this can be accomplished.

First, the server may simply inherit the login context, including the principal identity, of the user who invoked it. In this case there is no need for server key management as such because the key is derived from the human user's password, and the user is responsible for that management. The sample DCE application **timop** (described in Chapter 3) operates this way.

The second way a server can become a principal is by getting its own registry account. This is done by a system administrator running the **rgy_edit** command with the **ktadd** subcommand. This process, which consists of two separate steps (first, adding the account; then, creating the server's key) is described in detail both in the *OSF DCE Administration Guide* and in the *OSF DCE Adminstration Reference*. When this command is executed, a key data file is created for the server that contains its key. The default local file created by the system administrator as root is **/krb5/v5srvtab** on the local machine, where **rgy_edit** is run. The file can be created elsewhere by specifying a pathname relative to the current working directory:

```
rgy_edit => ktadd -p my_server_account -f /krb5/mysrvtab
```

When first invoked, a server process uses the login context (that is, a handle to the principal identity and secret key) of the user who invoked it until it can access its own secret key. This initial login context must have access to the file or device that stores the key. The procedure that is followed is described in Section 2.3.4. (See also Step B12, Section 1.5.17 in Chapter 1 of this guide.)

2.3.4 Acquiring a Login Context

As previously mentioned, a server when first invoked inherits the login context of the principal that invoked it. This identity may be sufficient for the application's purposes; however, if it needs to assume its own identity, which is what key management is all about, then it has to call the following routines in order to accomplish the switchover:

1. **sec_login_setup_identity( )**

   The server passes its own principal name to this routine, and it receives a login context (**sec_login_handle_t** structure), which is one of the things it will need to validate its new (true) identity.

2. **sec_key_mgmt_get_key( )**

   The server retrieves its password (key) in a **sec_passwd_rec_t** structure.

3. **sec_login_validate_identity( )**

   This call establishes the server's network credentials (the principal's ticket-granting ticket received from the authentication service).

4. **sec_login_get_current_context( )**

   Retrieves the server's login context.

If all has gone well, the server has successfully switched to its ''own'' identity, and can use its login context to receive authenticated requests from clients or to authenticate itself to other servers. See Steps B7 (Section 1.5.12) and B8 (Section 1.5.13) in Chapter 1 of this guide for the former case, and Step C2 (Section 1.5.19) for the latter case.

2.3.5 Using the Key

For the server the central authentication routine, which must be called before clients can conduct authenticated RPC operations with it, is

```
rpc_server_register_auth_info(server_princ_name, authn_svc,
                              get_key_fn, arg, status);
```

which among other things tells the RPC runtime where the server wants its local key to be read from when tickets incoming from authenticated clients are decrypted. Thus, if a server wants to use its own (nondefault) local key file, it should create the file before making this call if it does not already exist.

A server can create a local key file by calling the key management routine

```
sec_key_mgmt_change_key(authn_service, arg, principal_name,
                        key_vno, keydata,
                        garbage_collect_time, status);
```

where, if the *arg* parameter is non-**NULL**, it is interpreted to specify the server's local key file. The server supplies the key in *keydata*. If a file is specified, it is created (if it does not already exist) with read/write protection for the owner. To use the default file, the server must be running as root, since this file is created and owned by root. The specified key file should always be local, not accessed across a remote DFS mount point; otherwise, file accesses will result in the key contents being transmitted across the network.

The other important parameters to **rpc_server_register_auth_info**( ) are

authn_service — Specifies which authentication service is used for the **rpc_server_register_auth_info**( ) call.

princ_name — The server's principal name (a string).

key_vno — When a server's key is changed, the former key is not automatically deleted from the local storage. Instead, each version of a key is tagged with a version number. Clients with tickets encrypted (by the authentication service) with an earlier version key can still be authenticated by the server runtime, as long as those earlier versions are retained in the local storage. In order to find out what the next eligible key version number is, **sec_key_mgmt_get_next_kvno**( ) can be called, or 0 (zero) can be passed to specify the next appropriate version number.

keydata — A pointer to a **sec_passwd_rec_t** structure that contains either the server's new plaintext password or a pre-encrypted (in some arbitrary manner by the server) buffer.

garbage_collect_time This is an output parameter; it informs the server when it will have to call **sec_key_mgmt_garbage_collect**( ) to get rid of obsoleted keys.

get_key_fn When this parameter is non-**NULL**, it specifies the address of a server-supplied key retrieval function.

See also the reference pages for the key management routines in the *OSF DCE Application Development Reference* for information about other key parameters.

The server may wish to generate and store its key in some other hardware-specific way. The authentication mechanism provides for this by allowing a server-supplied key retrieval routine to be specified in the **rpc_server_register_auth_info**( ) call. This now becomes the routine that the RPC runtime will call to get the server key for decrypting incoming tickets. However, doing this means also that the **sec_key_mgmt_...**( ) routines can no longer be used to manage that server's key storage.

A server that installs its own key retrieval routine becomes completely responsible for the generation and maintenance of its key, as well as its synchronization with the registry copy of the key. It must provide its own functionality and mechanisms for all these things. When it changes its local key copy, it will have to call a routine like **sec_rgy_acct_passwd**( ) to update the registry copy. Maintaining earlier key versions, garbage collection of outdated keys, and so on, must all be implemented by the server.

The server-supplied routine is expected to be called by the runtime in the following form:

```
get_key_fn(arg, princ_name, key_type, key_ver, key, status);
```

where **get_key_fn**( ) is the name of the server-supplied function. Its parameters are

arg Pathname to the server-maintained local key storage.

princ_name Server's principal name.

key_type A pointer to a **sec_passwd_type_t** indicating the encipherment system with which the key is to be used.

key_ver Indicates the key version number. This is contained in the **sec_passwd_rec_t** structure.

key Returned by the routine; this is a pointer to an array of **sec_passwd_rec_t**.

status Returned by the routine; this is a pointer to a status code.

2.3.6 Typical Tasks in Managing the Key

The following subsections describe how the key management routines can be used and combined to perform useful tasks.

2.3.6.1 Updating a Key in Response to Cell Password Expiration Policy

Passwords do not usually last forever. Password expiration policy is set by the cell system administrator, and it affects the validity of server keys just as it does that of user passwords. Once a password's lifetime expires, it can no longer be used either to acquire a login context or as a key to encrypt or decrypt authentication tickets. (Note that this has nothing directly to do with ticket lifetimes; see Section 2.3.6.3 for information about maintaining previous version keys.)

If a server's password expires between invocations so that it does not have a valid login context for its principal name, then a context is created using the latest key available in the server's key file. If no such key is available, then the **sec_key_mgmt_e_key_unavailable** error is returned by the key management routines, meaning that the server process was unable to authenticate itself to the authentication service. A new password will then have to be created by the system administrator.

A more likely and potentially more troublesome problem is the expiration of a server's password, and hence the key derived from it, during a session. If this happens, the result will be not only that the server will not be able to acquire a login context and authenticate itself to other servers, but also that any outstanding tickets held by the server's clients will suddenly become invalid, and authenticated RPC will stop.

The **sec_key_mgmt_manage_key( )** routine will prevent this. The intention of this routine is to relieve server writers of the responsibility of determining when a server's key should be changed in response to the registry's password expiration policy.

This routine should be invoked from a server thread dedicated to this purpose. Once called, it will run indefinitely; it will never return during normal operation. The **sec_key_mgmt_manage_key( )** routine queries the registry for the expiration policy for the principal named in the call. It then idles until a short time before the server's current key is due to expire, when it calls **sec_key_mgmt_generate_key( )** to produce a new random key before the old one can expire. If necessary, it also calls **sec_key_mgmt_garbage_collect( )**.

Note again that a server providing its own key retrieval routine, specified in **rpc_server_register_auth_info( )**, is responsible for monitoring password expiration policy and taking appropriate action itself; it cannot use **sec_key_mgmt_manage_key( )** to do this.

2.3.6.2 Changing the Key

There is more than one way a server can change its key. The variations depend on the following:

- Whether the registry's copy is changed at the same time as the server's local copy or some time later
- Whether the server supplies its own new key value or requests the Security Service to generate a random value for it
- What is done about previous version keys in the server's local storage

The option of (temporarily) not changing the registry's server key copy while changing the local copy is useful for propagating a key change among slave replicas of a server.

It is up to the application to set its own key version maintenance policy, but there should be no reason for retaining outdated keys in the local storage; the **sec_key_mgmt_garbage_collect( )** routine should be used to delete them.

There is also the possibility that a server's key could be changed by a system administrator in response to some perceived security compromise. The server should be aware of this possibility with regard to any assumptions it makes about its current key value or key version number.

2.3.6.3 Maintaining Previous Version Keys

What happens if a server key is changed while clients still hold unused tickets to that server? Tickets have lifetimes, just as passwords do, and when clients' unused tickets are renewed automatically by the system, the renewals are issued against the current key. In order not to inconvenience clients holding unused unexpired tickets, the Security Service maintains key version numbers. Each server key has a version number, and tickets issued against that key also bear that key's version number. When a key is changed, the previous version is not automatically deleted. Thus, when outdated tickets are presented, the runtime applies the correct version key to them, if it still exists.

However, there is no reason to retain old version keys indefinitely. The **sec_key_mgmt_garbage_collect**( ) routine will, when called, delete all keys in the local storage that are older than the maximum ticket lifetime in effect.

Tickets presented by clients with key version numbers that no longer exist in the server's key file are not honored. A server can always delete either the current or an earlier version key from its storage. The **sec_key_mgmt_delete_key**( ) routine allows the caller to specify a key version number, and **sec_key_mgmt_get_nth_key**( ) can be used to scan the local storage for all existing key versions.

2.3.7 Key Management Routines

The following is a list of all the specific operations a server can perform on its key with the default key management interface; it is arranged by functionality. For complete information on each routine, refer to the *OSF DCE Application Development Reference*.

- Change to a new key:
  - Change both the local and registry copies:

    The **sec_key_mgmt_change_key( )** routine changes a key to a specified value.

    The **sec_key_mgmt_gen_rand_key( )** routine followed by **sec_key_mgmt_change_key( )** changes a key to a system-generated value.
  - Change the local copy:

    The **sec_key_mgmt_set_key( )** routine changes a key to a specified value.

    The **sec_key_mgmt_gen_rand_key( )** routine followed by **sec_key_mgmt_set_key( )** changes a key to a system-generated value.
- Retrieve a key from local storage:
  - Current key: **sec_key_mgmt_get_key( )**
  - Specific key: **sec_key_mgmt_get_nth_key( )**
- Delete a key from local storage:
  - Current key: **sec_key_mgmt_delete_key( )**
  - Current type key: **sec_key_mgmt_delete_key_type( )**
- Conform to cell password expiration policy:
  - **sec_key_mgmt_manage_key( )**
- Miscellaneous operations:
  - **sec_key_mgmt_free_key( )**
  - **sec_key_mgmt_garbage_collect( )**
  - **sec_key_mgmt_get_next_kvno( )**

2.4 Writing an ACL Manager

The following subsections contain some general information about how to write an ACL manager.

2.4.1 Introduction to Writing an ACL Manager

This text is intended to give some practical helpful hints on the most important things you must know in order to write your own ACL manager for a DCE application. For design and other information you should refer to Part 6 of this Guide and to the Security sections of the following books:

- *OSF DCE Application Development Reference*

  Contains reference pages for the ACL interface routines discussed later in this section.

- *OSF DCE User's Guide and Reference*

  Contains detailed discussions of ACL format and usage as well as reference pages for the **acl_edit** command.

Although several of the DCE components have their own ACL managers, these can be used to create and maintain ACLs only for those components' own objects. For example, if you add an entry to the namespace, then CDS will automatically attach an ACL to that entry, which CDS's ACL manager will be responsible for maintaining. The same thing is true when you add a principal to the registry, or when you create a DFS file or directory. All DCE components tie into the ACL interface described in the *OSF DCE Application Development Reference*. This interface is made up of all the **sec_acl_...()** calls (more information about this appears later in this section). This means that any application can use these calls on any of the DCE components' ACLs, provided that the application is properly bound to the desired server.

However, applications that define their own objects must provide their own ACL manager for those objects, if ACLs are desired. Consider the print service described earlier in this chapter. If each printer is given its own entry in the namespace, as recommended, then CDS will maintain ACLs for those entries; but the ACLs will pertain to the *entries*, not to the printer objects themselves. If it is desirable for the service to maintain ACLs on the printers, then the service must provide its own ACL manager to do so.

2.4.2 Design Guidelines

When designing an ACL manager, you should conform to the following guidelines:

- The DCE ACL guidelines as defined in the *OSF DCE User's Guide and Reference*.
- The standard DCE ACL interface as defined in the *OSF DCE Application Development Reference*. This interface is defined by the set of **rdacl_...()** calls. The application's ACL manager must support this interface.

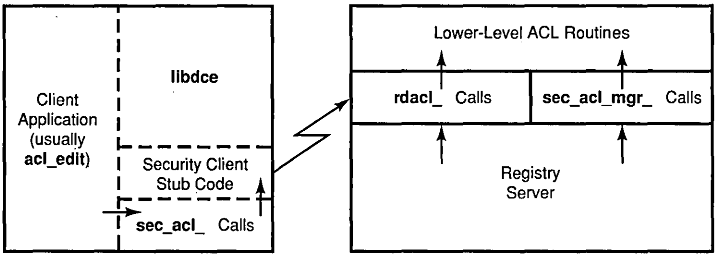
2.4.3 How ACL Interfaces Work in the Registry Server

The DCE Security Service, for its part, provides the **sec_acl_...()** calls as entry points in the DCE library (**libdce**). A remote client linked to this library (as all DCE applications should be) can now bind to and access this application's ACLs via the **sec_acl_...()** calls. The **acl_edit** command is a command-line interface to these same **sec_acl_...()** calls. The **sec_acl_mgr_...()** set is a third group of ACL-related calls described in the *OSF DCE Application Development Reference*. These are routines used locally within the server. The following subsections provide more information on the routines themselves, as well as the terminology used in regard to DCE ACLs.

2.4.3.1 ACL Interface Routines

Figure 2-12 illustrates how the **sec_acl_...()**, **rdacl_...()**, and **sec_acl_mgr_...()** ACL interfaces interact in the DCE registry server.

Figure 2–12. ACL Interfaces in the Registry Server



The Client application in the figure could be the **acl_edit** command, or it could be any other client that wants to access a registry ACL. The DCE library, **libdce**, contains (among many other things) the **sec_acl_...()** entry points and the **rdacl_...()** client stub code, which is used by the **sec_acl...()** calls. A client linked to **libdce.a** simply executes the **sec_acl_...()** calls as it would any local function.

These in turn call the remote **rdacl_...()** routines, which are implemented in the registry server. In other words, the ACL routine calls made by the client pass through two interfaces: a local explicit one (the **sec_acl_...()** calls), and the remote one (the **rdacl_...()** calls), which is utilized by the **sec_acl_...()** routines. Applications never call the **rdacl_...()** routines; they call the **sec_acl_...()** routines. This arrangement relieves the client application of some of the details of managing server bindings and so on.

Within the server, the **sec_acl_mgr_...()** routines are a local interface into the ACL routines used by the server itself and by the **rdacl_...()** routines as necessary. In other words, there are two avenues into the direct ACL-manipulation routines: one via the **sec_acl_...()** calls for remote clients, and the other via the local **sec_acl_mgr_...()** calls for the server itself.

The lowest-level ACL routines, where the ACL storage is actually manipulated, do not constitute a formal interface and are not visible to any but the **rdacl_...()** and **sec_acl_mgr_...()** routines.

All three of these formal interfaces (**sec_acl_...()**, **rdacl_...()**, and **sec_acl_mgr_...()**) are documented in the *OSF DCE Application Development Reference*. The intended use of these three sets of reference pages is as follows:

sec_acl_...() When developing client application code, refer to these reference pages to learn how to make the necessary calls to access and manipulate ACLs.

rdacl_...() When developing server application code, refer to these reference pages to learn what ACL-manipulation routines you must implement, what their behavior and call signatures should be, and so forth.

sec_acl_mgr_...() When developing server application code, refer to these reference pages as a guide and example of the repertory of ACL-management calls a server should implement locally for its own use.

The organization of an application-specific ACL manager should be similar to this scheme. The **sec_acl_...()** calls executed by a client would still come from **libdce**, only now a different set of **rdacl_...()** routines would be remotely executed, namely the specific application's (assuming, of course, that the client is bound to this server). Implementing the **rdacl_...()** interface makes the application's ACLs accessible via the **acl_edit** command. It would be up to the application developer to decide whether to implement the **sec_acl_mgr_...()** interface for the server's use; doing so would help to organize the manager's internal functionality.

2.4.3.2 An Important Note on Terminology

DCE has proven to be in some respects more extensive than the English language. A result is that in a few cases terminology is shared by functionality that is not in fact similar in behavior.

The term ‘‘type manager’’ is used in the DCE RPC documentation to describe a way of allowing a server to offer multiple implementations of the same interface to its clients. Incoming remote calls from various clients, all of whom are calling through the same interface, are switched by the RPC runtime to the appropriate interface implementation in the server on the basis of object UUIDs in the incoming calls. For more information on how this works, see Step B2 (Section 1.5.7) in Chapter 1 of this guide. An object UUID of a given type (the typing is done by the server as part of its setup) will vector its RPC to the server's appropriate manager code; hence the term ''type manager.''

On the other hand, ACL managers often implement more than one type of ACL. The differences among these types are characterized by the different sets of possible privileges that are appropriate for the object that is to be protected. Thus, one can quite naturally speak of ''ACL type managers,'' which contain within a server's ACL manager the code that implements the different ACL formats. However, these ACL submanagers do not use the RPC vector-typing mechanism, and the two types of manager should not be confused because they are quite different.

To simplify this concept, ACL managers typically handle everything themselves. If ACLs in various formats are supported, then the ACL manager itself is responsible on receipt of an incoming client request for calling the correct subroutine to perform the request. The **sec_acl_...()** routines expect a *manager_type* parameter, by which the client can explicitly specify the ACL type desired.

Furthermore, the *manager_type* parameter should not be confused with the *sec_acl_type* parameter, which is used to distinguish among certain basic kinds of ACL that apply to all of the ACL manager types. The following list will perhaps make the distinctions clearer:

manager_type Specifies a particular ACL type among several that may be implemented by an ACL manager. For example, a print server might implement ACLs both on individual printers and on groups of printers; the two types of ACL would have different sets of privileges, and would be implemented by different routines within the manager.

sec_acl_type Specifies one of three basic kinds of ACLs that are common to all the manager types:

- The Object ACL controls access to an object.
- The Initial Container Creation ACL serves as a default template for ACLs on newly created objects that can contain other objects.

- The Initial Object Creation ACL serves as a default template for ACLs on noncontainer objects.

When ACLs are created, they are always created as a side effect of creating an instance of the object they are associated with. Thus, there must be a set of default templates at hand for an ACL manager to use when objects are created; the *sec_acl_type* parameter is the specifier for the desired template in a **sec_acl_...**( ) call.

2.4.4 IDL Definitions

If you are developing an ACL manager that is intended to use the standard ACL interface, making it accessible both to users via the **acl_edit** command and to applications via the **sec_acl_...**( ) routines, there is no need to write an **.idl** file. All you need to do is compile (with IDL) the **rdaclif.idl** file supplied with DCE, which is located at

dce-root-dir/**install**/*machine_name*/**opt/dce***n.n*/**share/include/dce**

and link the server stub output with your server code. The client-side stubs are part of **libdce** and so are automatically linked to any client application. For more information on the IDL process, see Steps A1 to A5 (Sections 1.5.1 to 1.5.5) in Chapter 1, and also Chapter 3 of this guide. In addition, Part 3 of this guide contains chapters on using IDL.

Under no circumstances should you generate a new UUID for this interface. One of the things that make the standard DCE ACL interface work is that all implementations of the interface are identified by the same interface UUID, and all ACL clients bind through it. If you were to generate your ‘‘own’’ interface UUID and build an **.idl** file around it that you then compiled and linked to your server application, clients would never be able to bind to your manager (at least not using the standard ACL library routines) because the standard ACL interface UUID that the client-side **libdce** code would be seeking to bind through would not be the one exported by your server.

However, you *do* have to code your own implementations of the ACL interface. In doing so, you should refer to the reference pages for the **rdacl_...()** routines in the *OSF DCE Application Development Reference*. These routines describe the operations you must implement, namely

- **rdacl_lookup()**
- **rdacl_replace()**
- **rdacl_get_access()**
- **rdacl_test_access()**
- **rdacl_test_access_on_behalf()**
- **rdacl_get_manager_types()**
- **rdacl_get_printstring()**
- **rdacl_get_referral()**

2.4.5 Representation of Objects with ACLs in the Namespace

The binding requirements for an ACLed object are summed up in the reference page for **sec_acl_bind()**, which is the routine that user applications call to obtain a binding to such an object. The ACL editor command **acl_edit** uses this same routine. An ACLed object must be bindable *by name*, which means that clients must be able to obtain an unambiguous binding to the object (actually, to the server that manages that object) by importing from that object's entry in the namespace.

Thus, there are three general rules that must be observed by applications that maintain ACLs on objects:

- If the object's ACL is to be generally accessible through the DCE user interface (the **sec_acl_...()** calls and the **acl_edit** command), then the object must have an entry in the namespace; **sec_acl_bind()** looks up and imports through an entry name, nothing else. (See the note at the end of this section on using namespace junctions.)
- Moreover, the binding(s) exported to the object's entry must contain an object UUID that is registered by the server and that uniquely identifies the object so that the **sec_acl_bind()** mechanism can unambiguously reach the object through its server.

- Finally, all object UUIDs for objects with ACLs must be registered with the ACL interface UUID at the endpoint mapper by using **rpc_ep_register**( ). (See Step B10 (Section 1.5.15) in Chapter 1 of this guide for an example of how this is done.)

The **sec_acl_bind**( ) routine specifies the **NULL** interface when it performs its import. This allows applications that offer the ACL interface to not export it via **rpc_ns_binding_export**( ), which would greatly increase the size of the namespace. Of course, the interface UUID is used in the actual ACL operations, and is checked in incoming stubs by clients' and servers' respective runtimes. This is why the server registers the interface using **rpc_ep_register**( ), although it does not export it (with **rpc_ns_binding_export**( )).

The result of all this is that the object-oriented namespace organization illustrated in Figure 2-8 and described in Section 2.1.10 would work fine with an ACL manager implemented in the application server. The ACL interface is not exported into the namespace; all that is necessary is to make sure that each object's entry is unambiguously identified by its own object UUID in the exported partial bindings.

Note: Objects with ACLs can also be made available to users and clients through a namespace junction, which is a way of implementing a server-private namespace. This can relieve CDS of the burden of having to maintain separate entries for many objects. For more information on junctions, see Section 2.1.3.

2.5 Additional Guidelines

DCE gives you a set of tools and services that compartmentalizes the huge number of interrelated tasks involved in designing and implementing a distributed application into a manageable set of integrated services, which you can combine to build powerful distributed applications. Ideally, the applications that you build with DCE should similarly, where possible, consist of reusable, combinable, and robust functionality that will be easy to use and maintain.

One of the most powerful of the DCE tools is the RPC Interface Definition Language (IDL), which allows you to design services and service characteristics with this principle of modularity in mind. Good interface design in the IDL sense means organizing things so that the set of calls processed by a single server consists of related calls. In this way, the service can be used by as many clients as possible (who, it must be remembered, could themselves also be servers). If the interfaces are generalized enough, you can reuse them among other servers and combine them so that you get new functionality and the same kind of synergy that you have with the traditional UNIX tools.

To do this, server writers should keep the following in mind:

- Server code should be ''production quality.'' That is, it should
  - — Be multithreaded to increase efficiency.
  - — Make use of security resources, including ACLs and authenticated RPC.
- Servers, once installed, should be locatable under (as nearly as possible) all circumstances.

The following subsections discuss some of the specific DCE topics related to accomplishing these things.

2.5.1 Initialization and Configuration

The following subsections contain discussions of various aspects of starting a server process, whether for the first time or after it has already been active for some time and perhaps accumulated stored state information.

2.5.1.1 Storage of Configuration Information

Configuration information such as

- The cell the server is in
- The server's principal name
- The server's group
- Database filename(s)

should be held externally in local files to minimize dependence on the network, and be read in by servers at startup. This allows the information and the application to be dynamically configurable.

A server should not rely on any particular namespace structure. The names for server entries, object or resource entries, and so on, should either be read in from some local external storage or prompted for from the user who is installing the application.

If a server is to run under its own principal identity, an entry will have to be created for it in the Security Registry. This is done with the **rgy_edit** command. For further information, see the *OSF DCE Administration Reference*. See also Section 2.3 of this chapter.

2.5.1.2 Registering Binding Information

There should always be bindings for *only* one server per entry in the namespace. Moreover, the entry should always contain enough information to ensure that a client can bind to the specific intended server, or instance of service, represented by that entry. This usually means including an object UUID in the handle as well. Since the ACL editor does not use **rpc_ep_resolve_binding**( ), the binding handle received from an import operation *must* contain an object UUID.

In order to support the ACL editor, all object UUIDs for objects with ACLs must be registered with the ACL interface at the endpoint mapper.

2.5.1.3 Choosing a Directory Name

The names that a distributed application depends on to store and retrieve server location information (bindings) should, as far as possible, be indirect; in other words, you should rely on the name service group and profile entry mechanisms to find entry names of servers. This makes it easy to reconfigure the application. Rather than changing some part of the server or client code, you change the group or profile entry with **cdscp**. It also puts less strain on the namespace by keeping as much ''hard'' location information as possible out in the open in the namespace itself where it can be maintained.

An obvious alternative to hardcoding would be to use system-level environment variables. But this too should be avoided because it increases management overhead. Names and paths will change, and applications should not deal directly with hardcoded names at all.

Try to keep entry names intuitive and clear. This applies to groups and profiles, as well as to the simple server entries. On the other hand, if you are using one or more levels of entry indirection, keep the names appropriately indirect as well. For example, it would not make sense to call a group printer entry **floor_3_printers** if the entry was intended to contain printers from any location.

From this basic principle the following guidelines are derived:

- Is a service always associated with a host (as, for example, a process server)? If so, then put it in the host directory. If not, put it somewhere else more appropriate. Do *not* use misleading names.

- Names of entries should *never* be encoded directly in programs because hardcoded names are not configurable and are not easily changeable. If the application is moved, it will probably have to be recompiled. Instead, the entry name should be extracted from some external place, such as a local file or an environment variable.

- You should *not* create names in specific locations in the namespace. In other words, names should *not* be hardcoded into specific paths in the namespace hierarchy. Hardcoded names will unnecessarily constrain the namespace and make it hard to maintain.

- Every name in a profile or group should be global so that the entry will work no matter what cell it happens to be installed in.

Names are often assigned by applications in response to user input of various kinds. The following guidelines apply to such assignments:

- If a name or name service entry is to represent an RPC server, the user should be allowed to determine what name to assign by using normal RPC binding mechanisms to locate the server.

- If the name or name service entry will represent a directory, a non-RPC server, or an entry other than a server, applications should be able to use configuration profiles to locate the assigned name of the resource.

2.5.1.4 Namespace Usage Guidelines

Following are some basic guidelines that a new user of DCE can use when setting up the DCE namespace.

1. To place the server principal in the security namespace, assuming that your server is called **my_server**, you could start by putting it at

   /.:/hosts/*hostname***/my_server**

   Then, after testing it, you could move it to

   /.:/subsys/my_company/my_server

   or to some other well-known place, such as

   /.:/applications/my_server

2. To place the server binding information in the cell namespace, assuming you call your server **my_server**, you could export the bindings to

   /.:/subsys/my_company/my_server

   This would be good especially if you used **/.:/subsys/my_company/my_server** (see step 1). You could then use the same name for the binding entry and the server principal name.

   The entries for principal and for binding would be distinct (the former being located physically in the security space, the latter under the CDS namespace cell root), even though the names were the same.

3. To place the files your application uses (for example, mail message files for a mail application, posting files for a bulletin board application, and so forth), if access to the files is only through the server, then you want the files to be somewhere in *dcelocal* (*not* in *dceshared*). If the files are to be operated on directly by users, they could go in

   /.:/fs/opt/my_company/my_application/

   The default path prefix for *dcelocal* is set to /**opt**/*dcelocal* during the DCE configuration process. Note that this is a pathname in the machine's file system, *not* a CDS pathname. For a discussion of the differences between the two, see Section 2.1.1.

   For further information about the structure of *dcelocal* and the DCE namespace in general, see the *OSF DCE Administration Guide*.

2.5.1.5 Changing a Server's Location

Consider whether it is likely that the server will ever be moved, either to a different location within the same cell or to a different cell. If the server depends on locally held databases that also will have to be moved, then provision will have to be made to move the files in a machine-independent way. For example, a ``transfer server'' could be brought up on the new host to receive the files sent to it by the old host, which it would then store locally. The data would have been automatically converted to the appropriate machine format by the server stubs.

One of the most important consequences of moving a server involves security. Authenticated RPC depends on a database (the Security Service Registry, which contains a principal entry for the server, if the server runs under its own identity) and a local file (the server's key data file). Moving the server to a different cell may make its registry entry unfindable, unless its principal name is expressed as a global name. Moving the server within the same cell may likewise cause it to lose its key data file. You can guard against this by expressing the name in cell-global form, or by designing your key management module so that it can create a new key data file if necessary. For details on the server's key data file, see Section 2.3.

2.5.1.6 Robustness

Ensuring that a server comes up smoothly with all its resources, or that it at least comes up (if at all possible), is a part of designing robust server code. The ability of a server to struggle to activate itself even under adverse circumstances may be defined as the complement of graceful degradation.

For example, if a resource or service that a server depends on is not available, the server should create a thread that will wait and try to access the resource or service again later. In any case where it is possible that access could be delayed because of the vagaries of network performance, access should be retried for a reasonable amount of time before giving up and failing. On the other hand, if access is denied, for example, for security reasons, then it is appropriate to fail the operation with an informative error message.

Or suppose two servers that are supposed to establish contact with each other are unable immediately to do so. Here again, it is a good idea to wait a while and then try again. A consequence of proceeding this way is that clients may fail in the meantime, but this is far preferable to having the *servers* fail instead, which would probably require the intervention of a system administrator to recover from.

If contact cannot be established with the name service, binding alternatives should be provided. This can work for both clients and servers. Clients can prompt the user for a hostname, and then combine this with a partial string binding to try to complete the call. If all else fails, a client should try (or allow the user to try) binding to a local instance of the server.

The CDS entry **/.:/hosts/***hostname***/self** (where **/.:/** is the DCE notation for the local cell name and *hostname* is the valid name of a host machine) is defined to always contain binding information that will allow clients of any service running on the specified host to import a binding handle for use in operations on servers running at that host. Importing from this entry should always allow a client to bind to a local version of the server.

Similarly, servers should always try to access a local version of a database when they cannot make contact with the remote version.

2.5.2 Availability and Performance of Services

The following subsections deal with aspects of server operation that depend on the presence or performance of various DCE or other network services.

2.5.2.1 Coping with Inconsistent Binding Data

Because the DCE Directory Service is a partially replicated database, data received from it may not be consistent throughout the replicas. As a result, applications must be prepared to encounter out-of-date data and deal with it in a reasonable way.

To give an internal DCE example: When a DFS (Distributed File Service) client looks up a DFS mount point, the name service returns the address of the server providing access to that mount point. However, this address could be out of date. This will be true, for example, if the mount point has changed but the change has not yet been propagated to the replica being used by the client. As a result, the attempted connection will fail. The DFS client is prepared to recover from this situation gracefully, and application clients should behave similarly in analogous circumstances.

It should always be remembered that an imported binding is not guaranteed to work. Servers can suddenly become inactive or unavailable for various reasons, leaving stale exported bindings behind in the namespace. Clients should always be prepared to retry failed bindings or import another when the one just imported has failed to work.

2.5.2.2 Slow Name Service Response

Applications should expect that name service requests will sometimes take a long time to complete.

Most simple name (server entry) or attribute (group or profile) lookups will respond quickly. However, some interactions can take considerably longer, such as a lookup on a branch of the namespace that is physically distant and has not been recently accessed (and is therefore not cached), or a search operation. Applications that are not prepared to accept arbitrary delays in completing a request should either

- Bound the time permitted for the request, or
- Be prepared to abandon the request (after warning the user, if appropriate) if it takes longer than desired to complete.

2.5.3 Management

The following two subsections discuss some management aspects of server design.

2.5.3.1 Binding with the Management Interface

The DCE RPC management interface consists of the **rpc_mgmt_...()** calls, which allow clients to perform various operations on, and find out various things about servers. *All* servers automatically offer the management interface; the IDL compiler sees to this.

However, as a result of this universal availability, the same ''ambiguous call'' problem that was discussed earlier in this chapter occurs when a client makes a management call (for example, **rpc_mgmt_binding_is_server_listening()**) with a partial binding. Since *all* the servers on any target host presumably export the management interface, the endpoint mapper at that host has no way to select a particular server that does so.

The solution is for the client to use the the **rpc_ep_resolve_binding()** call, which takes a (typically partial) binding and an interface, and contacts the endpoint mapper on the remote host to find a server that offers that interface; the call then returns with a completed binding.

Suppose a client, having just imported a partial binding to print server A, wanted to make a management call to that server. The client would call **rpc_ep_resolve_binding()**, passing the partial binding it just imported; if successful, the call would return a full binding to print server ''A.''

The client could now use this completed binding handle to perform any management operation on print server ''A''; the binding would get the call to the server, and the management interface UUID would select the desired interface among those offered by the server.

For more information on the management interface itself, see Step B9 (Section 1.5.14) in Chapter 1 of this guide, and the *OSF DCE Application Development Reference*.

2.5.3.2 Shutdown Considerations

If a server is going down, it should unexport its entries and unregister its endpoints. A filled-up or somewhat-overwritten namespace will cause robust clients to take longer; nonrobust clients will fail.

Chapter 3

A Sample DCE Application

This chapter consists of an introduction to the commented client and server source code for **timop**, a sample DCE application. The source files themselves are located at *dce-root-dir*/**src/test/sample**. The chapter begins with a discussion of IDL (Interface Definition Language) and the interface definition process.

3.1 Developing a DCE Application

As was explained at the beginning of Chapter 1, the first step in coding a DCE application is to define one or more interfaces through which the application's clients and servers will communicate. Interfaces are defined in a declarative C-like Interface Definition Language and then compiled by the IDL compiler.

Interfaces, like most other objects and entities in DCE, are identified by the system by associating each one with a 128-bit Universal Unique Identifier (UUID). Generating a UUID for your application's interface is the very first step in the IDL process.

Executing the **uuidgen** command with the **-i** option, as follows:

```
uuidgen -i > pandaemonium.idl
```

will cause a skeleton **.idl** file to be generated, containing a new UUID and very little else; it is your task to add the rest.

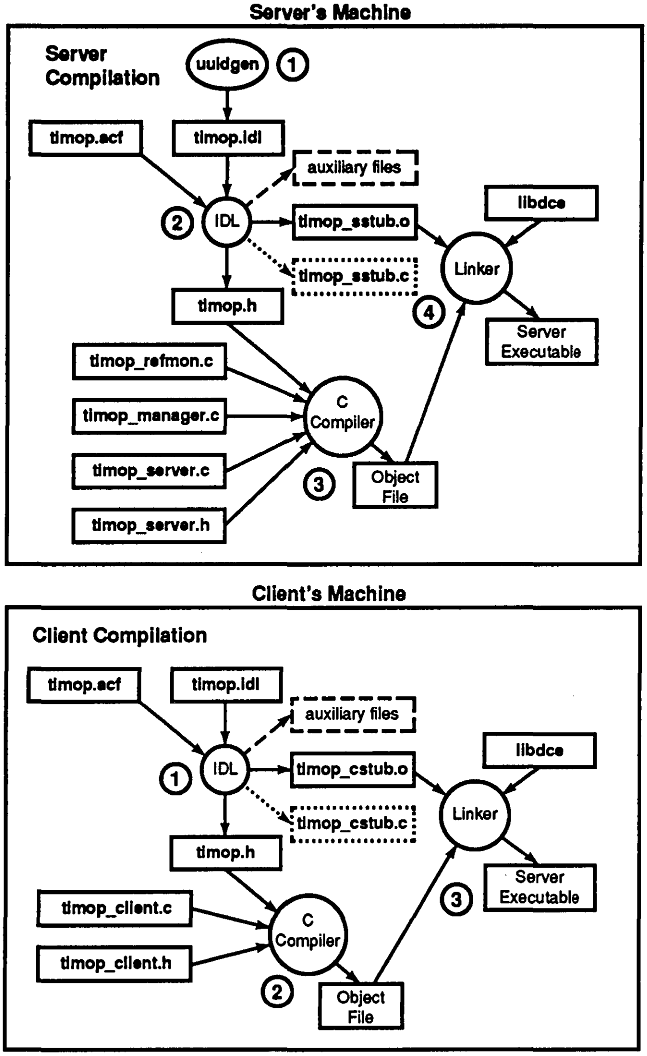
Thus, the development cycle for a DCE application is as follows:

1. Write and compile the **.idl** file.
2. Write and compile the server implementation code.
3. Write and compile the client implementation code.
4. Link the server object code with the server stub code and the DCE library.
5. Link the client object code with the client stub code and the DCE library.
6. Try running the compiled application.

Some of the steps may have to be executed repeatedly.

Figure 3-1 illustrates this process as it might be followed for both the server and the client modules of **timop**.

Figure 3–1. How an Executable DCE Application is Produced



Both the server and the client compilation phases are illustrated. As noted in the figure, these can occur on different machines. Note that the interface UUID is generated *only once*.

The circled numbers in boldface in the figure indicate the order of development steps, as follows:

1. Run **uuidgen** to get a skeleton **.idl** file containing a newly generated UUID. Complete the file with your interface operation definitions.
2. Compile the completed interface definition file with the IDL compiler.
3. Write the source code implementation of the interface operations in various **.c** and **.h** files, and compile them with the header file output by the IDL compiler.
4. Link the output of the previous step with the stub module produced by the IDL compiler, and the DCE library, **libdce**.

Of the server files shown in the figure, the application developer is responsible for writing the following:

timop.idl	A skeleton is generated by **uuidgen**
timop.acf	An optional file that affects interaction between the stub and code module
timop_manager.c	Implementation of interface operations
timop_server.c	Server setup and related routines
timop_refmon.c	Server reference monitor
timop_server.h	Server data declarations

Of the client files shown in the figure, the developer is responsible for writing (besides **timop.idl** and **timop.acf**, which are the same source files as were used for the server compilation) **timop_client.h**, **timop_client.c**, and finally a **timop_aux.h** auxiliary header file, which in **timop** is the same for both the server and the client.

An attribute configuration file (**timop.acf** in the figure) is usually optional; it contains input to the IDL compiler that alters the IDL output in various ways. Also optional are the auxiliary files (**aux**), which contain support routines for the client or server stub modules.

There is one other important option. The IDL compiler actually operates by first creating C source modules, and then invoking the C compiler to produce its object file output form the C source. Normally the C source files are then deleted. You can specify that the C source be kept, in which case the stub and auxiliary source files will appear as output too. This possibility is shown in dotted lines in the figure. Note that the IDL compiler's use of the C compiler is *not* shown in the figure.

A server can implement more than one interface. The interfaces would be defined in separate **.idl** files and compiled separately by the IDL compiler. The implemented interface operations in various source code files would then be linked with the IDL output.

3.1.1 The Purpose of Stub Files

The client and server stub files that are the output of the IDL compiler consist of RPC routines that handle all the mechanical details of packaging and unpackaging data into messages to be sent over the network, as well as the actual sending and receiving. All this is done in accordance with the specifications you made in the **.idl** and **.acf** input files. The **.idl** file specifications determine how the client/server interaction will occur over the network (the network protocol). The specifications in the **.acf** file, if the file exists, affect only the way the client's and/or server's application code interacts with what goes on in their respective stubs.

3.1.2 IDL Output Default Filenames

If the input **.idl** file to the IDL compiler is named **thorndyke.idl**, then the output files will have the following default names:

- Stub Files

  thorndyke_cstub.o and **thorndyke_cstub.c** for the client

  thorndyke_sstub.o and **thorndyke_sstub.c** for the server

- Header File

  thorndyke.h

- Auxiliary Files

  thorndyke_caux.o and **thorndyke_caux.c** for the client

  thorndyke_saux.o and **thorndyke_saux.c** for the server

It is usually a good idea to give the **.idl** files a name of the form *xxx*_**if.idl** (where the **if** signifies "interface file"), since the default name transformation in the IDL output can obscure the files' origin. That way you will always know that a file named *xyz*_**if.h** was generated from an **.idl** file.

3.2 A Complete Sample Application: timop

The **timop** program is a tutorial DCE application sample. It exercises the basic DCE technologies: Threads, RPC, Directory, Time, and Security. It is not intended to be a model of application techniques in general. A production application would probably feature such things as better fault management, the use of **getopt( )**, a Motif interface, internationalization, performance optimization, and so on; none of which are really important for this tutorial. The **timop** sample just tries to perform in a straightforward illustrative way, insofar as that is possible given the complexity of the technologies involved.

It is assumed that you have a DCE cell up and running. This means that your system must support thread-safe system interfaces (for example, for **printf( )**). You must also be registered as a DCE principal, or at least know the password of a principal in your cell, in order to do authenticated RPC; and you must be authorized to use certain of the cell's facilities (for example, to execute **rgy_edit** and place objects in the namespace).

3.2.1 What timop Does

The **timop** program has two parts, a client and a server, which are implemented by the **timop_client** and **timop_server** processes, respectively.

The server offers just one remote operation: clients can learn the span of time it takes the server to calculate the factorial of a random number specified by the client. The client spawns a number of threads, each of which makes parallel remote calls of this operation to designated servers. The client then prints out the name, invocation order, and time spans reported by the servers, and the random numbers it asked the servers to take the factorial of; it also prints out a total time span that encompasses all the job events at the servers and the sum of the random numbers.

The program uses only UDP (User Datagram Protocol) as a least common denominator transport provider. Authentication and integrity-secure RPC are used to make sure the communicated data is correct, and a small degree of authorization (named-based, not ACLs) is used as well. The Directory Service is used to identify the servers and to mediate the RPC binding between client and servers.

All time calculations are done in UTC with TDF = 0 (the Z (Zulu) or UTC reference time zone, corresponding to and generally equivalent to the classical UT GMT time zone), not local civil time, because the client(s) and server(s) may be in different time zones. Note that the server and client clocks are all different physical clocks, but they are all commensurable with one another because they are synchronized by DTS.

3.2.2 The timop Program and Security

Since **timop** uses the Security Service, the **timop** clients and servers must run as security principals. But in accordance with the tutorial goals of this example, only a minimal usage is made of security. With the code as supplied, **timop_client** is run as a principal named **/.../mycell/tclient**, and **timop_server** is run as a principal named **/.../mycell/tserver**. These names should be changed to suit your environment by modifying **timop_aux.h** (for example, both **tclient** and **tserver** could be the person executing the program).

The default login contexts used are **tclient** and **tserver**. In other words, when you execute **timop_client** or **timop_server**, you must **dce_login** as the principal **tclient** or **tserver**, respectively, to run the client or server. We run **timop_server** with the key of **tserver**; you therefore need to install the key of **tserver** into the key file **/tmp/tkeyfile**, for example, which you should have exclusive read/write permission to. (See the comments in **timop_server.h** in Section 3.2.8.6 for instructions on how to do this.)

Note that only a simple form of authorization is used, based on principal names, *not* ACLs; it is the programmer's responsibility to implement an ACL manager and use ACL-based authorization. Default source code for ACL management is supplied with DCE, but to have used it in this example would have made the code much too unwieldy.

3.2.3 Source Files

The **timop** program is built from the following source files:

- **Makefile.timop**

  The makefile

- **timop.idl**

  The IDL file

- **timop.acf**

  The ACF file

- **timop_aux.h**

  The auxiliary header file

- **timop_client.h**

  The client header file

- **timop_client.c**

  The client program

- **timop_server.h**

  The server header file

- **timop_server.c**

  The server program

- **timop_manager.c**

  The manager routines

- **timop_refmon.c**

  The server reference monitor

These files are located at *dce-root-dir*/**src/test/sample**.

''Manager'' is generic RPC terminology for the part of the server that actually handles the remote operation(s). In the usual practice, as illustrated here, **server.c** contains the nonapplication-specific routines that start up and initialize the server, and **manager.c** contains the application-specific routines that (among other things) implement the remote operations offered.

3.2.4 Building timop

To build **timop**, change the contents of **Makefile.timop** to suit your environment, then issue the following command:

```
make -f Makefile.timop
```

You will have to do this separately for every machine architecture you want to use.

3.2.5 Running timop

To run **timop**, install **timop_client** and **timop_server** on the machines you want to use, and issue commands something like the following, using names chosen to suit your environment.

On one machine, enter:

```
timop_server /.:/foo
```

where **/.:/foo** is the name in the namespace you want this server to have. You should do this either in the background (&), or on another terminal, or in another window.

Wait until you get the message:

```
Server /.:/foo ready.
```

then enter:

```
timop_client /.:/foo
```

This will print out results continuously.

On multiple machines in the same cell, enter:

```
timop_server /.:/foo         # on machine A
timop_server /.:/bar         # on machine B
timop_server /.:/zot         # on machine C
```

```
timop_client /.:/foo /.:/bar /.:/zot      # on machine D
timop_client /.:/zot /.:/bar /.:/foo      # on machine E
```

Note however that if the machines are *not* in the same cell, you must use fully qualified names beginning with **/...**, *not* beginning with **/.:** as in the example.

3.2.6 Stopping timop

You must kill clients and servers by hand, either by using the interrupt key or with a combination of the **ps** and **kill** commands. This will leave server binding information in the endpoint map and namespace, which is normal for persistent servers. The information can always be removed by hand later on with the **cdscp** and **rpccp** system administration commands, if necessary.

3.2.7 Further Exercises

After getting **timop** running, it would be a good exercise for you to figure out how it all works by modifying the code in various ways. In the process of doing this you can start to write your own applications. Some suggestions (other than the improved error-checking procedures, and so on, that were previously mentioned) are offered as follows:

- Get **timop** running over some transports other than UDP.
- Intentionally introduce some threads race conditions in order to experiment with the meaning of reentrancy. You can also fix the **asctime**( ) bug that was intentionally left in the code.
- Parallelize the client in a different way, perhaps by using **pthread_exit**( ) and **pthread_join**( ) instead of **pthread_cond_signal**( ) and **pthread_cond_wait**( ).
- Receive just one reply from one server, canceling the other outstanding jobs when the first reply arrives.
- Handle server returns from within the listen loop. Doing this means you will have to clean the server binding information from the endpoint map and namespace. You may want to experiment with the

pthread_signal_to_cancel_np( ) library routine and the exception handling interface (the **TRY**, **FINALLY**, and **ENDTRY** constructs). For more information, see Chapter 7 of this guide.

- Create a namespace service group, instead of a collection of individually named server instances.
- Create Version 1.1 of **timop** to contain an additional operation consisting of an additive version of the multiplicative factorial operation (*n* += *i* instead of *n* *= *i*).
- Use context handles and some DTS primitives to return per-client cumulative job times.
- Create a server that supports two managers, each offering a separate implementation of the factorial operation: one implementation remains the same as in the present version, while the new one (accessed by a different object UUID) computes the factorial in decreasing order.
- Working with some other users, make the clients and servers run under several principal identities. An even better way of doing this would be to have your security administrator create some extra identities for you to experiment with. (These extra identities would also be useful in the following exercise.)
- Implement an ACL manager for the **timop** service, add ACL entries for several principals and groups, and test the ACL manager by running the clients under various principal identities.
- Replace the no-op factorial operation with some operation or operations that would be really useful in your environment. This is the first step in creating your own DCE application.

3.2.8 The timop Program: A Sample DCE Application

The following subsections present the source code for **timop**.

3.2.8.1 The timop.idl Source File

Following are the IDL specifications for **timop**, contained in **timop.idl**:

```
/*
**      timop.idl
**
**      IDL interface specification for remote time operations.
*/

/* We need explicit handles in timop because our client has multiple
   (actually, multi-threaded) RPCs bound to multiple explicitly-specified
   servers. */

[uuid(0cf616d8-b858-11c9-8078-02608c0a03a7),
    version(1.0)]
interface timop
{
        /* DTS timestamps are already in a universal format,
           so are opaque to (the presentation layer of) the RPC
           (16 = sizeof(utc_t)). */
        const small             SIZEOF_TIMESTAMP = 16;
        typedef byte            timestamp_t[SIZEOF_TIMESTAMP];

        /* Failure value for remote status indications. */
        const long              TIMOP_ERR = -1;

        /* Get the time span to do a job (random factorial). */
        [idempotent]
        void timop_getspan(
                [in]    handle_t                handle,
                [in]    long                    rand,
                [out]   timestamp_t             timestamp,
                [out]   long                    *status_p,
                [in,out] error_status_t         *remote_status_p);
}
```

3.2.8.2 The timop.acf Source File

Following are the attribute configuration specifications for **timop**, contained in **timop.acf**:

```
/*
**      timop.acf
**
**      Attribute configuration file for timop interface.
*/

/* Do all marshalling out-of-line. */
[out_of_line]
interface timop
{
        /* Declare remote_status_p to be a comm_status and
           fault_status parameter. */
        timop_getspan(
                [comm_status,fault_status]      remote_status_p);
}
```

3.2.8.3 The timop_aux.h Source File

Following is the auxiliary information for **timop**, contained in **timop_aux.h**:

```
/*
**      timop_aux.h
**
**      Auxiliary info for timop example.
**      There are other ways to do these things, but we're just
**      illustrating the basics here.
*/

/* Principal names for this sample application.
   Change them to suit your environment. */
#define CLIENT_PRINC_NAME       (unsigned_char_t *)"/.../mycell/tclient"
#define SERVER_PRINC_NAME       (unsigned_char_t *)"/.../mycell/tserver"
```

```
/* Well-known object uuid for this sample application. */
#define OBJ_UUID (unsigned_char_t *)"2541af56-43a2-11ca-a9f5-02608c0ffe49"
```

3.2.8.4 The timop_client.h Source File

Following are the contents of **timop**'s client header file, **timop_client.h**:

```
/*
**      timop_client.h
**
**      Client header file for timop interface.
*/

#define MAX_SERVERS  10            /* single-digit server_num's, 0...9 */
#define CLIENT_NUM   -1            /* not equal to any server_num */
#define MAX_RANDOM  (10*1000*1000) /* big, to observe threads in action */
#define DO_WORK_OK    0            /* pass */
#define DO_WORK_ERR   1            /* fail */

/* Package up do_work() args in a struct, because
   pthreads start routines take only one argument. */
typedef struct work_arg {
        int                    server_num;   /* as ordered in arg list */
        unsigned_char_t        *server_name; /* as named in arg list */
        rpc_binding_handle_t   bind_handle;  /* binding handle to server */
        idl_long_int           rand;         /* input to the rpc call */
        int                    status;       /* returned from do_work() */
} work_arg_t;

/* Prototypes for client. */
int main(int _1, char *_2[]);
void do_work(work_arg_t *_1);
void print_report(unsigned_char_t *_1, int _2, utc_t *_3, long _4);
```

3.2.8.5 The timop_client.c Source File

Following is the source code for the **timop** client application, contained in **timop_client.c**:

```
/*
**      timop_client.c
**
**      Client program for timop interface.
*/

#include <errno.h>
#include <stdio.h>
#include <dce/rpc.h>
#include <pthread.h>
#include <time.h>
#include <dce/utc.h>
#include "timop.h"
#include "timop_aux.h"
#include "timop_client.h"

long                 Rand;          /* sum of random numbers */
int                  Workers;       /* number of active worker threads */
pthread_mutex_t      Work_mutex;    /* guard access to Workers, Rand */
pthread_cond_t       Work_cond;     /* condition variable for Workers==0 */


/*
 *      main()
 *
 *      Get started, and main loop.
 */

int
main(
        int                             argc,
        char                            *argv[])
{
        int                             server_num, nservers, ret;
        work_arg_t                      work_arg[MAX_SERVERS];
        unsigned_char_t                 *server_name[MAX_SERVERS],
                                  *string_binding, *protseq;
rpc_binding_handle_t            bind_handle[MAX_SERVERS];
unsigned32                      status;
utc_t                           start_utc, stop_utc, span_utc;
struct tm                       time_tm;
uuid_t                          obj_uuid;
rpc_ns_handle_t                 import_context;
pthread_t                       thread_id[MAX_SERVERS];


/* Check usage and initialize. */
if (argc < 2 || (nservers = argc-1) > MAX_SERVERS) {
        fprintf(stderr,
            "Usage: %s server_name ...(up to %d server_name’s)...\n",
            argv[0], MAX_SERVERS);
        exit(1);
}
for (server_num = 0; server_num < nservers; server_num += 1) {
        server_name[server_num] = (unsigned_char_t *)argv[1+server_num];
}

/* Initialize object uuid. */
uuid_from_string(OBJ_UUID, &obj_uuid, &status);
if (status != uuid_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}

/* Import binding info from namespace. */
for (server_num = 0; server_num < nservers; server_num += 1) {
        /* Begin the binding import loop. */
        rpc_ns_binding_import_begin(rpc_c_ns_syntax_dce,
            server_name[server_num], timop_v1_0_c_ifspec,
            &obj_uuid, &import_context, &status);
        if (status != rpc_s_ok) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }
        /* Import bindings one at a time. */
        while (1) {
                rpc_ns_binding_import_next(import_context,
                        &bind_handle[server_num], &status);
                    if (status != rpc_s_ok) {
                            fprintf(stderr, "FAULT: %s:%d\n", __FILE__,
                                __LINE__);
                            exit(1);
                    }
                    /* Select, say, the first binding over UDP. */
                    rpc_binding_to_string_binding(bind_handle[server_num],
                        &string_binding, &status);
                    if (status != rpc_s_ok) {
                            fprintf(stderr, "FAULT: %s:%d\n", __FILE__,
                                __LINE__);
                            exit(1);
                    }
                    rpc_string_binding_parse(string_binding, NULL,
                        &protseq, NULL, NULL, NULL, &status);
                    if (status != rpc_s_ok) {
                            fprintf(stderr, "FAULT: %s:%d\n", __FILE__,
                                __LINE__);
                            exit(1);
                    }
                    rpc_string_free(&string_binding, &status);
                    ret = strcmp(protseq, "ncadg_ip_udp");
                   rpc_string_free(&protseq, &status);
                    if (ret == 0) {
                           break;
                    }
            }
            /* End the binding import loop. */
            rpc_ns_binding_import_done(&import_context, &status);
            if (status != rpc_s_ok) {
                    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                    exit(1);
            }
    }

    /* Annotate binding handles for security. */
    for (server_num = 0; server_num < nservers; server_num += 1) {
            rpc_binding_set_auth_info(bind_handle[server_num],
                SERVER_PRINC_NAME, rpc_c_protect_level_pkt_integ,
                rpc_c_authn_dce_secret, NULL /*default login context*/,
                    rpc_c_authz_name, &status);
                if (status != rpc_s_ok) {
                        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                        exit(1);
                }
        }

        /* Initialize mutex and condition variable. */
        ret = pthread_mutex_init(&Work_mutex, pthread_mutexattr_default);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }
        ret = pthread_cond_init(&Work_cond, pthread_condattr_default);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }

        /* Initialize random number generator. */
        srandom(time(NULL));

        /* Initialize work args that are constant throughout main loop. */
        for (server_num = 0; server_num < nservers; server_num += 1) {
                work_arg[server_num].server_num = server_num;
                work_arg[server_num].server_name = server_name[server_num];
                work_arg[server_num].bind_handle = bind_handle[server_num];
        }

        /* Print out the year and date, just once. */
        ret = utc_gettime(&start_utc);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }
        ret = utc_gmtime(&time_tm, NULL, NULL, NULL, &start_utc);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }
        fprintf(stdout, "\n%24.24s UTC (Z time zone)\n\n", asctime(&time_tm));
```

```
/* Main loop -- never exits -- interrupt to quit. */
while (1) {
        /* Per-loop initialization.  We're single-threaded here, so
           locks and reentrant random number generator unnecessary. */
        Rand = 0;
        Workers = nservers;
        for (server_num = 0; server_num < nservers; server_num += 1) {
                work_arg[server_num].rand = random()%MAX_RANDOM;
        }

        /* Get client's start timestamp. */
        ret = utc_gettime(&start_utc);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }

        /* Spawn a worker thread for each server. */
        for (server_num = 0; server_num < nservers; server_num += 1) {
                ret = pthread_create(&thread_id[server_num],
                    pthread_attr_default, (void *)do_work,
                    (void *)&work_arg[server_num]);
                if (ret == -1) {
                        fprintf(stderr, "FAULT: %s:%d\n", __FILE__,
                            __LINE__);
                        exit(1);
                }
        }

        /* Reap the worker threads; pthread_cond_wait() semantics
           requires it to be coded this way. */
        ret = pthread_mutex_lock(&Work_mutex);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }
        while (Workers != 0) {
                ret = pthread_cond_wait(&Work_cond, &Work_mutex);
                if (ret == -1) {
                        fprintf(stderr, "FAULT: %s:%d\n", __FILE__,
                            __LINE__);
                exit(1);
        }
}
ret = pthread_mutex_unlock(&Work_mutex);
if (ret == -1) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}
/* Reclaim storage. */
for (server_num = 0; server_num < nservers; server_num += 1) {
        ret = pthread_detach(&thread_id[server_num]);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__,
                    __LINE__);
                exit(1);
        }
}

/* Any failures? */
for (server_num = 0; server_num < nservers; server_num += 1) {
        if (work_arg[server_num].status != DO_WORK_OK) {
                exit(1);
        }
}

/* Get client's stop timestamp. */
ret = utc_gettime(&stop_utc);
if (ret == -1) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}

/* Calculate the span of client's start and stop timestamps. */
ret = utc_spantime(&span_utc, &start_utc, &stop_utc);
if (ret == -1) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}

/* Print total results. */
print_report((unsigned_char_t *)"(client)", CLIENT_NUM,
                        &span_utc, Rand);
        }

        /* Not reached. */
}
/*
 *      do_work()
 *
 *      Do the work.  This is done in parallel threads, so we want it
 *      (and the subroutine print_report() it calls) to be reentrant.
 */

void
do_work(
        work_arg_t                      *work_arg_p)
{
        int                             server_num, *status_p, ret;
        unsigned_char_t                 *server_name;
        rpc_binding_handle_t            bind_handle;
        idl_long_int                    rand, status;
        error_status_t                  remote_status = rpc_s_ok;
        timestamp_t                     timestamp;


        /* Unpackage the args into local variables. */
        server_num = work_arg_p->server_num;
        server_name = work_arg_p->server_name;
        bind_handle = work_arg_p->bind_handle;
        rand = work_arg_p->rand;
        status_p = &work_arg_p->status;

        /* Do the RPC! */
        timop_getspan(bind_handle, rand, timestamp, &status, &remote_status);
        if (remote_status != rpc_s_ok) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                *status_p = DO_WORK_ERR;
                pthread_exit(NULL);
                /* Not reached. */
        }
        if (status != rand) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                *status_p = DO_WORK_ERR;
                pthread_exit(NULL);
                /* Not reached. */
        }

        /* Print report.  Not a critical section here because print_report()
           is supposed to be implemented to be reentrant. */
        print_report(server_name, server_num, (utc_t *)timestamp, rand);

        /* Update Rand and decrement Workers.  As implemented, it is a
           critical section, so must be locked. */
        ret = pthread_mutex_lock(&Work_mutex);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }
        Workers -= 1;
        if (Workers == 0) {
                /* Last worker signals main thread. */
                ret = pthread_cond_signal(&Work_cond);
                if (ret == -1) {
                        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                        exit(1);
                }
        }
        Rand += rand;
        ret = pthread_mutex_unlock(&Work_mutex);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }

        /* Done. */
        *status_p = DO_WORK_OK;
        pthread_exit(NULL);
        /* Not reached. */
}
/*
 *      print_report()
 *
 *      Print DTS timestamp interval, to millisecond granularity.
 *      Implemented this way so it is reentrant (assuming all the underlying
 *      OS subroutines it calls are reentrant).
 *      This kind of timestamp manipulation is always messy -- see the
 *      manual for the formats of structures and print-strings we use.
 */

void
print_report(
        unsigned_char_t         *server_name,
        int                     server_num,
        utc_t                   *utc_p,
        long                    rand)
{
#define LINE_LEN                78
#define COL1                    0
#define COL2                    44
#define COL3a                   47
#define COL3b                   60
#define COL4                    70
        char                    asctime_buf[26], ascinacc_buf[26],
                                    time_ns_buf[10], inacc_ns_buf[10],
                                    report[LINE_LEN+3];
        int                     inacc_sec, ret;
        long                    time_ns, inacc_ns;
        struct tm               time_tm, inacc_tm;


        /* Print server_name into report.  Pad or truncate as necessary. */
        sprintf(report+COL1, "%*.*s  ", COL2-2, COL2-2, (char *)server_name);

        /* Print server_num into report. */
        if (server_num != CLIENT_NUM) {
                sprintf(report+COL2, "%1.1d  ", server_num);
        } else {
                sprintf(report+COL2, "%1.1s  ", "*");
        }

        /* Format utc_p and print it into report. */
        ret = utc_gmtime(&time_tm, &time_ns, &inacc_tm, &inacc_ns, utc_p);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }
        memcpy(asctime_buf, asctime(&time_tm), 26);   /* reentrancy bug! */
        memcpy(ascinacc_buf, asctime(&inacc_tm), 26); /* reentrancy bug! */
        sprintf(time_ns_buf, "%9.9d", time_ns);
        sprintf(inacc_ns_buf, "%9.9d", inacc_ns);
        inacc_sec = inacc_tm.tm_yday*24*60*60 + inacc_tm.tm_hour*60*60 +
            inacc_tm.tm_min*60 + inacc_tm.tm_sec;
        sprintf(report+COL3a, "%8.8s.%3.3sI", asctime_buf+11,
            time_ns_buf);
        if (inacc_tm.tm_year != -1) {
                sprintf(report+COL3b, "%4.4d.%3.3s  ", inacc_sec,
                    inacc_ns_buf);
        } else {
                sprintf(report+COL3b, "%8.8s  ", "infinity");
        }

        /* Print rand into report. */
        if (server_num != CLIENT_NUM) {
                sprintf(report+COL4, "%8d\n", rand);
        } else {
                sprintf(report+COL4, "%8d\n\n", rand);
        }

        /* Output report. */
        fprintf(stdout, "%s", report);
        return;
}
```

3.2.8.6 The timop_server.h Source File

Following are the contents of the **timop** server's header file, **timop_server.h**:

```
/*
**      timop_server.h
**
```

```
**      Server header file for timop interface.
*/

#define NUM_OBJS                1       /* num of objs supported */
#define MAX_CONC_CALLS_PROTSEQ  5       /* max conc calls per protseq */
#define MAX_CONC_CALLS_TOTAL    10      /* max conc calls total */

/* Success/failure for remote procedures. */
#define GETSPAN_OK              0       /* pass */
#define GETSPAN_ERR             1       /* fail */

/* Defines for access control. */
#define GETSPAN_OP              1       /* requested operation */
#define GRANT_ACCESS            0       /* reference monitor success */
#define DENY_ACCESS             1       /* reference monitor failure */
#define IS_AUTHORIZED           0       /* authorization success */
#define  NOT_AUTHORIZED          1       /* authorization failure */

/* Server key table for this example.  Change name of keyfile to suit your
   environment, and populate it with "rgy_edit ktadd tserver /tmp/tkeyfile". */
#define KEYFILE                 "/tmp/tkeyfile"
#define KEYTAB                  "FILE:" ## KEYFILE

/* Prototypes for server. */
int main(int _1, char *_2[]);
void getspan_ep(rpc_binding_handle_t _1, idl_long_int _2, timestamp_t _3,
        idl_long_int *_4, error_status_t *_5);
int do_getspan(idl_long_int _1, timestamp_t _2);
int ref_mon(rpc_binding_handle_t _1, int _2);
int is_authorized(unsigned_char_t *_1, int _2);
```

3.2.8.7 The timop_server.c Source File

Following is the **timop** server application setup source code, contained in **timop_server.c**:

```
/*
**      timop_server.c
**
```

```
**       Server program for timop interface.
*/

#include <stdio.h>
#include <dce/rpc.h>
#include "timop.h"
#include "timop_aux.h"
#include "timop_server.h"

/* Declare manager EPV.  This EPV could be bulk-initialized here,
   but we do prefer to do it one operation at a time in main(). */
timop_v1_0_epv_t                manager_epv;


/*
 *       main()
 *
 *       Get started -- set up server the way we want it, and call listen loop.
 */

int
main(
        int                             argc,
        char                            *argv[])
{
        unsigned_char_t                 *server_name;
        rpc_binding_vector_t            *bind_vector_p;
        unsigned32                      status;
        int                             i;
        uuid_t                          type_uuid, obj_uuid;
        struct {
            unsigned32  count;
            uuid_t      *uuid[NUM_OBJS];
        }                               obj_uuid_vec = {NUM_OBJS, {&obj_uuid}};


        /* Check usage and initialize. */
        if (argc != 2) {
                fprintf(stderr, "Usage: %s namespace_server_name\n", argv[0]);
                exit(1);
        }
```

```
server_name = (unsigned_char_t *)argv[1];

/* Initialize manager EPV (just one entry point in this example). */
manager_epv.timop_getspan = getspan_ep;

/* Initialize object uuid (just one in this example). */
uuid_from_string(OBJ_UUID, &obj_uuid, &status);
if (status != uuid_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}

/* Initialize type uuid (just one in this example). */
uuid_create(&type_uuid, &status);
if (status != uuid_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}

/* Register object/type uuid associations with rpc runtime. */
rpc_object_set_type(&obj_uuid, &type_uuid, &status);
if (status != rpc_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}

/* Register interface/type_uuid/epv associations with rpc runtime. */
rpc_server_register_if(timop_v1_0_s_ifspec, &type_uuid,
    (rpc_mgr_epv_t)&manager_epv, &status);
if (status != rpc_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}

/* Tell rpc runtime we want to use all supported protocol sequences. */
rpc_server_use_all_protseqs(MAX_CONC_CALLS_PROTSEQ, &status);
if (status != rpc_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}
```

```
/* Ask the runtime which binding handle(s) it's going to let us use. */
rpc_server_inq_bindings(&bind_vector_p, &status);
if (status != rpc_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}

/* Register authentication info with rpc runtime. */
rpc_server_register_auth_info(SERVER_PRINC_NAME,
    rpc_c_authn_dce_secret, NULL /*default key retrieval function*/,
    KEYTAB /*server key table for this example*/, &status);
if (status != rpc_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}

/* Establish server's login context(s), if necessary.
   In this example we just use the default login context,
   so we do NOTHING here. */

/* Decide what to do upon server termination.  It would be prudent
   to handle signals and decide what to do if the listen loop returns
   (e.g., clean exported info out of endpoint map and namespace,
   something that is not usually done for a persistent server),
   but since this is just an example we don't do those things here. */

/* Register binding info with endpoint map. */
rpc_ep_register(timop_v1_0_s_ifspec, bind_vector_p,
    (uuid_vector_t *)&obj_uuid_vec,
    (unsigned_char_t *)"timop server, version 1.0", &status);
if (status != rpc_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}

/* Export binding info to the namespace. */
rpc_ns_binding_export(rpc_c_ns_syntax_dce, server_name,
    timop_v1_0_s_ifspec, bind_vector_p,
    (uuid_vector_t *)&obj_uuid_vec, &status);
if (status != rpc_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
```

```
                exit(1);
        }

        /* Listen for service requests (semi-infinite loop). */
        fprintf(stdout, "Server %s ready.\n", server_name);
        rpc_server_listen(MAX_CONC_CALLS_TOTAL, &status);
        if (status != rpc_s_ok) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                exit(1);
        }

        /* Returned from listen loop.  We haven't arranged for this. */
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        exit(1);
}
```

3.2.8.8 The timop_manager.c Source File

Following is the **timop** server application-specific source code, contained in **timop_manager.c**:

```
/*
**      timop_manager.c
**
**      Manager routines for timop interface.
*/

#include <stdio.h>
#include <dce/utc.h>
#include "timop.h"
#include "timop_aux.h"
#include "timop_server.h"


/*
 *      getspan_ep()
 *
 *      Entry point for timop_getspan() operation.
 *      Note it is reentrant, so we can have a multi-threaded server.
 */

void
getspan_ep(
        rpc_binding_handle_t            bind_handle,
        idl_long_int                    rand,
        timestamp_t                     timestamp,
        idl_long_int                    *status_p,
        error_status_t                  *remote_status_p)
{
        int                             ret;


        /* Call reference monitor, to make authorization decision. */
        ret = ref_mon(bind_handle, GETSPAN_OP);
        if (ret == DENY_ACCESS) {
                *status_p = TIMOP_ERR;
                return;
        }

        /* Service the request, i.e., do the actual remote procedure. */
        ret = do_getspan(rand, timestamp);
        if (ret == GETSPAN_ERR) {
                *status_p = TIMOP_ERR;
                return;
        }

        /* Return the input random number as a status value (!= TIMOP_ERR). */
        *status_p = rand;
        /* Return all results to client, and resume listen loop. */
        return;
}
/*
 *      do_getspan()
 *
 *      Do the actual remote procedure.
 */

int
do_getspan(
        idl_long_int                    rand,
        timestamp_t                     timestamp)
{
        long                            i;
        volatile long                   n;
        int                             ret;
        utc_t                           start_utc, stop_utc;


        /* Get server's start timestamp. */
        ret = utc_gettime(&start_utc);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                return(GETSPAN_ERR);
        }

        /* Do service (here a random factorial, but could be anything). */
        for (n = i = 1; i <= rand; i += 1) {
                n *= i;         /* Burn cpu -- use your imagination. */
        }

        /* Get server's stop timestamp. */
        ret = utc_gettime(&stop_utc);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                return(GETSPAN_ERR);
        }

        /* Calculate the span of server's start and stop timestamps. */
        ret = utc_spantime((utc_t *)timestamp, &start_utc, &stop_utc);
        if (ret == -1) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                return(GETSPAN_ERR);
        }

        /* Success. */
        return(GETSPAN_OK);
}
```

3.2.8.9 The timop_refmon.c Source Files

Following is the **timop** server application reference monitor source code, contained in **timop_refmon.c**:

```
/*
**      timop_refmon.c
**
**      Reference monitor for timop example.
*/

#include <stdio.h>
#include "timop_aux.h"
#include "timop.h"
#include "timop_server.h"


/*
 *      ref_mon()
 *
 *      Reference monitor for timop.
 *      It checks generalities, then calls is_authorized() to check specifics.
 */

int
ref_mon(
        rpc_binding_handle_t            bind_handle,
        int                             requested_op)
{
        int                             ret;
        rpc_authz_handle_t              privs;
        unsigned_char_t                 *client_princ_name, *server_princ_name;
        unsigned32                      protect_level, authn_svc, authz_svc,
                                            status;


        /* Get client auth info. */
        rpc_binding_inq_auth_client(bind_handle, &privs, &server_princ_name,
            &protect_level, &authn_svc, &authz_svc, &status);
        if (status != rpc_s_ok) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                return(DENY_ACCESS);
        }

        /* Check if selected authn service is acceptable to us. */
        if (authn_svc != rpc_c_authn_dce_secret) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                return(DENY_ACCESS);
        }

        /* Check if selected protection level is acceptable to us. */
        if (protect_level != rpc_c_protect_level_pkt_integ
        &&  protect_level != rpc_c_protect_level_pkt_privacy) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                return(DENY_ACCESS);
        }

        /* Check if selected authz service is acceptable to us. */
        if (authz_svc != rpc_c_authz_name) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                return(DENY_ACCESS);
        }
        /* If rpc_c_authz_dce were being used instead of rpc_c_authz_name,
           privs would be a PAC (sec_id_pac_t *), not a name as it is here. */
        client_princ_name = (unsigned_char_t *)privs;

        /* Check if selected server principal name is supported. */
        if (strcmp(server_princ_name, SERVER_PRINC_NAME) != 0) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                return(DENY_ACCESS);
        }

        /* Now that things seem generally OK, check the specifics. */
        ret = is_authorized(client_princ_name, requested_op);
        if (ret == NOT_AUTHORIZED) {
                fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
                return(DENY_ACCESS);
        }

        /* Cleared all the authorization hurdles -- grant access. */
        return(GRANT_ACCESS);
}
```

```
/*
 *       is_authorized()
 *
 *       Check authorization of client to the requested service.
 *       This could be arbitrarily application-specific, but we keep it simple.
 *       A normal application (i.e., one using PACs & ACLs) would be using
 *       sec_acl_mgr_is_authorized() instead of this function.
 */

int
is_authorized(
        unsigned_char_t                  *client_princ_name,
        int                               requested_op)
{
        /* Check if we want to let this client do this operation. */
        if (strcmp(client_princ_name, CLIENT_PRINC_NAME) == 0
        &&  requested_op == GETSPAN_OP) {
                /* OK, we'll let this access happen. */
                return(IS_AUTHORIZED);
        }

        /* Sorry, Charlie. */
        return(NOT_AUTHORIZED);
}
```

Part 2

DCE Threads

Chapter 4

Introduction to Multithreaded Programming

DCE Threads is a user-level (nonkernel) threads package based on the pthreads interface specified by POSIX in 1003.4a, Draft 4. This chapter introduces multithreaded programming, which is the division of a program into multiple threads (parts) that execute concurrently. In addition, this chapter describes four software models that improve multithreaded programming performance.

A thread is a single sequential flow of control within a program. It is the active execution of a designated routine, including any nested routine invocations. Within a single thread, there is a single point of execution. Most traditional programs consist of a single thread.

Threads are lightweight processes that share a single address space. Each thread shares all the resources of the originating process, including signal handlers and descriptors. Each thread has its own thread identifier, scheduling policy and priority, **errno** value, thread-specific data bindings, and the required system resources to support a flow of control.

4.1 Advantages of Using Threads

With a threads package, a programmer can create several threads within a process. Threads execute concurrently, and within a multithreaded process, there are at any time multiple points of execution. Threads execute within a single address space. Multithreaded programming offers the following advantages:

- Performance

  Threads improve the performance (throughput, computational speed, responsiveness, or some combination of these) of a program. Multiple threads are useful in a multiprocessor system where threads run concurrently on separate processors. In addition, multiple threads also improve program performance on single processor systems by permitting the overlap of input and output or other slow operations with computational operations.

  You can think of threads as executing simultaneously, regardless of the number of processors present. You cannot make any assumptions about the start or finish times of threads or the sequence in which they execute, unless explicitly synchronized.

- Shared Resources

  An advantage of using multiple threads over using separate processes is that the former share a single address space, all open files, and other resources.

- Potential Simplicity

  Multiple threads can reduce the complexity of some applications that are inherently suited for threads.

4.2 Software Models for Multithreaded Programming

The following subsections describe four software models for which multithreaded programming is especially well suited:

- Boss/worker model
- Work crew model
- Pipelining model
- Combinations of models

4.2.1 Boss/Worker Model

In a boss/worker model of program design, one thread functions as the boss because it assigns tasks to worker threads. Each worker performs a different type of task until it is finished, at which point the worker interrupts the boss to indicate that it is ready to receive another task. Alternatively, the boss polls workers periodically to see whether or not each worker is ready to receive another task.

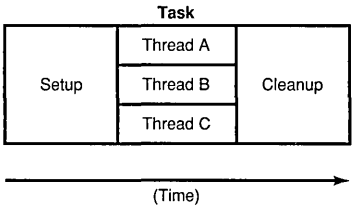
A variation of the boss/worker model is the work queue model. The boss places tasks in a queue, and workers check the queue and take tasks to perform. An example of the work queue model in an office environment is a secretarial typing pool. The office manager puts documents to be typed in a basket, and typists take documents from the basket to work on.

4.2.2 Work Crew Model

In the work crew model, multiple threads work together on a single task. The task is divided into pieces that are performed in parallel, and each thread performs one piece. An example of a work crew is a group of people cleaning a house. Each person cleans certain rooms or performs certain types of work (washing floors, polishing furniture, and so forth), and each works independently.

Figure 4-1 shows a task performed by three threads in a work crew model.

Figure 4–1. Work Crew Model



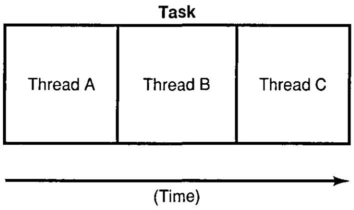
4.2.3 Pipelining Model

In the pipelining model, a task is divided into steps. The steps must be performed in sequence to produce a single instance of the desired output, and the work done in each step (except for the first and last) is based on the preceding step and is a prerequisite for the work in the next step. However, the program is designed to produce multiple instances of the desired output, and the steps are designed to operate in a parallel time frame so that each step is kept busy.

An example of the pipelining model is an automobile assembly line. Each step or stage in the assembly line is continually busy receiving the product of the previous stage's work, performing its assigned work, and passing the product along to the next stage. A car needs a body before it can be painted, but at any one time numerous cars are receiving bodies, and then numerous cars are being painted.

In a multithreaded program using the pipelining model, each thread represents a step in the task. Figure 4-2 shows a task performed by three threads in a pipelining model.

Figure 4–2. Pipelining Model



4.2.4 Combinations of Models

You may find it appropriate to combine the software models in a single program if your task is complex. For example, a program could be designed using the pipelining model, but one or more steps could be handled by a work crew. In addition, tasks could be assigned to a work crew by taking a task from a work queue and deciding (based on the task characteristics) which threads are needed for the work crew.

4.3 Potential Disadvantages with Multithreaded Programming

When you design and code a multithreaded program, consider the following problems and accommodate or eliminate each problem as appropriate:

- Potential Complexity

  The level of expertise required for designing, coding, and maintaining multithreaded programs may be higher than for most single-threaded programs because multithreaded programs may need shared access to resources, mutexes, and condition variables. Weigh the potential benefits against the complexity and its associated risks.

- Nonreentrant Software

  If a thread calls a routine or library that is not reentrant, use the global locking mechanism to prevent the nonreentrant routines from modifying a variable that another thread modifies. Section 6.4 discusses nonreentrant software in more detail.

  Note: A multithreaded program must be reentrant; that is, it must allow multiple threads to execute at the same time. Therefore, be sure that your compiler generates reentrant code before you do any design or coding work for multithreading. (Many C, Ada, Pascal, and BLISS compilers generate reentrant code by default.)

  If your program is nonreentrant, any thread synchronization techniques that you use are not guaranteed to be effective.

- Priority Inversion

  Priority inversion prevents high-priority threads from executing when interdependencies exist among three or more threads. Section 6.5 discusses priority inversion in more detail.

- Race Conditions

  A type of programming error called a ''race condition'' causes unpredictable and erroneous program behavior. Section 6.6.1 discusses race conditions in more detail.

- Deadlocks

  A type of programming error called a ''deadlock'' causes two or more threads to be blocked from executing. Section 6.6.2 discusses deadlocks in more detail.

- Blocking Calls

  Certain system or library calls may cause an entire process to block while waiting for the call to complete, thus causing all other threads to stop executing. Section 6.1.2 discusses blocking in more detail.

Chapter 5

Thread Concepts and Operations

This chapter discusses concepts and techniques related to DCE Threads. The following topics are covered:

- Thread operations
- Attributes objects
- Synchronization objects
- One-time initialization code
- Thread-specific data
- Thread cancellation
- Thread scheduling

For detailed information on the multithreading routines referred to in this chapter, see the reference page for that routine in the *OSF DCE Application Development Reference*.

5.1 Thread Operations

A thread changes states as it runs, waits to synchronize, or is ready to be run. A thread is in one of the following states:

- Waiting

  The thread is not eligible to execute because it is synchronizing with another thread or with an external event.

- Ready

  The thread is eligible to be executed by a processor.

- Running

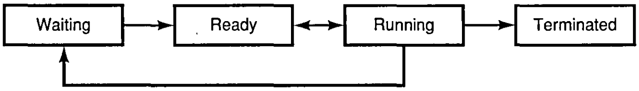
  The thread is currently being executed by a processor.

- Terminated

  The thread has completed all of its work.

Figure 5-1 shows the transitions between states for a typical thread implementation.

Figure 5–1. Thread State Transitions



The operations that you can perform include starting, waiting for, terminating, and deleting threads.

5.1.1 Starting a Thread

To start a thread, create it using the **pthread_create**( ) routine. This routine creates the thread, assigns specified or default attributes, and starts execution of the function you specified as the thread's start routine. A unique identifier (handle) for that thread is returned from the **pthread_create**( ) routine.

5.1.2 Terminating a Thread

A thread exists until it terminates and the **pthread_detach( )** routine is called for the thread. The **pthread_detach( )** routine can be called for a thread before or after it terminates. If the thread terminates before **pthread_detach( )** is called for it, then the thread continues to exist and can be synchronized (joined) until it is detached. Thus, the object (thread) can be detached by any thread that has access to a handle to the object.

Note that **pthread_detach( )** must be called to release the memory allocated for the thread objects so that this storage does not build up and cause the process to run out of memory. For example, after a thread returns from a call to join, it detaches the joined-to thread if no other threads join with it. Similarly, if a thread has no other threads joining with it, it detaches itself so that its thread object is deallocated as soon as it terminates.

A thread terminates for any of the following reasons:

- The thread returns from its start routine; this is the usual case.
- The thread calls the **pthread_exit( )** routine.

  The **pthread_exit( )** routine terminates the calling thread and returns a status value, indicating the thread's exit status to any potential joiners.
- The thread is canceled by a call to the **pthread_cancel( )** routine.

  The **pthread_cancel( )** routine requests termination of a specified thread if cancellation is permitted. (See Section 5.6 for more information on canceling threads and controlling whether or not cancellation is permitted.)
- An error occurs in the thread.

5.1.3 Waiting for a Thread to Terminate

A thread waits for the termination of another thread by calling the **pthread_join( )** routine. Execution in the current thread is suspended until the specified thread terminates. If multiple threads call this routine and specify the same thread, all threads resume execution when the specified thread terminates.

If you specify the current thread with the **pthread_join()** routine, a deadlock results. (See Section 6.6.2 for more information.)

Do not confuse **pthread_join()** with other routines that cause waits and that are related to the use of a particular multithreading feature. For example, use **pthread_cond_wait()** or **pthread_cond_timedwait()** to wait for a condition variable to be signaled or broadcast (see Section 5.3.2 for information about condition variables).

5.1.4 Deleting a Thread

A thread is automatically deleted after it terminates; that is, no explicit deletion operation is required. Use **pthread_detach()** to free the storage of a terminated thread. Use **pthread_cancel()** to request that a running thread terminate itself.

If the thread has not yet terminated, the **pthread_detach()** routine marks the thread for deletion, and its storage is reclaimed immediately when the thread terminates. A thread cannot be joined or canceled after the **pthread_detach()** routine is called for the thread, even if the thread has not yet terminated.

If a thread that is not detached terminates, its storage remains so that other threads can join with it. Storage is reclaimed when the thread is eventually detached.

For more information, see Section 5.1.2.

5.2 Attributes Objects

An attributes object is used to describe the behavior of threads, mutexes, and condition variables. This description consists of the individual attribute values that are used to create an attributes object. Whether an attribute is valid depends on whether it describes threads, mutexes, or condition variables.

When you create an object, you can accept the default attributes for that object, or you can specify an attributes object that contains individual attributes that you have set. For a thread, you can also change one or more attributes after thread execution starts; for example, calling the **pthread_setprio( )** routine to change the priority that you specified with the **pthread_attr_setprio( )** routine.

The following subsections describe how to create and delete attributes objects and describe the individual attributes that you can specify for different objects.

5.2.1 Creating an Attributes Object

To create an attributes object, use one of the following routines, depending on the type of object to which the attributes apply:

- The **pthread_attr_create( )** routine for thread attributes objects
- The **pthread_condattr_create( )** routine for condition variable attributes objects
- The **pthread_mutexattr_create( )** routine for mutex attributes objects

These routines create an attributes object containing default values for the individual attributes. To modify any attribute values in an attributes object, use one of the set routines described in the following subsections.

Creating an attributes object or changing the values in an attributes object does not affect the attributes of objects previously created.

5.2.2 Deleting an Attributes Object

To delete an attributes object, use one of the following routines:

- The **pthread_attr_delete( )** routine for thread attributes objects
- The **pthread_condattr_delete( )** routine for condition variable attributes objects
- The **pthread_mutexattr_delete( )** routine for mutex attributes objects

Deleting an attributes object does not affect the attributes of objects previously created.

5.2.3 Thread Attributes

A thread attributes object allows you to specify values for thread attributes other than the defaults when you create a thread with the **pthread_create()** routine. To use a thread attributes object, perform the following steps:

1. Create a thread attributes object by calling the **pthread_attr_create()** routine.
2. Call the routines discussed in the following subsections to set the individual attributes of the thread attributes object.
3. Create a new thread by calling the **pthread_create()** routine and specifying the identifier of the thread attributes object.

You have control over the following attributes of a new thread:

- Scheduling policy attribute
- Scheduling priority attribute
- Inherit scheduling attribute
- Stacksize attribute

5.2.3.1 Scheduling Policy Attribute

The scheduling policy attribute describes the overall scheduling policy of the threads in your application. A thread has one of the following scheduling policies:

- **SCHED_FIFO** (First In, First Out)

  The highest-priority thread runs until it blocks. If there is more than one thread with the same priority, and that priority is the highest among other threads, the first thread to begin running continues until it blocks.

- **SCHED_RR** (Round Robin)

  The highest-priority thread runs until it blocks; however, threads of equal priority, if that priority is the highest among other threads, are timesliced. (Timeslicing is a mechanism that ensures that every thread is allowed time to execute by preempting running threads at fixed intervals.)

- **SCHED_OTHER, SCHED_FG_NP** (Default)

  All threads are timesliced. **SCHED_OTHER** and **SCHED_FG_NP** do the same thing; however, **SCHED_FG_NP** is simply more precise terminology. The **FG** stands for foreground and the **NP** stands for nonportable. All threads running under the **SCHED_OTHER** and **SCHED_FG_NP** policy, regardless of priority, receive some scheduling. Therefore, no thread is completely denied execution time. However, **SCHED_OTHER** and **SCHED_FG_NP** threads can be denied execution time by **SCHED_FIFO** or **SCHED_RR** threads.

  Routines implemented by DCE Threads that are not specified by Draft 4 of the POSIX 1003.4a standard are indicated by an **_np** suffix on the name. These routines are not portable.

- **SCHED_BG_NP** (Background)

  Like **SCHED_OTHER** and **SCHED_FG_NP**, **SCHED_BG_NP** ensures that all threads, regardless of priority, receive some scheduling. However, **SCHED_BG_NP** can be denied execution by the **SCHED_FIFO** or **SCHED_RR** policies. The **BG** stands for background and the **NP** stands for nonportable.

The following two methods are used to set the scheduling policy attribute:

- Set the scheduling policy attribute in the attributes object, which establishes the scheduling policy of a new thread when it is created. To do this, call the **pthread_attr_setsched( )** routine.

- Change the scheduling policy of an existing thread (and at the same time, the scheduling priority) by calling the **pthread_setscheduler( )** routine.

Section 5.7 describes and shows the effect of scheduling policy on thread scheduling.

5.2.3.2 Scheduling Priority Attribute

The scheduling priority attribute specifies the execution of a thread. This attribute is expressed relative to other threads on a continuum of minimum to maximum for each scheduling policy.

A thread's priority falls within one of the following ranges, which are implementation defined:

- **PRI_FIFO_MIN** to **PRI_FIFO_MAX**
- **PRI_RR_MIN** to **PRI_RR_MAX**
- **PRI_OTHER_MIN** to **PRI_OTHER_MAX**
- **PRI_FG_MIN_NP** to **PRI_FG_MAX_NP**
- **PRI_BG_MIN_NP** to **PRI_BG_MAX_NP**

Section 5.7 describes how to specify priorities between the minimum and maximum values, and it also discusses how priority affects thread scheduling.

The following two methods are used to set the scheduling priority attribute:

- Set the scheduling priority attribute in the attributes object, which establishes the execution priority of a new thread when it is created. To do this, call the **pthread_attr_setprio( )** routine.
- Change the scheduling priority attribute of an existing thread by calling the **pthread_setprio( )** routine. (Call the **pthread_setscheduler( )** routine to change both the scheduling priority and scheduling policy of an existing thread.)

5.2.3.3 Inherit Scheduling Attribute

The inherit scheduling attribute specifies whether a newly created thread inherits the scheduling attributes (scheduling priority and policy) of the creating thread (the default), or uses the scheduling attributes stored in the attributes object. Set this attribute by calling the **pthread_attr_setinheritsched( )** routine.

5.2.3.4 Stacksize Attribute

The stacksize attribute is the minimum size (in bytes) of the memory required for a thread's stack. The default value is machine dependent. Set this attribute by calling the **pthread_attr_setstacksize**( ) routine.

5.2.4 Mutex Attributes

A mutex attributes object allows you to specify values for mutex attributes other than the defaults when you create a mutex with the **pthread_mutex_init**( ) routine.

The mutex type attribute specifies whether a mutex is fast, recursive, or nonrecursive. (See Section 5.3.1 for definitions.) Set the mutex type attribute by calling the **pthread_mutexattr_setkind_np**( ) routine. (Any routine with the **_np** suffix is nonportable.) If you do not use a mutex attributes object to select a mutex type, calling the **pthread_mutex_init**( ) routine creates a fast mutex by default.

5.2.5 Condition Variable Attributes

Currently, attributes affecting condition variables are not defined. You cannot change any attributes in the condition variable attributes object.

Section 5.3.2 describes the purpose and uses of condition variables.

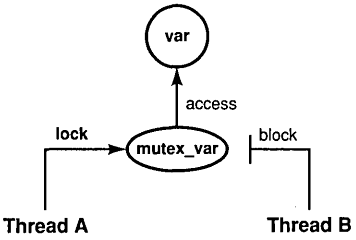
5.3 Synchronization Objects

In a multithreaded program, you must use synchronization objects whenever there is a possibility of corruption of shared data or conflicting scheduling of threads that have mutual scheduling dependencies. The following subsections discuss two kinds of synchronization objects: mutexes and condition variables.

5.3.1 Mutexes

A mutex (*mut*ual *ex*clusion) is an object that multiple threads use to ensure the integrity of a shared resource that they access, most commonly shared data. A mutex has two states: locked and unlocked. For each piece of shared data, all threads accessing that data must use the same mutex; each thread locks the mutex before it accesses the shared data and unlocks the mutex when it is finished accessing that data. If the mutex is locked by another thread, the thread requesting the lock is blocked when it tries to lock the mutex if you call **pthread_mutex_lock( )** (see Figure 5-2). The blocked thread continues and is not blocked if you call **pthread_mutex_trylock( )**.

Figure 5–2. Only One Thread Can Lock a Mutex



Each mutex must be initialized. (To initialize mutexes as part of the program's one-time initialization code, see Section 5.4.) To initialize a mutex, use the **pthread_mutex_init( )** routine. This routine allows you to specify an attributes object, which allows you to specify the mutex type. The following are types of mutexes:

- A fast mutex (the default) is locked only once by a thread. If the thread tries to lock the mutex again without first unlocking it, the thread waits for itself to release the first lock and deadlocks on itself.

  This type of mutex is called ''fast'' because it can be locked and unlocked more rapidly than a recursive mutex. It is the most efficient form of mutex.

- A recursive mutex can be locked more than once by a given thread without causing a deadlock. The thread must call the **pthread_mutex_unlock( )** routine the same number of times that it called the **pthread_mutex_lock( )** routine before another thread can lock the mutex. Recursive mutexes have the notion of a mutex owner. When a thread successfully locks a recursive mutex, it owns that mutex and the lock count is set to 1. Any other thread attempting to lock the mutex blocks until the mutex becomes unlocked. If the owner of the mutex attempts to lock the mutex again, the lock count is incremented, and the thread continues running. When an owner unlocks a recursive mutex, the lock count is decremented. The mutex remains locked and owned until the count reaches 0 (zero). It is an error for any thread other than the owner to attempt to unlock the mutex.

A recursive mutex is useful if a thread needs exclusive access to a piece of data, and it needs to call another routine (or itself) that needs exclusive access to the data. A recursive mutex allows nested attempts to lock the mutex to succeed rather than deadlock.

This type of mutex requires more careful programming. Never use a recursive mutex with condition variables because the implicit unlock performed for a **pthread_cond_wait( )** or **pthread_cond_timedwait( )** may not actually release the mutex. In that case, no other thread can satisfy the condition of the predicate.

- A nonrecursive mutex is locked only once by a thread, like a fast mutex. If the thread tries to lock the mutex again without first unlocking it, the thread receives an error. Thus, nonrecursive mutexes are more informative than fast mutexes because fast mutexes block in such a case, leaving it up to you to determine why the thread no longer executes. Also, if someone other than the owner tries to unlock a nonrecursive mutex, an error is returned.

To lock a mutex, use one of the following routines, depending on what you want to happen if the mutex is locked:

- The **pthread_mutex_lock( )** routine

  If the mutex is locked, the thread waits for the mutex to become available.

- The **pthread_mutex_trylock( )** routine

  If the mutex is locked, the thread continues without waiting for the mutex to become available. The thread immediately checks the return status to see if the lock was successful, and then takes whatever action is appropriate if it was not.

When a thread is finished accessing a piece of shared data, it unlocks the associated mutex by calling the **pthread_mutex_unlock( )** routine.

If another thread is waiting on the mutex, its execution is unblocked. If more than one thread is waiting on the mutex, the scheduling policy (for more information, see Section 5.2.3.1) and the thread scheduling priority (for more information, see Section 5.2.3.2) determine which thread acquires the mutex.

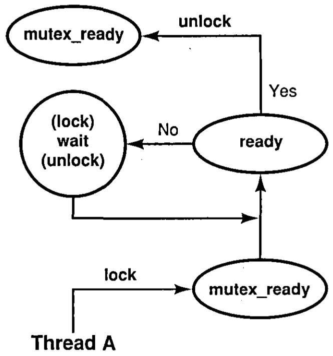
You can delete a mutex and reclaim its storage by calling the **pthread_mutex_destroy( )** routine. Use this routine only after the mutex is no longer needed by any thread. Mutexes are automatically deleted when the program terminates.

5.3.2 Condition Variables

A condition variable allows a thread to block its own execution until some shared data reaches a particular state. Cooperating threads check the shared data and wait on the condition variable. For example, one thread in a program produces work-to-do packets and another thread consumes these packets (does the work). If the work queue is empty when the consumer thread checks it, that thread waits on a work-to-do condition variable. When the producer thread puts a packet on the queue, it signals the work-to-do condition variable.

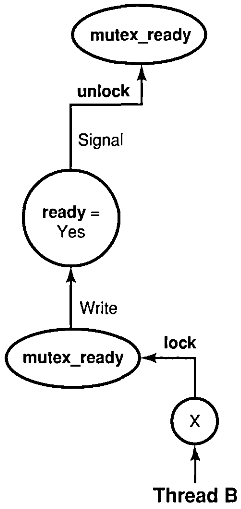
A condition variable is used to wait for a shared resource to assume some specific state (a predicate). A mutex, on the other hand, is used to reserve some shared resource while the resource is being manipulated. For example, a thread A may need to wait for a thread B to finish a task X before thread A proceeds to execute a task Y. Thread B can tell thread A that it has finished task X by using a variable they both have access to, a condition variable. When thread A is ready to execute task Y, it looks at the condition variable to see if thread B is finished (see Figure 5-3).

Figure 5–3. Thread A Waits on Condition Ready, Then Wakes Up and Proceeds



First, thread A locks the mutex named **mutex_ready** that is associated with the condition variable. Then it reads the predicate associated with the condition variable named **ready**. If the predicate indicates that thread B has finished task X, then thread A can unlock the mutex and proceed with task Y. If the condition variable predicate indicated that thread B has not yet finished task X, however, then thread A waits for the condition variable to change. Thread A calls the **wait** primitive. Waiting on the condition variable automatically unlocks the mutex, allowing thread B to lock the mutex when it has finished task X (see Figure 5-4).

Figure 5–4. Thread B Signals Condition Ready



Thread B updates the predicate named **ready** associated with the condition variable to the state thread A is waiting for. It also executes a signal on the condition variable while holding the mutex **mutex_ready**. Thread A wakes up, verifies that the condition variable is in the correct state, and proceeds to execute task Y (see Figure 5-3).

Note that although the condition variable is used for explicit communications among threads, the communications are anonymous. Thread B does not necessarily know that thread A is waiting on the condition variable that thread B signals. And thread A does not know that it was thread B that woke it up from its wait on the condition variable.

Use the **pthread_cond_init**( ) routine to create a condition variable. To create condition variables as part of the program's one-time initialization code, see Section 5.4.

Use the **pthread_cond_wait**( ) routine to cause a thread to wait until the condition is signaled or broadcast. This routine specifies a condition variable and a mutex that you have locked. (If you have not locked the mutex, the results of **pthread_cond_wait**( ) are unpredictable.) This routine unlocks the mutex and causes the calling thread to wait on the condition variable until another thread calls one of the following routines:

- The **pthread_cond_signal**( ) routine to wake one thread that is waiting on the condition variable
- The **pthread_cond_broadcast**( ) routine to wake all threads that are waiting on a condition variable

If you want to limit the time that a thread waits for a condition to be signaled or broadcast, use the **pthread_cond_timedwait**( ) routine. This routine specifies the condition variable, mutex, and absolute time at which the wait should expire if the condition variable is not signaled or broadcast.

You can delete a condition variable and reclaim its storage by calling the **pthread_cond_destroy**( ) routine. Use this routine only after the condition variable is no longer needed by any thread. Condition variables are automatically deleted when the program terminates.

5.3.3 Other Synchronization Methods

There is another synchronization method that is not anonymous: the **join** primitive. This allows a thread to wait for another specific thread to complete its execution. When the second thread is finished, the first thread unblocks and continues its execution. Unlike mutexes and condition variables, the **join** primitive is not associated with any particular shared data.

5.4 One-Time Initialization Routines

You probably have one or more routines that must be executed *before* any thread executes code in your application, but must be executed *only once* regardless of the sequence in which threads start executing. For example, you may want to create mutexes and condition variables (each of which must be created only once) in an initialization routine. Multiple threads can call the **pthread_once**( ) routine, or the **pthread_once**( ) routine can be called multiple times in the same thread, resulting in only one call to the specified routine.

Use the **pthread_once**( ) routine to ensure that your application initialization routine is executed only a single time; that is, by the first thread that tries to initialize the application. This routine is the only way to guarantee that one-time initialization is performed in a multithreaded environment on a given platform. The **pthread_once**( ) routine is of particular use for runtime libraries, which are often called for the first time after multiple threads are created.

5.5 Thread-Specific Data

The thread-specific data interfaces allow each thread to associate an arbitrary value with a shared key value created by the program.

Thread-specific data is like a global variable in which each thread can keep its own value, but is accessible to the thread anywhere in the program.

Use the following routines to create and access thread-specific data:

- The **pthread_keycreate**( ) routine to create a unique key value
- The **pthread_setspecific**( ) routine to associate data with a key
- The **pthread_getspecific**( ) routine to obtain the data associated with a key

The **pthread_keycreate**( ) routine generates a unique key value that is shared by all threads in the process. This key is the identifier of a piece of thread-specific data. Each thread uses the same key value to assign or retrieve a thread-specific value. This keeps your data separate from other thread-specific data. One call to the **pthread_keycreate**( ) routine creates a cell in all threads. Call this routine to specify a routine to be called to destroy the context value associated with this key when the thread terminates.

The **pthread_setspecific**( ) routine associates the address of some data with a specific key. Multiple threads associate different data (by specifying different addresses) with the same key. For example, each thread points to a different block of dynamically allocated memory that it has reserved.

The **pthread_getspecific**( ) routine obtains the address of the thread-specific data value associated with a specified key. Use this routine to locate the data associated with the current thread's context.

5.6 Thread Cancellation

Canceling is a mechanism by which one thread terminates another thread (or itself). When you request that a thread be canceled, you are requesting that it terminate as soon as possible. However, the target thread can control how quickly it terminates by controlling its general cancelability and its asynchronous cancelability.

The following is a list of the pthread calls that are cancellation points:

- The **pthread_setasynccancel( )** routine
- The **pthread_testcancel( )** routine
- The **pthread_delay_np( )** routine
- The **pthread_join( )** routine
- The **pthread_cond_wait( )** routine
- The **pthread_cond_timedwait( )** routine

General cancelability is enabled by default. A thread is canceled only at specific places in the program; for example, when a call to the **pthread_cond_wait( )** routine is made. If general cancelability is enabled, request the delivery of any pending cancel request by using the **pthread_testcancel( )** routine. This routine allows you to permit cancellation to occur at places where it may not otherwise be permitted under general cancelability, and it is especially useful within very long loops to ensure that cancel requests are noticed within a reasonable time.

If you disable general cancelability, the thread cannot be terminated by any cancel request. Disabling general cancelability means that a thread could wait indefinitely if it does not come to a normal conclusion; therefore, be careful about disabling general cancelability.

Asynchronous cancelability, when it is enabled, allows cancels to be delivered to the enabling thread at any time, not only at those times that are permitted when just general cancelability is enabled. Thus, use asynchronous cancellation primarily during long processes that do not have specific places for cancel requests. Asynchronous cancelability is disabled by default. Disable asynchronous cancelability when calling threads routines or any other runtime library routines that are not explicitly documented as cancel-safe.

Note: If general cancelability is disabled, the thread cannot be canceled, regardless of whether asynchronous cancelability is enabled or disabled. The setting of asynchronous cancelability is relevant only when general cancelability is enabled.

Use the following routines to control the canceling of threads:

- The **pthread_setcancel( )** routine to enable and disable general cancelability
- The **pthread_testcancel( )** routine to request delivery of a pending cancel to the current thread
- The **pthread_setasynccancel( )** routine to enable and disable asynchronous cancelability
- The **pthread_cancel( )** routine to request that a thread be canceled

5.7 Thread Scheduling

Threads are scheduled according to their scheduling priority and how the scheduling policy treats those priorities. To understand the discussion in this section, you must understand the concepts in the following sections of this chapter:

- The ``Scheduling Policy Attribute'' section (5.2.3.1) discusses scheduling policies, including how each policy handles thread scheduling priority.
- The ``Scheduling Priority Attribute'' section (5.2.3.2) discusses thread scheduling priorities.
- The ``Inherit Scheduling Attribute'' section (5.2.3.3) discusses inheritance of scheduling attributes by created threads.

To specify the minimum or maximum priority, use the appropriate symbol; for example, **PRI_OTHER_MIN** or **PRI_OTHER_MAX**. To specify a value between the minimum and maximum priority, use an appropriate arithmetic expression. For example, to specify a priority midway between the minimum and maximum for the default scheduling policy, specify the following concept using your programming language's syntax:

```
pri_other_mid = (PRI_OTHER_MIN + PRI_OTHER_MAX)/2
```

If your expression results in a value outside the range of minimum to maximum, an error results when you use it. Priority values are integers.

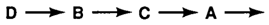
To show results of the different scheduling policies, consider the following example: a program has four threads, called threads A, B, C, and D. For each scheduling policy, three scheduling priorities have been defined: minimum, middle, and maximum. The threads have the priorities shown in the following table:

Thread	Priority
A	Minimum
B	Middle
C	Middle
D	Maximum

Figures 5-5 through 5-7 show execution flows, depending on whether the threads use the **SCHED_FIFO**, **SCHED_RR**, or **SCHED_OTHER** (default) scheduling policy. Assume that all waiting threads are ready to execute when the current thread waits or terminates and that no higher priority thread is awakened while a thread is executing (during the flow shown in each figure).

Figure 5-5 shows a flow with **SCHED_FIFO** (First In, First Out) scheduling.

Figure 5–5. Flow with SCHED_FIFO Scheduling



All four threads are timesliced. Threads with higher priority are generally scheduled when more than one thread is ready to run; however, to ensure fairness, all threads are given some time. The effective priority of threads may be modified over time by the scheduler, depending on the use of processor resources.

Figure 5-6 shows a flow with **SCHED_RR** (Round Robin) scheduling.

Figure 5–6. Flow with SCHED_RR Scheduling

D → B → C → B → C → A →

Thread D executes until it waits or terminates; then threads B and C are timesliced because they both have middle priority; then thread A executes.

Figure 5-7 shows a flow with **SCHED_OTHER** (default) scheduling.

Figure 5–7. Flow with SCHED_OTHER Scheduling

D → B → C → A → B → C → . . .

Thread D executes until it waits or terminates; then threads B, C, and A are timesliced, even though thread A has a lower priority than the other two. Thread A receives less execution time than thread B or C if either is ready to execute as often as thread A is. However, the default scheduling policy protects thread A against being blocked from executing indefinitely.

Because low-priority threads eventually run, the default scheduling policy protects against the problem of priority inversion discussed in Section 6.5.

Chapter 6

Programming with Threads

This chapter discusses issues you face when writing a multithreaded program and how to deal with those issues.

The topics discussed in this chapter are as follows:

- Calling UNIX services
- Using signals
- Nonthreaded libraries
- Avoiding nonreentrant software
- Avoiding priority inversion
- Using synchronization objects
- Signaling a condition variable

6.1 Calling UNIX Services

On a UNIX system that does not have kernel support for threads, making system and library calls from within a multithreaded program raises the following issues:

- System calls may not be thread-reentrant.
- If a system call blocks, it blocks the entire process instead of blocking the calling thread only.

6.1.1 Jacket Routines

To resolve the previous two issues, DCE Threads provides jacket routines for a number of UNIX system calls. Threads call the jacket routine instead of the UNIX system service; this allows DCE Threads to take action on behalf of the thread before or after calling the system service. For example, the jacket routines ensure that only one thread calls any particular service at a time to avoid problems with system calls that are not thread-reentrant.

Jacket routines are provided for UNIX input and output system calls (documented in the *UNIX Programmer's Manual*) and the **fork**( ) and **sigaction**( ) system calls. Jackets are not provided for any other UNIX system calls or for any of the C runtime library services (documented in the *UNIX Programmer's Manual*). See **/usr/include/dce/cma_ux.h** for the full list of jacket routines.

6.1.1.1 Input and Output Jacket Routines

Jacket routines are provided for routines that perform input and output operations. Examples of these operations are as follows:

- Open or create files, pipe symbols, and sockets
- Send and receive messages on sockets
- Read and write files and pipe symbols

Jacket routines are provided for Input/Output services so that DCE Threads can determine when to issue or block the service call based on the results of the **select()** system call. For these UNIX services, DCE Threads can determine whether issuing the system call causes the process to block. If the system call causes the process to block, DCE Threads blocks only the calling thread and schedules another thread to run in its place.

Periodically, DCE Threads checks whether the original calling thread can issue its operation without blocking the process. When the thread runs without blocking the process, that thread is placed back into the queue of ready threads, and at its turn, the thread resumes execution and issues the system call. Therefore, the jacket routines provide thread-synchronous I/O operations where otherwise the system calls block the entire process.

6.1.1.2 The fork( ) Jacket Routine

Jackets are provided for the **fork()** system call. A specific thread environment must exist in the forked process when it resumes (begins) execution. These jacket routines allow code to be executed in the context of the new process before the user code resumes execution in it.

6.1.1.3 The atfork( ) Routine

The **atfork()** routine allows an application or library to ensure predicted behavior when the **fork()** routine is used in a multithreaded environment. Using the **fork()** routine from a threaded application or from an application that uses threaded libraries can result in unpredictable behavior. For example, one thread has a mutex locked, and the state covered by that mutex is inconsistent while another thread calls the **fork()** routine. In the child process, the mutex will be in the locked state, and it cannot be unlocked because only the forking thread exists in the child process. Having the child reinitialize the mutex is unsatisfactory because this approach does not resolve the question of how to correct the inconsistent state in the child.

The **atfork( )** routine provides a way for threaded applications or libraries to protect themselves when a **fork( )** occurs. The **atfork( )** routine allows you to set up routines that will run at the following times:

- Prior to the **fork( )** in the parent process
- After the **fork( )** in the child process
- After the **fork( )** in the parent process

Within these routines you can ensure that all mutexes are locked prior to the **fork( )** and that they are unlocked after the **fork( )**, thereby protecting any data or resources associated with the mutexes. You can register any number of sets of **atfork( )** routines; that is, any number of libraries or user programs can set up **atfork( )** routines and they will all execute at **fork( )** time.

Note: Using the **atfork( )** routine can potentially cause a deadlock if two applications or libraries call into one another using calls that require locking. Specifically, when these component's routines use the **atfork( )** routine to run prior to the fork in the parent process, a deadlock may occur when these routines are executing.

6.1.1.4 Using the Jacketed System Calls

You do not have to rename your system calls to take advantage of the jacket routines. Macros put the jacket routines into place when you compile your program; these macros rename the jacketed system calls to the name of the DCE Threads jacket routine. Thus, a reference to the DCE Threads jacket routine is compiled into your code instead of a reference to the system call. When the code is executed, it calls the jacket routine, which then calls the system on your code's behalf.

If you do not wish to use any of the jacket routines, you can add the following line to your program before any of the thread header files: `#define _CMA_NOWRAPPERS_`. By adding this definition, you prevent the jacket routines from being substituted for the real routines.

If you wish to use most of the jackets but do not wish to use a specific jacket, you can undefine a specific jacket by adding the following directive after the thread header files: `#undef <routine_name>`. For example, to not use the fork jacket, you can add: `#undef fork`.

6.1.2 Blocking System Calls

DCE Threads provides jacket routines that make certain system calls thread-synchronous. If calling one of these jacketed system calls would normally block the process, the jacket routine ensures that only the calling thread is blocked and that the process remains available to execute other threads. Examples of jacketed system calls include **read( )**, **write( )**, **open( )**, **socket( )**, **send( )**, and **recv( )**.

If a thread makes a call to any of the other nonjacketed blocking system calls (or if it calls one of the jacketed system calls without going through the jacket), then when the system call blocks the thread, it blocks the whole process, preventing any other threads in the process from executing. Examples of nonjacketed system calls include **wait( )**, **sigpause( )**, **msgsnd( )**, **msgrcv( )**, and **semop( )**.

Some care must be used when calling nonjacketed blocking system calls from a multithreaded program. Other threads in the program may not be able to tolerate not running for an extended period of time while the process blocks for the system call. If your program must make use of such system calls, the calling thread should specify a nonblocking or polling option to the system call. If the call is not successful, then the calling thread should retry; however, to prevent the retry code from becoming a hot loop, a yield or delay function call should be inserted into the path. This gives other threads in the program a chance to run between poll attempts.

6.1.3 Calling fork( ) in a Multithreaded Environment

The **fork( )** system call creates an exact duplicate of the address space from which it is called, resulting in two address spaces executing the same code. Problems can occur if the forking address space has multiple threads executing at the time of the **fork( )**. When multithreading is a result of library invocation, threads are not necessarily aware of each other's presence, purpose, actions, and so on. Suppose that one of the other threads (any thread other than the one doing the **fork( )**) has the job of deducting money from your checking account. Clearly, you do not want this to happen twice as a result of some other thread's decision to call **fork( )**.

Because of these types of problems, which in general are problems of threads modifying persistent state, POSIX defined the behavior of **fork( )** in the presence of threads to propagate only the forking thread. This solves the problem of improper changes being made to persistent state. However, it causes other problems, as discussed in the next paragraph.

In the POSIX model, only the forking thread is propagated. All the other threads are eliminated without any form of notice; no cancels are sent and no handlers are run. However, all the other portions of the address space are cloned, including all the mutex state. If the other thread has a mutex locked, the mutex will be locked in the child process, but the lock owner will not exist to unlock it. Therefore, the resource protected by the lock will be permanently unavailable.

The fact that there may be mutexes outstanding only becomes a problem if your code attempts to lock a mutex that could be locked by another thread at the time of the **fork( )**. This means that you cannot call outside of your own code between the call to **fork( )** and the call to **exec( )**. Note that a call to **malloc( )**, for example, is a call outside of the currently executing application program and may have a mutex outstanding. The following code obeys these guidelines and is therefore safe:

```
fork ();
a = 1+2;  /* some inline processing */
exec();
```

Similarly, if your code calls some of your own code that does not make any calls outside of your code and does not lock any mutexes that could possibly be locked in another thread, then your code is safe.

One solution to the problem of calling **fork()** in a multithreaded environment exists. (Note that this method will not work for server application code or any other application code that is invoked by a callback from a library.) Before an application performs a **fork( )** followed by something other than **exec( )**, it must cancel all of the other threads. After it joins the canceled threads, it can safely **fork( )** because it is the only thread in existence. This means that libraries that create threads must establish cancel handlers that propagate the cancel to the created threads and join them. The application should save enough state so that the threads can be recreated and restarted after the **fork( )** processing completes.

6.2 Using Signals

The following subsections cover three topics: types of signals, DCE Threads signal handling, and alternatives to using signals.

6.2.1 Types of Signals

Signals are delivered as a result of some event. UNIX signals are grouped into the following four categories of pairs that are orthogonal to each other:

- Terminating and synchronous
- Terminating and asynchronous
- Nonterminating and synchronous
- Nonterminating and asynchronous

The action that DCE Threads takes when a particular signal is delivered depends on the characteristics of that signal.

6.2.1.1 Terminating Signals

Terminating signals result in the termination of the process by default. Whether a particular signal is terminating or not is independent of whether it is synchronously or asynchronously delivered.

6.2.1.2 Nonterminating Signals

Nonterminating signals do not result in the termination of the process by default.

Nonterminating signals represent events that can be either internal or external to the process. The process may require notification or ignore these events. When a nonterminating asynchronous signal is delivered to the process, DCE Threads awakens any threads that are waiting for the signal. This is the only action that DCE Threads takes because, by default, the signal has no effect.

6.2.1.3 Synchronous Signals

Synchronous signals are the result of an event that occurs inside a process and are delivered synchronously with respect to that event. For example, if a floating-point calculation results in an overflow, then a **SIGFPE** (floating-point exception signal) is delivered to the process immediately following the instruction that resulted in the overflow.

Synchronous, terminating signals represent an error that has occurred in the currently executing thread.

6.2.1.4 Asynchronous Signals

Asynchronous signals are the result of an event that is external to the process and are delivered at any point in a thread's execution when such an event occurs. For example, when a user running a program types the interrupt character at the terminal (generally **<Ctrl-c>**), a **SIGINT** (interrupt signal) is delivered to the process.

Asynchronous, terminating signals represent an occurrence of an event that is external to the process, and if unhandled, results in the termination of the process. When an asynchronous terminating signal is delivered, DCE Threads catches it and checks to see if any threads are waiting for it. If threads are waiting, they are awakened, and the signal is considered handled and is dismissed. If there are no waiting threads, then DCE Threads causes the process to be terminated as if the signal had not been handled.

6.2.2 DCE Threads Signal Handling

DCE Threads provides the POSIX **sigwait()** service to allow threads to perform activities similar to signal handling without having to deal with signals directly. It also provides a jacket for **sigaction()** that allows each thread to have its own handler for synchronous signals.

In order to provide these mechanisms, DCE Threads installs signal handlers for most of the UNIX signals during initialization.

DCE Threads do not provide handlers for several UNIX signals. Those signals and a reason why handlers are not provided are listed in the following table.

Signal	Reason Handler Is Not Provided
SIGKILL and **SIGSTOP**	These signals cannot be caught by user mode code.
SIGTRAP	Catching this signal interferes with debugging.
SIGTSTP and **SIGQUIT**	These signals are caught only while a thread has issued a **sigwait()** call because their default actions are otherwise valuable.

6.2.2.1 The POSIX sigwait() Service

The DCE Threads implementation of the POSIX **sigwait()** service allows any thread to block until one of a specified set of signals is delivered. A thread waits for any of the asynchronous signals, except for **SIGKILL** and **SIGSTOP**.

A thread cannot wait for a synchronous signal. This is because synchronous signals are the result of an error during the execution of a thread, and if the thread is waiting for a signal, then it is not executing. Therefore, a synchronous signal cannot occur for a particular thread while it is waiting, and so the thread waits forever. POSIX stipulates that the thread must block the signals (using the UNIX system service **sigprocmask( )**) it waits for before calling **sigwait( )**.

6.2.2.2 The POSIX sigaction( ) Service

The DCE Threads implementation of the POSIX **sigaction( )** service allows for per-thread handlers to be installed for catching synchronous signals. The **sigaction( )** routine only modifies behavior for individual threads and only works for synchronous signals. Setting the signal action to **SIG_DFL** for a specific signal will restore the thread's default behavior for that signal. Attempting to set a signal action for an asynchronous signal is an error.

6.2.2.3 The itimer VTALARM

DCE Threads installs a handler for the **itimer VTALARM**. Therefore, **VTALARM** is unavailable for use by other applications.

6.2.3 Alternatives to Using Signals

Avoid using UNIX signals in multithreaded programs. DCE Threads provides alternatives to signal handling. These alternatives are discussed in more detail in Sections 6.6 and 6.7.

Note: In order to implement these alternatives, DCE Threads must install its own signal handlers. These are installed when DCE Threads initializes itself, typically on the first thread-function call. At this time, any existing signal handlers are replaced.

Following are several reasons for avoiding signals:

- They cannot be used in a modular way in a multithreaded program.
- They are unnecessary when used as an asynchronous programming technique in a multithreaded program.

- There are almost no threads services available at signal level.
- There is no reliable, portable way to modify predicates.
- The signal-handler interface is unsuitable for use with threads. (For example, there is one signal action per signal per process, there is one signal mask per process, and **sigpause( )** blocks the whole process.)

In a multithreaded program, signals cannot be used in a modular way because, on most current UNIX implementations, signals are inherently a process construct. There is only one instantiation of each signal and of each signal handler routine for all of the threads in an application. If one thread handles a particular signal in one way, and a different thread handles the same signal in a different way, then the thread that installs its signal handler last handles the signal. This applies only to asynchronously generated signals; synchronous signals can be handled on a per-thread basis using the DCE Threads **sigaction( )** jacket.

Do not use asynchronous programming techniques in conjunction with threads, particularly those that increase parallelism such as using timer signals and I/O signals. These techniques can be complicated. They are also unnecessary because threads provide a mechanism for parallel execution that is simpler and less prone to error where concurrence can be of value. Furthermore, most of the threads routines are not supported for use in interrupt routines (such as signal handlers), and portions of runtime libraries cannot be used reliably inside a signal handler.

6.3 Nonthreaded Libraries

As programming with threads becomes common practice, you need to ensure that threaded code and nonthreaded code (code that is not designed to work with threads) work properly together in the same application. For example, you may write a new application that uses threads (for example, an RPC server), and link it with a library that does not use threads (and is thus not thread-safe). In such a situation you can do one of the following:

- Work with the nonthreaded software.
- Change the nonthreaded software to be thread-safe.

6.3.1 Working with Nonthreaded Software

Thread-safe code is code that works properly in a threaded environment. To work with nonthread-safe code, associate the global lock with all calls to such code.

You can implement the lock on the side of the routine user or the routine provider. For example, you can implement the lock on the side of the routine user if you write a new application like an RPC server that uses threads, and you link it with a library that does not. Or, if you have access to the nonthreaded code, the locks can be placed on the side of the routine provider, within the actual routine. Implement the locks as follows:

1. Associate one lock, a global lock, with execution of such code.
2. Require all of your threads to lock prior to execution of nonthreaded code.
3. Perform an unlock when execution is complete.

By using the global lock you ensure that only one thread executes in outside libraries, which may call each other, and in unknown code. Using a single global lock is safer than using multiple local locks because it is difficult to be aware of everything a library may be doing or of the interactions that library can have with other libraries.

6.3.2 Changing Nonthreaded Code to be Thread-Reentrant

Thread-reentrant code is code that works properly while multiple threads execute it concurrently. Thread-reentrant code is thread-safe, but thread-safe code may not be thread-reentrant. Document your code as being thread-safe or thread-reentrant.

More work is involved in making code thread-reentrant than in making code thread-safe. To make code thread-reentrant, do the following:

1. Use proper locking protocols to access global or static variables.
2. Use proper locking protocols when you use code that is not thread-safe.
3. Store thread-specific data on the stack or heap.

4. Ensure that the compiler produces thread-reentrant code.
5. Document your code as being thread-reentrant.

6.4 Avoiding Nonreentrant Software

The following subsections discuss two methods to help you avoid the pitfalls of nonreentrant software. These methods are as follows:

- Global lock
- Thread-specific storage

6.4.1 Global Lock

Use a global lock, which has the characteristics of a recursive mutex, instead of a regular mutex when calling routines that you think are nonreentrant. (When in doubt, assume the code is nonreentrant.)

The **pthread_lock_global_np**( ) routine is a locking protocol that is used to call nonreentrant routines, often found in existing library packages that were not designed to run in a multithreaded environment.

The way to call a library function that is not reentrant from a multithreaded program is to protect the function with a mutex. If every function that calls a library locks a particular mutex before the call and releases the mutex after the call, then the function completes without interference. However, this is difficult to do successfully because the function may be called by many libraries. A global lock solves this problem by providing a universal lock. Any code that calls any nonreentrant function uses the same lock.

To lock a global lock, call the **pthread_global_lock_np**( ) routine. To unlock a global lock, call the **pthread_global_unlock_np**( ) routine.

Note: Many COBOL and FORTRAN compilers generate inherently nonreentrant code. Many C, Ada, Pascal, and BLISS compilers generate reentrant code by default. It is possible to write nonreentrant code in the reentrant languages by not following a locking protocol.

6.4.2 Thread-Specific Storage

To avoid nonreentrancy when writing new software, avoid using global variables to store data that is thread-specific data. (See Section 5.5 for more information.)

Alternatively, allocate thread-specific data on the stack or heap and explicitly pass its address to called routines.

6.5 Avoiding Priority Inversion

Priority inversion occurs when interaction among three or more threads blocks the highest-priority thread from executing. For example, a high-priority thread waits for a resource locked by a low-priority thread, and the low-priority thread waits while a middle-priority thread executes. The high-priority thread is made to wait while a thread of lower priority (the middle-priority thread) executes.

To avoid priority inversion, associate a priority with each resource and force any thread using that object to first raise its priority to that associated with the object. This method of avoiding priority inversion is not a complete solution because all threads will then block at the same ceiling priority and be unblocked in FIFO order rather than by their actual priority.

The **SCHED_OTHER** (default) scheduling policy prevents priority inversion from causing a complete blockage of the high-priority thread because the low-priority thread is permitted to execute and release the resource. The **SCHED_FIFO** and **SCHED_RR** policies, however, do not force resumption of the low-priority thread if the middle-priority thread executes indefinitely.

6.6 Using Synchronization Objects

The following subsections discuss the use of mutexes to prevent two potential problems: race conditions and deadlocks. Also discussed is why you should signal a condition variable with the associated mutex locked.

6.6.1 Race Conditions

A race condition occurs when two or more threads perform an operation, and the result of the operation depends on unpredictable timing factors; specifically, when each thread executes and waits and when each thread completes the operation.

An example of a race condition is as follows:

1. Both A and B are executing (X = X + 1).
2. A reads the value of X (for example, X = 5).
3. B comes in and reads the value of X and increments it (making X = 6).
4. A gets rescheduled and now increments X. Based on its earlier read operation, A thinks (X+1 = 5+1 = 6). X is now 6. It should be 7 because it was incremented once by A and once by B.

To avoid race conditions, ensure that any variable modified by more than one thread has only one mutex associated with it. Do not assume that a simple add operation can be completed without allowing another thread to execute. Such operations are generally not portable, especially to multiprocessor systems. If it is possible for two threads to share a data point, use a mutex.

6.6.2 Deadlocks

A deadlock occurs when one or more threads are permanently blocked from executing because each thread waits on a resource held by another thread in the deadlock. A thread can also deadlock on itself.

The following is one technique for avoiding deadlocks:

1. Associate a sequence number with each mutex.
2. Lock mutexes in sequence.
3. Do not attempt to lock a mutex with a sequence number lower than that of a mutex the thread already holds.

Another technique, which is useful when a thread needs to lock the same mutex more than once before unlocking it, is to use a recursive mutex. This technique prevents a thread from deadlocking on itself.

6.7 Signaling a Condition Variable

When you are signaling a condition variable and that signal may cause the condition variable to be deleted, it is recommended that you signal or broadcast with the mutex locked.

The recommended coding for signaling a condition variable appears at the end of this chapter. The following two C code fragments show coding that is *not recommended*. The following C code fragment is executed by a releasing thread:

```
pthread_mutex_lock (m);
/* Change shared variables to allow */
/* another thread to proceed */
pthread_mutex_unlock (m);        <---- Point A
pthread_cond_signal (cv);        <---- Statement 1
```

The following C code fragment is executed by a potentially blocking thread:

```
pthread_mutex_lock (m);
while (!predicate ...
        pthread_cond_wait (cv, m);

pthread_mutex_unlock (m);
```

Note: It is possible for a potentially blocking thread to be running at *Point A* while another thread is interrupted. The potentially blocking thread can then see the predicate true and therefore not become blocked on the condition variable.

Signaling a condition variable without first locking a mutex is not a problem. However, if the released thread deletes the condition variable without any further synchronization at *Point A*, then the releasing thread will fail when it attempts to execute *Statement 1* because the condition variable no longer exists.

This problem occurs when the releasing thread is a worker thread and the waiting thread is the boss thread, and the last worker thread tells the boss thread to delete the variables that are being shared by boss and worker.

The following C code fragment shows the *recommended* coding for signaling a condition variable while the mutex is locked:

```
pthread_mutex_lock (m);
/* Change shared variables to allow */
/* some other thread to proceed */

pthread_cond_signal (cv);    <---- Statement 1
pthread_mutex_unlock (m);
```

Chapter 7

Using the DCE Threads Exception-Returning Interface

DCE Threads provides the following two ways to obtain information about the status of a threads routine:

- The routine returns a status value to the thread.
- The routine raises an exception.

Before you write a multithreaded program, you must choose only one of the preceding two methods of receiving status. These two methods cannot be used together in the same code module.

The POSIX P1003.4a (pthreads) draft standard specifies that errors be reported to the thread by setting the external variable **errno** to an error code and returning a function value of -1. The threads reference pages document this status-value-returning interface (see the *OSF DCE Application Development Reference*). However, an alternative to status values is provided by DCE Threads in the exception-returning interface.

This chapter introduces and provides conventions for the modular use of the exception-returning interface to DCE Threads.

7.1 Syntax for C

Access to exceptions from the C language is defined by the macros in the **exc_handling.h** file. The **exc_handling.h** header file is included automatically when you include **pthread_exc.h** (see Section 7.2).

The following example shows the syntax for handling exceptions:

```
TRY
    try_block
[CATCH (exception_name)
    handler_block]...
[CATCH_ALL
    handler_block]
ENDTRY
```

A **try_block** or a **handler_block** is a sequence of statements, the first of which may be declarations, as in a normal block. If an exception is raised in the **try_block**, the catch clauses are evaluated in order to see if any one matches the current exception.

The **CATCH** or **CATCH_ALL** clauses absorb an exception; that is, they catch an exception propagating out of the **try_block**, and direct execution into the associated **handler_block**. Propagation of the exception, by default, then ends. Within the lexical scope of a handler, it is possible to explicitly cause propagation of the same exception to resume (this is called ''reraising'' the exception), or it is possible to raise some new exception.

The **RERAISE** statement is allowed in any handler statements and causes the current exception to be reraised. Propagation of the caught exception resumes.

The **RAISE** (**exception_name**) statement is allowed anywhere and causes a particular exception to start propagating. For example:

```
TRY
    sort(); /* Call a function that may raise an exception.
             * An exception is accomplished by longjumping
             * out of some nested routine back to the TRY
             * clause.  Any output parameters or return values
             * of the called routine are therefore indeterminate.
             */
```

```
CATCH (pthread_cancel_e)
    printf("Alerted while sorting\n"); RERAISE;

CATCH_ALL
    printf("Some other exception while sorting\n"); RERAISE;

ENDTRY
```

In the preceding example, if the **pthread_cancel_e** exception propagates out of the function call, the first **printf** is executed. If any other exception propagates out of sort, the second **printf** is executed. In either situation, propagation of the exception resumes because of the **RERAISE** statement. (If the code is unable to fully recover from the error, or does not understand the error, it needs to do what it did in the previous example and further propagate the error to its callers.)

The following shows the syntax for an epilogue:

```
TRY        try_block
[FINALLY   final_block]
ENDTRY
```

The **final_block** is executed whether the **try_block** executes to completion without raising an exception, or if an exception is raised in the **try_block**. If an exception is raised in the **try_block**, propagation of the exception is resumed after executing the **final_block**.

Note that a **CATCH_ALL** handler and **RERAISE** could be used to do this, but the epilogue code would then have to be duplicated in two places, as follows:

```
TRY
     try_block
CATCH_ALL
     final_block
     RERAISE;
ENDTRY
{ final_block }
```

A **FINALLY** statement has exactly this meaning, but avoids code duplication.

Note: The behavior of **FINALLY** along with the **CATCH** or **CATCH_ALL** clauses is undefined. Do *not* combine them for the same **try_block**.

Another example of the **FINALLY** statement is as follows:

```
pthread__mutex_lock (some_object.mutex);
some_object.num_waiters = some_object.num_waiters + 1;
TRY
    while (! some_object.data_available)
        pthread_cond_wait (some_object.condition);
    /* The code to act on the data_available goes here */
FINALLY
    some_object.num_waiters = some_object.num_waiters - 1;
    pthread_mutex_unlock (some_object.mutex);
ENDTRY
```

In the preceding example, the call to **pthread_cond_wait()** could raise the **pthread_cancel_e** exception. The **final_block** ensures that the shared data associated with the lock is correct for the next thread that acquires the mutex.

7.2 Invoking the Exception-Returning Interface

To use the exception-returning interface, replace

```
#include <pthread.h>
```

with the following include statement:

```
#include <pthread_exc.h>
```

7.3 Operations on Exceptions

An exception is an object that describes an error condition. Operations on exception objects allow errors to be reported and handled. If an exception is handled properly, the program can recover from errors. For example, if an exception is raised from a parity error while reading a tape, the recovery action may be to retry 100 times before giving up.

The DCE Threads Exception-Returning Interface allows you to perform the following operations on exceptions:

- Declare and initialize an exception object
- Raise an exception
- Define a region of code over which exceptions are caught
- Catch a particular exception or all exceptions
- Define epilogue actions for a block
- Import a system-defined error status into the program as an exception

These operations are discussed in the following subsections.

7.3.1 Declaring and Initializing an Exception Object

Declaring and initializing an exception object documents that a program reports or handles a particular error. Having the error expressed as an exception object provides future extensibility as well as portability. Following is an example of declaring and initializing an exception object:

```
EXCEPTION parity_error;          /* Declare it */
EXCEPTION_INIT (parity_error);   /* Initialize it */
```

7.3.2 Raising an Exception

Raising an exception reports an error, not by returning a value, but by propagating an exception. Propagation involves searching all active scopes for code written to handle the error or code written to perform scope-completion actions in case of any error, and then causing that code to execute. If a scope does not define a handler or epilogue block, then the scope is simply torn down as the exception propagates through the stack. This is sometimes referred to as ''unwinding the stack.'' DCE Threads exceptions are terminating; there is no option to make execution resume at the point of the error. (Execution resumes at the point where the exception was caught.)

If the exception is unhandled, the thread is terminated. This provides increased manageability by confining an error to a well-defined portion of a program. An example of raising an exception is as follows:

```
RAISE (parity_error);
```

7.3.3 Defining a Region of Code Over Which Exceptions Are Caught

Defining a region of code over which exceptions are caught allows you to call functions that can raise an exception and specify the recovery action.

Following is an example of defining an exception-handling region (without indicating any recovery actions):

```
TRY {
   read_tape ();
    }
ENDTRY;
```

7.3.4 Catching a Particular Exception or All Exceptions

It is possible to discriminate among errors and perform different actions for each error.

Following is an example of catching a particular exception and specifying the recovery action (in this case, a message). The exception is reraised (passed to its callers) after catching the exception and executing the recovery action:

```
TRY {
   read_tape ();
    }
CATCH (parity_error) {
   printf ("Oops, parity error, program terminating\n");
   printf ("Try cleaning the heads!\n");
   RERAISE;
   }
ENDTRY
```

7.3.5 Defining Epilogue Actions for a Block

A **FINALLY** mechanism is provided so that multithreaded programs can restore invariants as certain scopes are unwound; for example, restoring shared data to a correct state and releasing locks. This is often the ideal way to define, in one place, the cleanup activities for normal or abnormal exit from a block that has changed some invariant.

Following is an example of specifying an invariant action whether or not there is an error:

```
lock_tape_drive (t);
TRY
    TRY
       read_tape ();
    CATCH (parity_error)
       printf ("Oops, parity error, program terminating\n");
       printf ("Try cleaning the heads!\n");
       RERAISE;
```

```
        ENDTRY
        /* Control gets here only if no exception is raised */
        /* ... Now we can use the data off the tape */
    FINALLY
        /* Control gets here normally, or if any exception is raised */
        unlock_tape_drive (t);
    ENDTRY
```

7.3.6 Importing a System-Defined Error Status into the Program as an Exception

Most systems define error messages by integer-sized status values. Each status value corresponds to some error message text that should be expressed in the user's own language. The capability to import a status value as an exception permits the DCE Threads Exception-Returning Interface to raise or handle system-defined errors as well as programmer-defined exceptions.

An example of importing an error status into an exception is as follows:

```
exc_set_status (&parity_error, EPARITY);
```

The **parity_error** exception can then be raised and handled like any other exception.

7.4 Rules and Conventions for Modular Use of Exceptions

The following rules ensure that exceptions are used in a modular way so that independent software components can be written without requiring knowledge of each other:

- Use unique names for exceptions.

  A naming convention makes sure that the names for exceptions that are declared **EXTERN** from different modules do not clash.

The following convention is recommended:

```
<facility-prefix>_<error_name>_e
```

For example, **pthread_cancel_e**.

- Avoid putting code in a **TRY** routine that belongs before it.

  The **TRY** only guards statements for which the statements in the **FINALLY**, **CATCH**, or **CATCH_ALL** clauses are always valid.

  A common misuse of **TRY** is to put code in the **try_block** that needs to be placed before **TRY**. An example of this misuse is as follows:

```
TRY
    handle = open_file (file_name);
    /* Statements that may raise an exception here */
FINALLY
    close (handle);
ENDTRY
```

  The preceding **FINALLY** code assumes that no exception is raised by **open_file**. This is because the code accesses an invalid identifier in the **FINALLY** part if **open_file** is modified to raise an exception. The preceding example needs to be rewritten as follows:

```
handle = open_file (file_name);
TRY
    {
    /* Statements that may raise an exception here */
    }
FINALLY
    close (handle);
ENDTRY
```

  The code that opens the file belongs prior to **TRY**, and the code that closes the file belongs in the **FINALLY** statement. (If **open_file** raises exceptions, it may need a separate **try_block**.)

- Raise exceptions to their proper scope.

  Write functions that propagate exceptions to their callers so that the function does not modify any persistent process state before raising the exception. A call to the matching **close** call is required only if the **open_file** operation is successful in the current scope.

If **open_file** raises an exception, the identifier will not be written, so **open_file** must not require that **close** be called when **open_file** raises an exception; that is, **open_file** should not be part of the **TRY** clause because that means **close** is called if **open_file** fails, and you cannot close an unopened file.

- Do not place a **RETURN** or nonlocal **GOTO** between **TRY** and **ENDTRY**.

  It is invalid to use **RETURN** or **GOTO**, or to leave by any other means, a **TRY**, **CATCH**, **CATCH_ALL**, or **FINALLY** block. Special code is generated by the **ENDTRY** macro, and it must be executed.

- Use the ANSI C volatile attribute.

  Variables that are read or written by exception-handling code must be declared with the ANSI C volatile attribute. Run your tests with the optimize compiler option to ensure that the compiler thoroughly tests your exception-handling code.

- Reraise exceptions that are not fully handled.

  You need to reraise any exception that you catch, unless your handler performs the complete recovery action for the error. This rule permits an unhandled exception to propagate to some final default handler that prints an error message to terminate the offending thread. (An unhandled exception is an exception for which recovery is incomplete.)

  A corollary of this rule is that **CATCH_ALL** handlers must reraise the exception because they may catch any exception, and usually cannot do recovery actions that are proper for every exception.

  Following this convention is important so that you also do not absorb a cancel or thread-exit request. These are mapped into exceptions so that exception handling has the full power to handle all exceptional conditions from access violations to thread exit. (In some applications, it is important to be able to catch these to work around an erroneously written library package, for example, or to provide a fully fault-tolerant thread.)

- Declare only static exceptions.

  For compatibility with C++, you need to only declare static exceptions.

7.5 DCE Threads Exceptions and Definitions

Table 7-1 lists the DCE Threads exceptions and briefly explains the meaning of each exception. Exception names beginning with **pthread_** are raised as the result of something happening internal to the DCE Threads facility and are not meant to be raised by your code. Exceptions beginning with **exc_** are generic and belong to the exception facility, the underlying system, or both. The pthread-specific extensions are listed followed by the generic extensions, each in alphabetical order.

Table 7–1. DCE Threads Exceptions

Exception	Definition
pthread_badparam_e	An improper parameter was used.
pthread_cancel_e	A thread cancellation is in progress.
pthread_defer_q_full_e	No space is currently available to process an interrupt request.
pthread_existence_e	The object referenced does not exist.
pthread_in_use_e	The object referenced is already in use.
pthread_nostackmem_e	No space is currently available to create a new stack.
pthread_stackovf_e	An attempted stack overflow was detected.
pthread_unimp_e	This is an unimplemented feature.
pthread_use_error_e	The requested operation is improperly invoked.
exc_decovf_e	An unhandled decimal overflow trap exception occurred.
exc_exquota_e	The operation failed due to an insufficient quota.
exc_fltdiv_e	An unhandled floating-point division by zero trap exception occurred.
exc_fltovf_e	An unhandled floating-point overflow trap exception occurred.
exc_fltund_e	An unhandled floating-point underflow trap exception occurred.
exc_illaddr_e	The data or object could not be referenced.

Exception	Definition
exc_insfmem_e	There is insufficient virtual memory for the requested operation.
exc_intdiv_e	An unhandled integer divide by zero trap exception occurred.
exc_intovf_e	An unhandled integer overflow trap exception occurred.
exc_nopriv_e	There is insufficient privilege for the requested operation.
exc_privinst_e	An unhandled privileged instruction fault exception occurred.
exc_resaddr_e	An unhandled reserved addressing fault exception occurred.
exc_resoper_e	An unhandled reserved operand fault exception occurred.
exc_SIGBUS_e	An unhandled bus error signal occurred.
exc_SIGEMT_e	An unhandled EMT trap signal occurred.
exc_SIGFPE_e	An unhandled floating-point exception signal occurred.
exc_SIGILL_e	An unhandled illegal instruction signal occurred.
exc_SIGIOT_e	An unhandled IOT trap signal occurred.
exc_SIGPIPE_e	An unhandled broken pipe signal occurred.
exc_SIGSEGV_e	An unhandled segmentation violation signal occurred.
exc_SIGSYS_e	An unhandled bad system call signal occurred.
exc_SIGTRAP_e	An unhandled trace or breakpoint trap signal occurred.
exc_SIGXCPU_e	An unhandled CPU time limit exceeded signal occurred.
exc_SIGXFSZ_e	An unhandled file-size limit exceeded signal occurred.
exc_subrng_e	An unhandled subscript out-of-range trap exception occurred.
exc_uninitexc_e	An uninitialized exception was raised.

Chapter 8

DCE Threads Example

The example in this chapter shows the use of DCE Threads in a C program that performs a prime number search. The program finds a specified number of prime numbers, then sorts and displays these numbers. Several threads participate in the search: each thread takes a number (the next one to be checked), sees if it is a prime, records it if it is prime, and then takes another number, and so on.

This program shows the work crew model of programming (see Section 4.2.2). The workers (threads) increment a number (**current_num**) to get their next work assignment, which in this case is the same task as before, but with a different number to check for a prime. As a whole, the worker threads are responsible for finding a specified number of prime numbers, at which point their work is completed.

8.1 Details of Program Logic and Implementation

The number of workers to be used and the requested number of prime numbers to be found are defined constants. A macro is used to check for bad status (bad status returns a value of 1), and to print a given string and the associated error value upon bad status. Data to be accessed by all threads (mutexes, condition variables, and so forth) are declared as global items.

Worker threads execute the prime search routine, which begins by synchronizing with the boss (or parent) thread using a predicate and a condition variable. Always enclose a condition wait in a predicate loop to prevent a thread from continuing if it receives a spurious wakeup. The lock associated with the condition variable must be held by the thread when the condition wait call is made. The lock is implicitly released within the condition wait call and acquired again when the thread resumes. The same mutex must be used for all operations performed on a specific condition variable.

After the parent sets the predicate and broadcasts, the workers begin finding prime numbers until canceled by a fellow worker who has found the last requested prime number. Upon each iteration the workers increment the current number to be worked on and take the new value as their work item. A mutex is locked and unlocked around getting the next work item. The purpose of the mutex is to ensure the atomicity of this operation and the visibility of the new value across all threads. This type of locking protocol needs to be performed on all global data to ensure its visibility and protect its integrity.

Each worker thread then determines if its current work item (a number) is prime by trying to divide numbers into it. If the number proves to be nondivisible, it is put on the list of primes. Cancels are explicitly turned off while working with the list of primes in order to better control any cancels that do occur. The list and its current count are protected by locks, which also protect the cancellation process of all other worker threads upon finding the last requested prime. While still under the prime list lock, the current worker checks to see if it has found the last requested prime, and if so unsets a predicate and cancels all other worker threads. Cancels are then reenabled. The canceling thread falls out of the work loop as a result of the predicate that it unsets.

The parent thread's flow of execution is as follows: set up the environment, create worker threads, broadcast to them that they can start, join each thread as it finishes, and sort and print the list of primes.

- Setting up of the environment requires initializing mutexes and the one condition variable used in the example.
- Creation of worker threads is straightforward and utilizes the default attributes (**pthread_attr_default**). Note again that locking is performed around the predicate on which the condition variable wait loops. In this case, the locking is simply done for visibility and is not related to the broadcast function.
- As the parent joins each of the returning worker threads, it receives an exit value from them that indicates whether a thread exited normally or not. In this case the exit values on all but one of the worker threads are -1, indicating that they were canceled.
- The list is then sorted to ensure that the prime numbers are in order from lowest to highest.

The following pthread routines are used in this example:

- **pthread_cancel( )**
- **pthread_cond_broadcast( )**
- **pthread_cond_init( )**
- **pthread_cond_wait( )**
- **pthread_create( )**
- **pthread_detach( )**
- **pthread_exit( )**
- **pthread_join( )**
- **pthread_mutex_init( )**
- **pthread_mutex_lock( )**
- **pthread_mutex_unlock( )**
- **pthread_setcancel( )**
- **pthread_testcancel( )**
- **pthread_yield( )**

The following is the DCE Threads example:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
/*
 * Constants used by the example.
 */
#define     workers     5           /* Threads to perform prime check  */
#define     request     110         /* Number of primes to find        */
/*
 * Macros
 */
#define check(status,string)       if (status == -1) perror (string)
/*
 * Global data
 */
pthread_mutex_t prime_list;     /* Mutex for use in accessing the prime     */
pthread_mutex_t current_mutex;  /* Mutex associated with current number     */
pthread_mutex_t cond_mutex;     /* Mutex used for ensuring CV integrity     */
read_cond_t     cond_var;       /* Condition variable for thread start      */
int             current_num= -1;/* Next number to be checked, start odd     */
int             thread_hold= 1; /* Number associated with condition state   */
int             count=0;        /* Count of prime numbers - index to primes */
int             primes[request];/* Store prime numbers - synchronize access */
pthread_t       threads[workers];/* Array of worker threads                 */
/*
 * Worker thread routine.
 *
 * Worker threads start with this routine, which begins with a condition
 * wait designed to synchronize the workers and the parent.  Each worker
 * thread then takes a turn taking a number for which it will determine
 * whether or not it is prime.
 *
 */
void
prime_search (pthread_addr_t arg)
    {
    div_t   div_results;            /* DIV results: quot and rem        */
    int     numerator;              /* Used for determing primeness     */
    int     denominator;            /* Used for determing primeness     */
int     cut_off;                /* Number being checked div 2       */
int     notifiee;               /* Used during a cancellation       */
int     prime;                  /* Flag used to indicate primeness */
int     my_number;              /* Worker thread identifier         */
int     status;                 /* Hold status from pthread calls  */
int     not_done=1;             /* Work loop predicate              */
my_number = (int)arg;
/*
 * Synchronize threads and the parent using a condition variable, for
 * which the predicate (thread_hold) will be set by the parent.
 */
status = pthread_mutex_lock (&cond_mutex);
check(status,"1:Mutex_lock bad status\n");

while (thread_hold) {
    status = pthread_cond_wait (&cond_var, &cond_mutex);
    check(status,"3:Cond_wait bad status\n");
    }

status = pthread_mutex_unlock (&cond_mutex);
check(status,"4:Mutex_unlock bad status\n");
/*
 * Perform checks on ever larger integers until the requested
 * number of primes is found.
 */
while (not_done) {

    /* cancellation point */
    pthread_testcancel ();

    /* Get next integer to be checked */
    status = pthread_mutex_lock (&current_mutex);
    check(status,"6:Mutex_lock bad status\n");

    current_num = current_num + 2;              /* Skip even numbers */
    numerator = current_num;

    status = pthread_mutex_unlock (&current_mutex);
    check(status,"9:Mutex_unlock bad status\n");

    /* Only need to divide in half of number to verify not prime */
```

```
cut_off = numerator/2 + 1;
prime = 1;

/* Check for prime; exit if something evenly divides */
for (denominator = 2; ((denominator < cut_off) && (prime));
                                                denominator++) {
    prime = numerator % denominator;
    }
if (prime != 0) {

    /* Explicitly turn off all cancels */
    pthread_setcancel(CANCEL_OFF);

    /*
     * Lock a mutex and add this prime number to the list. Also,
     * if this fulfills the request, cancel all other threads.
     */
    status = pthread_mutex_lock (&prime_list);
    check(status,"10:Mutex_lock bad status\n");

    if (count < request)  {
        primes[count] = numerator;
        count++;
        }
    else if (count == request) {
        not_done = 0;
        count++;
        for (notifiee = 0; notifiee < workers; notifiee++) {
            if (notifiee != my_number) {
                status = pthread_cancel ( threads[notifiee] );
                check(status,"12:Cancel bad status\n");
                }
            }
        }

    status = pthread_mutex_unlock (&prime_list);
    check(status,"13:Mutex_unlock bad status\n");

    /* Reenable cancels */
    pthread_setcancel(CANCEL_ON);
    }
```

```
        pthread_testcancel ();
        }
    pthread_exit (my_number);
    }
main()
    {
    int     worker_num;     /* Counter used when indexing workers   */
    int     exit_value;     /* Individual worker's return status    */
    int     list;           /* Used to print list of found primes   */
    int     status;         /* Hold status from pthread calls       */
    int     index1;         /* Used in sorting prime numbers        */
    int     index2;         /* Used in sorting prime numbers        */
    int     temp;           /* Used in a swap; part of sort         */
    int     not_done;       /* Indicates swap made in sort          */

     * Create mutexes
     */
    status = pthread_mutex_init (&prime_list, pthread_mutexattr_default);
    check(status,"15:Mutex_init bad status\n");
    status = pthread_mutex_init (&cond_mutex, pthread_mutexattr_default);
    check(status,"16:Mutex_init bad status\n");
    status = pthread_mutex_init (&current_mutex, pthread_mutexattr_default);
    check(status,"17:Mutex_init bad status\n");

    /*
     * Create conditon variable
     */
    status = pthread_cond_init (&cond_var, pthread_condattr_default);
    check(status,"45:Cond_init bad status\n");

    /*
     * Create the worker threads.
     */
    for (worker_num = 0; worker_num < workers; worker_num++) {
        status = pthread_create (
            &threads[worker_num],
            pthread_attr_default,
            prime_search,
            (pthread_addr_t)worker_num);
        check(status,"19:Pthread_create bad status\n");
    }
```

```
/*
 * Set the predicate thread_hold to zero, and broadcast on the
 * condition variable that the worker threads may proceed.
 */
status = pthread_mutex_lock (&cond_mutex);
check(status,"20:Mutex_lock bad status\n");

thread_hold = 0;

status = pthread_cond_broadcast (&cond_var);
check(status,"20.5:cond_broadcast bad status0);

status = pthread_mutex_unlock (&cond_mutex);
check(status,"21:Mutex_unlock bad status\n");
/*
 * Join each of the worker threads inorder to obtain their
 * summation totals, and to ensure each has completed
 * successfully.
 *
 * Mark thread storage free to be reclaimed upon termination by
 * detaching it.
 */
for (worker_num = 0; worker_num < workers; worker_num++) {

  status = pthread_join (
    threads[worker_num],
    &exit_value );
  check(status,"23:Pthread_join bad status\n");

  if (exit_value == worker_num) printf("thread terminated normally\n");

  status = pthread_detach ( &threads[worker_num] );
  check(status,"25:Pthread_detach bad status\n");
  }

/*
 * Take the list of prime numbers found by the worker threads and
 * sort them from lowest value to highest.  The worker threads work
 * concurrently; there is no guarantee that the prime numbers
 * will be found in order. Therefore, a sort is performed.
 */
```

```
not_done = 1;
for (index1 = 1; ((index1 < request) && (not_done)); index1++) {
    for (index2 = 0; index2 < index1; index2++) {
        if (primes[index1] < primes[index2]) {
            temp = primes[index2];
            primes[index2] = primes[index1];
            primes[index1] = temp;
            not_done = 0;
            }
        }
    }

/*
 * Print out the list of prime numbers that the worker threads
 * found.
 */
printf ("The list of %d primes follows:\n", request);
printf("%d",primes[0]);

for (list = 1; list < request; list++) {
    printf (",%d", primes[list]);
    }

printf ("\n");
}
```

Part 3

DCE Remote Procedure Call

Part 3A

Using Remote Procedure Call

Part 3A describes the Remote Procedure Call (RPC) model and the basic concepts of DCE RPC. It contains a brief tutorial on how to develop RPC applications. This part also discusses the basic DCE RPC operations, the impact of remoteness on RPC applications, the use of the directory service interface, advanced RPC topics, and the use of exception handling.

Chapter 9

Introduction to Remote Procedure Calls

The Remote Procedure Call (RPC) model is a well-tested, industry-wide framework for distributing applications. An RPC executes a procedure located in a separate address space from the calling code. This is a remote procedure.

Applications that use remote procedure calls (RPC applications) look and behave much like local applications. However, an RPC application is divided into two parts: an RPC server, which offers one or more sets of remote procedures, and an RPC client, which makes remote procedure calls to RPC servers. A server and its clients generally reside on separate systems and communicate over a network. RPC applications depend on the RPC runtime. Any RPC runtime controls network communications for RPC applications. The DCE RPC runtime supports additional tasks, such as finding servers for clients and managing servers.

An RPC application uses dispersed computing resources such as CPUs, databases, devices, and services. The following are examples of RPC applications:

- A calendar-management application that allows authorized users to access the personal calendars of other users.

- A graphics application that processes data on central CPUs and displays the results on workstations.

- A manufacturing application that shares changing information about assembly components among design, inventory, scheduling, and accounting programs located on different computers.

9.1 General Requirements for Distributing an Application

RPC technology meets the basic requirements of a distributed application. These requirements include the following:

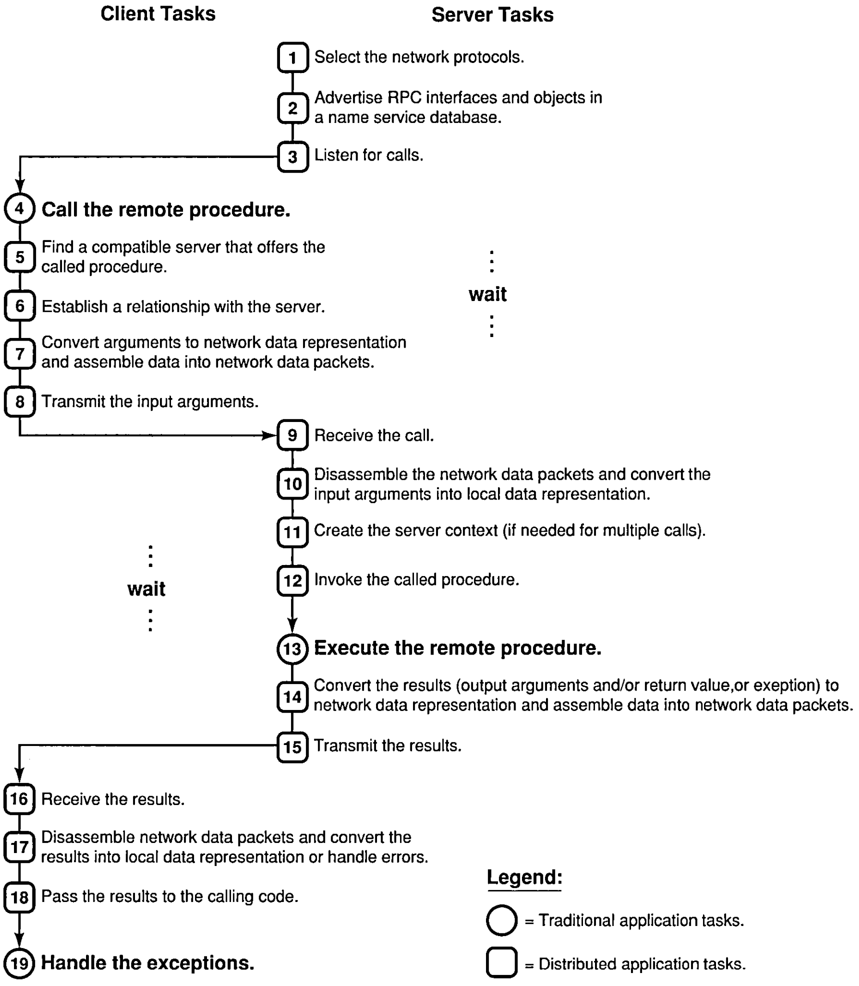
- Clients finding the appropriate servers
- Data conversion for operating in a heterogeneous environment
- Network communications

Distributing an application involves performing tasks such as managing communications, finding servers, providing security, and so forth. Without a convenient mechanism for these distributed computing tasks, writing distributed applications is difficult, expensive, and error-prone. A standalone distributed application needs to perform all of these tasks itself. RPC software performs distributed computing tasks for RPC applications, which can focus on issuing remote procedure calls, executing called procedures, and handling exceptions.

RPC software provides flexible code fragments that perform a full range of distributed computing tasks. RPC code fragments resemble code fragments of any high-level language and can be linked with client and server application code to form an RPC application.

Figure 9-1 shows the basic tasks that are necessary for distributing an application.

Figure 9–1. Tasks for Distributing an Application


Client Tasks
Server Tasks
1 Select the network protocols.
2 Advertise RPC interfaces and objects in a name service database.
3 Listen for calls.
4 Call the remote procedure.
5 Find a compatible server that offers the called procedure.
6 Establish a relationship with the server.
7 Convert arguments to network data representation and assemble data into network data packets.
8 Transmit the input arguments.
wait
9 Receive the call.
10 Disassemble the network data packets and convert the input arguments into local data representation.
11 Create the server context (if needed for multiple calls).
12 Invoke the called procedure.
wait
13 Execute the remote procedure.
14 Convert the results (output arguments and/or return value,or exeption) to network data representation and assemble data into network data packets.
15 Transmit the results.
16 Receive the results.
17 Disassemble network data packets and convert the results into local data representation or handle errors.
18 Pass the results to the calling code.
19 Handle the exceptions.
Legend:
○ = Traditional application tasks.
□ = Distributed application tasks.

9.2 The RPC Model

The RPC model is derived from the programming model of local procedure calls and takes advantage of the fact that every procedure contains a procedure declaration. The procedure declaration defines the interface between the calling code and the called procedure. All calls to a procedure must conform to the procedure declaration.

The procedure declaration defines the call syntax and parameters of the procedure; for example, consider the function **get_sum** written here in the C language:

```
long get_sum(long first, long second)       ①
{
    /* Add two input numbers and return their sum. */       ②
    long sum;
    sum = first + second;
    return(sum);
}
```

The preceding example shows the following:

1. Procedure declaration
2. Operations

9.2.1 RPC Interfaces

Traditionally, calling code and called procedures share the same address space and are linked. In an RPC application, the calling code and the called remote procedures are not linked; rather, they communicate indirectly through an RPC interface. An RPC interface is a logical grouping of operations, data types, and constants that serves as a unique network contract for a set of remote procedures. DCE RPC interfaces are compiled from formal interface definitions written by application developers using the DCE Interface Definition Language (IDL). Each RPC interface contains a Universal Unique Identifier (UUID), which is a hexadecimal number that can identify an entity. A UUID that identifies an RPC interface is known as an interface UUID. The interface UUID ensures that the interface can be uniquely identified across all possible network configurations.

In addition to an interface UUID, each RPC interface contains major and minor version numbers. Together, the interface UUID and version numbers form an interface identifier that identifies an instance of an RPC interface across systems and through time.

The following example shows the use of IDL in a simple interface definition for a math application.

```
[uuid(A01D0280-2D27-11C9-9FD3-08002B0ECEF1)]         ①
interface math

   {
   const long ARRAY_SIZE = 10;        ②

   typedef long array_type[ARRAY_SIZE];

   long get_sum([in] long first,        ③
                [in] long second);

   void get_sums([in] array_type a,        ④
                 [in] array_type b,
                 [out] array_type c);
   }
```

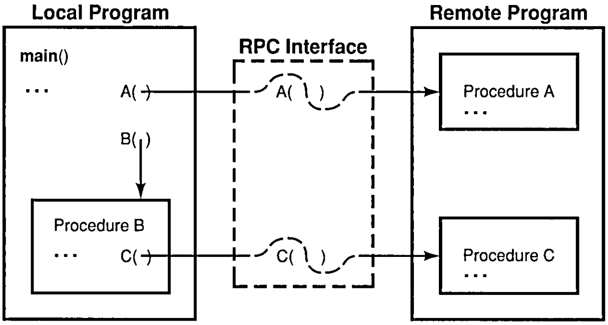
The interface definition contains the following:

1. Interface header
2. Constant and data type declarations
3. Operation declaration of **get_sum** remote procedure
4. Operation declaration of **get_sums** remote procedure

An RPC interface exists independently of specific applications. Each RPC interface can be implemented by any set of procedures that conforms to the interface definition. The operations of an interface are exactly the same for all implementations of the save version of the interface. This makes it possible for clients from different applications to call the same interface, and servers from different applications to offer the same interface.

Figure 9-2 shows the role of an RPC interface in a remote procedure call. The client contains calling code that makes two procedure calls. The first is a remote procedure call to Procedure A. The second is a local procedure call to Procedure B, which then makes a remote procedure call to Procedure C.

Figure 9–2. Role of the RPC Interface



Clients can use any practical combination of RPC interfaces, whether offered by the same or different servers. For example, using a database access interface, a client on a graphics workstation can call a remote procedure on a database server to fetch data from a central database. Then, using a statistics interface, the client can call a procedure on another server on a parallel processor to analyze the data from the central database and return the results to the client for display.

9.2.2 RPC Services

The simplest RPC application uses only one RPC interface. However, an application can use multiple RPC interfaces, and frequently, an integral set of RPC interfaces work together as an RPC service. An RPC service is a logical grouping of one or more RPC interfaces. For example, you can write a calendar service that contains only a personal calendar interface or a calendar service that contains additional RPC interfaces such as a scheduling interface for meetings.

Different services can share one or more RPC interfaces. For example, an administrative-support application can include an RPC interface from a calendar service. A client that calls that calendar interface service, without specifying a specific calendar, may use a server for either the calendar service or the administrative service.

9.2.3 RPC Objects

DCE RPC enables clients to find servers that offer specific RPC objects. An RPC object is an entity that an RPC server defines and identifies to its clients. Frequently, an RPC object is a distinct computing resource such as a particular database, directory, device, process, or processor. Identifying a resource as an RPC object enables an application to ensure that clients can use an RPC interface to operate on that resource. An RPC object can also be an abstraction that is meaningful to an application such as a service or the location of a server.

RPC objects are defined by application code. The RPC runtime provides substantial flexibility to applications about whether, when, and how they use RPC objects. RPC applications generally use RPC objects to enable clients to find and access a specific server. When servers are completely interchangeable, using RPC objects may be unnecessary. However, when clients need to distinguish between two servers that offer the same RPC interface, RPC objects are essential. If the servers offer distinct computing resources, each server can identify itself by treating its resources as RPC objects. Alternatively, each server can establish itself as an RPC object that is distinct from other instances of the same server.

RPC objects also enable a single server to distinguish among alternative implementations of an RPC interface, as long as each implementation operates on a distinct type of object. To offer multiple implementations of an RPC interface, a server must identify RPC objects, classify them into types, and associate each type with a specific implementation.

The set of remote procedures that implements an RPC interface for a given type of object is known as a ''manager.'' The tasks performed by a manager depend on the type of object on which the manager operates. For example, a manager of a queue-management interface may operate on print queues, while another manager may operate on batch queues.

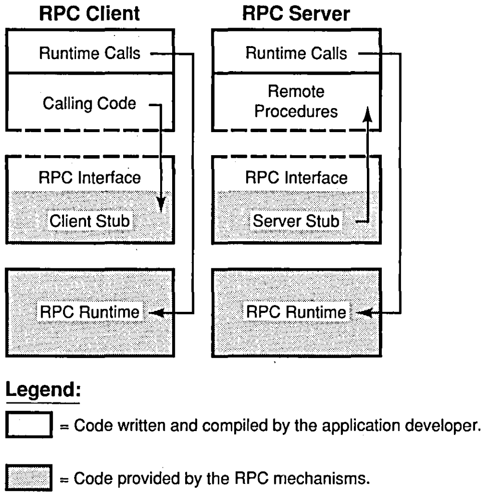
9.3 The Parts of an RPC Application

An RPC server or client contains application code, one or more RPC stubs, and a copy of the RPC runtime.

An RPC stub is an interface-specific code module that uses an RPC interface to pass and receive arguments. A server and a client contain complementary stubs for each RPC interface they share. RPC application code is the code written for a specific RPC application by the application developer. Application code implements and calls remote procedures, and also calls any RPC runtime routines the application needs. The DCE RPC runtime manages communications for RPC applications. In addition, a library of runtime routines enables RPC applications to set up their communications, manipulate information about servers, and perform optional tasks such a remotely managing servers and accessing security information.

Figure 9-3 shows the relationship of application code, stubs, and the RPC runtime in the server and client portions of an RPC application.

Figure 9–3. The Parts of an RPC Application



9.3.1 RPC Application Code

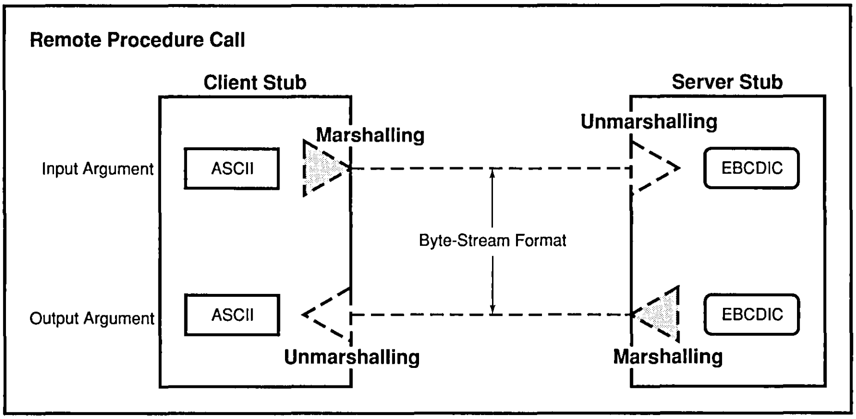
RPC application code differs for servers and clients. Minimally, server application code contains the remote procedures that implement one RPC interface, and the corresponding client contains calls to those remote procedures.

9.3.2 Stubs

A stub uses its RPC interface to pass call arguments. The stub performs basic support functions for remote procedure calls. For instance, stubs prepare input and output arguments for transmission between systems with different forms of data representation. The stubs use the RPC runtime to send and receive remote procedure calls. The client stub can also use the runtime to find servers for the client.

When a client application calls a remote procedure, the client stub first prepares the input arguments for transmission. The process for preparing arguments for transmission is known as ‘‘marshalling.’’ Marshalling converts call arguments into a byte-stream format and packages them for transmission. Upon receiving call arguments, a stub unmarshalls them. Unmarshalling is the process by which a stub disassembles incoming network data and converts it into application data using a format that the local system understands. Marshalling and unmarshalling both occur twice for each remote procedure call; that is, the client stub marshalls input arguments and unmarshalls output arguments, and the server stub unmarshalls input arguments and marshalls output arguments. Marshalling and unmarshalling permit client and server systems to use different data representations for equivalent data. For example, the client system can use ASCII characters and the server system can use EBCDIC characters (see Figure 9-4).

Figure 9–4. Marshalling and Unmarshalling Between ASCII and EBCDIC Data



The DCE IDL compiler (a component of the DCE RPC software) generates stubs by compiling an RPC interface definition written by application developers. The compiler generates marshalling and unmarshalling routines for IDL data types. For application-specific types of data, a developer must supply user-defined marshalling routines.

To build the client for an RPC application, a developer links client application code to a client stub for each RPC interface of the application; to build the server, the developer links the server application code to the corresponding server stubs.

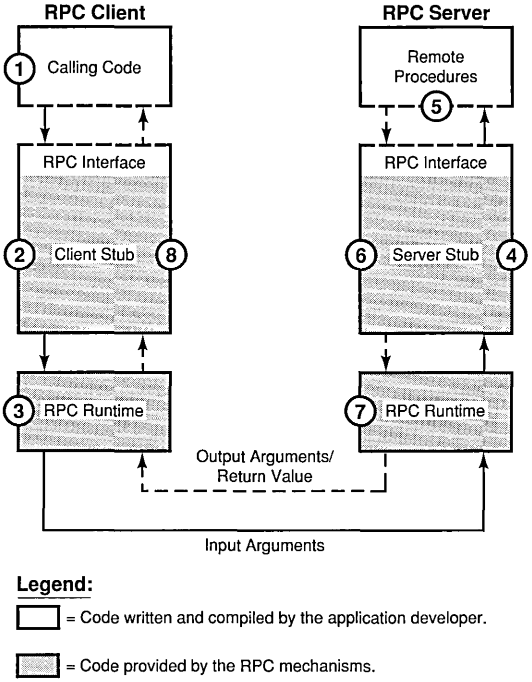
9.3.3 The RPC Runtime

In addition to one or more RPC stubs, every RPC server and RPC client is linked with a copy of the RPC runtime. Runtime operations perform tasks such as controlling communications between clients and servers and finding servers for clients on request. Stubs exchange arguments through their local RPC runtimes. The client runtime transmits remote procedure calls to the server. The server runtime receives the calls and dispatches each call to the appropriate server stub. The server runtime passes the call results to the client runtime. The DCE RPC runtime supports an Application Programming Interface (API) used by RPC application code to call runtime routines.

Server application code must also contain server initialization code that calls RPC runtime routines when the server is starting up and shutting down. Client application code can also call RPC runtime routines. Server and client application code can also contain calls to RPC stub-support routines. Stub-support routines allow applications to perform programming tasks such as allocating and freeing memory storage.

Figure 9-5 shows the roles of application code, RPC stubs, and RPC runtimes during a remote procedure call.

Figure 9–5. Interrelationships During a Remote Procedure Call



The following steps describe the interrelationships of the components of RPC applications, as shown in the previous figure:

1. The client's application code makes a remote procedure call, passing the input arguments to the stub for the called RPC interface.

2. The client's stub marshalls the input arguments and dispatches the call to the client's RPC runtime.
3. The client's RPC runtime transmits the input arguments over the communications network to the server's RPC runtime, which dispatches the call to the server stub for the called RPC interface.
4. The server's stub uses its copy of the RPC interface to unmarshall the input arguments and pass them to the called remote procedure.
5. The procedure executes and returns any results (output arguments or a return value or both) to the server's stub.
6. The server's stub marshalls the results and passes them to the server's RPC runtime.
7. The server's RPC runtime transmits the results over the communications network to the client's RPC runtime, which dispatches them to the client's stub.
8. The client's stub uses its copy of the RPC interface to unmarshall output arguments and pass them to the calling code.

9.4 DCE RPC and the Distributed Computing Environment

DCE RPC is a fully integrated part of the distributed computing environment. The communications capabilities of DCE RPC are used by clients and servers of other DCE components. In turn, RPC uses services provided by the following other DCE components: the Threads Service, the Cell Directory Service, and the Security Service.

To help RPC clients find RPC servers, RPC applications typically use a namespace. A namespace is a collection of information about applications, systems, and any other relevant computing resources. A namespace is maintained by a directory service such as the Cell Directory Service (CDS). DCE RPC provides a Name Service Interface (NSI) that is independent of any particular directory service. CDS, however, is the only directory service available for DCE RPC Version 1.0 applications.

NSI communicates with supported directory services for both RPC applications and the RPC control program. NSI insulates RPC applications from the intricacies of using a directory service. An RPC server uses NSI to store information about itself in a namespace, and a client uses NSI to access information about a server that meets the client's requirements for a specific RPC interface and object, among other things. The client uses this information to establish a relationship, known as a ''binding,'' with the server.

Thread services are also important to DCE RPC. A thread is a single sequential flow of control with one point of execution on a single processor at any instant. Multiple threads can coexist in a single process. DCE RPC uses threads internally for its own operations. For a discussion of threads and remote procedure calls, see Chapter 14, which describes advanced DCE RPC topics. DCE RPC also provides an environment where RPC applications can use thread services.

The DCE RPC runtime provides RPC applications with a programming interface to the DCE Security Service. The RPC authentication interface enables RPC clients and servers to mutually authenticate (that is, prove the identity of) each other. An authenticated remote procedure call provides client authorization information and authentication information to servers. Authorization information includes the privileges a client has and the identities a client is associated with at the time of a call. By comparing client authorization information to access control lists, a server can find out whether a client is eligible to use a requested remote procedure. Client authentication information identifies a client to a server.

9.5 Overview of DCE RPC Development Tasks

The tasks involved in developing an RPC application resemble those involved in developing a local application. As an RPC developer, you perform the following basic tasks:

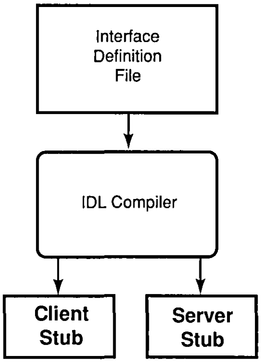
1. Design your application, deciding what procedures you need and which, if any, will be remote procedures.

2. Use the Universal Unique Identifier (UUID) generator to generate a UUID for your new interface.

3. Use the Interface Definition Language (IDL) to describe the RPC interface for the planned remote procedures.

4. Use the DCE IDL compiler to compile the IDL code to generate object code for the client stub and server stub. Figure 9-6 illustrates this task.

Figure 9–6. Generating Stubs



Note: Optionally, instead of generating object code for stubs, the DCE IDL compiler can generate the stubs as C source code. Stub source code is ANSI C compliant. It contains conditional preprocessor logic that allows most C compilers to compile them.

5. Write or modify application code using a compatible programming language; that is, a language that can be linked with C and can invoke C procedures, so the application code works with the stubs.

   Application code includes several kinds of code, as follows:

   a. Remote procedures

   b. Remote procedure calls

   c. Initialization code (calls to RPC stub-support or runtime routines)

   d. Any non-RPC code your application requires

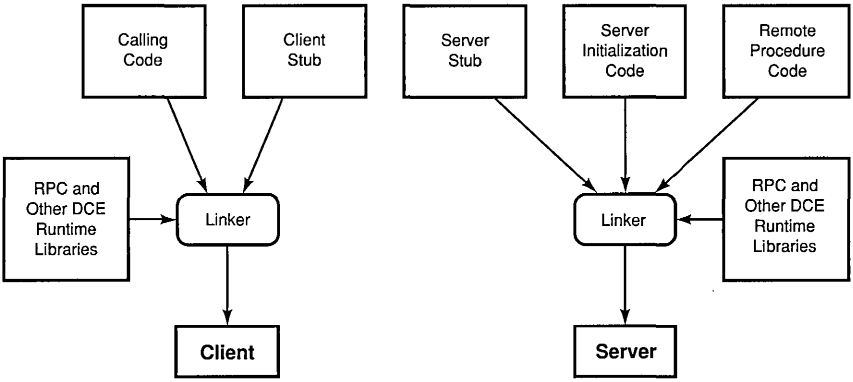
6. Generate object code from application code.

7. Create an executable client and server from the object files. Figure 9-7 illustrates this task.

For the client, link object code of the client stub(s) and the client application with the RPC runtime and any other needed runtime libraries.

For the server, link object code for the server stub(s), the initialization routines, and the set(s) of remote procedures with the RPC runtime and any other needed runtime libraries.

Figure 9–7. Building a Simple Client and Server


Calling Code
Client Stub
Server Stub
Server Initialization Code
Remote Procedure Code
RPC and Other DCE Runtime Libraries
Linker
Linker
RPC and Other DCE Runtime Libraries
Client
Server

8. After initial testing, distribute the new application by separately installing the server and client executable images on systems on the network.

Chapter 10

Basic DCE RPC Components

This chapter introduces the following DCE RPC components:

- DCE UUID generator
- DCE RPC Interface Definition Language
- DCE IDL compiler
- DCE RPC daemon
- Network Data Representation (NDR) transfer syntax
- DCE RPC runtime
- DCE RPC control program

10.1 DCE UUID Generator

The UUID generator is an interactive utility that creates UUIDs (Universal Unique Identifiers). A UUID is a hexadecimal number that contains information that makes it unique from all other UUIDs. This information includes a timestamp of the UUID's creation time and an identifier of the host of origin. The significance of a given UUID depends entirely on its context; for example, a UUID is an interface UUID only if so declared in an interface definition, as follows:

```
uuid(D07B6948-85BA-11CA-80AE-08002B245A28)
```

Various uses of UUIDs are discussed in Chapter 12 of this guide.

Run the UUID generator by using the **uuidgen** command. This command offers several options, including creating a template RPC interface definition file (an **.idl** file) containing a newly generated interface UUID. For example, the following command generates a template for the **Calendar** interface:

```
$ uuidgen -i -o Calendar.idl
```

The resulting **Calendar.idl** file contains the following text:

```
[
uuid(2FAC8900-31F8-11CA-B331-08002B13D56D),
version(1.0)
]
interface INTERFACENAME
{

}
```

The RPC interface developer replaces the text **INTERFACENAME** with the actual interface name; in this example, **Calendar**.

Note: A recommended convention is that you should use the RPC interface name defined in the file in the name of the **.idl** file; for example, **Calendar.idl**.

10.2 DCE RPC Interface Definition Language

Developing an RPC application involves writing and compiling an interface definition for a specific RPC interface that is written in the DCE Interface Definition Language (IDL). IDL is a high-level descriptive language whose syntax resembles that of ANSI C. DCE RPC interface definitions contain two basic components:

- An interface header

  An RPC interface header contains an interface UUID, interface version numbers, and an interface name. An RPC interface name is an easy-to-read local name that is not guaranteed to be unique; it is merely a convenience. It is helpful if the interface name reflects the nature or purpose of the RPC interface.

- An interface body

  An RPC interface body declares any application-specific data types and constants, and contains directives for including data types and constants from other RPC interfaces. The interface body also contains the operation declaration of each remote procedure to be accessed through the RPC interface. An operation declaration identifies the parameters of a procedure in terms of their data types, access method, and call order, and declares the data type of the return value (if any).

For more information, see Chapter 17, which discusses IDL syntax and its usage.

The following example shows the RPC interface definition, **Calendar.idl**, of the **Calendar** interface.

```
    /*        Calendar.idl        */
[
    uuid(2FAC8900-31F8-11CA-B331-08002B13D56D),
    version(1.0)
] interface Calendar
{
    /*        Type Declarations        */

    /* The opaque calendar object implemented as a
     * context handle to provide resource cleanup, and
     * implicit binding information.  */
    typedef [context_handle] void *Cal_Calendar_t;

    /* The name of the owner of a calendar */
    typedef [string] char Cal_String_t[];
    typedef Cal_String_t Cal_Username_t;

    /* enumeration of months for use in Cal_Time_t */
    typedef enum {
        January, February, March, April, May, June, July,
        August, September, October, November, December
        } Cal_Month_t;

    /* Specification of a time */
    typedef struct {
        short       year;
        Cal_Month_t month;
        short       day;
        short       hour;
        short       minute;
        } Cal_Time_t;

    /* Specification of an entry on the calendar */
    typedef struct {
        Cal_Time_t time;
        long       length;      /* in minutes */
        [ptr] Cal_String_t *description;
        } Cal_Appointment_t;

    /* List of values returned from calendar operations */
    typedef enum {
        Cal_s_ok,
        Cal_s_no_such_appointment,
        Cal_s_no_current_appointment,
        Cal_s_no_such_calendar
        } Cal_Status_t;

                  ▪  ▪  ▪
```

```
                    . . .
    /*       Operations Declarations        */

    /* Create the calendar for a user and return a calendar object. */
    Cal_Calendar_t Cal_create(
        [in] Cal_Username_t user
        );

    /*  Open the calendar of a specific user and return a
     *  calendar object.  Set the current calendar location
     *  to the first appointment that has not yet passed.  */
    Cal_Calendar_t Cal_open(
        [in] Cal_Username_t user
        );

    /*  Close a calendar  */
    void Cal_close(
        [in,out] Cal_Calendar_t *calendar
        );

    /*  Add a new appointment in the calendar.  This
     *  has no effect upon the current calendar location. */
    Cal_Status_t Cal_add_appointment(
        [in] Cal_Calendar_t calendar,
        [in] Cal_Appointment_t *appointment
        );

    /*  Get the first appointment in the calendar and reset the
     *  current calendar location to the specified appointment. */
    Cal_Status_t Cal_get_first(
        [in] Cal_Calendar_t calendar,
        [out] Cal_Appointment_t *appointment
        );

    /*  Get the next appointment in the calendar based
     *  upon the current calendar location.  */
    Cal_Status_t Cal_get_next(
        [in] Cal_Calendar_t calendar,
        [out] Cal_Appointment_t *appointment
        );

    /*  Delete the appointment at the current calendar location. */
    Cal_Status_t Cal_delete_current_appointment(
        [in] Cal_Calendar_t calendar
        );
}
```

Using IDL, a programmer can write a definition of an RPC interface for any set of procedures. An RPC interface such as this sample **Calendar** interface can be implemented using any programming language, such as Pascal or FORTRAN, under the following conditions:

- The object code must be linkable with the C object code of the stubs.
- The procedure declarations must conform to the operation declarations of the RPC interface definition and the calling sequences must be compatible.

10.3 DCE IDL Compiler

The DCE IDL compiler (**idl**) processes RPC interface definitions written in IDL and generates header files and stub object code. (The compiler can generate source code for the stubs written in ANSI C.) The code generated from an RPC interface definition by the compiler includes client and server stubs that contain the RPC interface.

The compiler also generates a data structure called the interface specification, which contains identifying and descriptive information about the compiled interface, and creates a companion global variable, the interface handle, which is a reference to the interface specification. Each header file generated by the IDL compiler contains the reference the application code needs to access the interface handle. The interface handle allows the application code to refer to the interface specification in calls to the RPC runtime. Runtime operations obtain required information about the interface, such as its UUID and version numbers, directly from the interface specification.

Developers can tailor how an RPC interface appears to local application code and how the local application code interacts with the RPC interface. Along with the interface definition file, the compiler searches for an optional attribute-configuration file. The Attribute Configuration File (ACF) is written in the Attribute Configuration Language, which is a companion language to IDL. An ACF modifies how the compiler interprets an RPC interface definition. For example, an ACF can specify a subset of operations declarations for a client stub so that the client stub contains declarations for only the operations that the client application code needs for its remote procedure calls.

The following are some of actions that can be specified using an ACF:

- Omitting operations from the client stub's copy of the RPC interface

  Limiting the client's access to the remote procedures offered by servers reduces the size of the client stub.

- Representing a local data type as a network data type

  An application can equate an application-specific local data type to an IDL-specific data type that is declared in an RPC interface definition. The application must define the local data type to the stub. An interface definition can define the local data type, or the attribute configuration file can contain an include statement for a file containing a definition of the local type.

  The application must also provide the routines that switch between the local data type and the network data type. The stubs call these routines whenever the application passes the local data type in a remote procedure call.

- Defining how a client establishes a binding with a server that implements the called interface

  The available methods of managing bindings are discussed in Chapter 13.

- Specifying how arguments of nonscalar data types are marshalled for transmission and unmarshalled

  By default, every data type is marshalled and unmarshalled by inline code, which is part of the direct control flow in the stubs and which recurs for every parameter of the data type. However, for any nonscalar data type declared in an RPC interface definition, an ACF can relocate marshalling and unmarshalling functions into out-of-line code. Out-of-line code occurs only once per server and once per client. The same code is used to marshall and unmarshall all parameters of the out-of-line data type. The out-of-line code for a given data type resides in an auxiliary file that is linked into the client and server, along with the stubs that use it. For large data structures used for parameters of more than one operation, out-of-line code reduces the amount of code in an application, but also reduces execution speed.

For information on the ACF attributes that produce these and other actions in the stubs of a given RPC interface definition, see Chapter 18, which discusses the Attribute Configuration Language syntax and usage.

The following example shows an ACF (**Calendar.acf**), which is intended to be used when compiling **Calendar.idl**. For two data types declared in the IDL file, the ACF file declares out-of-line marshalling and unmarshalling.

```
/* Calendar.acf */

[auto_handle] interface Calendar
{
        typedef [out_of_line] Cal_String_t;
        typedef [out_of_line] Cal_Appointment_t;
}
```

10.4 DCE RPC Daemon

The RPC daemon (**rpcd**) is a process that provides the endpoint map service. This service maintains the local endpoint map for local RPC servers and looks up endpoints for RPC clients. An endpoint is the address of a specific instance of a server that is executing in a particular address space on a given system (a server instance). Each endpoint can be used on a system by only one server at a time.

An endpoint map is a system-specific database where servers register their endpoints and associated addressing information (the host address, information about communications protocol, and so on). A server registers endpoints separately for each of its RPC interfaces and any RPC objects offered with the interface.

If a client makes a remote procedure call to a system without providing an endpoint, the endpoint map service searches its endpoint map for the endpoint of a compatible server. Among other things, a compatible server offers the requested RPC interface, and if requested, the RPC object. On finding a suitable endpoint, the map service returns the endpoint to the client's runtime, which sends the call to the server at that endpoint.

When a server stops running, its map elements become outdated. Although the endpoint map service routinely removes any map element containing an outdated endpoint, a lag time exists during which stale entries remain. If a remote procedure call uses an endpoint from an outdated map element, the call fails to find a server. To prevent clients from receiving stale data from the endpoint map, before a server stops, it should remove its own map elements.

10.5 Network Data Representation Transfer Syntax

A transfer syntax is a set of encoding rules used for the network transmission of data and the conversion to and from different local data representations. A shared transfer syntax enables communications between systems that represent local data differently. DCE RPC currently uses a single transfer syntax, Network Data Representation (NDR). NDR encodes data into a byte stream for transmission over a network. A transfer syntax such as NDR enables machines with different formats to exchange data successfully.

Note: The DCE RPC communications protocols support the negotiation of transfer syntax. However, at present, the outcome of a transfer-syntax negotiation is always NDR.

10.6 DCE RPC Runtime

Every system running an RPC server or client must possess an RPC runtime. The DCE RPC runtime manages communications for RPC application. In addition, the DCE RPC runtime provides a library of routines to support RPC applications. An Application Programming Interface (API) enables server application code and client application code to call RPC routines to access runtime operations.

The basic classes of runtime operations include the following:

- Communications operations that establish communications links, transmit and receive remote procedure calls, and affect how data is transmitted

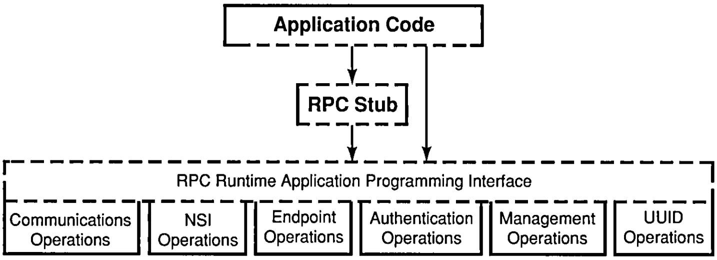
- Name Service Interface (NSI) operations that access namespaces for RPC applications

- Endpoint operations that allow servers to add server addressing information to and remove it from the local endpoint map
- Authentication operations that affect the type of authentication, protection level, and type of authorization used for communications between a client and server
- Miscellaneous classes of operations, such as the UUID operations, which allow applications to manipulate UUIDs, and the management operations, which provide a number of management operations, such as stopping servers

Note: In addition to the API, stubs use a private Stub Programming Interface (SPI). The SPI routines are unavailable to application code.

Figure 10-1 shows the relationship of RPC application and stub code to the basic kinds of runtime operations.

Figure 10–1. Relationship of RPC Application and Stub Code to Runtime Operations


Application Code
RPC Stub
RPC Runtime Application Programming Interface
Communications Operations
NSI Operations
Endpoint Operations
Authentication Operations
Management Operations
UUID Operations

10.6.1 Communications Operations

A binding (the relationship between an RPC client and server involved in a remote procedure call) requires that a communications link (a network pathway) exist between the client and the server. The communications operations use the underlying layers of network software to establish a communications link and transfer data between a client and a server. To initialize, RPC servers must make a number of calls to communications operations; for example, for selecting the protocol sequences to use and registering with the endpoint map.

10.6.2 Directory Service Interface Operations

To advertise RPC interfaces and objects to clients, servers use a directory service interface. A directory service interface maintains a namespace. A namespace is a repository of uniquely named entries that store information about computing resources for applications. Distributing an application often involves installing copies of the server on several systems whose locations may be subject to change and are often unknown to clients. The operations of the DCE RPC Name Service Interface (NSI) provide a means for using a directory service to store and access information about RPC servers. NSI routines insulate RPC applications from the intricacies of naming. An RPC server can use NSI to advertise one or more of its RPC interfaces, its RPC objects, and its server addressing information in a namespace. Using the NSI export operation, an RPC server can place information about its interfaces, objects, and addresses into its own namespace entry (known as a server entry). RPC clients can access that information using the NSI import operation.

10.6.3 Endpoint Operations

An endpoint is an address of a specific server instance. DCE RPC endpoint operations allow servers to register and remove (unregister) their own endpoints from the local endpoint map.

10.6.4 Authentication Operations

The most common use of authentication operations is to prove the identity of a client to a server and of a server to a client so that appropriate authorization decisions can be made. For example, before a banking application can transfer money from one account to another, the server must verify the identity of the client that is requesting the transfer and determine whether or not the client is authorized for that transaction. The RPC authentication operations manipulate authentication attributes that are used by RPC applications. Also, the RPC authentication operations dictate how identities are communicated between clients and servers.

Note: DCE RPC Version 1.0 supports authenticated RPC for the connectionless (datagram) protocol only.

10.6.5 Miscellaneous Runtime Operations

The RPC runtime offers several other kinds of operations to RPC applications. The most significant are operations that manipulate UUIDs and that manage RPC applications. The runtime also provides other operations such as requesting the error text of a given status code.

10.6.5.1 UUID Operations

UUID operations provide a set of operations to applications for creating and manipulating UUIDs.

Note: The UUID runtime routines are distinct from the UUID generator, which is an interactive utility that uses the UUID operations.

10.6.5.2 Management Operations

The RPC runtime performs management tasks on request such as setting communications timers, gathering selected kinds of information, and stopping a server from listening for remote procedure calls.

The management operations are separated into local and remote operations. Local management operations operate on only the calling application. Remote management operations allow remote control of servers from the local system or from remote systems.

A remote management interface makes the remote management operations accessible to remote management calls coming from the network. Linking server application code and stubs with the RPC runtime library automatically incorporates this remote management interface into every RPC server. Therefore, any server on the target system is capable of processing remote management calls. A caller must specify either an endpoint or object UUID when calling any routine of the remote management interface. If the client specifies an object UUID that is offered by several servers, the endpoint map service selects the endpoint of one of those servers for the client.

Note: The term ''manager'' refers to a set of remote procedures. The term ''management program'' refers to a program that manages an RPC application or the RPC runtime by calling management routines such as the **rpc_mgmt_is_server_listening( )** routine or the **rpc_mgmt_ep_unregister( )** routine.

10.7 DCE RPC Control Program

The RPC control program (**rpccp**) is an interactive management utility for the administrators and users of RPC applications. The control program provides one means for managing namespace entries and endpoint mapping. Many operations of the RPC directory service interface are accessible using the control program. Individuals with the necessary permission can add entries to and remove them from a namespace, can add information to and remove it from those entries, and can retrieve information. Also, the control program enables showing and unregistering endpoint map elements (or mappings) from the local endpoint map and any remote endpoint map.

The **rpccp** command used alone starts the RPC control program, as follows:

```
$ rpccp
```

By prefacing specific control program commands with **rpccp**, you can enter commands at the system prompt or from within a shell script (command procedure), for example:

```
$ rpccp show server /.:/building_1/server_entry_name
```

You can enter the specific commands of the control program interactively at the rpccp> prompt, for example:

```
rpccp> show server /.:/department/server_entry_name
```

Chapter 11

Building an Application

This chapter explains how to write an interface definition in the DCE RPC Interface Definition Language (IDL) and illustrates the basic features of IDL. As an example, we present an interface definition for **binop** (binary operations), a very simple application that uses RPC to add pairs of integers on a remote server. The remainder of the chapter describes how to develop, build, and run the **binop** client and server programs.

11.1 Writing an Interface Definition

The first step in developing a distributed application is to write an interface definition file in IDL. The IDL compiler, **idl**, uses the information in an interface definition to generate a header file, client stub files, and server stub files. The IDL compiler produces header files in C and can produce stubs either as C source files or as object files. For applications that use certain data types or certain features of RPC, the IDL compiler also generates client or server auxiliary files, or both, which contain support routines that are called by the stubs. (The **binop** example does not require auxiliary files.)

For some applications, you may also need to write an Attribute Configuration File (ACF) to accompany the interface definition. If an ACF exists, the IDL compiler interprets the ACF when it compiles the interface definition. Information in the ACF is used to modify the code that the compiler generates. (The **binop** example does not require an ACF.)

An IDL interface definition consists of a header and a body. The body can contain the following constructs:

- Import declarations
- Constant declarations
- Type declarations
- Operation declarations

IDL declarations resemble declarations in ANSI C. IDL is purely a declarative language, so, in some ways, an IDL interface definition is like a C header file. However, an IDL interface definition must specify the extra information that is needed by the remote procedure call mechanism. Most of this information is expressed via IDL attributes. IDL attributes can apply to types, to type members, to operations, to operation parameters, or to an interface as a whole. An attribute is represented in [ ] (brackets) before the item to which it applies.

A comment can be inserted at any place in an interface definition where white space is permitted. IDL comments, like C comments, begin with /* (a slash and an asterisk) and end with */ (an asterisk and a slash).

The remainder of this section briefly explains how to create an interface definition and gives simple examples of each kind of IDL declaration. For a detailed description of IDL, refer to Chapter 17. For information on the IDL compiler, see the **idl(1rpc)** reference page in the *OSF DCE Application Development Reference*.

11.1.1 Generating an Interface UUID

The first step in building an RPC application is to generate a skeletal interface definition file and a UUID for the interface. Every interface in an RPC application must have a Universal Unique Identifier (UUID). When you define a new interface, you must generate a new UUID for it. (In an object-oriented application, every object and object type must also have a non-nil UUID.)

Typically, you run **uuidgen** with the **-i** option. The **-i** option produces a skeletal interface definition file and includes the generated UUID for the interface. For example:

```
$ uuidgen -i > binop.idl
$ cat binop.idl

[
uuid(443f4b20-a100-11c9-baed-08001e0218cb),
version(1)
]
interface INTERFACENAME {

}
```

The first part of the skeletal definition is the header, which specifies a UUID, a version number, and a name for the interface. The last part of the definition is { } (an empty pair of braces); import, constant, type, and operation declarations go between these braces.

By convention, the names of interface definition files end with the suffix **.idl**. To construct names for its output files, the IDL compiler replaces **.idl** with other suffixes, which by default are as follows:

- **.h** for header files
- **_cstub.c** for client stub files
- **_sstub.c** for server stub files
- **_caux.c** for client auxiliary files
- **_saux.c** for server auxiliary files

For example, compilation of a **chess.idl** interface definition file would produce a **chess.h** header file, a **chess_cstub.c** client stub file, and a **chess_sstub.c** server stub file. (The IDL compiler can also produce stubs in object format by invoking a C compiler, in which case the **.c** portion of the suffix is typically replaced by **.o**.)

For more information on the UUID generator, see the **uuidgen(1rpc)** reference page in the *OSF DCE Application Development Reference*.

11.1.2 Naming the Interface

After you have used **uuidgen** to generate a skeletal interface definition, replace the dummy string **INTERFACENAME** with the name of your interface.

By convention, the name of an interface definition file is the same as the name of the interface it defines, with the suffix **.idl** appended. For example, the definition for a **bank** interface would reside in a **bank.idl** interface definition file, and if the application required an ACF, its name would be **bank.acf**.

The IDL compiler incorporates the interface name in identifiers it constructs for various data structures and data types, so the length of an interface name can be at most 17 characters. (Most IDL identifiers have a maximum length of 31 characters.)

11.1.3 Specifying Interface Attributes

Interface attributes are defined within [ ] (brackets) in the header of the interface definition. The definition for any remote interface needs to specify the **uuid** and **version** interface attributes, so these are included in the skeletal definition that **uuidgen** produces.

If an interface is called only locally and never remotely, it can be given the **local** attribute, which causes the compiler to generate only header files, not stubs, for the interface.

If an interface is exported by servers on well-known endpoints, these endpoints must be specified via the **endpoint** attribute. Interfaces that use dynamic endpoints do not have this attribute. (A well-known endpoint is a stable address on the host, while a dynamic endpoint is an address that the RPC runtime requests for the server.)

For detailed information about all the interface attributes, see Chapter 17.

11.1.4 Import Declarations

The IDL **import** declaration specifies another interface definition whose types and constants are used by the importing interface.

The **import** declaration allows you to collect declarations for types and constants that are used by several interfaces into one common file. For example, if you are defining two database interfaces named **dblookup** and **dbupdate**, and these interfaces have many constants in common, you can declare those constants in a **dbconstants.idl** file and import this file in the **dblookup.idl** and **dbupdate.idl** interface definitions. For example:

```
import "dbconstants.idl";
```

By default, the IDL compiler resolves relative pathnames of imported files by looking first in the current working directory and then in the system IDL directory. The **-I** option of the IDL compiler allows you to specify additional directories to search. You can thereby avoid putting absolute pathnames in your interface definitions. For example, if an imported file has the absolute pathname **/dbproject/src/dbconstants.idl**, then the IDL compiler option **-I/dbproject/src** allows you to import the file by its leaf name, **dbconstants.idl**.

11.1.5 Constant Declarations

The IDL **const** declaration allows you to declare integer, Boolean, character, string, and null pointer constants, as in the following examples:

```
const short TEN = 10;
const boolean VRAI = TRUE;
const char* JSB = "Johann Sebastian Bach";
```

11.1.6 Type Declarations

To support application development in a variety of languages and to support the special needs of distributed applications, IDL provides an extensive set of data types, including the following:

- Simple types, such as integers, floating-pointing numbers, characters, Booleans, and the primitive binding-handle type **handle_t** (equivalent to **rpc_binding_handle_t**)
- Constructed types, such as strings, structures, unions, arrays, pointers, and pipes
- Predefined types, including ISO international character types and the error status type **error_status_t**

The IDL **typedef** declaration lets you give a name to any of these types.

The general form of a type declaration is

typedef [*type_attribute,...*] *type_specifier type_declarator,...*;

where the bracketed list of type attributes is optional. The *type_specifier* specifies a simple type, a constructed type, a predefined type, or a named type. Each *type_declarator* is a name for the type being defined. As in C, arrays and pointers are declared by the *type_declarator* constructs [ ] (brackets) and an * (asterisk) rather than by *type_specifier* constructs.

The following type declaration defines **integer32** as a name for a 32-bit integer type:

```
typedef long integer32;
```

The *type_specifier* constructs for structures and unions permit the application of attributes to members. In the following example, one member of a structure is a conformant array (an array without a fixed upper bound), and the **size_is** attribute names another member of the structure that at runtime provides information about the size of the array:

```
typedef struct {
    long dsize;
    [size_is(dsize)] float darray[];
    } dataset;
```

11.1.7 Operation Declarations

Operation declarations specify the signature of each operation in the interface, including the operation name, the type of data returned (if any), and the types of all parameters passed (if any) in a call.

The general form of an operation declaration is

[*operation_attribute*, ...] *type_specifier operation_identifier* (*parameter_declaration*, ...);

where the bracketed list of operation attributes is optional. Among the possible attributes of an operation are **idempotent**, **broadcast**, and **maybe**, which specify semantics to be applied by the RPC runtime mechanism when the operation is called. If an operation when called once can safely be executed more than once, the IDL declaration of the operation needs to specify the **idempotent** attribute; **idempotent** semantics allow remote procedure calls to execute more efficiently.

The *type_specifier* specifies the type of data returned by the operation.

The *operation_identifier* names the operation. Although operation names are arbitrary, a common convention is to use the name of an interface as a prefix for the names of its operations. For example, a **bank** interface may have operations named **bank_open**, **bank_close**, **bank_deposit**, **bank_withdraw**, and **bank_balance**.

Each *parameter_declaration* in an operation declaration declares a parameter of the operation. A *parameter_declaration* takes the following form:

[*parameter_attribute*, ...] *type_specifier parameter_declarator*

Every parameter attribute must have at least one of the parameter attributes **in** and **out** to specify whether the parameter is passed as an input, as an output, or in both directions. The *type_specifier* and *parameter_declarator* specify the type and name of the parameter.

Output parameters must be passed by reference and therefore must be declared as pointers via the pointer operator * (an asterisk).

In applications that use explicit handles, you must supply a handle as the first parameter in each operation declaration, as in the following example:

```
void exp_op(
     [in] handle_t h,
     [in] long a,
     [in] long b,
     [out] long *c
     );
```

In applications that use an implicit handle or use automatic handles (an ACF feature), operations do not require handle parameters:

```
void imp_op(
     [in] long a,
     [in] long b,
     [out] long *c
     );
```

11.1.8 The binop Interface Definition

Following is the IDL definition for the **binop** interface, which resides in the **binop.idl** file:

```
[
uuid(443f4b20-a100-11c9-baed-08001e0218cb),
version(1.0)
]
interface binop
{
[idempotent]
void binop_add
(
     [in] handle_t h,
     [in] long a,
     [in] long b,
     [out] long *c
);
}
```

The **binop** IDL definition defines Version 1.0 of the **binop** interface. No well-known endpoints are specified in the interface header; servers exporting this interface listen on dynamic endpoints.

The interface contains one operation, **binop_add**, which is declared to be **idempotent**. Because the **binop** application uses explicit handles, the operation has a handle as its first parameter. The handle is of the primitive handle type **handle_t**. The other inputs and the output are 32-bit integers.

The **binop** interface requires no import, constant, or type declarations.

11.2 Running the IDL Compiler

After you have written an interface definition, run the IDL compiler to generate header and stub files.

To run the IDL compiler, you need to have the environment variable **NLSPATH** set so that the compiler can find its catalog of diagnostic messages. The value of this variable must include the string *dir*/%N, where *dir* is the directory in which the **idl.cat** file resides. A general C shell command that includes the string *dir*/%N and that sets **NLSPATH** follows:

```
% setenv NLSPATH dceshared/nls/msg/LANG/%N
```

Note that *LANG* is a variable instead of a literal. A specific example of this general command follows:

```
% setenv NLSPATH /usr/lib/nls/msg/en_US.88591/%N
```

The compiler offers many options that, for example, allow you to choose what C compiler or C preprocessor commands are run, what directories are searched for imported files, which of the possible output files are generated, and how the output files are named.

When you compile the definition of a remote interface, you must ensure that the system IDL directory is among those that the compiler searches for imported files because any remote interface implicitly imports **rpc.idl**.

The **binop.idl** interface definition can be compiled by the following command:

```
% idl binop.idl -keep c_source
```

This compilation produces a header file, **binop.h**; a client stub file, **binop_cstub.c**; and a server stub file, **binop_sstub.c**. Because the IDL compiler option **-keep c_source** is specified, the compiler produces stubs in C source code rather than in object format.

For complete information on running the IDL compiler, see the **idl(1rpc)** reference page in the *OSF DCE Application Development Reference*.

11.3 Writing the Client Code

The following subsections describe the client program for the **binop** application, whose interface definition was shown earlier in this chapter.

11.3.1 Overview of the binop Client Program

The **binop** client program takes three arguments and is invoked as follows:

$ **client** *protseq hostid passes*

The **binop** application requires users of the client program to specify a host on which the server program is running. The client program therefore does not need to use RPC directory service interface calls to obtain information about server locations from the DCE CDS namespace.

As specified in the **binop.idl** interface definition, the **binop** application uses explicit handles, and these handles are of the primitive RPC binding handle type. The client therefore passes a binding handle of type **rpc_binding_handle_t** as the first parameter of the **binop_add** operation. The client uses information from the command line to compose an RPC string binding, then converts the string binding to a primitive binding handle. At runtime, when the client makes its first remote procedure call, the handle is only partially bound because the client does not know the particular endpoint on which the server is listening; for delivery of its requests to the server endpoint, the client depends on the endpoint mapping service of an **rpcd** on the server host.

Because **binop** exists just for pedagogical purposes, users of the client program do not actually supply inputs to be added. The client program loops through a set of inputs that it generates.

11.3.2 The client.c Source Code

This section presents **client.c**, the source code module for the **binop** client. The module is printed in three parts: the code that precedes the **main** function, the **main** function, and a **do_calls** function.

The **client.c** module includes **binop.h**, a header file for the **binop** interface generated by the IDL compiler.

```
#include <stdio.h>
#include "binop.h"

#define CALLS_PER_PASS 100
```

The client **main** function performs the following major steps:

1. It checks the command-line arguments.
2. It calls **rpc_binding_from_string_binding( )** to convert the string binding into a primitive binding handle.
3. It prints the string binding, just for diagnostic purposes, then calls **rpc_string_free( )** to free the memory that was allocated for the string by the RPC runtime.
4. It calls the **do_calls** function for the number of passes specified on the command line, then prints the following summary of results:

```
int main(int ac, char *av[])
{
    rpc_binding_handle_t    bh;
    error_status_t          st, error_inq_st;
    idl_char                *string_binding;
    int                     pass, passes, failures = 0;
```

```
        if (ac != 3) {
            fprintf(stderr, "Usage: %s address passes\n", av[0]);
            exit(1);
        }

        passes = atoi(av[2]);

        string_binding = (idl_char *) argv[1];

        rpc_binding_from_string_binding(string_binding, &bh, &st);

             <Check for errors.>

        printf("Bound to %s\n", string_binding);

        rpc_string_free(&string_binding, &st);

             <Check for errors.>

        for (pass = 0; pass < passes; pass++) {
            printf("PASS %d\n", pass);
            failures += do_calls(bh, passes);
        }

        printf("Summary: %d passes, %d failures\n", passes, failures);
}
```

After each call to an RPC runtime routine, the client program checks the returned status parameter, and if the status is not **error_status_ok**, it calls **dce_error_inq_text( )** and prints the returned error message text.

Note: Unlike production-quality software, the **binop** example does not do any exception handling. As a result, it may terminate abnormally if unexpected events occur during a remote procedure call. For example, if the **binop** server is stopped while the client is still running, the client will abort. Production-quality applications should place all remote procedure calls within the scope of a **TRY ... CATCH** exception handler, thus guarding against unexpected runtime events. Applications can also check for server failures by using the **comm_status** attribute in the ACF.

The **do_calls** function makes a series of remote procedure calls to perform the **binop_add** operation and check the sums:

```
do_calls(rpc_binding_handle_t h, int passes)
{
     idl_long_int i, n;
     char buf[100];
     int failures;

     failures = 0;

     for (i = 1; i <= CALLS_PER_PASS; i++) {
         binop_add(h, i, i, &n);
         if (n != i+i) {
             printf("Two times %ld is NOT %ld\n", i, n);
             if (failures == 0) failures = 1;
         }
     }

     return(failures);
}
```

11.4 Writing the Server Code

The following subsections describe the server program for the **binop** application.

11.4.1 Overview of the binop Server Program

The **binop** server program takes one argument and is invoked as follows:

$ **server** *protseq*

Because the **binop** client program derives a binding handle from information supplied by its users, the **binop** server program does not need to establish an entry for itself in the DCE CDS namespace. However, because the **binop** server uses dynamic endpoints that are not known to the client, the server does need to register its endpoint in the endpoint map on its host.

The **binop** server program has two user-written modules:

- The **server.c** module contains the server **main** function and performs the initialization and registration required to export the **binop** interface.
- The **manager.c** module contains the code that actually implements the **binop_add** operation.

11.4.2 The server.c Source Code

In this section, the **server.c** module is printed in several successive pieces, with an explanation preceding each piece.

Like **client.c**, the **server.c** module includes **binop.h**:

```
#include <stdio.h>
#include <dce/exc_handling.h>
#include "binop.h"
```

The server declares as external variables the manager EPV, which is defined in the **manager.c** module, and the nil UUID, which it will supply in registrations as an object UUID and object-type UUID:

```
#define MAX_CONCURRENT_CALLS 5

extern binop_v1_0_epv_t binop_v1_0_manager_epv;
extern uuid_t uuid_nil;
```

In the first part of the **main** function, the server calls **rpc_network_is_protseq_valid**( ) to check that its argument specifies a protocol sequence that is supported on its host both by the RPC runtime library and by the operating system:

```
int main(int ac, char *av[])
{
     rpc_binding_vector_p_t  bvec;
     error_status_t          st, error_inq_st;
     idl_boolean             validfamily;
```

```
    idl_char                  *string_binding;
    int                       i;

    if (ac != 2) {
        fprintf(stderr, "Usage: %s family\n", av[0]);
        exit(1);
  }

    validfamily = rpc_network_is_protseq_valid((idl_char *)av[1], &st);
    if (st != error_status_ok) {
        fprintf(stderr, "Cannot check protocol sequence - %s\n",
                dce_error_inq_text(st, &error_inq_st));
        exit(1);
    }
    if (!validfamily) {
        fprintf(stderr, "Protocol sequence %s is not valid\n", av[1]);
        exit(1);
    }
```

The server calls **rpc_server_use_protseq()** to obtain an endpoint on which to listen:

```
rpc_server_use_protseq((idl_char *)av[1], MAX_CONCURRENT_CALLS, &st);
if (st != error_status_ok) {
   fprintf(stderr, "Cannot use protocol sequence - %s\n",
          dce_error_inq_text(st, &error_inq_st));
   exit(1);
}
```

The server calls **rpc_server_register_if()**, supplying its interface specifier (defined in **binop.h**), to register its interface with the RPC runtime:

```
rpc_server_register_if(binop_v1_0_s_ifspec, &uuid_nil,
       NULL, &st);
if (st != error_status_ok) {
   printf("Cannot register interface - %s\n",
          dce_error_inq_text(st, &error_inq_st));
   exit(1);
}
```

To obtain a vector of binding handles that it can use when registering its endpoint, the server calls **rpc_server_inq_bindings()**. It then obtains, prints, and frees a string binding:

```
rpc_server_inq_bindings(&bvec, &st);
if (st != error_status_ok) {
    printf("Cannot inquire bindings - %s\n",
            dce_error_inq_text(st, &error_inq_st));
    exit(1);
}

printf("Bindings:\n");
for (i = 0; i < bvec->count; i++) {
    rpc_binding_to_string_binding(bvec->binding_h[i],
            &string_binding, &st);


    printf("%s\n", (char *)string_binding);

    rpc_string_free(&string_binding, &st);

}
```

A call to **rpc_ep_register**() registers the server endpoint in the local endpoint map:

```
rpc_ep_register(binop_v1_0_s_ifspec, bvec, (uuid_vector_p_t) NULL,
        (unsigned_char_p_t) "binop version 1.0 server", &st);
```

To begin listening for remote procedure call requests, the server calls **rpc_server_listen()**. This call is placed within the **TRY** of a **TRY, CATCH_ALL, ENDTRY** sequence so that if the server receives a signal while it is listening, it can unregister its interface and its endpoint before it exits, as follows:

```
TRY {

    printf("Listening...\n");
    rpc_server_listen(MAX_CONCURRENT_CALLS, &st);
    if (st != error_status_ok)
```

```
                fprintf(stderr, "Error: %s\n",
                        dce_error_inq_text(st, &error_inq_st));

        } CATCH_ALL {

            printf("Unregistering endpoint\n");
            rpc_ep_unregister(binop_v1_0_s_ifspec, bvec,
                (uuid_vector_p_t) NULL, &st);

        } ENDTRY;
}
```

For information on the macros for exception handling, see Part 2 of this guide.

11.4.3 The manager.c Source Code

The **manager.c** module includes **binop.h**, where the EPV type **binop_v1_0_epv_t** is defined. The name of the EPV type, **binop_v1_0_epv_t**, was constructed by the IDL compiler when it compiled the **binop.idl** interface definition.

The manager module also defines the function **binop_add**, as follows:

```
#include "binop.h"

void binop_add(
    handle_t h,
    idl_long_int a,
    idl_long_int b,
    idl_long_int *c)

{
     *c = a + b;
}
```

11.5 A Sample binop Application

The following subsections describe how to build and run the **binop** programs.

11.5.1 Building the binop Programs

The client side of the **binop** application is the **client** program, which is built from the following:

- The user-written **client.c** client module
- The IDL-compiler-generated **binop_cstub.c** client stub module
- Libraries for the RPC runtime, for IDL stub support, and for the Threads facility

The server side of the **binop** application is the **server** program, which is built from the following:

- The user-written **server.c** server module
- The user-written **manager.c** manager module
- The IDL-compiler-generated **binop_sstub.c** server stub module
- Libraries for the RPC runtime, for IDL stub support, and for the Threads facility

These programs can be built by **make** with a makefile such as the following:

```
IF = binop

IDL = /opt/dce/bin/idl
IFLAGS = -keep c_source
CFLAGS = -g -I.


FROMIDL = $(IF).h $(IF)_cstub.c $(IF)_sstub.c

COBJ = $(IF)_cstub.o client.o
SOBJ = $(IF)_sstub.o server.o manager.o
```

```
default: client server

client: $(COBJ)
 $(CC) -o client $(COBJ) $(LIBS)

server: $(SOBJ)
 $(CC) -o server $(SOBJ) $(LIBS)

client.o server.o manager.o: $(IF).h

$(FROMIDL): $(IDL) $(IF).idl
 $(IDL) $(IF).idl $(IFLAGS)

clean:
 $(RM) -f $(FROMIDL) $(COBJ) $(SOBJ)
```

11.5.2 Running the binop Programs

Running the **binop** application involves starting the server program and running the client program. Before starting the server program, you need to ensure that the **rpcd** process is running on the server host. For more information, see the description of the Remote Procedure Call daemon (**rpcd**) earlier in Chapter 10, and in the *OSF DCE Administration Guide* and the **rpcd(8rpc)** reference page in the *OSF DCE Administration Reference*.

The server program might be started on an IP host named **lulu** as follows:

```
$ ./server ncadg_ip_udp

Bindings:
ncadg_ip_udp:1.2.3.4[1234]
Listening...
```

The client might then be run on another host as follows:

```
$ client ncadg_ip_udp:1.2.3.4[1234] 5

Bound to @ncadg_ip_udp:lulu[]
PASS 0
PASS 1
PASS 2
PASS 3
PASS 4
Summary: 5 passes, 0 failures
$
```

The server program can be terminated at any time by a quit signal, which on many systems can be generated by **<Ctrl-c>**:

```
$ ./server ncadg_ip_udp

Bindings:
ncadg_ip_udp:1.2.3.4[1234]
Listening...<Ctrl-c>
Unregistering endpoint
$
```

Many errors can occur when applications such as **binop** execute. In general, errors that occur when a remote procedure call executes are reported as exceptions. For example, exceptions that the client side of **binop** could raise if the server suddenly and unexpectedly halts include (but are not limited to) **rpc_x_comm_failure** and **rpc_x_call_timeout**. Other ways to respond to these errors are available through the **comm_status** and **fault_status** attributes in an interface definition or attribute configuration file. Explanations of these attributes appear in Chapter 18. Also, see Chapter 16, which explains the guidelines for error handling.

As mentioned earlier in this chapter, Part 2 of this guide contains information about the macros for exception handling. If an exception occurs that the client application does not handle, it causes the client to terminate with an error message. The client's termination could include a core dump or other system-dependent error reporting method. Detailed explanations of RPC status codes and RPC exceptions are in the **rpc_status_codes(7rpc)** reference page in the *OSF DCE Application Development Reference*.

Chapter 12

Effects of Remoteness

DCE RPC provides a call environment that behaves essentially like a local call environment. However, some special requirements are imposed on remote procedure calls by the remoteness of calling code to the called procedure. ''Remoteness'' refers to the distribution of calling and called code among different address spaces that usually reside in physically separate computers linked by communications networks. Therefore, a remote procedure call may not always behave exactly like a local procedure call.

This chapter discusses the following topics:

- Direct implications of remoteness
- Communications protocols
- Universal unique identifiers
- Binding information
- Obtaining binding information of a compatible server
- Endpoints
- Context handles
- Execution semantics

- Communications failures
- Scaling

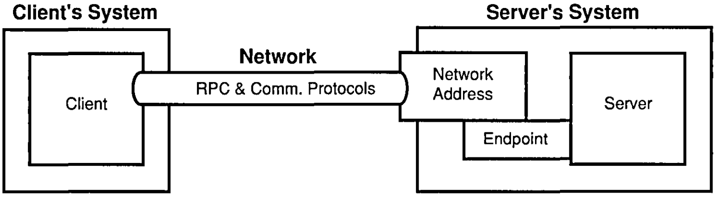
12.1 Direct Implications of Remoteness

Remoteness has the following direct implications:

- Client/server relationship: binding

  Like a local procedure call, a remote procedure call depends on a static relationship between the calling code and the called procedure. In a local application, this relationship is established by linking the calling and called code. Linking gives the calling code access to the address of each procedure to be called. Enabling a remote procedure call to go to the right procedure requires a similar relationship (called a ''binding'') between a client and a server. A binding is a temporary relationship that depends on a communications link between the client and server RPC runtimes. A client establishes a binding over a specific protocol sequence to a specific host system and endpoint. Figure 12-1 illustrates a binding.

Figure 12–1. A Binding



- Lack of shared memory

  The calling code and called remote procedure reside in different address spaces, generally on separate systems. The calling and called code cannot share global variables or other global program states such as open files. All data shared between the caller and the called remote procedure must be specified as procedure parameters. Unlike a local procedure call that commonly uses the call-by-reference passing mechanism for input/output parameters, remote procedure calls with input/output parameters have copy-in/copy-out semantics due to the differing address spaces of the calling and called code. These two passing mechanisms are only slightly different, and most procedure calls are not sensitive to the differences between call-by-reference and copy-in/copy-out semantics.

- Independent failure

  Distributing a calling program and the called procedures to physically separate machines increases the complexity of procedure calls. Remoteness introduces issues such as a remote system crash, communications links, naming and binding issues, security problems, and protocol incompatibilities. Such issues can require error handling that is unnecessary for local procedure calls. Also, as with local procedure calls, remote procedure calls are subject to execution errors that arise from the procedure call itself.

12.2 Communications Protocols

A communications link depends on a set of communications protocols. A communications protocol is a clearly defined set of operational rules and procedures for communications.

Communications protocols include a transport protocol (from the Transport Layer of the OSI network architecture) such as the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP); and the corresponding network protocol (from the OSI Network Layer) such as the Internet Protocol (IP).

For an RPC client and server to communicate, their RPC runtimes must use at least one identical communications protocol, including a common RPC protocol, transport protocol, and network protocol. An RPC protocol is a communications protocol that supports the semantics of the DCE RPC API and runs over specific combinations of transport and network protocols. DCE RPC provides two RPC protocols: the connectionless RPC protocol and the connection-oriented RPC protocol.

- Connectionless (Datagram) RPC protocol

  This protocol runs over a connectionless transport protocol such as UDP. The connectionless protocol supports broadcast calls.

- Connection-oriented RPC protocol

  This protocol runs over a connection-oriented transport protocol such as TCP.

Each binding uses a single RPC protocol and a single pair of transport and network protocols. Only certain combinations of communications protocols are functionally valid (are actually useful for interoperation); for instance, the RPC connectionless protocol cannot run over connection-oriented transport protocols such as TCP. DCE RPC supports the following combinations of communications protocols:

- RPC connection-oriented protocol over the Internet Protocol Suite, Transmission Control Protocol (TCP/IP)
- RPC connectionless protocol over the Internet Protocol Suite, User Datagram Protocol (UDP/IP)

12.3 Universal Unique Identifiers

A Universal Unique Identifier (UUID) is a hexadecimal number. Each UUID contains information, including a timestamp and a host identifier.

Applications use UUIDs to identify many kinds of entities. DCE RPC identifies several kinds of UUIDs, according to the kind of entities each identifies:

- Interface UUID

  A UUID that identifies a specific RPC interface. An interface UUID is declared in an RPC interface definition and is a required element of the interface. For example:

  ```
  uuid(2FAC8900-31F8-11CA-B331-08002B13D56D),
  ```

- Object UUID

  A UUID that identifies an entity for an application; for example, a resource, a service, or a particular instance of a server. An application defines an RPC object by associating the object with its own UUID known as an object UUID. The object UUID exists independently of the object, unlike an interface UUID. If different servers offer the same RPC object, the servers typically use different object UUIDs to identify it. A server usually generates UUIDs for its objects as part of initialization. A given object UUID is meaningful only while a server is offering the corresponding RPC object to clients.

Note: To distinguish a specific use of an object UUID, a UUID is sometimes labeled for the entity it identifies. For example, an object UUID that is used to identify a particular instance of a server is known as an instance UUID.

- Type UUID

  A UUID that identifies a class of RPC objects and an associated manager (the set of remote procedures that implements an RPC interface for objects of that type).

Servers can create object and type UUIDs by calling the **uuid_create**( ) routine.

12.4 Binding Information

In general terms, binding information is information about one or more potential bindings. Binding information includes a set of information that identifies a server to a client or a client to a server. Each instance of binding information contains all or part of a single address. The RPC runtime maintains binding information for RPC servers and clients. To make a specific instance of locally maintained binding information available to a given server or client, the runtime creates a local reference known as a binding handle. Servers and clients use binding handles to refer to binding information in runtime calls or remote procedure calls. A server obtains a complete list of its binding handles from its RPC runtime. A client obtains one binding handle at a time from its RPC runtime.

Binding information includes the following components:

- A protocol sequence: A valid combination of communications protocols represented by a character string. Each protocol sequence typically includes a network protocol, a transport protocol, and an RPC protocol that works with them.

  An RPC server tells the runtime which protocol sequences to use when listening for calls to the server, and its binding information contains those protocol sequences.

- Network addressing information: Includes the network address and the endpoint of a server.
  - The network address identifies a specific host system on a network. The format of the address depends on the network protocol portion of the protocol sequence.
  - The endpoint acts as the address of a specific server instance on the host system. The format of the endpoint depends on the transport protocol portion of the protocol sequence. For each protocol sequence a server instance uses, it requires a unique endpoint. A given endpoint can be used by only one server per system, on a first-come, first-served basis.
- Transfer Syntax: The server's RPC runtime must use a transfer syntax that matches one used by the client's RPC runtime.
- RPC protocol version numbers: The client and server runtimes must use compatible versions of the RPC protocol specified by the client in the protocol sequence. The major version number of the RPC protocol used by the server must equal the specified major version number. The minor version number of the RPC protocol used by the server must be greater than or equal to the specified minor version number.

12.4.1 Server Binding Information

Binding information for a server is known as server binding information. A binding handle that refers to server binding information is known as a server binding handle. The use of server binding handles differs on servers and clients.

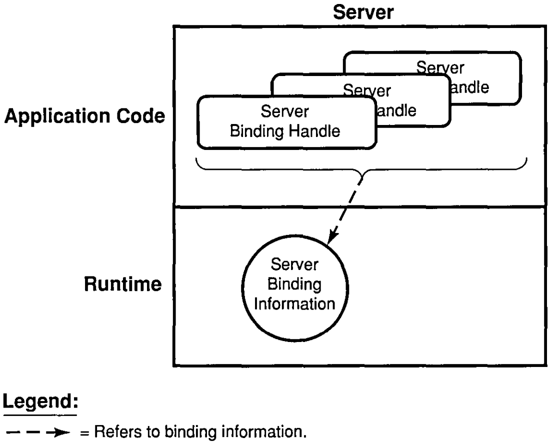
- On a server

  Servers use a list of server binding handles. Each represents one way to establish a binding with the server. Before exporting binding information to a namespace, a server tells the RPC runtime which RPC protocol sequences to use for the RPC interfaces the server supports. For each protocol sequence, the server runtime creates one or more server binding handles. Each server binding handle refers to binding information for a single potential binding, including a protocol sequence, a network (host) address, an endpoint (server address), a transfer syntax, and an RPC protocol version number.

A server obtains a list of its server binding handles from the RPC runtime. The server uses this list for asking the RPC runtime to perform operations that involve binding information such as exporting it to a server entry in the namespace or registering endpoints.

Figure 12-2 illustrates how a server uses server binding handles to refer to all of its own binding information.

Figure 12–2. Server Binding Information and Binding Handles on a Server

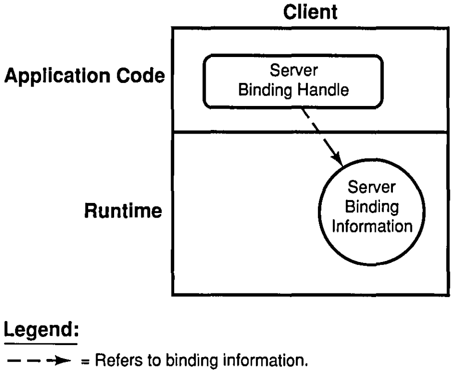


- On a client

  A client uses a single server binding handle that refers to the server binding information the client needs for making one or more remote procedure calls to a given server (see Figure 12-2). Server binding information on a client contains binding information for one potential binding.

  Figure 12-3 illustrates how a client uses a server binding handle to refer to server binding information from which the client establishes a particular binding with a specific server.

Figure 12–3. Server Binding Information and a Binding Handle on a Client


Client
Application Code
Server Binding Handle
Runtime
Server Binding Information
Legend:
– – → = Refers to binding information.

On a client, server binding information always includes a protocol sequence and the network address of the server's host system. However, sometimes a client obtains binding information that lacks an endpoint, resulting in a partially bound binding handle. A partially bound binding handle corresponds to a system, but not to a particular server instance. When a client makes a remote procedure call using a partially bound binding handle, the client runtime gets an endpoint either from the interface specification or from the endpoint map on the server's system. Adding the endpoint to the server binding information results in a fully bound binding handle, which contains an endpoint and corresponds to a specific server instance.

12.4.2 Defining a Compatible Server

Compatible binding information identifies a server whose communications capabilities (RPC protocol and protocol major version number, network and transport protocols, and transfer syntax) are compatible with those of the client. Compatible binding information is sufficient for establishing a binding. However, binding information is insufficient for ensuring that the binding is to a compatible server; that is, a server that also offers the requested RPC interface and RPC object (if any).

When requesting a binding, a client imposes requirements on its RPC runtime that ensure that the binding is with a compatible server. This additional information includes an RPC interface identifier and an object UUID, as follows:

- Interface identifier: The interface UUID and version numbers of an RPC interface:

  — Interface UUID: The interface UUID, unlike the interface name, clearly identifies the RPC interface across time and space.

  — Interface version number: The combined major and minor version numbers identify one generation of an interface.

    Version numbers allow multiple versions of an RPC interface to coexist. Strict rules govern valid changes to an interface and determine whether different versions of an interface are compatible. For a description of these rules, see Chapter 17 on IDL syntax and usage.

    The runtime uses the version number of an RPC interface to decide whether the version offered by a given server is compatible with the version requested by a client. The offered and requested interface are compatible under the following conditions:

    — The interface requested by the client and the interface offered by the server have the same major version number.

    — The interface requested by the client has a minor version number less than or equal to that of the interface offered by the server.

- An object UUID: A Universal Unique Identifier that identifies a particular object.

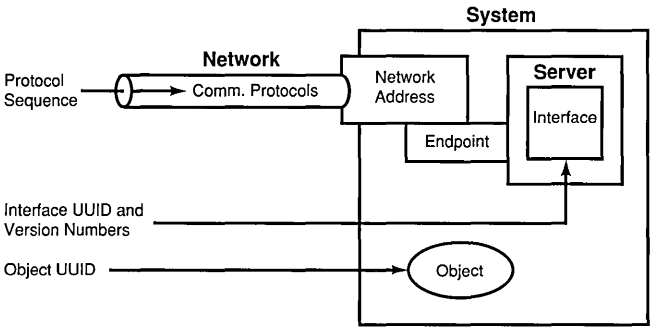
  An object is a distinct computing resource, such as a particular database, a specific RPC service that a remote procedure can access, and so on; for example, personal calendars are RPC objects to a calendar service. Accessing an object requires including its object UUID with the binding information used for establishing a binding. A client can request a specific RPC object when requesting new binding information, or the client can ask the runtime to associate an object UUID with binding information the client already has available.

  Sometimes the object UUID is the nil UUID, which contains only zeros, `00000000-0000-0000-0000-000000000000`; when calling an RPC runtime routine, you can represent the nil UUID by specifying **NULL**. In this case, the object UUID does not represent any object. Often, however, the object UUID represents a specific RPC object and is a non-nil value. To create a non-nil object UUID, a server calls the **uuid_create**( ) routine, which returns a non-nil UUID that the server then associates with a particular object.

If a client requests a non-nil object UUID, the client runtime uses that UUID as one of the criteria for a compatible server. When searching the namespace for server binding information, the client runtime looks for the requested interface identifier and object UUID. The endpoint map service uses this same information to help identify a map element corresponding to a compatible server.

Figure 12-4 illustrates the aspects of a server and its system that is identified by the client's server binding information, requested interface identifier, and requested object UUID.

Figure 12–4. Information Used to Identify a Compatible Server



12.4.3 Obtaining Binding Information of a Compatible Server

When a client initiates a series of related remote procedure calls, the RPC runtime tries to establish a binding, which requires the address of a compatible server. Establishing a binding requires that the client possess binding information of a compatible server.

A compatible server is a server that meets the following criteria:

- Offers the requested RPC interface
- Offers the requested RPC object (if any)
- Shares a common communications environment with the client; that is, the client and server RPC runtimes must support the following:
    - At least one common pair of transport and network protocols such as UDP/IP or TCP/IP
    - At least one common transfer syntax such as NDR
    - The same DCE RPC protocol (connection-oriented or connectionless protocol) and RPC protocol major version number

An RPC client can use compatible binding information obtained from either a namespace or from a string representation of the binding information.

- A namespace

  Usually, a server exports binding information for one or more of its interface identifiers and its object UUIDs, if any, to an entry in a namespace. The namespace is provided by a directory service such as the DCE Cell Directory Service (CDS). The namespace entry to which a server exports binding information is known as a server entry.

  To learn about a server that offers a given RPC interface and object, if any, a client can import binding information from a server entry belonging to that server. A client can delegate the finding of servers from the namespace to a stub. In this case, if a binding is accidentally broken, the RPC runtime automatically tries to establish a new binding with a compatible server.

- A string representation of binding information

  Occasionally, a client can receive binding information in the form of a string (also known as a ''string binding''). The client can receive a string binding (or the information to compose a string binding) from many sources; for example, an application-specific environment variable, a file, or the application user. The client must call the RPC runtime to convert a string binding to a binding handle. The RPC runtime stores the binding information from the string binding and creates a binding handle that refers to the binding information. The runtime returns this binding handle to the client to use for remote procedure calls.

Establishing a binding can also involve requesting an endpoint from the RPC daemon of the server's system.

12.4.3.1 Format of String Representations of Binding Information

String representations of binding information have several possible components. The binding information can include an RPC protocol sequence, a network address, and an endpoint. The protocol sequence is mandatory; the endpoint is optional; and for a server on the client's system, the network address is optional. Also, a string binding optionally associates an object UUID with the binding information.

The string bindings have the following format:

obj-uuid@*rpc-protocol-seq*:*network-addr*[*endpoint*,*option-name*=*opt-value*...]

or

obj-uuid@*rpc-protocol-seq*:*network-addr*[**endpoint**=*endpoint*,*option-name*=*opt-value*...]

The following example string binding contains all possible components:

```
B07122E2-83DF-11C9-BE29-08002B1110FA@ncacn_ip_tcp:16.20.15.25[2001]
```

The following example string binding contains only the protocol sequence and network address:

```
ncacn_ip_tcp:16.20.15.25
```

For more information about this format, see the RPC introduction reference page, **intro(3rpc)**, in the *OSF DCE Application Development Reference*.

12.4.3.2 Evaluation of Mechanisms for Obtaining Binding Information

Sometimes, string bindings are useful, for example, when developing and testing an application. However, string bindings are inappropriate as the principal way of providing binding information to clients. For moderate to large environments and for small environments that may expand, applications should use the directory service to advertise binding information.

Table 12-1 summarizes the distinctions between using string bindings and using the Directory Service.

Table 12–1. Assessment of Mechanisms for Obtaining Binding Information

String Bindings	**Directory Service**
Convenient for small RPC environments; for example, a development or test environment. Eliminates the overhead of installing, configuring, and understanding a directory service.	Convenient for large RPC environments. Initial overhead of understanding and configuring a directory service is balanced by easier management over time.
Requires mutual management.	Management of data in a directory service is more automated.
Suitable only for static end-user environments.	Effective in dynamic end-user environments.
Binding information has no user-friendly name associated with it. A complicated string binding must be communicated out of band (in a file, on paper, in a mail message, and so forth) to every client that wants to use the binding information. If the binding information is modified, all the users must update their copy of the information manually.	Binding information is stored in a named server entry. Data can be dynamic. Servers can automatically place their binding information in the namespace. Changes in binding information are made once by a server or administrator and then propagated automatically by the directory service to the replicas of the data.

String Bindings	Directory Service
Decentralized ad hoc administration of binding information. Limited ability to control access to the information.	Centralized administration of data in a namespace. Sophisticated access control is possible.
Limited selection of servers and services; limited ability to control or influence selection.	Supports searching for and choosing services based on an interface identifier and object UUID. Clients access data by specifying an entry name. NSI groups and profiles in directory service entries provide search paths for importing binding information.

12.4.4 Client Binding Information

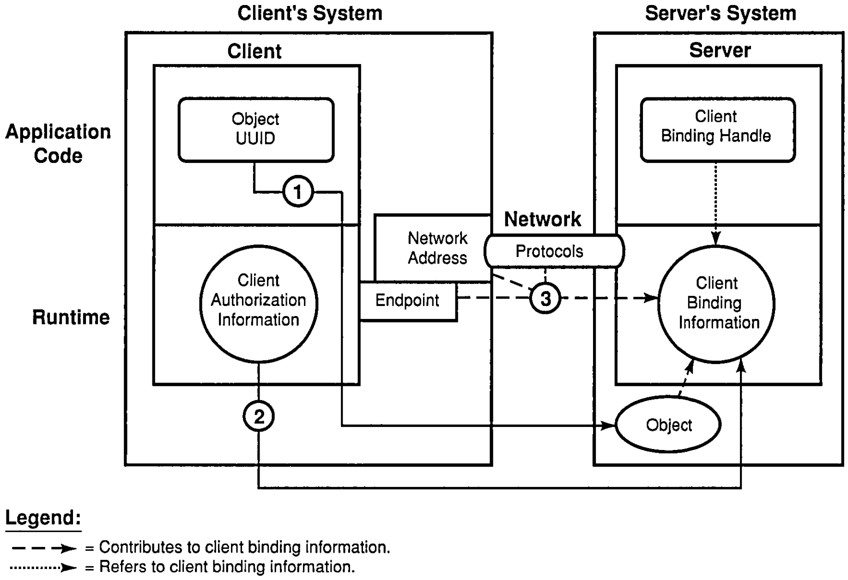
When making a remote procedure call, the client runtime provides information about the client to the server runtime. This information, known as client binding information, includes the following information:

- The address where the call originated
- The RPC protocol used by the client for the call
- The object UUID that a client requests
- The client authentication information (if present)

The server runtime maintains the client binding information and makes it available to the server application by a client binding handle.

Figure 12-5 illustrates the relationships between what a client supplies when establishing a binding and the corresponding client binding information.

Figure 12–5. Client Binding Information Resulting from a Remote Procedure Call


Client's System
Server's System
Client
Server
Application Code
Runtime
Object UUID
Client Binding Handle
1
Network
Network Address
Protocols
Client Authorization Information
Endpoint
3
Client Binding Information
2
Object
Legend:
– – → = Contributes to client binding information.
··········→ = Refers to client binding information.

The callouts in the figure refer to the following:

1. The requested object UUID, which may be the nil UUID
2. Client authentication information, which is optional
3. The address from which the client is making the remote procedure call, which the communications protocols supply to the server

A server application can use the client binding handle to ask the RPC runtime about the object UUID requested by a client or about the client's authentication information.

12.5 Endpoints

An endpoint is the address of a specific server instance on a host system. The following kinds of endpoints exist: well-known endpoints and dynamic endpoints.

12.5.1 Well-Known Endpoints

A well-known endpoint is a preassigned stable address that a server can use every time it runs. Well-known endpoints typically are assigned by a central authority responsible for a transport protocol; for example, the ARPANET Network Information Center assigns endpoint values for the IP family of protocols. If you use well-known endpoints for a server, you should register them with the appropriate authority.

Well-known endpoints can be declared for an interface (in the interface declaration) or for a server instance, as follows:

- Any interface definition can store one or more endpoints, along with the RPC protocol sequence corresponding to each endpoint.

  When compiling an interface, the IDL compiler stores each combination of endpoint and protocol sequence in the interface specification. If a call is made using binding information that lacks an endpoint, the RPC runtime automatically looks in the interface specification for a well-known endpoint specified for the protocol sequence obtained from the binding information. If the interface specification contains an appropriate endpoint, the runtime adds it to the binding information.

- Alternatively, server-specific, well-known endpoints can be defined in server application code. When asking the runtime to use a given protocol sequence, the server supplies the corresponding endpoints to the RPC runtime. On a given system, each endpoint can be used by only one server at a time. If server application code contains a hardcoded endpoint or the server's installers always specify the same well-known endpoint, only one instance of the server can run per system.

When a server exports its binding information to a server entry, the export operation includes any well-known endpoints within the server binding information stored in the server entry.

12.5.2 Dynamic Endpoints

A dynamic endpoint is an endpoint that is requested and assigned at runtime. For some transport protocols, the number of endpoints is limited; for example, TCP/IP and UDP/IP use a 16-bit number for endpoints, which allows only 65,536 endpoints. When the supply of endpoints for a transport protocol is limited, the protocol ensures an adequate supply of endpoints by limiting the portion that can be reserved as well-known endpoints. A transport, on request, dynamically makes its remaining endpoints available on a first-come, first-served basis to specific processes such as RPC server instances.

When a server requests dynamic endpoints, the server's RPC runtime asks the operating system for a unique dynamic endpoint for each protocol sequence the server is using. For a given protocol sequence, the local implementation of the corresponding transport protocol provides the requested endpoints. When an RPC server with dynamic endpoints stops running, its dynamic endpoints are released.

Because of the transient nature of dynamic endpoints, NSI does not export them to a namespace; however, NSI does export the rest of the server's binding information. References to expired endpoints would remain indefinitely in server entries, causing clients to import and try, unsuccessfully, to establish bindings to nonexistent endpoints. Also, updating transient data in namespace entries impairs the performance of a directory service. Therefore, the export operation removes dynamic endpoints before adding binding information to a server entry; the exported server address contains only network addressing information. The import operation returns a partially bound binding handle. The client makes its first remote procedure call with the partially bound handle, and the endpoint map service on the server's system attempts to resolve the binding handle with the endpoint of a compatible server. To make dynamic endpoints available to clients using partially bound binding handles, a server must register its dynamic endpoints in the local endpoint map.

Note: Register all endpoints to enable administrators to view all the endpoints of RPC servers by showing the endpoint map elements. To do this, use the **rpccp show mapping** command of the RPC control program.

By using object UUIDs, a server can ensure that a client that imports a partially bound handle obtains one of a particular server's endpoints. This requires that the server do the following:

1. Specify a list of one or more object UUIDs that are unique to the server.
2. Export the list of object UUIDs.
3. Supply the list of object UUIDs to the endpoint map service when registering endpoints.
4. If the server provides different managers that implement an interface for different types of objects, the server must set the type of each object (see Section 13.4.1 on assigning types to objects.)

To request binding information for a particular server, a client specifies one of the server's object UUIDs, which is then associated with the server binding information the client uses for making a remote procedure call.

Note: If a client requests the nil object UUID when importing from a server entry containing object UUIDs, the import (or lookup) operation selects one of those object UUIDs and associates it with the imported server binding information. This object UUID guarantees that the call goes to the server that exported the binding information and object UUID to the server entry.

12.6 Context Handles

Server application code can store information it needs for a particular client, such as the state of RPC the client is using, as part of a client context. During a series of remote procedure calls, the client may need to refer to the client context maintained by a specific server instance. To provide a client with a means of referring to its client context, the client and server pass back and forth an RPC-specific parameter called a ''context handle.'' A context handle is a reference to the server instance and the client context of a particular client. A context handle ensures that subsequent remote procedure calls from the client can reach the server instance that is maintaining context for the client.

On completing the first procedure in a series, the server passes a context handle to the client. The context handle identifies the client context that the server uses for subsequent operations. The client stores the handle and can return it unchanged in subsequent calls to the same server. Using the handle, the server finds the context and provides it to the called remote procedure.

The server maintains the client context for a client until one of the following occurs:

- The client calls an operation that terminates use of the context.
- The server crashes.
- Communications are lost and the server's runtime invokes a context rundown procedure.

12.7 Execution Semantics

Execution semantics identify the ability of a procedure to execute more than once during a given remote procedure call. The communications environment that underlies remote procedure calls affects their reliability. A communications link can break for a variety of reasons such as a server termination, a remote system crash, a network failure, and so forth; all invocations of remote procedures risk disruption due to communications failures. However, some procedures are more sensitive to such failures, and their impact depends partly on how reinvoking an operation affects its results.

To maximize valid outcomes for its operations, the operation declarations of an RPC interface definition indicate the effect of multiple invocations on the outcome of the operations.

Table 12-2 summarizes the execution semantics for DCE RPC calls.

Table 12–2. Execution Semantics for DCE RPC Calls

<table>
<tr><th>Semantics</th><th colspan="2">Meaning</th></tr>
<tr><td>**at-most-once**</td><td colspan="2">The operation must execute either once, partially, or not at all; for example, adding or deleting an appointment from a calendar can use **at-most-once** semantics. This is the default execution semantics for remote procedure calls.</td></tr>
<tr><td rowspan="4">**idempotent**</td><td colspan="2">The operation can execute more than once; executing more than once using the same input arguments produces identical outcomes without undesirable side effects; for example, an operation that reads a block of an immutable file is **idempotent**. DCE RPC supports **maybe** semantics and **broadcast** semantics as special forms of **idempotent** operations.</td></tr>
<tr><td>**Semantics**</td><td>**Meaning**</td></tr>
<tr><td>**maybe**</td><td>The caller neither requires nor receives any response or fault indication for an operation, even though there is no guarantee that the operation completed. An operation with **maybe** semantics is implicitly idempotent and must lack output parameters.</td></tr>
<tr><td>**broadcast**</td><td>The operation is always broadcast to all host systems on the local network, rather than delivered to a specific system. An operation with **broadcast** semantics is implicitly **idempotent**; **broadcast** semantics are supported only by connectionless protocols.</td></tr>
</table>

With the RPC communications protocols, a **maybe** or **broadcast** call lacks guarantees; an **idempotent** call guarantees that the data for a remote procedure call is received and processed in order zero or more times; and an **at-most-once** call guarantees that the call data is received and processed in order zero or one time.

12.8 Communications Failures

If a server detects a communications failure during a remote procedure call, the server runtime attempts to terminate the now orphaned call by sending a cancel to the called procedure. A cancel is a mechanism by which a client thread of execution notifies a server thread of execution (the canceled thread) to terminate as soon as possible. A cancel sent by the RPC runtime after a communications failure initiates orderly termination for a remote procedure call. (For a brief discussion of how cancels work with remote procedure calls, see Chapter 14; for detailed information, see Part 2 of this guide.)

Applications that use context handles to establish a client context require a context rundown procedure to enable the server to clean up client context when it is no longer needed. A type declaration for the context rundown procedure is declared in the interface definition; this ensures that the stub knows about the procedure in the server application code. If a communications link with a client is lost while a server is maintaining context for the client, the RPC runtime will inform the server to invoke the context rundown procedure.

12.9 Scaling

Unlike local applications, RPC applications require network resources, which are possible bottlenecks to scaling an RPC application. RPC clients and servers require network resources that are not required by local programs. The main network resources to consider are network bandwidth, endpoints, network descriptors (the identifiers of potential network channels such as UNIX sockets), kernel buffers, and for a connection-oriented transport, the connections. Also, RPC applications place extra demands on system resources such as memory buffers, various quotas, and the CPU.

The number of remote procedure calls that a server can support depends on various factors, such as the following:

- The resources of the server and the network
- The requirements of each call

- The number of calls that can be concurrently offered at some level of service
- The performance requirements

An accurate analysis of the requirements of a given server involves detailed work load and resource characterization and modeling techniques. Although measurement of live configurations under load will offer the best information, general guidelines apply. You should consider the connection, buffering, bandwidth, and CPU resources as the most likely RPC bottlenecks to scaling. Use these application requirements to scale resources.

Many system implementations limit the number of network connections per process. This limit provides an upper bound on the number of clients that can be served concurrently using the connection-oriented protocol. Some UNIX derived systems set this limit at 64. However, except for applications that use context handles, the connection-oriented RPC runtime allows pooling of connections. Pooling permits simultaneously supporting more clients than the maximum number of connections, provided they do not all make calls at the same instant and occasionally can wait briefly.

Chapter 13

Basic DCE RPC Runtime Operations

This chapter introduces a number of basic DCE RPC directory service, communications, and authentication operations and discusses major usage issues important for developing DCE RPC applications.

Note: DCE RPC Version 1.0 supports authenticated RPC for the connectionless (datagram) protocol only.

This chapter discusses the following topics:

- Overview of basic operations
- Basic tasks of an unauthenticated remote procedure call
- Basic runtime routines
- Server initialization tasks
- Methods for managing bindings
- Obtaining server binding handles
- Using authenticated RPC

13.1 Overview of Basic Operations

This section summarizes the major concerns of RPC communications, NSI, and authenticated RPC, as follows:

- Basic operations of RPC communications protocols

  The DCE RPC runtime provides the following communications operations for RPC applications:

  — Managing communications for RPC applications

    As part of server initialization, a server sets up its communications capabilities by a series of calls to the RPC runtime. These runtime calls register the server's RPC interfaces, tell the RPC runtime what combination of communications protocols to use for the server, and register the endpoints of the server for each of its interfaces. After completing these and any other initialization tasks, the server tells the runtime to begin listening for incoming calls.

  — Managing binding information

    A variety of communications operations allow servers to access and manipulate binding information. In addition, a set of communications operations enables applications to manipulate string representations of binding information (string bindings).

- Basic operations of the RPC Name Service Interface (NSI)

  The NSI routines perform operations on a namespace for RPC applications. The fundamental operations include the following:

  — Creating and deleting entries in namespaces

  — Exporting

    A server uses the NSI export operation to place binding information associated with its RPC interfaces and objects into the namespace used by the RPC application.

  — Importing

    Clients can search for exported binding information associated with an interface and object using the NSI import operation or lookup operation. These two operations are collectively referred to as the NSI search operations.

- — Unexporting

  The unexport operation enables a server to remove some or all of its binding information from a server entry.

- — Managing information in a namespace

  Applications use the NSI interface to place information about server entries into a namespace and to inquire about and manage that information.

For information about the Cell Directory Service, see the *OSF DCE Administration Guide.*

- Basic operations of authenticated RPC

The authenticated RPC routines provide a mechanism for establishing secure communications between clients and servers.

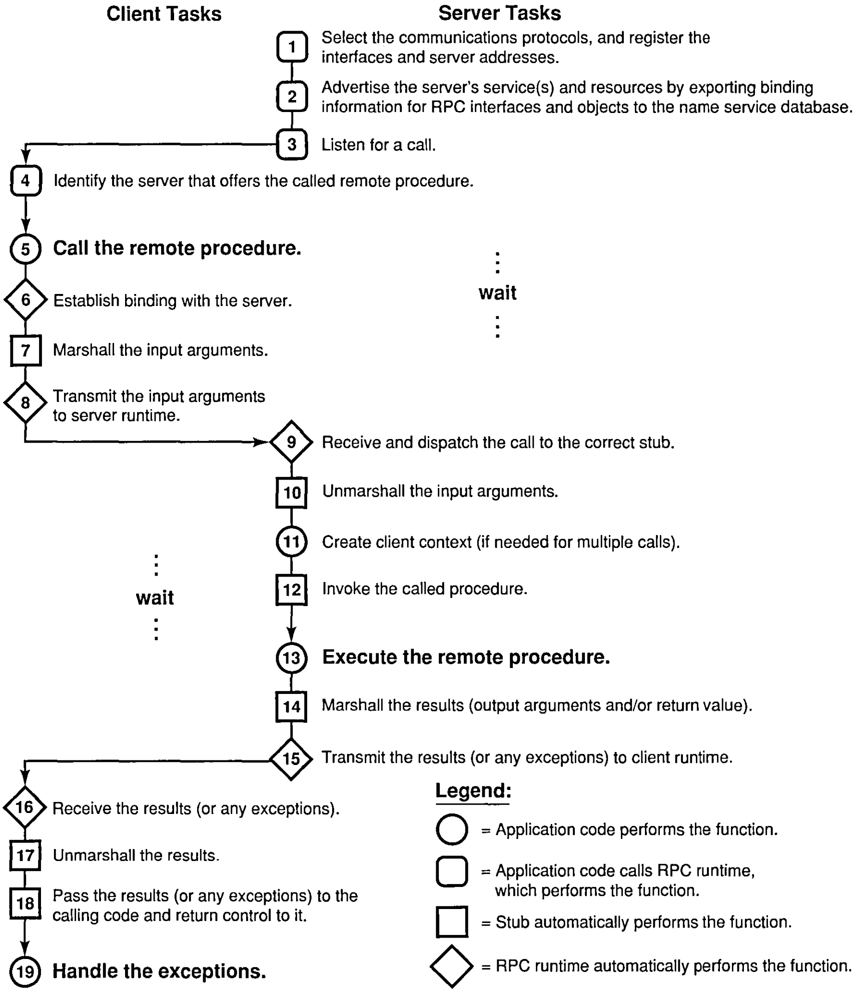
To engage in authenticated RPC, a client and server must agree on the authentication service to be used. The server's responsibility is to register its principal name and the authentication service to be supported with the RPC runtime. The client's responsibility is to establish the authentication service, a given protection level, and an authorization service for the server binding handle. The protection level determines the degree of protection applied to individual messages between the client and server. The authorization service determines the form in which the client's credentials will be presented to the server (for access checking).

Once authenticated RPC has been established between a client and server, the client issues remote procedure calls in the usual fashion, with all authentication and protection being handled by the DCE Security Service component and the RPC runtime.

13.2 Basic Tasks of an Unauthenticated Remote Procedure Call

Figure 13-1 summarizes the basic tasks of an unauthenticated remote procedure call. Use the legend to the figure to learn what portion of an RPC application is concerned with each task.

Figure 13–1. Basic Tasks of a Remote Procedure Call


Client Tasks
Server Tasks
1 Select the communications protocols, and register the interfaces and server addresses.
2 Advertise the server's service(s) and resources by exporting binding information for RPC interfaces and objects to the name service database.
3 Listen for a call.
4 Identify the server that offers the called remote procedure.
5 Call the remote procedure.
wait
6 Establish binding with the server.
7 Marshall the input arguments.
8 Transmit the input arguments to server runtime.
9 Receive and dispatch the call to the correct stub.
10 Unmarshall the input arguments.
11 Create client context (if needed for multiple calls).
wait
12 Invoke the called procedure.
13 Execute the remote procedure.
14 Marshall the results (output arguments and/or return value).
15 Transmit the results (or any exceptions) to client runtime.
16 Receive the results (or any exceptions).
17 Unmarshall the results.
18 Pass the results (or any exceptions) to the calling code and return control to it.
19 Handle the exceptions.
Legend:
○ = Application code performs the function.
▢ = Application code calls RPC runtime, which performs the function.
□ = Stub automatically performs the function.
◇ = RPC runtime automatically performs the function.

13.3 Basic Runtime Routines

Table 13-1 relates several of the RPC runtime operations to specific routines or classes of routines.

Table 13–1. Runtime Routines Associated with Basic Runtime Operations

Description of Operation	Usage	Routine Name(s)
Communications Routines		
Setting the type of an RPC object with the RPC runtime	Server	**rpc_object_set_type( )**
Registering RPC interfaces	Server	**rpc_server_register_if( )**
Selecting RPC protocol sequences	Server	**rpc_network_inq_protseqs( )** and **rpc_server_use_*protseq*_...( )**
Obtaining server binding handles	Server	**rpc_server_inq_bindings( )**
Registering endpoints	Server	**rpc_ep_register( )** and **rpc_ep_register_no_replace( )**
Unregistering endpoints	Server	**rpc_ep_unregister( )**
Listening for calls	Server	**rpc_server_listen( )**
Manipulating string representations of binding information (string bindings)	Client	**rpc_binding_from_string_binding( )**
	Client, Server	**rpc_binding_to_string_binding( ),** **rpc_string_binding_compose( ),** and **rpc_string_binding_parse( )**
Changing the RPC object in server binding information	Client	**rpc_binding_set_object( )**
Converting a client binding handle to a server binding handle	Server	**rpc_binding_server_from_client( )**

Description of Operations	Usage	Routine Name(s)
Name Service Interface Routines		
Exporting binding information to a namespace	Server	**rpc_ns_binding_export( )**
Searching a namespace for binding information	Client	**rpc_ns_binding_import_...( ),** **rpc_ns_binding_lookup_...( ),** and **rpc_ns_binding_select( )**
Authentication Routines		
Authentication and authorization	Server, Client	**rpc_*auth...( )**

13.4 Server Initialization Tasks

Before an RPC server can receive any remote procedure calls, the server initializes itself by calling the RPC runtime routines. The server initialization code, written by the application developer, varies among servers. However, every server must set up its communications capabilities, which minimally involves all or most of the following tasks:

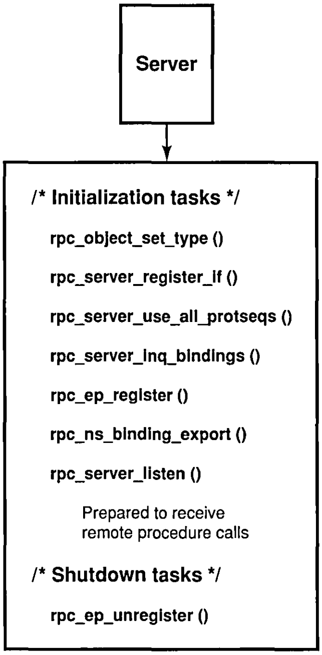
- Assigning types to objects
- Registering at least one interface
- Specifying which protocol sequences the server will use
- Obtaining a list of references to a server's binding information (a list of binding handles)
- Registering endpoints

  Note: Before stopping, a server should unregister these endpoints.

- Exporting binding information to a server entry or entries in the namespace
- Listening for remote procedure calls

Figure 13-2 illustrates the calls a server makes to accomplish these basic initialization tasks.

Figure 13–2. Typical Initialization Calls of an RPC Server


Server
/* Initialization tasks */
rpc_object_set_type ()
rpc_server_register_if ()
rpc_server_use_all_protseqs ()
rpc_server_inq_bindings ()
rpc_ep_register ()
rpc_ns_binding_export ()
rpc_server_listen ()
Prepared to receive remote procedure calls
/* Shutdown tasks */
rpc_ep_unregister ()

13.4.1 Assigning Types to Objects

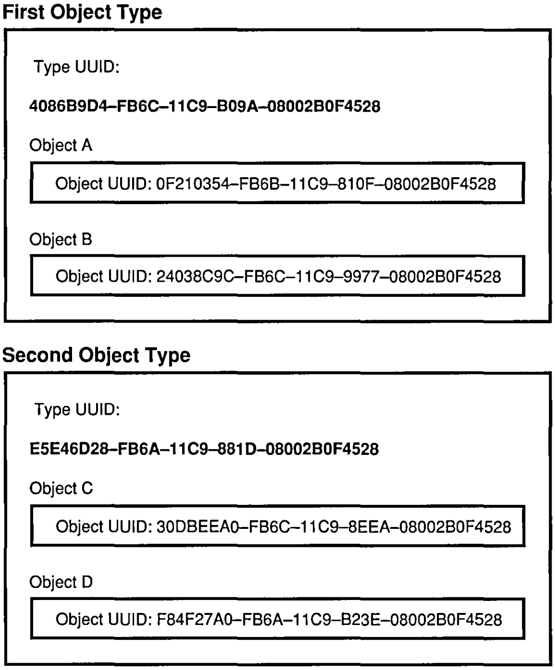
A ‘‘type’’ is a mechanism for associating a set of RPC objects and the manager whose remote procedures implement an RPC interface for those objects. Types allow an application to cluster objects, such as computing resources, according to any relevant criteria. For example, a single accounting interface can be implemented to operate on accounting databases that contain equivalent information but that are formatted differently; each database format represents a distinct type.

By default, objects have the nil type. Only a server that implements different managers for different objects or sets of objects needs to type classify its RPC objects. To type classify an object, a server associates the object UUID of the object with a single type5.25ID by calling the **rpc_object_set_type**( ) procedure separately for each object. To create a type UUID, a server calls the **uuid_create( )** routine. A server uses a single type UUID for every object of an identical type.

To simultaneously offer alternative implementations of an RPC interface for different types of objects, a server uses alternative managers. Servers that implement each of their interfaces with only one manager can usually avoid the tasks associated with assigning object types. However, when a server offers multiple managers, each manager must be dedicated to operating on a separate type of object. In this case, a server must classify some or all of its objects into types; for example, a calendar application that specifies one non-nil type UUID for departmental calendars and another non-nil type UUID for personal calendars.

Figure 13-3 illustrates how objects correspond to types. The server associates each object with a particular type. When the server receives an incoming call that specifies an object UUID, the server dispatches the call to the manager for the type to which the object belongs.

Figure 13–3. How Objects Correspond to Types


First Object Type
Type UUID:
4086B9D4–FB6C–11C9–B09A–08002B0F4528
Object A
Object UUID: 0F210354–FB6B–11C9–810F–08002B0F4528
Object B
Object UUID: 24038C9C–FB6C–11C9–9977–08002B0F4528
Second Object Type
Type UUID:
E5E46D28–FB6A–11C9–881D–08002B0F4528
Object C
Object UUID: 30DBEEA0–FB6C–11C9–8EEA–08002B0F4528
Object D
Object UUID: F84F27A0–FB6A–11C9–B23E–08002B0F4528

For information on how a type is used to select a manager for an incoming call, see Chapter 14 on advanced DCE RPC topics.

13.4.2 Registering Interfaces

To register an interface, a server calls the **rpc_server_register_if( )** routine to tell the RPC runtime about a specific RPC interface. Registering an interface informs the runtime that the server is offering that interface and makes it available to clients. A server can register any number of interfaces with the RPC runtime by calling the **rpc_server_register_if( )** routine once for each set of procedures, or manager, that implements an interface.

To offer more than one manager for an interface, a server must register each manager separately.

When registering an interface, the server provides the following information:

- Interface specification

  This is a reference to information about an RPC interface as offered by its server stub. The DCE IDL compiler generates an interface specification as part of the stub code. For a specific version of an interface, all managers use the same interface specification. Information in an interface specification that concerns application developers includes the following:

  — The interface identifier (UUID and major and minor version numbers)

  — The supported transfer syntaxes

  — A list of any well-known endpoints (and their associated protocol sequences) specified in the interface definition (**.idl**) file

  — The interface's default manager endpoint vector (manager EPV), if present

  A manager EPV is a list of the addresses (the entry points of the remote procedures provided by the manager). A manager EPV must contain exactly one entry point for each procedure defined in the interface definition. A default manager EPV, constructed using the operation names of the interface definition, is typically generated for stubs by the DCE IDL compiler (a compiler option can suppress this feature).

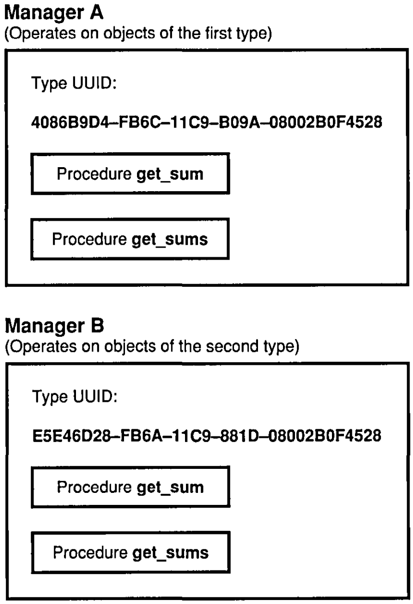
- The manager EPV and type for the interface

  A server can register a given interface more than once by specifying a different manager EPV and type each time. The server can use the default manager EPV only once, and only for a manager that uses the procedure names declared in the interface definition. For any additional manager of the RPC interface, the server must create and register a unique manager EPV. Also, each manager must be associated with a distinct type of object. To associate a type with a manager EPV, a server passes the type UUID of the associated type of object to the RPC runtime.

The exact operations performed by managers can vary with the type of object on which each manager operates. For example, a queue-management interface may be implemented to manage print queues as objects in one case and to manage batch queues as objects in another.

Figure 13-4 illustrates the use of type UUIDs to identify the types of two managers. These types correspond to the two object types illustrated in Figure 13-3.

Figure 13–4. Manager Types


Manager A
(Operates on objects of the first type)
Type UUID:
4086B9D4–FB6C–11C9–B09A–08002B0F4528
Procedure **get_sum**
Procedure **get_sums**
Manager B
(Operates on objects of the second type)
Type UUID:
E5E46D28–FB6A–11C9–881D–08002B0F4528
Procedure **get_sum**
Procedure **get_sums**

For a discussion of how a server uses object and manager types when routing incoming calls, see Section 14.4.4, which tells you how to select the appropriate manager.

13.4.3 Selecting RPC Protocol Sequences

For an overview of protocol sequences, see the following subsections on binding information.

13.4.3.1 Inquiring About Supported Protocol Sequences

A server can inquire about the protocol sequences that the local RPC runtime supports. The server can ask the RPC runtime for a list of all protocol sequences supported by both the RPC runtime and the operating system. After requesting a list of protocol sequences, a server is responsible for releasing the memory the runtime uses to store the requested information. The server can also inquire whether a given protocol sequence is available for receiving remote procedure calls.

13.4.3.2 Selecting Protocol Sequences

To receive remote procedure calls, a server tells the the RPC runtime to use at least one protocol sequence. For each protocol combination, the RPC runtime creates one or more binding handles. The server uses a list of these binding handles to register dynamic endpoints and to export its binding information.

As an option, an interface can contain one or more well-known endpoints, each of which is accompanied by a protocol sequence. A server can use any protocol sequence declared in an interface endpoint declaration, or the server can ignore the endpoint declarations, as long as it registers at least one endpoint.

13.4.4 Obtaining a List of Server Binding Handles

After a server passes the protocol sequences over which it will listen for remote procedure calls to the RPC runtime, the RPC runtime constructs server binding handles. Each binding handle refers to a complement of binding information that defines one potential binding; that is, a specific

RPC protocol sequence, RPC protocol major version, network address, endpoint, and transfer syntax that an RPC client can use to establish a binding with an RPC server.

Before registering endpoints or exporting binding information, a server must obtain a list of its binding handles from the RPC runtime. The server passes this list back to the runtime as an argument when registering endpoints and exporting binding information.

When a server requests a list of binding handles, the RPC runtime allocates memory for the list, and the application is responsible for freeing that memory.

13.4.5 Registering Endpoints

Servers can use well-known or dynamic endpoints with any protocol sequence.

When a server asks the runtime to use a dynamic endpoint with a protocol sequence, the runtime asks the operating system to generate the endpoint. To use the dynamic endpoints, a server must register the server's binding information, including the endpoints. For each combination of RPC interface identifier, object UUID, and binding information that the server offers, the endpoint map service creates an element in the local endpoint map.

To register an endpoint, a server places the following information into the local endpoint map:

- The RPC interface identifier, which contains the interface UUID and major and minor version numbers
- The list of binding handles for the interface
- The list of the server's object UUIDs (if any)

When a server asks the runtime to use a well-known endpoint with a protocol sequence, the runtime either uses an interface specification to look up the endpoint or receives the endpoint in a variable from the server. A server does not necessarily need to register well-known endpoints; however, by registering well-known endpoints, the server ensures that clients can always obtain them. Registration also makes the endpoints accessible to administrators, who can use the RPC control program to show the map elements of an endpoint map by using the **rpccp show mapping** command.

Servers, clients, or management modules can remove map elements from a local endpoint map by using the **rpc_ep_unregister()** routine. Servers should unregister endpoints after they stop listening. The RPC control program enables its users to remove map elements from an endpoint map by using the **rpccp remove mapping** command.

13.4.6 Making Binding Information Accessible to Clients

A server needs to make its binding information accessible to clients. Usually, a server uses the NSI export operation to place its binding information into a server entry. However, it is also possible for servers to make string bindings accessible to clients.

13.4.6.1 Using String Bindings to Provide Binding Information

While implementing and debugging a server program you may temporarily want to communicate binding information to clients using string bindings. This allows a server to establish a client/server relationship without registering endpoints in the local endpoint map or exporting binding information to a namespace.

After asking for a list of binding handles by calling the **rpc_server_inq_bindings()** routine, the server can obtain binding information from the server runtime. The server can convert each binding handle in the list into a string binding by calling **rpc_binding_to_string_binding()**. The resulting string binding is always fully bound. The server then makes some or all of its string bindings available to clients somehow; for example, by placing the string bindings in a file to be read by clients or users or both.

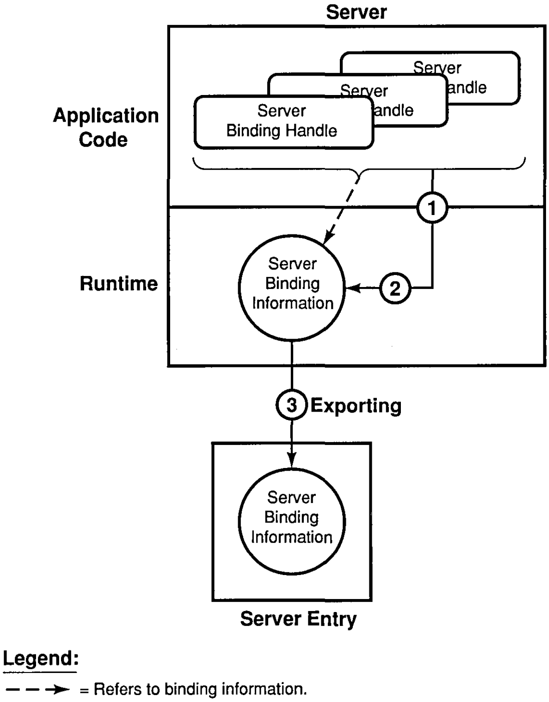
13.4.6.2 Exporting Binding Information

Servers can export binding information (and interface identifiers) or objects or both by calling the **rpc_ns_binding_export( )** routine. To export binding information associated with a given RPC interface, a server uses an interface handle. The interface handle is created by the IDL compiler as a reference to information about the interface that the compiler stores in an interface specification. To refer to binding information, the application code obtains a list of server binding handles from the RPC runtime and passes the list to the export operation. The list contains binding handles for all the protocol sequence and endpoint combinations that the server has requested; it does this by calling the use-protocol-sequence operations. However, the server can remove any of those binding handles from the list before exporting it. This enables a server to export the binding information associated with a subset of its binding handles.

To export object UUIDs, a server application must provide a list of object UUIDs for the RPC objects it offers. The server can generate these object UUIDs itself or obtain them from some application-specific source such as an object-UUID database. All object UUIDs in a given server entry are associated with every interface UUID and server address in the entry.

Figure 13-5 illustrates the use of server binding handles to refer to server binding information to be exported.

Figure 13–5. Exporting Server Binding Information



The callouts in the figure refer to the following operations:

1. The server application code calls the export operation, having previously inquired for a list of binding handles. Along with the name of a server entry, the application passes the export operation a list of server binding handles and an interface handle, a list of object UUIDs, or both.

2. The export operation uses the binding handles to identify the binding information to export.

3. The export operation places binding information, the associated interface identifier, and the associated list of object UUIDs into the designated server entry.

A server entry must belong exclusively to a server running on a given host. If there are identical, interchangeable instances of a server on the host, they can share a single set of server entries. However, if clients need to distinguish between coexisting instances of a server (for example, when each offers a different RPC object), each instance requires its own server entry.

Note: CDS data is subject to access control. To access CDS entries, you need Access Control List (ACL) permissions. Depending on the NSI operation, you need ACL permissions to the parent directory, the CDS object entry, or both. If you need ACL permissions, see your CDS administrator.

The ACL permissions are as follows:

- To create an entry, you need insert permission to the parent directory.
- To read an entry, you need read permission to the CDS object entry.
- To write to an entry, you need write permission to the CDS object entry.
- To delete an entry, you need delete permission either to the CDS object entry or to the parent directory.
- To test an entry, you need either test permission or read permission to the CDS object entry.

Note that write permission does not imply read permission.

13.4.7 Listening for Calls

When a server is ready to accept remote procedure calls, it initiates listening, specifying the maximum number of calls it can execute concurrently; it does this by calling the **rpc_server_listen()** routine. If a server allows concurrent calls, its remote procedures are responsible for concurrency control. If executing a set of remote procedures concurrently requires concurrency control and a server lacks this control, the server must allow only one call at a time.

The RPC runtime continues listening for new remote procedure calls to the server's registered interfaces until one of the following events occurs:

- Any of the server's procedures makes a local management call to stop a server from listening for future remote procedure calls.

- For applications whose servers enable clients to stop servers from listening, a client makes a remote management call to stop a server from listening for future remote procedure calls.

On receipt of a stop listening request, the RPC runtime stops accepting new remote procedure calls for all registered interfaces. However, currently executing calls are allowed to complete. After all executing calls complete, the listen operation stops listening and returns control to the server. Servers should unregister endpoints after they stop listening.

13.5 Methods for Managing Bindings

The client manages the bindings for its remote procedure calls. (For a discussion of bindings, see Chapter 12, which describes the effects of remoteness.) DCE RPC provides various methods of managing bindings for remote procedure calls. These methods include the automatic, implicit, and explicit methods. The automatic method requires the server to store binding information in server entries in a namespace; the implicit and explicit methods work with any source of binding information. These methods are described in the following text:

- Automatic method

  This is the simplest method of managing the binding for remote procedure calls. With the automatic method, the server exports its binding information to a namespace, and the client stub automatically manages a binding for the application code.

  The automatic method completely hides binding management from client application code. The stub calls the import operation and obtains a binding handle that refers to the imported binding information. The stub passes this binding handle to the runtime with the remote procedure call, and the runtime uses the binding handle to retrieve the associated binding information. If the client makes a series of remote procedure calls, the stub passes the same binding handle with each call.

  With the automatic method, a disrupted call can sometimes be automatically rebound. The automatic rebinding requires either that the remote procedure never begins to execute or that the operation is idempotent. If the call meets either of these requirements, the RPC runtime automatically tries to rebind the client to another server (if one is available).

Note: Using a context handle prevents automatic rebinding.

When a call made using the automatic method experiences an error that prevents automatic rebinding, the caller is informed that the call failed, and the caller can choose to reissue the call. If the client reissues a failed call, the automatic method selects a new server to try.

To make authenticated calls with the automatic binding method, the client calls the **rpc_ss_register_auth_info**( ) routine before making the remote procedure call. This routine places the client authentication information in the interface specification for the client runtime to access.

Note: The **rpc_ss_register_auth_info**( ) routine is unavailable in DCE RPC Version 1.0.

- Implicit method

  This is a relatively simple method of managing a binding. With the implicit method, prior to making any remote procedure calls, the client application code calls runtime routines to initialize a server binding handle. The runtime obtains server binding information from a namespace or a string binding and makes the information available to the client application by a server binding handle. The client application assigns the server binding handle to a global variable in the client application. When calling a remote procedure using the implicit method, the client stub passes this global binding handle to the runtime.

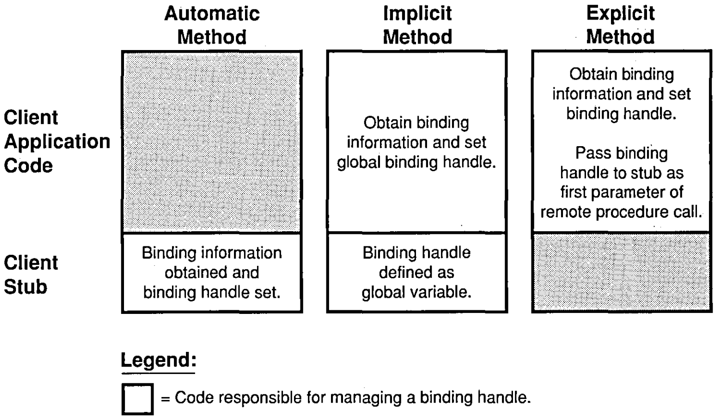
  Note: Multithreaded clients must be careful not to allow one thread to change the value of the shared global binding handle while another thread is using it.

- Explicit method

  This is a more complex and more flexible method of managing a binding. As with the implicit method, the explicit method requires that the client application code call runtime routines to initialize a binding handle. In the explicit method, however, this binding handle is supplied by the application code as a parameter to the remote procedure call. By allowing a client to manage bindings for individual calls, the explicit method enables clients to meet specialized binding requirements.

Figure 13-6 shows the distribution of responsibility for binding management in each of the three methods. For each method, the top portion of the box represents the client application code written by the developer. The bottom portion of each box represents the client stub code generated from an IDL interface definition. A binding handle is visible to the shaded portions of the code.

Figure 13–6. Methods of Binding Management


Automatic Method
Implicit Method
Explicit Method
Client Application Code
Client Stub
Obtain binding information and set global binding handle.
Obtain binding information and set binding handle.
Pass binding handle to stub as first parameter of remote procedure call.
Binding information obtained and binding handle set.
Binding handle defined as global variable.
Legend:
□ = Code responsible for managing a binding handle.

The automatic and implicit methods are interface wide and therefore mutually exclusive; that is, for a given interface, a client can use only one of these interface-wide methods. A client that uses either the automatic or implicit method for an interface can also use the explicit method for some or all of the remote procedure calls to that interface. The explicit method takes precedence over either the automatic or implicit methods of managing bindings.

The method(s) of binding management for an interface is specified using the interface definition, the Attribute Configuration File (ACF), or both. In the interface definition, the explicit method can be specified for the whole interface, or for an operation by declaring a binding handle (using the IDL type **handle_t**) as the first parameter of the operation declaration.

By default, an operation uses the automatic method of binding, unless the client passes either a context handle as one of the parameters or a binding handle as the first parameter. However, at compile time, declarations in an

ACF can override this default. The Attribute Configuration Language provides the **automatic_handle**, **implicit_handle**, and **explicit_handle** attributes. In an ACF you can declare any one of these attributes as an attribute of the whole interface (an ACF interface attribute). When you specify either the automatic method or the implicit method as an ACF interface attribute, you can also specify the explicit method for any individual operation by using **explicit_handle** as an attribute of the particular operation (an ACF operation attribute).

With the automatic method, binding management belongs completely to the client-stub code generated by the DCE IDL compiler. With the explicit method, the application developer is completely responsible for binding management. The implicit method provides the application developer with some control over binding management without having to pass a binding handle as a call argument.

A server binding handle that the runtime provides directly to an application is a primitive binding handle. To declare a primitive binding handle, application code uses the predefined RPC binding handle data type **rpc_binding_handle_t**, and an interface definition uses the IDL data type **handle_t**. Primitive binding handles offer a simple means of referring to binding information, which works in most cases. The automatic method of binding management always uses primitive binding handles.

Applications that use the implicit or explicit methods of binding management can choose to store primitive binding handles in an application-specific data structure known as a customized binding handle. Customized binding handles enable application developers to manage binding information to meet the special needs of a specific application. For example, a customized binding handle can be the handle of a file whose records contain the information required to construct a string binding.

Using customized binding handles requires the application developer to perform several special tasks. The RPC interface definition must include a declaration of the customized binding handle as a data structure with a handle data type; this is done by using the **handle** attribute. The client application code must contain specialized procedures that the client stub calls to obtain a primitive binding handle from the customized handle and to release any resources, such as memory, used for the customized handle.

When a customized binding handle is used with the explicit method, responsibility for setting the binding handle shifts to the client stub. The client code provides procedures for obtaining the primitive binding handle from the customized handle and for freeing the primitive binding handle after the call completes. However, it is the stub that calls these procedures to set and free the primitive binding handle.

13.6 Obtaining Server Binding Handles

A client can obtain server binding information in string format from an application-specific source such as a file. Alternatively, a client runtime can obtain server binding information from a namespace. Runtime routines enable client applications to obtain server binding handles that refer to server binding information obtained from either source.

13.6.1 Using String Bindings to Obtain Binding Information

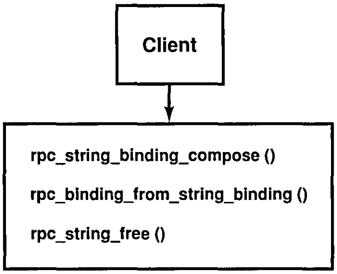
To use a string binding, a client starts with either an existing string binding or with the components of the binding information. When starting with the components, the client calls the RPC binding compose operation to get the string representation of binding information.

Do *not* hardcode string bindings into application code. Rather, specify them at runtime using a command argument, environment variable, file, or other means. The simplest way to specify a string binding is for a user to supply a string binding manually to a client. However, this manual approach is awkward for users who must know how to obtain and manipulate the string bindings. Also, if binding information changes, the users are responsible for updating any string bindings used by their clients. Reducing manual intervention in the use of string bindings requires that an application provide its own mechanisms for storing, maintaining, and accessing binding information. In contrast, a directory service such as CDS provides these mechanisms automatically to applications that store binding information in a namespace.

Regardless of how a client obtains a string binding, before establishing a binding, the client must ask the RPC runtime for a binding handle that refers to the server binding information depicted in the string binding. The client converts the string binding into a server binding handle by calling the **rpc_binding_from_string_binding**( ) routine.

Figure 13-7 lists the calls for composing a string binding and for using it to obtain a server binding handle.

Figure 13–7. Basic String Binding Calls of an RPC Client


Client
rpc_string_binding_compose ()
rpc_binding_from_string_binding ()
rpc_string_free ()

13.6.2 Searching a Namespace

To obtain binding information from a namespace, a client can do one of the following:

- The client must use the automatic method of binding management to make the client stub transparently manage binding information.

  In this case, the application code lacks any calls to the NSI interface. However, the automatic method does require the client to identify the directory service entry at which to begin the search for binding information. The client must specify the starting entry name as the value of the NSI-defined **RPC_DEFAULT_ENTRY** environment variable.

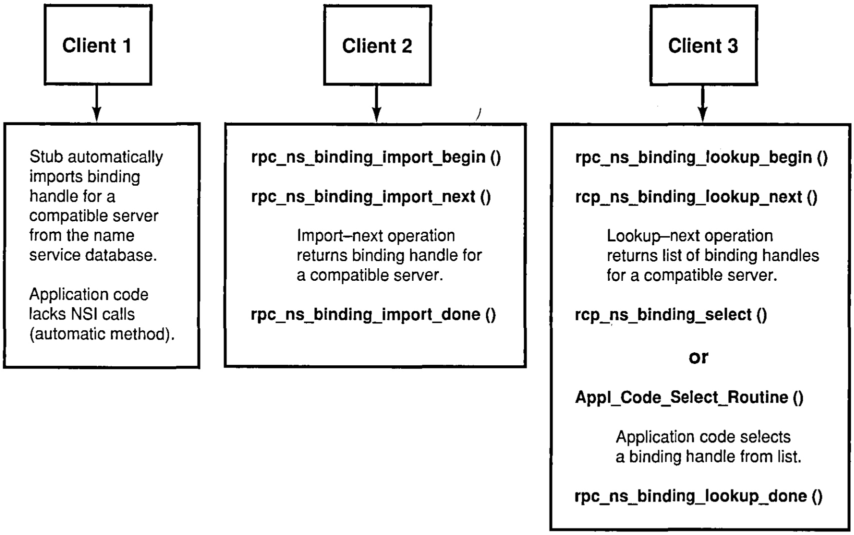
- The client must call the import routines **rpc_ns_binding_import_begin**( ), **rpc_ns_binding_import_next**( ), and **rpc_ns_binding_import_done**( ) to obtain a binding handle for a compatible server.

- The client must call the lookup routines **rpc_ns_binding_lookup_begin**( ), **rpc_ns_binding_lookup_next**( ), and **rpc_ns_binding_lookup_done**( ) to obtain a list of binding handles for a compatible server. Select a binding handle from the list by calling either of the following:

- — The NSI select routine **rpc_ns_binding_select**( ), which selects a binding handle at random
- — A user-defined select routine, which implements an application-specific selection algorithm

Figure 13-8 lists the NSI calls (where present) associated with these alternatives.

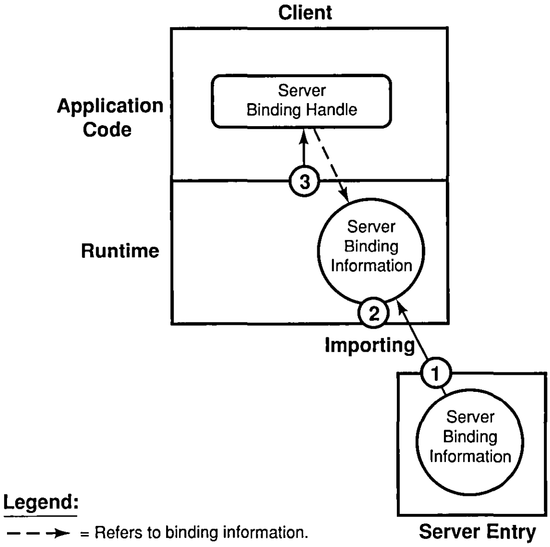
Figure 13–8. Calls for NSI Search Operations by RPC Clients


Client 1
Client 2
Client 3
Stub automatically imports binding handle for a compatible server from the name service database.
Application code lacks NSI calls (automatic method).
rpc_ns_binding_import_begin ()
rpc_ns_binding_import_next ()
Import–next operation returns binding handle for a compatible server.
rpc_ns_binding_import_done ()
rpc_ns_binding_lookup_begin ()
rcp_ns_binding_lookup_next ()
Lookup–next operation returns list of binding handles for a compatible server.
rcp_ns_binding_select ()
or
Appl_Code_Select_Routine ()
Application code selects a binding handle from list.
rpc_ns_binding_lookup_done ()

An NSI import or lookup operation searches server entries for a compatible server. On finding such a server entry, the search operation copies the server binding information associated with the requested interface and an object UUID. The search operation then creates a randomly ordered list of server binding handles to refer to the potential bindings represented by the binding information.

Figure 13-9 illustrates the use of a server binding handle to refer to server binding information selected by an import operation.

Figure 13–9. Importing Server Binding Information


Client
Application Code
Server Binding Handle
3
Runtime
Server Binding Information
2
Importing
1
Server Binding Information
Server Entry
Legend:
— — ➤ = Refers to binding information.

The callouts in the figure refer to the following operations:

1. The import operation looks up binding information of a server that is compatible with the client.

   The import operation finds a server entry based on the specified interface identifier, and then looks at the list of object UUIDs. If the importing client specifies a non-nil object UUID, the import operation looks for and returns that object UUID. If the client specifies the nil object UUID and the server entry contains any object UUIDs, the import operation selects and returns one UUID at random. If the entry lacks any object UUIDs, the import operation returns the nil UUID.

2. The import operation fetches the compatible binding information and creates a binding handle for each potential binding represented in the binding information.

3. The import operation then selects a binding handle at random and passes it to the client application.

13.7 Using Authenticated RPC

DCE RPC supports authenticated communications between clients and servers. Authenticated RPC works with the authentication and authorization services provided by the DCE Security Service.

On the application level, a server makes itself available for authenticated communications by registering its principal name and the authentication service that it supports with the RPC runtime. The server principal name is the name used to identify the server as a principal to the Registry Service provided by DCE Security Service. In practice, this name is usually the same as the name that the server uses to register itself with the DCE Directory Service.

A client must establish the authentication service, protection level, and authorization service that it wishes to use in its communications with a server. The client identifies the intended server by means of the principal name that the server has registered with the RPC runtime. Once the required authentication, protection, and authorization parameters have been established for the server binding handle, the client issues remote procedure calls to the server as it normally does.

The DCE Security Service, in conjunction with the RPC runtime, assumes responsibility for the following:

- Authenticating the client and server in accordance with the requested authentication service
- Applying the requested level of protection to communications between the client and server
- Providing client authorization data to the server in a form determined by the requested authorization service

Note: For a detailed discussion of authenticated RPC within the context of DCE Security, refer to Part 6 of this guide.

13.7.1 Authentication

When a client establishes authenticated RPC, it must indicate the authentication service that it wants to use. The possible values are the following:

- **rpc_c_authn_none**: No authentication
- **rpc_c_authn_dce_secret**: DCE shared-secret key authentication
- **rpc_c_authn_dce_public**: DCE public key authentication
- **rpc_c_authn_default**: DCE default authentication service

The value **rpc_c_authn_none** is used to turn off authentication already established for a binding handle. The default authentication is DCE shared-secret authentication, which is described in detail in Part 6 of this guide.

Before a client and server can engage in authenticated RPC, they must ''agree'' on which authentication service to use. Specifically, the server must register the ''agreed on'' authentication service with the RPC runtime, along with the server's principal name. For its part, the client must select the same service for the server's binding handle. The client indicates the appropriate server by supplying the server's principal name. If the client does not know the server's name, it can use the **rpc_mgmt_inq_server_princ_name**( ) routine to determine the name. The actual RPC routines used by both the client and the server to establish authenticated RPC are described under Section 13.7.3.

13.7.1.1 Cross-Cell Authentication

A client can engage in authenticated RPC with a target server that is in the client's cell or in a foreign cell. In the case of cross-cell authentication, DCE Security performs the necessary additional steps on behalf of the client.

To establish authenticated RPC with a foreign server, a client must supply the fully qualified principal name of the server. A fully qualified name includes the name of the cell as well as the name of the principal and takes the form

/.../cell_name/principal_name

13.7.1.2 Protection Levels

When a client establishes authenticated RPC, it can specify the level of protection to be applied to its communications with the server. The protection level determines how much of client/server messages are encrypted. As a rule, the more restrictive the protection level, the greater the impact on performance. Different levels are provided so that applications can control the protection versus performance tradeoffs.

Note that the protection level is entirely a client responsibility. When a server registers its supported authentication service with the RPC runtime, it does not specify any protection information for that service. However, the server can include the protection level used for a particular operation when deciding if the caller is authorized to perform the operation.

Authenticated RPC supports the following protection levels:

- **rpc_c_protect_level_default**: Uses the default protection level for the specified authentication service.
- **rpc_c_protect_level_none**: There is no protection level.
- **rpc_c_protect_level_connect**: Performs protection only when the client establishes a relationship with the server. This level performs an encrypted handshake when the client first communicates with the server. Encryption or decryption is not performed on the data sent between the client and server. The fact that the handshake succeeds indicates that the client is active on the network.
- **rpc_c_protect_level_call**: Performs protection only at the beginning of each remote procedure call when the server receives the request. This level attaches a verifier to each client call and server response.

  This level does not apply to remote procedure calls made over a connection-based protocol sequence; that is, **ncacn_ip_tcp**. If this level is specified and the binding handle uses a connection-based protocol sequence, the routine uses the **rpc_c_protect_level_pkt** level instead.

- **rpc_c_protect_level_pkt**: Ensures that all data received is from the expected client. This level attaches a verifier to each message.
- **rpc_c_protect_level_pkt_integrity**: Ensures and verifies that none of the data transferred between client and server has been modified. This level computes a cryptographic checksum of each message to verify that none of the data transferred between the client and server has been modified in transit.

  This is the highest protection level that is guaranteed to be present in the RPC runtime.
- **rpc_c_protect_level_pkt_privacy**: Performs protection as specified by all of the previous levels and also encrypts each remote procedure call argument value. This level encrypts all user data in each call.

  This is the highest protection level, but it may not be available in the RPC runtime.

If a client wants to use the default protection level but does not know what this level is, it can use the **rpc_mgmt_inq_dflt_protect_level**( ) routine to determine what the default level is.

13.7.2 Authorization

Authorization is the process of checking a client's permissions to an object that is controlled by the server. Access checking is entirely a server responsibility and involves matching the client's privilege attributes against the permissions associated with the object. A client's privilege attributes consist of the principal ID and group memberships contained in the client's network login context.

Authenticated RPC supports the following options for making client authorization information available to servers for access checking:

- **rpc_c_authz_none**: No authorization information is provided to the server, usually because the server does not perform access checking.
- **rpc_c_authz_name**: Only the client principal name is provided to the server. The server can then perform authorization based on the provided name. This form of authorization is sometimes referred to as ''name-based'' authorization.
- **rpc_c_authz_dce**: The client's DCE Privilege Attribute Certificate (PAC) is provided to the server with each remote procedure call made using the binding parameter. The server performs authorization using the client PAC. Generally, access is checked against DCE ACLs.

When a client establishes authenticated RPC, it must indicate which authorization option it wants to use.

It is the server's responsibility to implement the type of authorization appropriate for the objects that it controls. When the server calls **rpc_binding_inq_auth_client( )** to return information about an authenticated client, it gets back either the client's principal name or a pointer to the data structure that contains the client's PAC. The value that is returned depends on which type of authorization the client specified on its call to establish authenticated RPC with that server.

Each server is responsible for implementing its own access checking by means of ACL managers. When a server receives a client request for an object, the server invokes the ACL manager appropriate for that type of object and passes the manager the client's authorization data. The manager compares the client authorization data to the permissions associated with the object and either refuses or permits the requested operation. In the case of certified (PAC-based) authorization, servers must implement access checking using the ACL facility provided by the DCE Security Service. ACL managers are described in more detail in Part 6 of this guide.

13.7.2.1 Name-Based Authorization

Name-based authorization (**rpc_c_authz_name**) provides a server with the client's principal name. The server call to **rpc_binding_inq_auth_client**( ) retrieves the name from the binding handle associated with the client and returns it as a character string.

It is not recommended that names be used for authorization. To perform access checking using client principal names, the names must be stored in the access lists associated with the protected objects. Each time a name is changed the change has to be propagated through all the access lists in which the name is defined.

13.7.2.2 DCE Authorization

DCE authorization (**rpc_c_authz_dce**) provides a server with the client's PAC.

PACs offer a trusted mechanism for conveying client authorization data to authenticated servers. The DCE Security Service generates a client PAC in a tamper-proof manner. When a server receives a client PAC, it knows that the PAC has been certified by DCE Security.

PACs are designed to be used with the DCE ACL facility. The ACL facility provides an editor and a set of API routines that support the implementation of access control lists and the managers to control them.

For a detailed description of PACs and their use with DCE ACL facility, refer to Part 6 of this guide.

13.7.3 Authenticated RPC Routines

Authenticated RPC is implemented as a set of related RPC routines. Some of the routines are for use by clients, some are for use by servers and their managers, and some are for use by both clients and servers. The authenticated RPC routines are as follows:

- **rpc_binding_set_auth_info**( ): A client calls this routine to establish an authentication service, protection level, and authorization service for a server binding handle. The client identifies the server by supplying the server's principal name. The RPC runtime, in conjunction with the DCE Security Service, applies the authentication service and protection level to all subsequent remote procedure calls made using the binding handle.

- **rpc_ss_register_auth_info( )**: A client calls this routine to register an authentication service, protection level, and authorization service for an interface specification. After a client calls this routine, the RPC runtime automatically applies the authentication and authorization information to all remote procedure calls that the client makes using implicit binding handles. An example is when the client uses an implicit handle with the IDL **auto_handle** attribute or a customized handle with the **handle** attribute.

- **rpc_binding_inq_auth_info( )**: A client calls this routine to return the authentication service, protection level, and authorization service that are in effect for a specified server binding handle. This routine also returns the principal name of the server associated with the binding handle.

- **rpc_mgmt_inq_dflt_protect_level( )**: A client or a server calls this routine to learn the default protection level that is in force for a given authentication service.

- **rpc_mgmt_inq_server_princ_name( )**: A client, a server, or a server manager can call this routine to return the principal name that a server has registered with the RPC runtime via the **rpc_server_register_auth_info( )** routine. A client can identify the desired server by supplying a server binding handle and the authentication service associated with the registered principal name.

- **rpc_server_register_auth_info( )**: A server calls this routine to register an authentication service that it wants to support and the server principal name to be associated with the registered service. The server can also supply the address of a key retrieval routine to be called by the DCE Security Service as part of the client authentication process. The routine is a user-supplied function whose purpose is to provide the server's key to the DCE Security runtime.

  Note that the server registers only an authentication service. It does not establish a protection level or an authorization service. These are the responsibilities of the client.

- **rpc_binding_inq_auth_client( )**: A server calls this routine to return the authentication service, protection level, and authorization service that is associated with the binding handle of an authenticated client. This call also returns the server principal name specified by the client on its call to **rpc_binding_set_auth_info( )**.

- **rpc_mgmt_set_authorization_fn( )**: A server calls this routine to establish a user-supplied authorization function to validate remote client calls to the server's management routines. For example, the user function can call **rpc_binding_inq_auth_client( )** to return authentication and authorization information about the calling client. The RPC runtime calls the user-supplied function whenever it receives a client request to execute one of the following server management routines:

  — **rpc_mgmt_inq_if_ids( )**

  — **rpc_mgmt_inq_server_princ_name( )**

  — **rpc_mgmt_inq_stats( )**

  — **rpc_mgmt_is_server_listening( )**

  — **rpc_mgmt_stop_server_listening( )**

Chapter 14

Advanced DCE RPC Topics

This chapter discusses aspects of the internal behavior of remote procedure calls that are significant for advanced RPC programmers, including the following topics:

- Advanced Name Service Interface (NSI) topics
- Threads of execution in RPC applications
- Nested remote procedure calls
- Routing remote procedure calls at the server's system

14.1 Advanced Name Service Interface Topics

The following subsections discuss the structure of NSI directory service entries and the mechanics of NSI searches. More information about these entries appears at the beginning of Chapter 15.

14.1.1 Structure of NSI Name Service Entries: NSI Attributes

Usually, the distinct server entries, groups, and profiles concepts are adequate for using NSI. However, the way NSI stores RPC information allows you to combine server entries, groups, and profiles into a single directory service entry. To store information about RPC applications in a directory service entry, the RPC directory service interface defines several RPC-specific directory service attributes, or NSI attributes. NSI attributes contain information about RPC applications in a directory service entry. The NSI attributes are as follows:

- NSI binding attribute

  The binding attribute stores binding information and interface identifiers (interface UUID and version numbers) exported to the server entry. This attribute identifies a directory service entry as a server entry.

- NSI object attribute

  The object attribute stores a list of one or more object UUIDs. Whenever a server exports any object UUIDs to a server entry, the server entry contains an object attribute as well as a binding attribute. When a client imports from that entry, the import operation returns an object UUID from the list stored in the object attribute.

- NSI group attribute

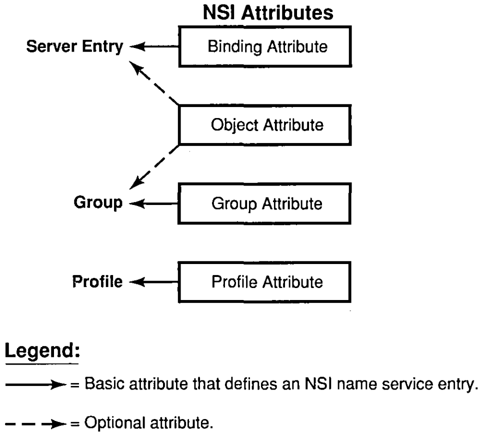
  The group attribute stores the entry names of the members of a single group. This attribute identifies a directory service entry as an RPC group.

- NSI profile attribute

  The profile attribute stores a set of profile elements. This attribute identifies a directory service entry as an RPC profile.

Figure 14-1 represents the correspondence between NSI attributes and the different directory service entries: server entries, groups, and profiles.

Figure 14–1. NSI Attributes



Any directory service entry can contain any combination of the four NSI attributes. However, to facilitate administrating directory service entries, avoid creating binding, group, and profile attributes in the same entry. Instead, use distinct directory service entries for server entries, groups, and profiles. The object attribute, in contrast, is designed as an adjunct to another NSI attribute, especially the binding attribute.

When implementing the resource model or when used to distinguish server instances, a server entry contains an object attribute as well as a binding attribute. On finding a server entry whose binding attribute contains compatible binding information, an NSI search operation also looks in the entry for an object attribute. For groups whose membership is selected according to a shared object or set of objects, it may be useful to export those objects to the group. In this case, the directory service entry of the group contains both group and object attributes. For reading the object UUIDs in the NSI object attribute in any directory service entry, NSI provides a set of object inquiry operations, called using the **rpc_ns_entry_object_inq_{begin,next,done}**( ) routines.

Using separate entries facilitates administration of the namespace; for example, by enabling entry names to specifically describe their contents. Keeping server entries, profiles, and groups separate allows clear references to each of them.

Note: In addition to any NSI attributes, a directory service entry contains other kinds of directory service attributes. Every entry in a namespace contains standard attributes created by the directory service. NSI operations rely on some standard attributes to identify and use an entry. Any directory service entry can also contain additional attributes specified by non-RPC applications; these are ignored by NSI operations.

14.1.2 Searching the Namespace for Binding Information

Searching the namespace for binding information requires that a client specify a starting point for the search. A client can start with a specific server entry. However, this is a limiting approach because the client is restricted to using one server. To avoid this, a client can start searching with a group or a profile instead of with a server entry. Searches that start with a profile or a group should encounter the server entry of a compatible server. If such an entry is not encountered, a search operation returns the **rpc_s_no_more_bindings** status code to the client. When calling the **rpc_ns_binding_import_next( )** or **rpc_ns_binding_lookup_next( )** routine, a client must track whether the routine returns this status code.

14.1.2.1 The import_next and lookup_next Search Algorithm

The NSI search operations (**import_next** and **lookup_next**) traverse one or more entries in the namespace when searching for compatible binding information. In each directory service entry, these operations ignore non-RPC attributes and process the NSI attributes in the following order:

1. Binding attribute (and object attribute, if present)
2. Group attribute
3. Profile attribute

If an NSI search path includes a group attribute, the search path can encompass every entry named as a group member. If a search path includes a profile attribute, the search path can encompass every entry named as the member of a profile element that contains the target interface identifier. A search finishes only when it finds a server entry containing compatible binding information and the non-nil object UUID, if requested. Search operations take the following steps when traversing a directory service entry:

Step 1: Binding attribute

In each entry, the search operation starts by searching for a compatible interface identifier in the binding attribute, if present.

The absence of a binding attribute or of any compatible interface identifier causes the search operation to go directly to step 2.

The presence of any compatible interface identifier indicates that compatible potential bindings may exist in the binding attribute. At this point, object UUIDs may impact the search, as follows:

- If the client specified the nil object UUID, object UUIDs do not affect the success or failure of the search. The search returns compatible binding information for one or more potential bindings.
- If the client specified a non-nil object UUID, the search reads the object attribute, if present, to look for the requested object UUID. This search for an object UUID has one of the following outcomes:
    - — On finding the specified object UUID, the search returns the object UUID along with compatible binding information for one or more potential bindings.
    - — If a requested object UUID is absent, the search continues to step 2.

Note: If a search involves a series of **import_next** or **lookup_next** operations, a subsequent next operation resumes the search at the point in the search path where the preceding operation left off.

Step 2: Group attribute

If the binding attribute does not lead to compatible binding information or if a series of **import_next** or **lookup_next** operations exhausts the compatible binding information, the search continues by reading the group attribute, if present; if the directory service entry lacks a group attribute, the search goes directly to step 3.

The search operation selects a member of the group at random, goes to the entry of that member, and resumes the search at step 1. Unless a group member leads the search to compatible binding information, the search looks at all the members of the group, one by one in random order, until none remain.

Step 3: Profile attribute

If the binding and group attributes do not lead to compatible binding information, the search continues by reading the profile attribute, if present; if the directory service entry lacks a profile attribute, the search fails.

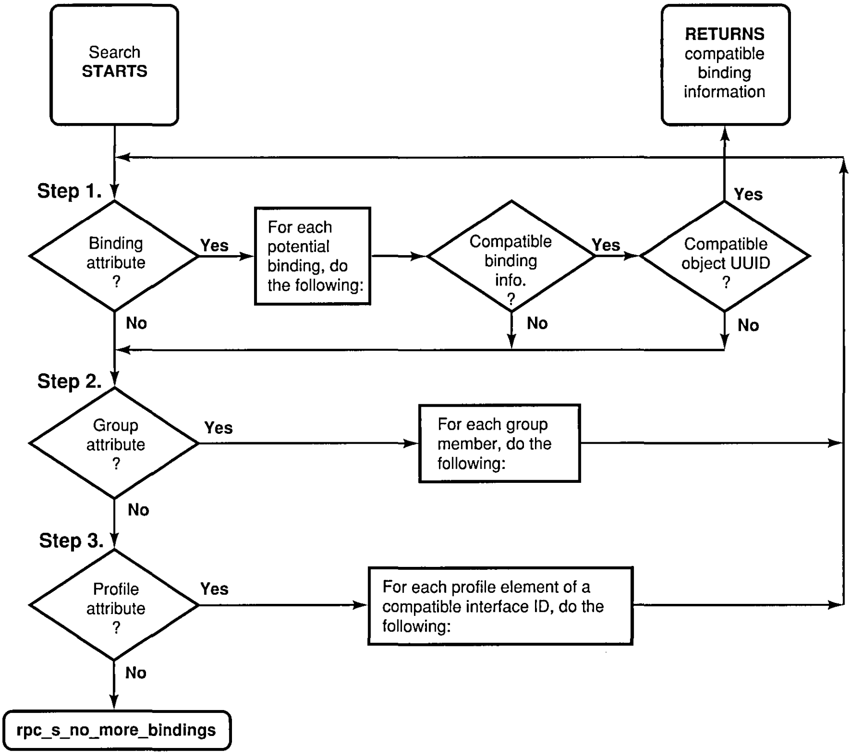
The search operation identifies all the profile elements containing the requested interface identifier and searches them in the order of their priority, beginning with the 0 (zero) priority elements. Profile elements of a given priority are searched in random order. For the selected profile element, the search reads the member name and goes to the corresponding directory service entry. There, the search resumes at step 1. Unless a profile element leads the search to compatible binding information, the search eventually looks at all the profile elements with the requested interface identifier, one by one, until none remain.

If the starting entry does not contain NSI attributes, or if none of the steps satisfies the search, the search operation returns an **rpc_s_no_more_bindings** status code to the client.

Note: The inquire next (**inq_next**) operations for objects, groups, or profiles look at only the entry specified in its corresponding inquire begin (**inq_begin**) operation. The search ignores nested groups or nested profiles.

Figure 14-2 illustrates the three steps of the **import_next** and **lookup_next** search operations.

Figure 14–2. The import_next and lookup_next Search Algorithm Within a Single Entry


Search STARTS
RETURNS compatible binding information
Step 1.
Binding attribute ?
Yes
No
For each potential binding, do the following:
Compatible binding info. ?
Yes
No
Compatible object UUID ?
Yes
No
Step 2.
Group attribute ?
Yes
No
For each group member, do the following:
Step 3.
Profile attribute ?
Yes
No
For each profile element of a compatible interface ID, do the following:
rpc_s_no_more_bindings

14.1.2.2 Examples of Searching for Server Entries

This subsection provides several examples of how the NSI **import_next** and **lookup_next** operations search for binding information associated with a given RPC interface and object in a namespace.

Conventions: The examples in this guide use the following conventions:

- To simplify the following examples, each member name is represented by a leaf name preceded by the symbol that represents the local cell (**/.:**).

For example, the full global name of the group for the **Bulletin_board_interface** is as follows:

/.../C=US/O=uw/OU=MadCity/LandS/bb_grp

The abridged name is **/.:/LandS/bb_grp**.

Note: For a summary of global name syntax, see Section 15.1.5 on naming directory service entries.

- Except for the nil interface UUID of the default profile, the examples avoid string representations of actual UUIDs. Instead, the examples represent a UUID as a value consisting of the name of the interface and the string *if-uuid* or of the name of the object and the string *object-uuid*; for example:

  calendar-if-uuid,1.0

  laser-printer-object-uuid

- Profile elements in the examples are organized as follows (annotations are not displayed):

  interface-identifier   member-name   priority

  For example,

```
2FAC8900-31F8-11CA-B331-08002B13D56D,1.0 /.:/LandS/C_host_7    0
```

  which, in the following examples, is represented as:

  calendar-if-uuid,1.0   /.:/LandS/C_host_7   0

  Note: The priority is a value of 0 to 7, with 0 having the highest search priority and 7 having the lowest priority.

The first two examples begin with the personal profile of a user, Molly O'Brian, whose username is **molly_o** and whose profile has the leaf name of **molly_o_profile**. To use this profile, Molly must specify its entry name to the client. Usually, a client either uses the predefined RPC environment variable **RPC_DEFAULT_ENTRY** or prompts for an entry name. For a client to use **RPC_DEFAULT_ENTRY**, the client or user must have already set the variable to a directory service entry.

The following example illustrates six profile elements from the individual user profile used in the first two examples. The six elements include five nondefault elements for some frequently used interfaces and a default profile element. Each profile element is displayed on three lines, but in an actual profile all the fields occupy a single record. The fields are the interface identifier (interface UUID and version numbers), member name, priority, and annotation.

```
/.:/LandS/anthro/molly_o_profile contents:

EC1EEB60-5943-11C9-A309-08002B102989,1.0
   /.../C=US/O=uw/OU=MadCity/LandS/Cal_host_7
   0  Calendar_interface_V1.0

EC1EEB60-5943-11C9-A309-08002B102989,2.0
   /.../C=US/O=uw/OU=MadCity/LandS/Cal_host_4
   1  Calendar_interface_V2.0

62251DDD-51ED-11CA-852C-08002B1BB4F6,2.0
   /.../C=US/O=uw/OU=MadCity/bb_grp
   0  Bulletin_board_interface_V2.0

62251DDD-51ED-11CA-852C-08002B1BB4F6,2.1
   /.../C=US/O=uw/OU=MadCity/bb_grp
   1  Bulletin_board_interface_V2.1

9E18D295-51EC-11CA-9CC0-08002B1BB4F5,1.0
   /.../C=US/O=uw/OU=MadCity/LandS/anthro/Zork_host_2
   0  Zork_interface_V1.0

00000000-0000-0000-0000-000000000000,0.0
   /.../C=US/O=uw/OU=MadCity/cell_profile
   0  Default_profile_element
```

Example 1: Importing for an Interface with Multiple Versions

Target Interface: Calendar V2.0

1. The search for binding information associated with Calendar V2.0 starts with the entry **molly_o_profile**:

```
/.../C=US/O=uw/OU=MadCity/LandS/anthro/molly_o_profile contents:
```

```
calendar-if-uuid,1.0    /.:/LandS/C_host_7    0
calendar-if-uuid,2.0    /.:/LandS/C_host_4    1

bulletin_board-if-uuid,2.0    /.:/LandS/bb_grp    2
bulletin_board-if-uuid,2.1    /.:/LandS/bb_grp    3

Zork-if-uuid,1.0    /.:/Eng/Zork_host_2    0
00000000-0000-0000-0000-000000000000,0.0    /.:/cell_profile    0
```

The search operation examines only the two profile elements that refer to the Calendar interface:

a. The operation rejects the first profile element for the interface because it refers to the wrong version numbers.

b. In the next profile element, the operation finds the correct version numbers (**2.0**). The search proceeds to the associated server entry, **/.:/LandS/Cal_host_4**.

2. The search ends with the indicated server entry, where the binding information requested by the client resides:

```
/.:/LandS/Cal_host_4 contents:
   calendar-if-uuid,2.0

   binding-information
```

Example 2: Using a Default Profile for Importing an Interface

Target Interface: Statistics V1.0

1. The search for binding information associated with Statistics V1.0 starts with the entry **molly_o_profile**. However, the profile lacks any elements for the interface. Therefore, the search reaches the default profile element, which provides the entry name for the default profile, **/.:/cell_profile**:

```
/.:/LandS/anthro/molly_o_profile contents:

   calendar-if-uuid,1.0    /.:/LandS/C_host_7    0
   calendar-if-uuid,2.0    /.:/LandS/C_host_4    1

   bulletin_board-if-uuid,2.0    /.:/LandS/bb_grp    2
```

```
bulletin_board-if-uuid,2.1    /.:/LandS/bb_grp    3

Zork-if-uuid,1.0    /.:/Eng/Zork_host_2    0
00000000-0000-0000-0000-000000000000,0.0    /.:/cell_profile    0
```

2. The search continues to the indicated default profile, **/.:/cell_profile**, which contains a profile element for the requested Statistics V1.0 interface:

   ```
   /.:/LandS/cell_profile contents:
       .
       .
       .
       Statistics-if-uuid,1.0    /.:/LandS/Stats_host_6    0
       .
       .
       .
   ```

3. The search ends at the indicated server entry, **/.:/LandS/Stats_host_6**, where a server address for the requested interface resides:

   ```
   /.:/LandS/Stats_host_6 contents:

       Statistics-if-uuid,1.0

       binding-information
   ```

Example 3: Importing an Interface and an Object

Target Interface: Print Server V2.1

Target Object: Laser Printer Print Queue

1. The search starts with the entry **/.:/Bldg/Print_queue_grp**, which contains the entry names of several server entries that advertise the **Print_server** interface and the object UUID of a given **Laser_printer** print queue. The search begins by randomly selecting a member name. In this instance, the search selects **/.:/Bldg/Print_server_host_3**:

   ```
   /.:/Bldg/Print_queue_grp contents:
   ```

```
/.:/Bldg/Print_server_host_3
/.:/Bldg/Print_server_host_7
/.:/Bldg/Print_server_host_9
```

2. The search continues with the **/.:/Bldg/Print_server_host_3** entry. There, it finds the requested Version 2.1 of the **Print_server** interface. However, the search continues because the entry lacks the object UUID of the requested **Laser_printer** queue:

   `/.:/Bldg/Print_server_host_3 contents:`

   print_server-if-uuid, `2.1`

   binding-information

   line_printer_queue-object-uuid

3. The search goes back to the previous entry, **/.:/Bldg/Print_queue_grp**, to select another entry name, in this instance, **/.:/Bldg/Print_server_host_9**:

   `/.:/Bldg/Print_queue_grp contents:`

   ```
   /.:/Bldg/Print_server_host_3
   /.:/Bldg/Print_server_host_7
   /.:/Bldg/Print_server_host_9
   ```

4. The search selects the **/.:/Bldg/Print_server_host_9** entry. This entry contains both a server address for the requested Version 2.1 of the interface and the requested object UUID of the **Laser_printer** queue:

   `/.:/Bldg/Print_server_host_9 contents:`

   print_server-if-uuid, 2.1

   binding-information

   laser_printer_queue-object-uuid

   The search returns binding information from this entry to the client.

14.1.2.3 Expiration Age of a Local Copy of Directory Service Data

To prevent accessing a namespace unnecessarily, previously requested directory service data is sometimes stored on the system where the request originated. A local copy of directory service data is not automatically updated at each request. Automatic updating of the local copy occurs only when it exceeds its expiration age. The expiration age is the amount of time that a local copy of directory service data from an NSI attribute can remain unchanged before a request from an RPC application for the attribute requires updating of the local copy. When an RPC application begins running, the RPC runtime randomly specifies a value between 8 and 12 hours as the default expiration age for that instance of the application. Most applications use only this default expiration age, which is global to the application.

An expiration age is used by an NSI next operation, which reads data from directory service attributes. For a given search or inquire operation, you can override the default expiration age by calling the **rpc_ns_mgmt_handle_set_exp_age**( ) routine after the operation's begin routine. Note that specifying a low default age will result in increased network updates among the name servers in your cell. This will adversely affect the performance of all network traffic. Therefore, use the default whenever possible. If you must override the default age, specify a number that is high enough to avoid frequent updates of local data.

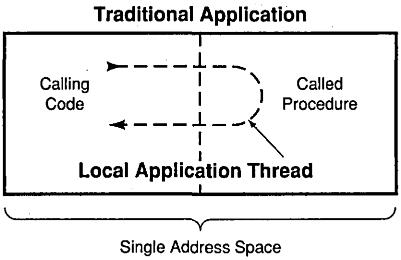
An NSI next operation usually starts by looking for a local copy of the attribute data being requested by an application. In the absence of a local copy, the NSI next operation creates one with fresh attribute data from the namespace. If a local copy already exists, the operation compares its actual age to the expiration age used by the application. If the actual age exceeds the expiration age, the operation automatically tries to update the local copy with fresh attribute data. If updating is impossible, the old local data remains in place and the NSI next operation fails, returning the **rpc_s_name_service_unavailable** status code.

14.2 Threads of Execution in RPC Applications

Each remote procedure call occurs in an execution context called a ''thread.'' A thread is a single sequential flow of control with one point of execution on a single processor at any instant. A thread created and managed by application code is an ''application thread.''

Traditional processing occurs exclusively within local application threads. Local application threads execute within the confines of one address space on a local system and pass control exclusively among local code segments, as illustrated by Figure 14-3.

Figure 14–3. Local Application Thread During a Procedure Call



RPC applications also use application threads to issue both remote procedure calls and runtime calls, as follows:

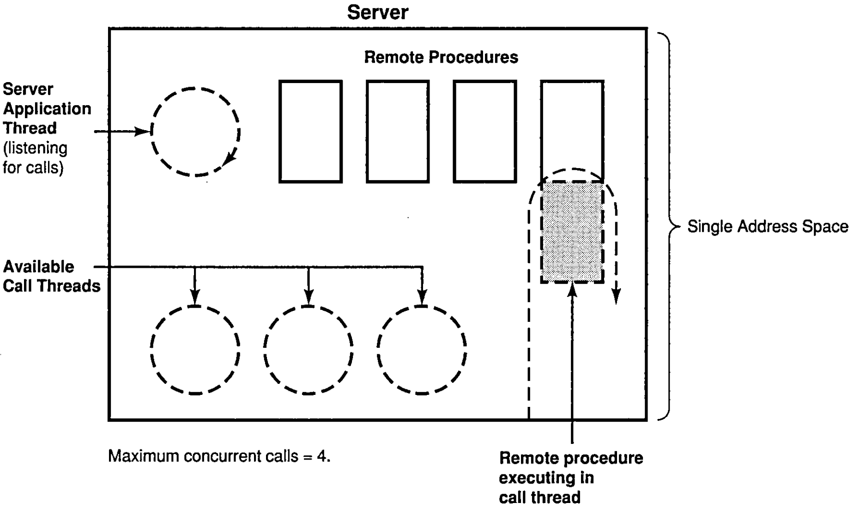
- An RPC client contains one or more client application threads; that is, a thread that executes client application code that makes one or more remote procedure calls.
- A DCE RPC Version 1.0 server contains one server application thread; that is, a thread that executes the server application code that listens for incoming calls.

In addition, for executing called remote procedures, an RPC server uses one or more call threads that the RPC runtime provides. As part of initiating listening, the server application thread specifies the maximum number of concurrent calls it will execute. Single-threaded applications have a maximum of one call thread. The maximum number of call threads in multithreaded applications depends on the design of the application. The RPC runtime creates the same number of call threads in the server process.

The number of call threads is significant to application code. When using only one call execution thread, application code does not have to protect itself against concurrent resource use. When using more than one call thread, application code must protect itself against concurrent resource use.

Figure 14-4 shows a multithreaded server with a maximum of four concurrently executing calls. Of the four call threads for the server, only one is currently in use; the other three threads are available for executing calls.

Figure 14–4. Server Application Thread and Multiple Call Threads

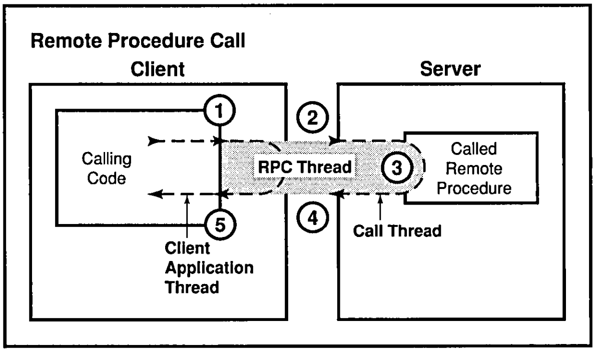

Server
Remote Procedures
Server Application Thread (listening for calls)
Available Call Threads
Single Address Space
Maximum concurrent calls = 4.
Remote procedure executing in call thread

14.2.1 Remote Procedure Call Threads

In distributed processing, a call extends to and from client and server address spaces. Therefore, when a client application thread calls a remote procedure, it becomes part of a logical thread of execution known as an RPC thread. An RPC thread is a logical construct that encompasses the various phases of a remote procedure call as it extends across actual threads of execution and the network. After making a remote procedure call, the calling client application thread becomes part of the RPC thread. Usually, the RPC thread maintains execution control until the call returns.

The RPC thread of a successful remote procedure call moves through the execution phases illustrated in Figure 14-5.

Figure 14–5. Execution Phases of an RPC Thread



The execution phases of an RPC thread in the preceding figure include the following operations:

1. The RPC thread begins in the client process, as a client application thread makes a remote procedure call to its stub; at this point, the client thread becomes part of the RPC thread.

2. The RPC thread extends across the network to the server address space.

3. The RPC thread extends into a call thread, where the remote procedure executes.

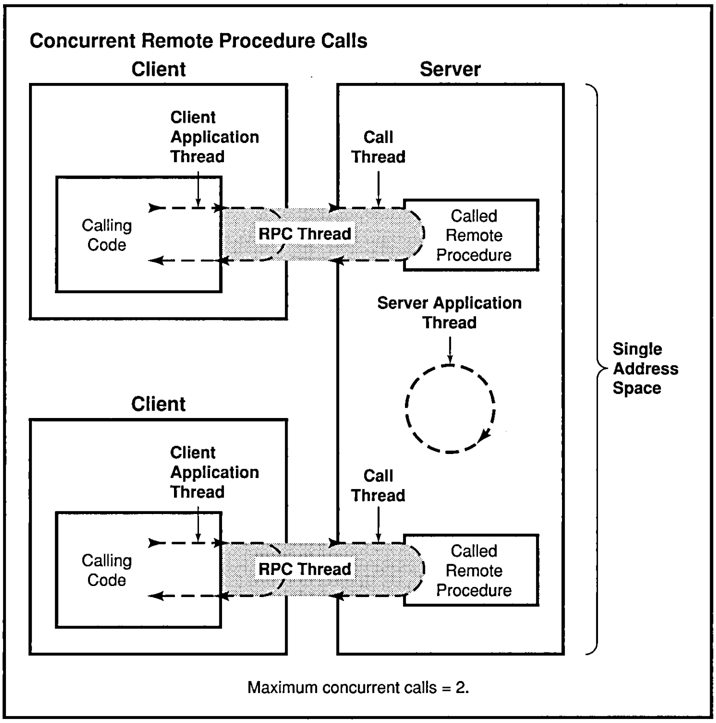
   While a called remote procedure is executing, the call thread becomes part of the RPC thread. When the call finishes executing, the call thread ceases being part of the RPC thread.

4. The RPC thread then retracts across the network to the client.

5. When the RPC thread arrives at the calling client application thread, the remote procedure call returns any call results and the client application thread ceases to be part of the RPC thread.

Figure 14-6 shows a server executing remote procedures in its two call threads, while the server application thread listens.

Figure 14–6. Concurrent Call Threads Executing in Shared Address Space



Note: Although a remote procedure can be viewed logically as executing within the exclusive control of an RPC thread, some parallel activity does occur in both the client and server.

An RPC server can concurrently execute as many remote procedure calls as it has call threads. When a server is using all of its call threads, the server application thread continues listening for incoming remote procedure calls. While waiting for a call thread to become available, DCE RPC server runtimes can queue incoming calls. Queuing incoming calls avoids remote procedure calls failing during short-term congestion. The queue capacity for incoming calls is implementation dependent; most implementations offer a small queue capacity. The queuing of incoming calls is discussed in Section 14.4.3, which describes the routing of incoming calls.

14.2.2 Cancels

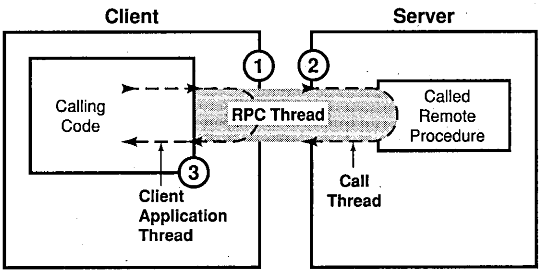
DCE RPC uses and supports the synchronous cancel capability provided by POSIX threads (pthreads). A cancel is a mechanism by which a thread informs another thread (the canceled thread) to terminate as soon as possible. Cancels operate on the RPC thread exactly as they would on a local thread, except for an application-specified, cancel-time-out period. A cancel-time-out period is an optional value that limits the amount of time the canceled RPC thread has before it releases control.

During a remote procedure call, if its thread is canceled and the cancel-time-out period expires before the call returns, the calling thread regains control and the call is orphaned at the server. An orphaned call may continue to execute in the call thread. However, the call thread is no longer part of the RPC thread, and the orphaned call is unable to return results to the client.

A client application thread can cancel any other client application thread in the same process (it is possible, but unlikely, for a thread to cancel itself.) While executing as part of an RPC thread, a call thread can be canceled only by a client application thread.

A cancel goes through several phases. Figure 14-7 indicates the point in the RPC thread where each of these phases occurs.

Figure 14–7. Phases of a Cancel in an RPC Thread



The phases of a cancel in the preceding figure include the following:

1. A cancel that becomes pending at the client application thread at the start of or during a remote procedure call becomes pending for the entire RPC thread. Thus, while still part of the RPC thread, the call thread also has this cancel pending.

2. If the call thread of an RPC thread makes a cancelable call when cancels are not deferred and a cancel is pending, the cancel exception is raised.

3. The RPC thread returns to the canceled client application thread with one of the following outcomes:

   a. If a cancel exception has not been taken, the RPC thread returns normal call results (output arguments, return value, or both) with a pending cancel.

   b. If the remote procedure is using an exception handler, a cancel exception can be handled. The procedure resumes, and the RPC thread returns normal call results without pending any cancel. (For information on the use of exception handlers, see Part 2 of this guide.)

   c. If the remote procedure failed to handle a raised cancel exception, the RPC thread returns with the cancel exception still raised. This is returned as a fault.

d. If the cancel-time-out period expires, the RPC thread returns either a cancel-time-out exception or status code, depending on how the application sets up its error handling. This is true for all cases where any abnormal termination is returned.

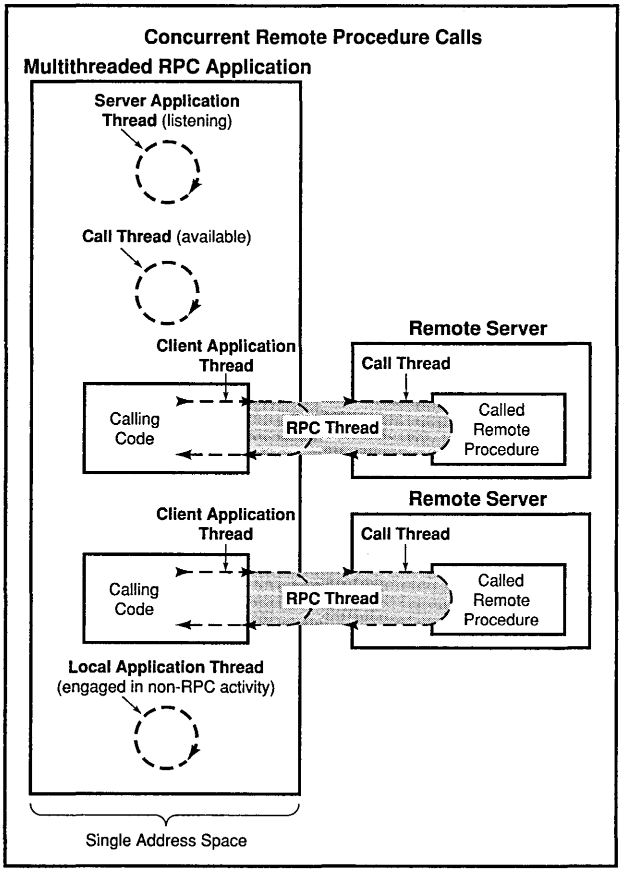
14.2.3 Multithreaded RPC Applications

DCE RPC provides an environment for RPC applications that create multiple application threads (multithreaded applications). The application threads of a multithreaded application share a common address space and much of the common environment. If a multithreaded application must be thread-safe (guarantee that multiple threads can execute simultaneously and correctly), the application is responsible for its own concurrency control. Concurrency control involves programming techniques such as controlling access to code that can share a data structure or other resource to prevent conflicting overlapping access by separate threads.

A multithreaded RPC application can have diverse activities going on simultaneously. A multithreaded client can make concurrent remote procedure calls and a multithreaded server can handle concurrent remote procedure calls. Using multiple threads allows an RPC client or server to support local application threads that continue processing independently of remote procedure calls. Also, multithreading enables the server application thread and the client application threads of an RPC application to share a single address space as a joint client/server instance. A multithreaded RPC application can also create local application threads that are uninvolved in the RPC activity of the application.

Figure 14-8 shows an address space where application threads are executing concurrently.

Figure 14–8. A Multithreaded RPC Application Acting as Both Server and Client


Concurrent Remote Procedure Calls
Multithreaded RPC Application
Server Application Thread (listening)
Call Thread (available)
Client Application Thread
Calling Code
RPC Thread
Remote Server
Call Thread
Called Remote Procedure
Client Application Thread
Calling Code
RPC Thread
Remote Server
Call Thread
Called Remote Procedure
Local Application Thread (engaged in non-RPC activity)
Single Address Space

The application threads in the preceding figure are performing the following activities:

1. The server application thread is listening for calls.
2. A call thread is available to execute an incoming remote procedure call.

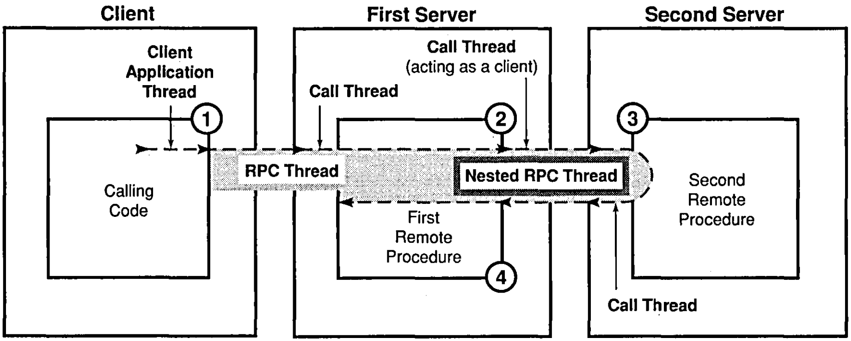
3. One client application thread has separated from an RPC thread and another is currently part of an RPC thread.
4. A local application thread is engaging in non-RPC activity.

14.3 Nested Remote Procedure Calls

A called remote procedure can call another remote procedure. The call to the second remote procedure is nested within the first call; that is, the second call is a nested remote procedure call. A nested call involves the following general phases (see Figure 14-9):

1. A client makes an initial remote procedure call to the first remote procedure.
2. The first remote procedure makes a nested call to the second remote procedure.
3. The second remote procedure executes the nested call and returns it to the first remote procedure.
4. The first remote procedure then resumes executing the initial call.

Figure 14–9. Phases of a Nested RPC Call

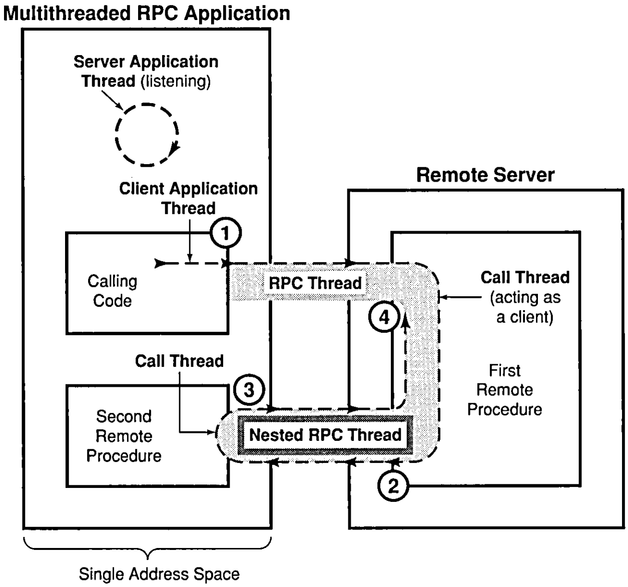


A specialized form of a nested remote procedure call involves a called remote procedure that is making a remote procedure call to the address space of the calling client application thread. Calling the client's address space requires that a server application thread be listening in that address space. Also, the second remote procedure needs a server binding handle for the address space of the calling client.

The remote procedure can ask the local RPC runtime to convert the client binding handle, provided by the server runtime, into a server binding handle. This is done by calling the **rpc_binding_server_from_client( )** routine. This routine returns a partially bound binding handle (the server binding information lacks an endpoint). For a nested remote procedure call to find the address space of the calling client, the application must ensure that the partially bound binding handle is filled in with the endpoint of that address space. The reference page for the **rpc_binding_server_from_client( )** routine in the *OSF DCE Application Development Reference* discusses alternatives for ensuring that the endpoint is obtainable for a nested remote procedure call.

Using the server binding handle, a remote procedure can attempt a nested remote procedure call. The nested call involves the general phases illustrated by Figure 14-10.

Figure 14–10. Phases of a Nested RPC Call to Client Address Space



The application threads in the preceding figure are performing the following activities:

1. A client application thread from a multithreaded RPC application makes an initial remote procedure call to the first remote procedure.
2. After converting the client binding handle into a server binding handle and obtaining the endpoint for the address space of the calling client application thread, the first remote procedure makes a nested call to the second remote procedure at that address space.
3. The second remote procedure executes the nested call and returns it to the first remote procedure.
4. The first remote procedure then resumes executing the initial call.

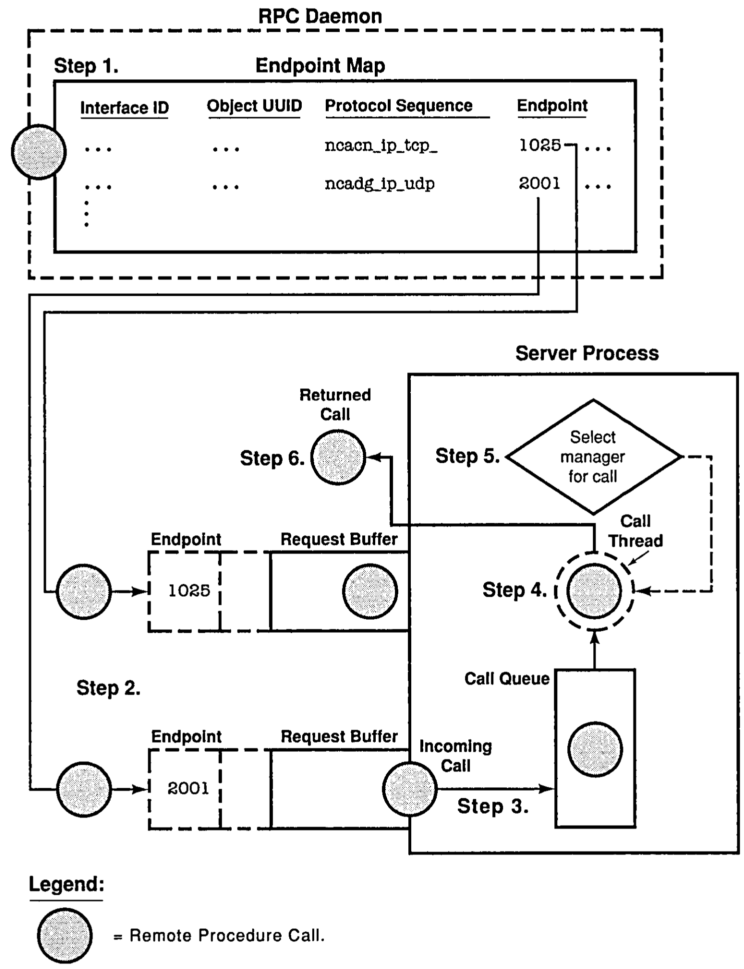
14.4 Routing Remote Procedure Calls

The following subsections discuss routing incoming remote procedure calls between their arrival at a server's system and the server's invocation of the requested remote procedure. The following routing steps are discussed:

1. If a client has a partially bound server binding handle, before sending a call request to a server, the client runtime must get the endpoint of a compatible server from the endpoint map service of the server's system. This endpoint becomes the server address for a call request.
2. When the request arrives at the endpoint, the server's system places it in a request buffer belonging to the corresponding server.
3. As one of its scheduled tasks, the server gets the incoming calls from the request buffer. The server either accepts or rejects an incoming call, depending on available resources. If no call thread is available, an accepted call is queued to wait its turn for an available call thread.
4. The server then allocates an available call thread to the call.
5. The server identifies the appropriate manager for the called remote procedure and invokes the procedure in that manager to execute the call.
6. When the call thread finishes executing a call, the server returns the call's output arguments and control to the client.

Figure 14-11 illustrates these steps.

Figure 14–11. Steps in Routing Remote Procedure Calls



The concepts in the following subsections are for the advanced RPC developer. Section 14.4.1 discusses how clients obtain endpoints when using partially bound binding handles. Sections 14.4.2 and 14.4.3 discuss how a system buffers call requests and how a server queues incoming calls;

this information is relevant mainly to advanced RPC developers. Section 14.4.4 discusses how a server selects the manager to execute a call; it is relevant for developing an application that implements an interface for different types of RPC objects.

14.4.1 Obtaining an Endpoint

The RPC daemon provides the endpoint map service that maintains the local endpoint map. The endpoint map is composed of elements. Each map element contains fully bound server binding information for a potential binding and an associated interface identifier and object UUID, which may be nil. Optionally, a map element can also contain an annotation such as the interface name.

Servers use the local endpoint map service to register their binding information. Each interface for which a server must register binding information requires a separate call to an **rpc_ep_register...()** routine, which calls the endpoint map service. The endpoint map service uses a new map element for every combination of binding information specified by the server. Figure 14-12 shows the correspondence between server binding information specified by a server and a graphic representation of the resulting endpoint map elements.

Figure 14–12. Mapping Information and Corresponding Endpoint Map Elements

Server's Inputs to Endpoint Register Operation

interface_handle	**Interface ID:** 2FAC8900-31F8-11CA-B331-08002B13D56D,1.0
*binding_handle_list**	**Server Addresses:** ncacn_ip_tcp:16.20.15.25[1025] ncadg_ip_udp:16.20.15.25[2001]
object_UUID_list	**Object UUIDs:** 47F40D10-E2E0-11C9-BB29-08002B0F4528 30DBEEA0-FB6C-11C9-8EEA-08002B0F4528 16977538-E257-11C9-8DC0-08002B0F4528

* Binding handles also enable the endpoint map service to learn the server's RPC protocol version and transfer syntaxes; this information is identical for every map element, however, and is ignored here to simplify the following representation of endpoint map elements.For the same reason, the network address of the server's host system is omitted from this representation of map elements.

Corresponding Representation of Endpoint Map Elements

Interface ID	Object UUID	Protocol Sequence	Endpoint
2FAC8900-31F8-11CA-B331-08002B13D56D, 1.0	47F40D10-E2E0-11C9-BB29-08002B0F4528	ncacn_ip_tcp	1025
2FAC8900-31F8-11CA-B331-08002B13D56D, 1.0	47F40D10-E2E0-11C9-BB29-08002B0F4528	ncadg_ip_udp	2001
2FAC8900-31F8-11CA-B331-08002B13D56D, 1.0	16977538-E257-11C9-8DC0-08002B0F4528	ncacn_ip_tcp	1025
2FAC8900-31F8-11CA-B331-08002B13D56D, 1.0	16977538-E257-11C9-8DC0--08002B0F4528	ncadg_ip_udp	2001
2FAC8900-31F8-11CA-B331-08002B13D56D, 1.0	30DBEEA0-FB6C-11C9-8EEA-08002B0F4528	ncacn_ip_tcp	1025
2FAC8900-31F8-11CA-B331-08002B13D56D, 1.0	30DBEEA0-FB6C-11C9-8EEA-08002B0F4528	ncadg_ip_udp	2001

A remote procedure call made with server binding information that lacks an endpoint uses an endpoint from the endpoint map service. This endpoint must come from binding information of a compatible server. The map element of a compatible server contains the following:

- A compatible interface identifier

  The requested interface UUID and compatible version numbers are necessary. For the version to be compatible, the major version number requested by the client and registered by the server must be identical and the requested minor version number must be less than or equal to the registered minor version number.

- The requested object UUID, if registered for the interface
- A server binding handle that refers to compatible binding information that contains the following:
  - — A protocol sequence from the client's server binding information
  - — The same RPC protocol major version number that the client runtime supports
  - — At least one transfer syntax that matches one used by the client's system

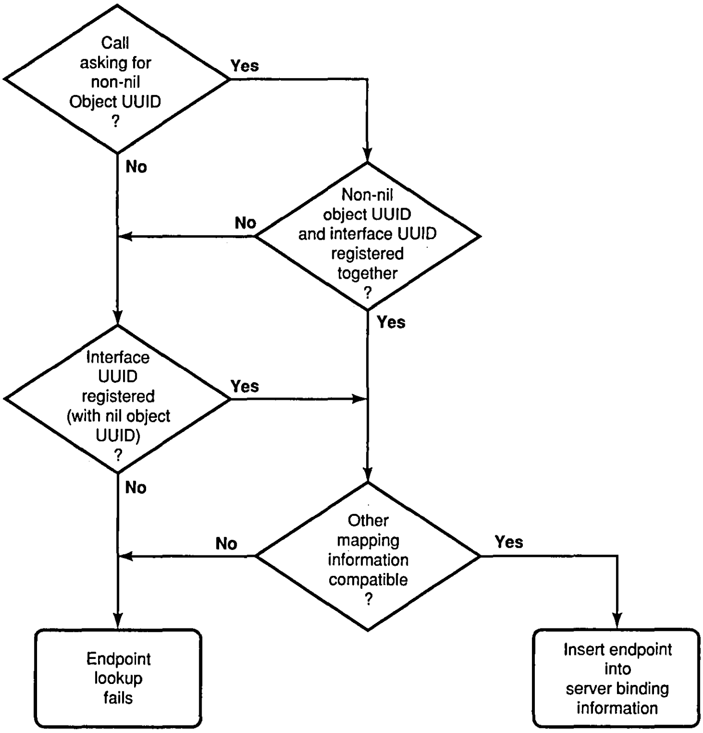
To identify the endpoint of a compatible server, the endpoint service uses the following rules:

1. If the client requests a non-nil object UUID, the endpoint map service begins by looking for a map element that contains both the requested interface UUID and object UUID.
   a. On finding an element containing both of the UUIDs, the endpoint map service selects the endpoint from that element for the server binding information used by the client.
   b. If no element contains both UUIDs, the endpoint map service discards the object UUID and starts over (see rule 2).
2. If the client requests the nil object UUID (or if the requested non-nil object UUID is not registered), the endpoint map service looks for an element containing the requested interface UUID and the nil object UUID.
   a. On finding that element, the endpoint map service selects the endpoint from the element for the client's server binding information.
   b. If no such element exists, the lookup fails.

The RPC protocol service inserts the endpoint of the compatible server into the client's server binding information.

Figure 14-13 illustrates the decisions the endpoint map service makes when looking up an endpoint for a client.

Figure 14–13. Decisions for Looking Up an Endpoint


Call asking for non-nil Object UUID ?
Yes
No
Non-nil object UUID and interface UUID registered together ?
No
Yes
Interface UUID registered (with nil object UUID) ?
Yes
No
Other mapping information compatible ?
No
Yes
Endpoint lookup fails
Insert endpoint into server binding information

You can design a server to allow the coexistence on a host system of multiple interchangeable instances of a server. Interchangeable server instances are identical, except for their endpoints; that is, they offer the same RPC interfaces and objects over the same network (host) address and protocol sequence pairs. For clients, identical server instances are fully interchangeable.

Usually, for each such combination of mapping information, the endpoint map service stores only one endpoint at a time. When a server registers a new endpoint for mapping information that is already registered, the endpoint map service replaces the old map element with the new one.

For interchangeable server instances to register their endpoints in the local endpoint map, they must instruct the endpoint map service not to replace any existing elements for the same interface identifier and object UUID. Each server instance can create new map elements for itself by calling the **rpc_ep_register_no_replace( )** routine.

When a client uses a partially bound binding handle, load sharing among interchangeable server instances depends on the RPC protocol the client is using.

- Connectionless (datagram) protocol

  The map service selects the first map element with compatible server binding information. If necessary, a client can achieve a random selection among all the map elements with compatible binding information. However, this requires that before making a remote procedure call, the client needs to resolve the binding by calling the **rpc_ep_resolve_binding( )** routine.

- Connection-oriented protocol

  The client RPC runtime uses the **rpc_ep_resolve_binding( )** routine, and the endpoint map service selects randomly among all the map elements of compatible servers.

For an alternative selection criteria, a client can call the **rpc_mgmt_ep_elt_inq_{begin,next,done}( )** routines and use an application-specific routine to select from among the binding handles returned to the client.

When a server stops running, its map elements become outdated. Although the endpoint map service routinely removes any map element containing an outdated endpoint, a lag time exists when stale entries remain. If a remote procedure call uses an endpoint from an outdated map element, the call fails to find a server. To avoid clients getting stale data from the endpoint map, before a server stops, it should remove its own map elements.

A server also has the option of removing any of its own elements from the local endpoint map and continuing to run. In this case, an unregistered endpoint remains accessible to clients that know it.

14.4.2 Buffering Call Requests

Call requests for RPC servers come into the RPC runtime over the network. For each endpoint that a server registers (for a given protocol sequence), the runtime sets up a separate request buffer. A request buffer is a first-in, first-out queue where an RPC system temporarily stores call requests that arrive at an endpoint of an RPC server. The request buffers allow the runtime to continue to accept requests during heavy activity. However, a request buffer may fill up temporarily, causing the system to reject incoming requests until the server fetches the next request from the buffer. In this case, the calling client can try again, with the same server or a different server. The client does not know why the call is rejected, nor does the client know when a server is available again.

Each server process regularly dequeues requests, one by one, from all of its request buffers. At this point, the server process recognizes them as incoming calls. The interval for removing requests from the buffers depends on the activities of the system and of the server process.

How the runtime handles a given request depends partly on the communications protocol over which it arrives, as follows:

- A call over a connectionless transport is routed by the server's system to the call request buffer for the endpoint specified in the call.
- A call over a connection-oriented transport may be routed by the server's system to a request buffer or the call may go directly to the server process.

  Whether a remote procedure call goes to the request buffer depends on whether the client sends the call over an established connection. If a client makes a remote procedure call without an established connection, the server's system treats the call request as a connection request and places the call request into a request buffer. If an established connection is available, the client uses it for the remote procedure call; the system handles the call as an incoming call and sends it directly to the server process that owns the connection.

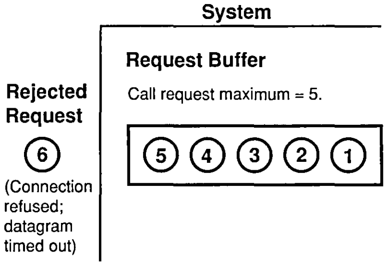
Whether a server gets an incoming call from a request buffer or over an existing connection, the server process manages the call identically. A server process applies a clear set of call-routing criteria to decide whether to dispatch a call immediately, queue it, or reject it (if the server is extremely busy). These call-routing criteria are discussed in Section 14.4.3 on routing incoming calls.

When telling the RPC runtime to use a protocol sequence, a server specifies the number of calls it can buffer for the specified communications protocol (at a given endpoint). Usually, it is best for a server to specify a default buffer size, represented by a literal whose underlying value depends on the communications protocol. The default equals the capacity of a single socket used for the protocol by the server's system.

The default usually is adequate to allow the RPC runtime to accept all the incoming call requests. For a well-known endpoint, the size of a request buffer cannot exceed the capacity of a single socket descriptor (the default size); specifying a higher number causes a runtime error. For well-known endpoints, specify the default for the maximum number of call requests.

For example, consider the request buffer at full capacity as represented in Figure 14-14. This buffer has the capacity to store five requests. In this example, the buffer is full, and the runtime rejects incoming requests, as is happening to the sixth request.

Figure 14–14. A Request Buffer at Full Capacity


System
Request Buffer
Call request maximum = 5.
Rejected Request
6
(Connection refused; datagram timed out)
5
4
3
2
1

14.4.3 Queuing Incoming Calls

Each server process uses a first-in, first-out call queue. When the server is already executing its maximum number of concurrent calls, it uses the queue to hold incoming calls. The capacity of queues for incoming calls is implementation dependent; most implementations offer a small queue capacity, which may be a multiple of the maximum number of concurrently executing calls.

A call is rejected if the call queue is full. The appearance of the rejected call depends on the RPC protocol the call is using, as follows:

- Connectionless (datagram) protocol

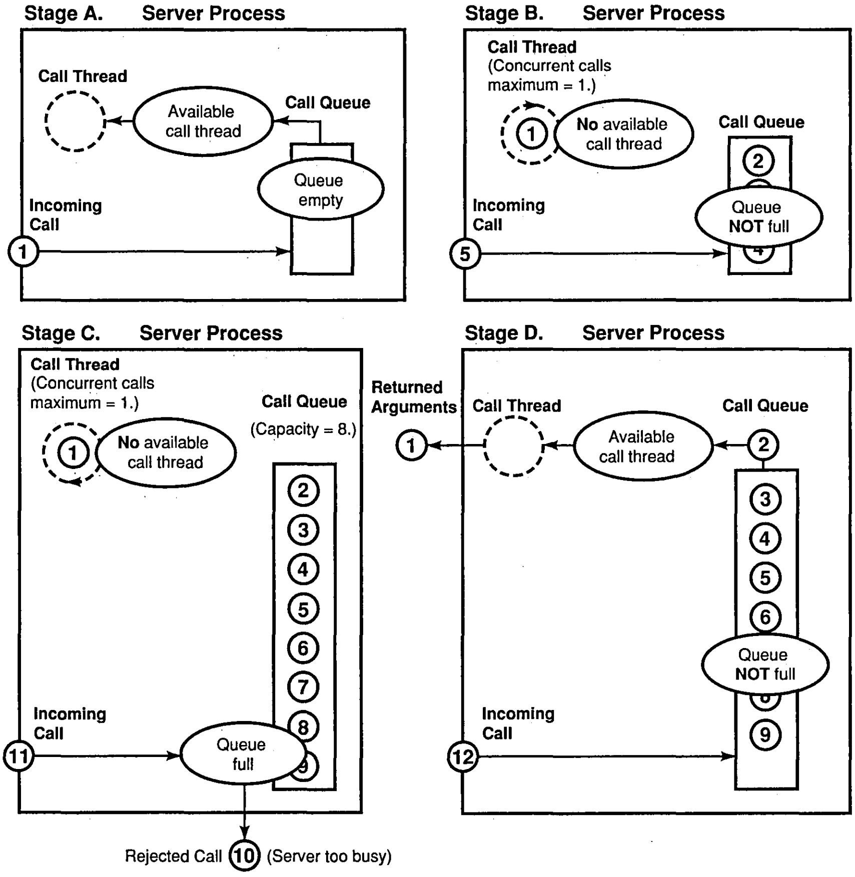
  The server does not notify the client about this failure. The call fails as if the server does not exist, returning an **rpc_s_comm_failure** communications status code (**rpc_x_comm_failure** exception).

- Connection-oriented protocol

  The server rejects the call with an **rpc_s_server_too_busy** communications status code (**rpc_x_server_too_busy** exception).

The server process routes each incoming call as it arrives. Call routing is illustrated by the server in Figure 14-15. This server has the capacity to execute only one call concurrently. Its call queue has a capacity of eight calls. This figure consists of four stages (A through D) of call routing by a server process. On receiving any incoming call, the server begins by looking at the call queue.

Figure 14–15. Stages of Call Routing by a Server Process



The activities of the four stages in the preceding figure are described as follows:

1. In stage **A**, call **1** arrives at a server that lacks any other calls. When the call arrives, the queue is empty and a call thread is available. The server accepts the call and immediately passes it to a call thread. The requested remote procedure executes the call in that thread, which becomes temporarily unavailable.

2. In stage **B**, call **5** arrives. The call queue is partially full, so the server accepts the call and adds it to the end of the queue.
3. In stage C, call **11** arrives. The queue is full, so the server rejects this call, as it rejected the previous call, **10**. (The caller can try again with the same or a different server.)
4. In stage **D**, the called procedure has completed call **1**, making the call thread available. The server has removed call **2** from the queue and is passing it to the call thread for execution. Thus, the queue is partially empty as call **12** arrives, so the server accepts the call and adds it to the queue.

14.4.4 Selecting a Manager

Unless an RPC interface is implemented for more than one specific type of object, selecting a manager for an incoming call is a simple process. When registering an interface with a single manager, the server specifies the nil type UUID for the manager type.[1] In the absence of any other manager, all calls, regardless of whether they request an object, go to the nil type manager.

The situation is more complex when a server registers multiple managers for an interface. The server runtime must select from among the managers for each incoming call to the interface. The DCE RPC dispatching mechanism requires a server to set a non-nil type UUID for a set of objects and for any interface that will access the objects in order to register a manager with the same type UUID.

To dispatch an incoming call to a manager, a server does the following:

1. If the call contains the nil object UUID, the server looks for a manager registered with the nil type UUID (the nil type manager).
    a. If the nil type manager exists for the requested interface, the server dispatches the call to that manager.

[1] The API uses **NULL** to specify a synonym to the address of the nil UUID, which contains only zeros.

   b. Otherwise, the server rejects the call.

2. If the call contains a non-nil object UUID, the server looks to see whether it has set a type for the object (by assigning a non-nil type UUID).

   If the object lacks a type, the server looks for the nil type manager.

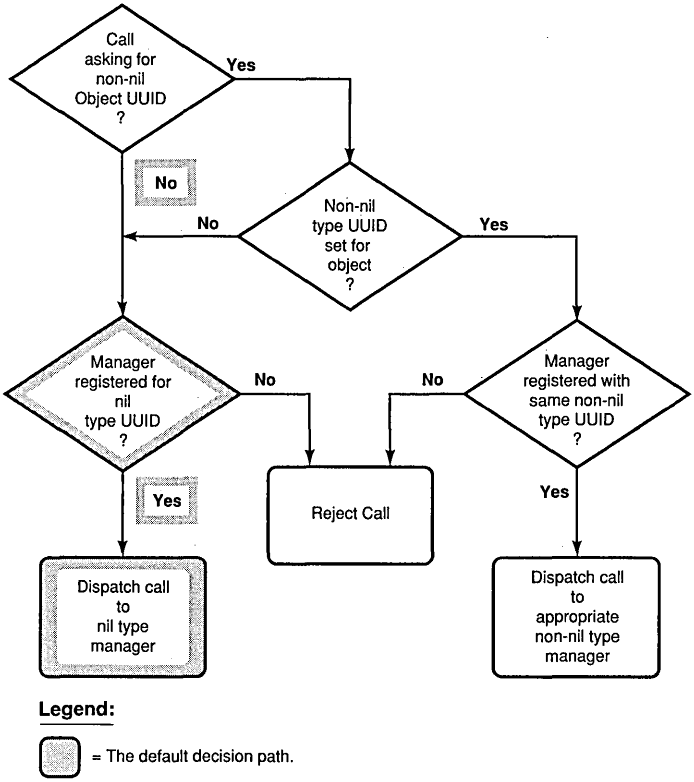
   a. If the nil type manager exists for the requested interface, the server dispatches the call to that manager.

   b. Otherwise, the server rejects the call.

3. If the object has a type, the call requires a remote procedure of a manager whose type matches the object's type. In its absence, the RPC runtime rejects the call.

Figure 14-16 illustrates the decisions a server makes to select a manager to which to dispatch an incoming call.

Figure 14–16. Decisions for Selecting a Manager


Call asking for non-nil Object UUID ?
Yes
No
Non-nil type UUID set for object ?
No
Yes
Manager registered for nil type UUID ?
No
No
Manager registered with same non-nil type UUID ?
Yes
Reject Call
Yes
Dispatch call to nil type manager
Dispatch call to appropriate non-nil type manager
Legend:
= The default decision path.

Chapter 15

Name Service Interface Usage

This chapter discusses how the DCE RPC Name Service Interface (NSI) configures directory service entries and how RPC applications can use those entries. The chapter considers the following topics:

- Directory service entries defined by NSI

  Describes the kinds of directory service entries NSI defines

- Strategies for using directory service entries

  Outlines strategies for using each kind of entry

- Models for defining RPC servers

  Introduces NSI usage models intended to guide application developers in assessing how to best use NSI for a given application

For information on the structure of NSI directory service entries and the mechanics of NSI searches, see Chapter 14, which describes advanced DCE RPC topics.

15.1 NSI Directory Service Entries

To store information about RPC servers, interfaces, and objects, NSI defines the following directory service entries in the namespace: server entries, groups, and profiles.

- A server entry is a directory service entry that stores binding information and object UUIDs for an RPC server.
- A group is a directory service entry that corresponds to one or more RPC servers that offer one or more RPC interfaces, type of RPC object, or both in common.
- A profile is a directory service entry that defines search paths in a namespace for a server that offers a particular RPC interface and object.

Note: When NSI is used with the Cell Directory Service (CDS), the directory service entries are CDS objects.

The use of server entries, groups, and profiles determines how clients view servers. A server describes itself to its clients by exporting binding information associated with interfaces and objects to one or more server entries. A group corresponds to servers that offer a given interface, service, or object. Profiles enable clients to access alternative directory service entries when searching for an interface or object. Used together, groups and profiles offer sophisticated ways for RPC applications to maintain and use directory service data.

15.1.1 Structure of Entry Names

Each entry in a namespace is identified by a unique global name comprising a cell name and a cell-relative name.

A cell is a group of users, systems, and resources that share common DCE services. A cell configuration includes at least one Cell Directory Server, one Security Server, and one Distributed Time Server. A cell's size can range from one system to thousands of systems. A host is assigned to its cell by a DCE configuration file. For information on cells, see the *OSF DCE Administration Guide*.

The following is an example of a global name:

/.../C=US/O=uw/OU=MadCity/LandS/anthro/Stats_host_2

The parts of a global name are as follows:

- Cell name (using X.500 name syntax):

  /.../C=US/O=uw/OU=MadCity

  The symbol **/...** begins a cell name. The letters before the = (equal signs) are abbreviations for Country (**C**), Organization (**O**), and Organization Unit (**OU**). For entries in the local cell, the cell name can be represented by a **/.:** prefix, in place of the actual cell name; for example:

  /.:/LandS/anthro/Stats_host_2

  The / (slash) to the right of the cell name represents the root of the cell directory (the cell root).

  For NSI operations on entries in the local cell you can omit the cell name.

- Cell-relative name (using DCE name syntax):

  Each directory service entry requires a cell-relative name, which contains a directory pathname and a leaf name.

  — A directory pathname follows the cell name and indicates the hierarchical relationship of the entry to the cell root.

  The directory pathname contains the names of any subdirectories in the path; each subdirectory name begins with a / (slash), as follows:

  /sub-dir-a-name/sub-dir-b-name/sub-dir-c-name

  Directory pathnames are created by directory service administrators. If an appropriate directory pathname does not exist, ask your directory service administrator to extend an existing pathname or create a new pathname. In a directory pathname, the name of a subdirectory should reflect its relationship to its parent directory (the directory that contains the subdirectory).

— A leaf name identifies the specific entry.

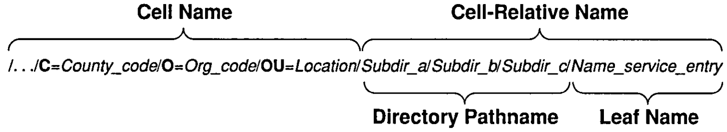
The leaf name constitutes the right-hand part of a global name, beginning with the rightmost / (slash).

For example, **/.:/LandS/anthro/Cal_host_4**, where **/.:/** represents the cell name, **/LandS/anthro** is the directory pathname, and **/Cal_host_4** is the leaf name. If the directory service entry is located at the cell root, the leaf name directly follows the cell name; for example, **/.:/cell_profile**.

Note: When NSI is used with CDS, the cell-relative name is a CDS name.

Figure 15-1 shows the parts of a global name.

Figure 15–1. Parts of a Global Name


Cell Name
Cell-Relative Name
/. . ./C=County_code/O=Org_code/OU=Location/Subdir_a/Subdir_b/Subdir_c/Name_service_entry
Directory Pathname
Leaf Name

15.1.2 Server Entries

NSI enables any RPC server with the necessary directory service permissions to create and maintain its own server entries in a namespace. A server can use as many server entries as it needs to advertise combinations of its RPC interfaces and objects.

Each server entry must correspond to a single server (or a group of interchangeable server instances) on a given system. Interchangeable server instances are instances of the same server running on the same system that offer the same RPC objects (if any). Only interchangeable server instances can share a server entry.

Each server entry must contain binding information. Every combination of protocol sequence and network addressing information represents a potential binding. The network addressing information can contain a network address, but lacks an endpoint, making the address partially bound.

A server entry can also contain a list of object UUIDs exported by the server. Each of the object UUIDs corresponds to an object offered by the server. In a given server entry, each interface identifier is associated with every object UUID, but with only the binding information exported with the interface.

Figure 15-2 represents a server entry. This server entry was created by two calls to the **rpc_ns_binding_export( )** routine. The first call created the first column of the top half of the figure. The routine's *binding_vec* parameter had three elements, each of which is paired with the routine's *if_handle* parameter. The vertical ellipsis points under the last box indicate that more elements in the routine's *binding_vec* parameter would have resulted in more interface UUID/binding information pairs in the first column.

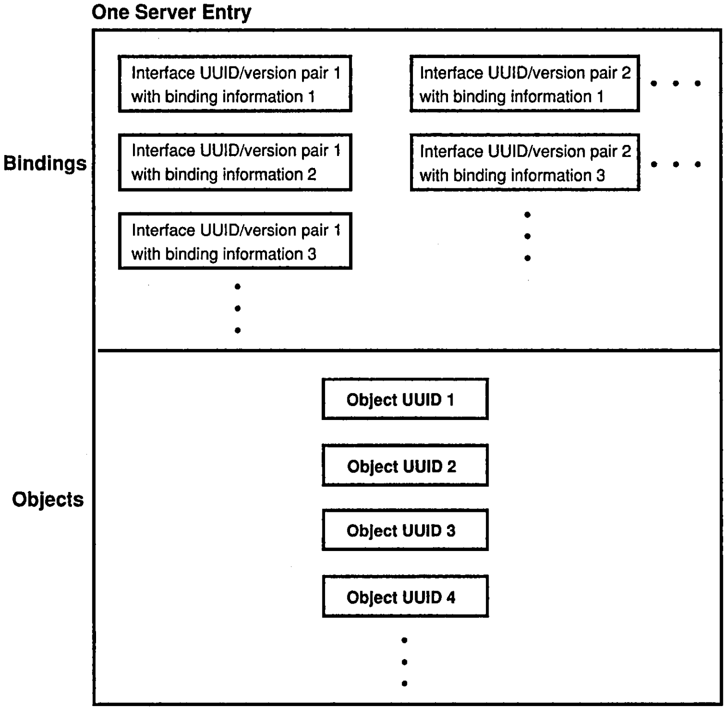
Similarly, the second call to the **rpc_ns_binding_export( )** routine created the second column of the top half of the figure. The routine's *binding_vec* parameter had two elements, each of which is paired with the routine's *if_handle* parameter. For example, the first element could have contained binding information with the **ncacn_ip_tcp** protocol sequence, and the second element could have contained binding information with the **ncadg_ip_udp** protocol sequence. As in the first column, more elements in the routine's *binding_vec* parameter would have resulted in more interface UUID/binding information pairs.

Third and subsequent calls to the **rpc_ns_binding_export( )** routine would create more columns; the two pairs of horizontal ellipsis points indicate this expansion.

Finally, note that the **rpc_ns_binding_export( )** routine optionally takes a vector of object UUIDs. The four object UUIDs in the bottom half of the figure came from the two calls to the routine, or from another call to the routine with no interface UUID/version and with no binding information, but with object UUIDs. The object UUIDs are associated with no particular binding. Instead, they are associated with all the bindings. Third and subsequent calls to the routine could create more object UUIDs; the vertical ellipsis points indicate this expansion.

Note: To distinguish among RPC objects when using the CDS ACL editor, export the RPC objects to separate directory service entries.

Figure 15–2. Possible Information in a Server Entry

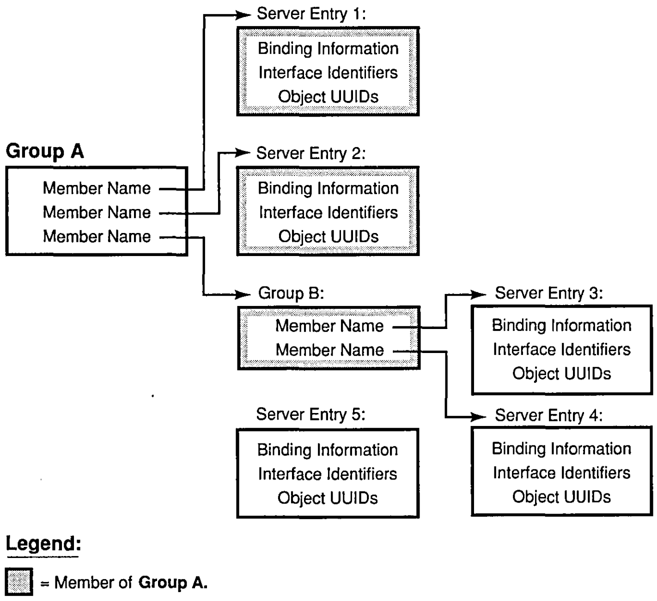

One Server Entry
Bindings
Interface UUID/version pair 1 with binding information 1
Interface UUID/version pair 2 with binding information 1
Interface UUID/version pair 1 with binding information 2
Interface UUID/version pair 2 with binding information 3
Interface UUID/version pair 1 with binding information 3
Objects
Object UUID 1
Object UUID 2
Object UUID 3
Object UUID 4

15.1.3 Groups

Administrators or users of RPC applications can organize searches of a namespace for binding information by having clients use an RPC group as the starting point for NSI search operations. A group provides NSI search operations (**import_next** or **lookup_next** operations) with access to the server entries of different servers that offer a common RPC interface or object. A group contains names of one or more server entries, other groups, or both. Since a group can contain group names, groups can be nested. Each server entry or group named in a group is a member of the group. A group's members must offer one or more RPC interfaces, the type of RPC object, or both in common.

Figure 15-3 shows an example of the kinds of members a group can contain and how those members correspond to database entries.

Figure 15–3. Possible Mappings of a Group



The members of Group A are Server Entry 1, Server Entry 2, and Group B. The members of the nested group, Group B, are Server Entry 3 and Server Entry 4. An additional server entry that advertises the common interface or object, Server Entry 5, is omitted from either group.

15.1.4 Profiles

Administrators or users of RPC applications can organize searches of a namespace for binding information by having clients use an RPC profile as the starting point for NSI search operations. A profile is an entry in a namespace that contains a collection of profile elements. A profile element is a database record that corresponds to a single RPC interface and that refers to a server entry, group, or profile. Each profile element contains the following information:

- Interface identifier

  This field is the key to the profile. The interface identifier consists of the interface UUID and the interface version numbers.

- Member name

  The entry name of one of the following kinds of directory service entries:

  — A server entry for a server offering the requested RPC interface

  — A group corresponding to the requested RPC interface

  — A profile

- Priority value

  The priority value (0 is the highest priority; 7 is the lowest priority) is designated by the creator of a profile element to help determine the order for using the element NSI search operations to select among like-priority elements at random.

- Annotation string

  The annotation string enables you to identify the purpose of the profile element. The annotation can be any textual information; for example, an interface name associated with the interface identifier or a description of a service or resource associated with a group.

  Unlike the interface identifier field, the annotation string is not a search key.

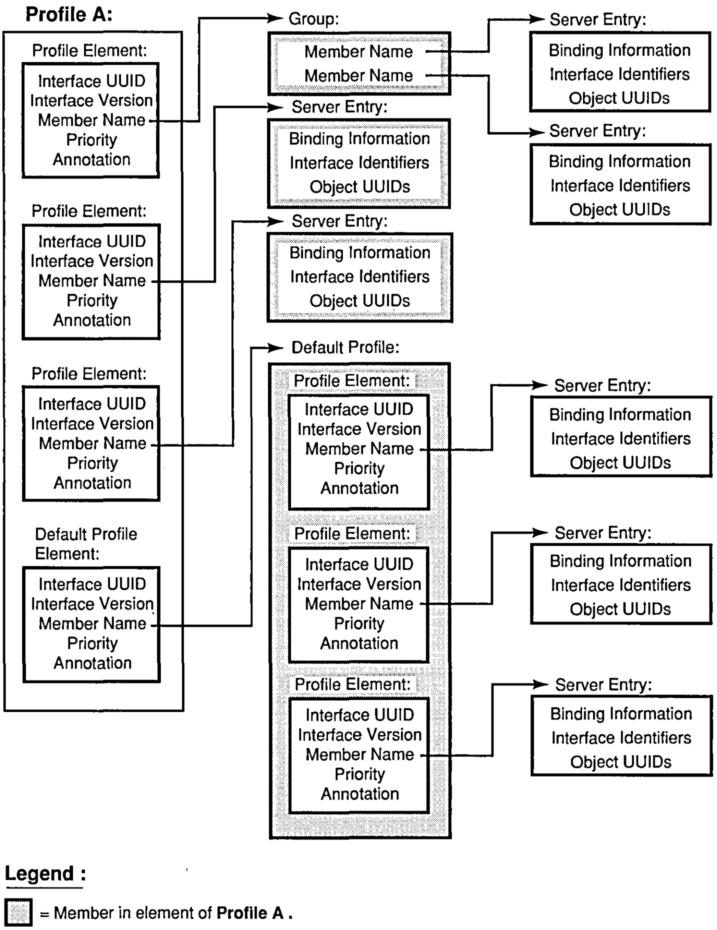
Optionally, a profile can contain one default profile element. A default profile element is the element that an NSI search operation uses when a search using the other elements of a profile finds no compatible binding information; for example, when the current profile lacks any element corresponding to the requested interface. A default profile element contains the nil interface identifier, a priority of 0, the entry name of a default profile, and an optional annotation.

A default profile is a backup profile, referred to by a default profile element in another profile. A profile designated as a default profile should be a comprehensive profile maintained by an administrator for a major set of users, such as the members of an organization or the owners of computer accounts on a local area network.

A default profile must not create circular dependencies between profiles; for example, when a public profile refers to an application's profile, the application's profile must not specify that public profile as a default profile.

Figure 15-4 shows an example of the kinds of elements a profile can contain and how those elements correspond to database entries.

Figure 15–4. Possible Mappings of a Profile



NSI search operations use a profile to construct an NSI search path. When an NSI search operation reads a profile, the operation dynamically constructs its NSI search path from the set of elements that correspond to a common RPC interface.

A profile element is used only once per NSI search path. The construction of NSI search paths depends partly on the priority rankings of the elements.

A search operation uses higher priority elements before lower priority elements. Elements of equal priority are used in random order, permitting some variation in the NSI search paths between searches for a given interface. If nondefault profile elements do not satisfy a search, the search path extends to the default profile element, if any. For samples of NSI search paths generated from profile elements, see Chapter 14, which describes advanced DCE RPC topics.

Profiles meet the needs of particular individuals, systems, LANs, sites, organizations, and so forth, with minimal configuration management. The administrator of a profile can set up NSI search paths that reflect the preferences of the profile's user or users. The profile administrator can set up profile elements that refer (directly or indirectly) to only a subset of the server entries that offer a given RPC interface. Also, the administrator can assign different search priorities to the elements for an interface.

15.1.5 Guidelines for Constructing Names of Directory Service Entries

A global name includes both a cell name and a cell-relative name composed of a directory pathname and a leaf name. The cell name is assigned to a cell root at its creation. When you specify only a cell-relative name to an NSI operation, NSI automatically expands the name into a global name by inserting the local cell name. When returning the name of a directory service entry, a group member, or a member in a profile element, NSI operations return global names.

The directory pathname and leaf name uniquely identify a directory service entry. The leaf name should somehow describe the entry; for example, by identifying its owner or its contents. The remainder of this section contains guidelines for choosing leaf names.

Note: Directory pathnames and leaf names are case sensitive.

Use the following guidelines for constructing names:

- Naming a server entry

  For a server entry that advertises an RPC interface or service offered by a server, the leaf name must distinguish the entry from the equivalent entries of other servers. When a single server instance runs on a host, you can ensure a unique name by combining the name of the service, interface (from the interface definition), or the system name for the server's host system.

For example, consider two servers, one offering a calendar service on host JULES, and one on host VERNE.

The server on JULES uses the following leaf name:

calendar_JULES

The server on VERNE uses the following leaf name:

calendar_VERNE

For servers that perform tasks on or for a specific system, an alternative approach is to create server entries in a system-specific host directory within the namespace. Each host directory takes the name of the host to which it corresponds. Because the directory name identifies the system, the leaf name of the server entry name does not need to include the host name, for example:

/.:/LandS/host_1/Process_control

To construct names for the server entries used by distinctive server instances on a single host, you can construct unique server entry names by combining the following information: the name of the server's service, interface, or object; the system name of the server's host system; and a reusable instance identifier such as an integer.

For example, the following leaf names distinguish two instances of a calendar service on the JULES system:

calendar_JULES_01

calendar_JULES_02

Avoid automatically generating entry names for the server entries of server instances; for example, by using unique data such as a timestamp (**calendar_verne_15OCT91_21:25:32**) or a process identifier (**calendar_jules_208004D6**). When a server incorporates such unique data into its server entry names, each server instance creates a separate server entry, causing many server entries. When a server instance stops running, it leaves an obsolete server entry that is not reused. The creation of a new entry whenever a server instance starts may impair performance.

A server can use multiple server entries to advertise different combinations of interfaces and objects. For example, a server can create a separate server entry for a specific object, and the associated interfaces. The name of such a server entry should correspond to a well-known name for the object. For example, consider a server that offers a horticulture bulletin board known to users as **horticulture_bb**. The server exports the **horticulture_bb** object, binding information, and the associated bulletin-board interface to a server entry whose leaf name identifies the object, as follows:

horticulture_bb

Note: An RPC server that uses RPC authentication can choose identical names for its principal name and its server entry. Use of identical names permits a client that calls the **rpc_binding_set_auth_info**( ) routine to automatically determine a server's principal name. (The client will assume the principal name to be the same as the server's entry name.) If a server uses different principal and server entry names, users must explicitly supply the principal name. For an explanation of principal names, see Part 6 of this guide.

- Naming a group

  The leaf name of a group should indicate the interface, service, or object that determines membership in the group. For example, for a group whose members are selected because they advertise an interface named **Statistics**, the following is an effective leaf name:

  Statistics

For a group whose members advertise laser printer print queues as objects, the following is an effective leaf name:

laser-printer

- Naming a profile

  The leaf name of a profile should indicate the profile users; for example, for a profile that serves the members of an accounting department, the following is an effective leaf name:

  accounting_profile

The following text describes the NSI **begin**, **next**, and **done** operations. NSI accesses a variety of search and inquire operations that read NSI attributes in directory service entries. An NSI attribute is an RPC-defined attribute of a directory service entry used by the DCE RPC directory service interface. An NSI attribute stores one of the following: binding information, object UUIDs, a group, or a profile. Reading information from any attribute involves an equivalent set of search or inquire operations; that is, an integral set of **begin**, **next**, and **done** operations. An RPC application uses these operations as follows:

1. The application creates a directory service handle (a reference to the context of the ensuing series of **next** operations) by calling an NSI **begin** operation.

2. The application calls the NSI **next** operation corresponding to the **begin** operation one or more times. Each **next** operation returns another value or list of values from the target RPC directory service attribute. For example, an **import_next** operation returns binding information from a binding attribute and an object from an object attribute.

   Each call to an NSI **next** operation requires the directory service handle created in the associated NSI **begin** operation. The directory service handle maintains state information for reading values from directory service attributes; this is analogous to the function of a file pointer in the C language.

3. The application deletes the directory service handle by calling the corresponding NSI **done** operation.

Note: Search and inquire operations are also accessible interactively from within the RPC control program.

Table 15-1 lists the NSI **next** operations used by RPC applications.

Table 15–1. NSI next Operations

Search Operation	**Attributes Traversed**
rpc_ns_binding_import_next( )	Searches for binding and object attributes of a compatible server; reads any NSI attribute in a search path. Returns a binding handle that refers to a potential binding for a compatible server, and also to a single object UUID.
rpc_ns_binding_lookup_next( )	Searches for binding and object attributes of a compatible server; reads any NSI attribute in a search path. Returns a list of binding handles, each of which refers to a potential binding for a compatible server, and also to a single object UUID. The same object UUID is associated with each potential binding. Note that after calling the **lookup_next** operation, the client must select one binding handle from the list. To select a binding handle at random, the client can call the NSI binding select routine **rpc_ns_binding_select( )**. For an alternative selection algorithm, the client can define and call its own application-specific select algorithm.
Inquire Operation	**Attributes Traversed**
rpc_ns_group_mbr_inq_next( )	Reads a group attribute and returns a member name.
rpc_ns_profile_elt_inq_next( )	Reads a profile attribute and returns the fields of a profile element.

15.1.6 Selecting the Starting Entry

When searching a namespace for an RPC interface and object, a client supplies the name of the directory service entry where the search begins. The entry can be a server entry, group, or profile. Generally, an NSI search starts with a group or profile. The group or profile defines a search path that ends at a server entry containing the requested interface identifier, object UUID, and compatible binding information.

A user may know in advance what server instance to use. In this case, starting with a server entry for the server instance is appropriate.

15.1.7 Environment Variables

DCE RPC provides predefined environment variables that a client can use for NSI operations. An environment variable is a variable that stores information, such as a name, about a particular environment. The NSI interface provides two environment variables, **RPC_DEFAULT_ENTRY** and **RPC_DEFAULT_ENTRY_SYNTAX**, which are described in the *OSF DCE Application Development Reference*. Used together, these environment variables identify an entry name and indicate its syntax.

When a client searches for binding information, the search starts with a specific entry name. Optionally, a client can specify this entry name as the value of the **RPC_DEFAULT_ENTRY** variable. A client can also specify the name syntax of the starting entry as the value of the **RPC_DEFAULT_ENTRY_SYNTAX** variable; the default name syntax is **dce**.

Note: The **dce** name syntax is the only syntax currently supported by the DCE Cell Directory Service. However, NSI is independent of any specific directory service. Therefore, NSI may support one or more alternative directory services that use different name syntaxes.

15.2 Strategies for Using Directory Service Entries

When developing an RPC application, decide how an application will use the namespace and design your application accordingly. The following subsections discuss issues associated with how servers use different types of directory service entries.

15.2.1 Using Server Entries

An application requires separate server entries for servers on different hosts. For example, if a server offering the calendar service runs on two hosts, JULES and VERNE, one server entry is necessary for the server on JULES and another is necessary for the server on VERNE.

Each server entry requires a unique cell-relative entry name. (See Section 15.1.5 for the guidelines for constructing names of directory service entries). If a server adheres to a simple and consistent arrangement of server entries, you may be able to use server initialization code to automatically generate a name for each server entry, and also to ensure that the entry exists. However, some servers will need to obtain the entry name of a server entry from an external source such as a command-line argument or a local database belonging to the application.

Note: Applications that obtain entry names and UUIDs as command-line arguments should accept user-defined values that represent them as an alternative to accepting the actual names.

Some applications, such as a process-control application, require only one server instance per system. Many applications, however, can accommodate multiple server instances on a system. When multiple instances of a server run simultaneously on a single system, all instances on a host can use a single server entry, every instance can use separate server entries, or the instances can be classified into subsets with a separate server entry. A client importing from a shared server entry cannot distinguish among the server instances that export to the entry. Therefore, the recommended strategy for a server on a given system depends on which server instances are viewed by clients as interchangeable entities and which are viewed as unique entities, as follows:

- Interchangeable server instances

  When clients consider all the server instances on a host as equivalent alternatives, all of the instances can (and should) share a server entry. For example, multiple instances of the calendar service running on host JULES can all export to the **calendar_JULES** entry.

- Unique server instances

  A unique server instance possesses a significant difference from other instances of the same host. Unique server instances require separate server entries. Each server instance must export unique information to its own server entry; this unique information can be either a server-specific, well-known endpoint or an object UUID belonging exclusively to the one server instance.

  Before exporting, each server instance must acquire the entry name of its server entry from an external source. When a unique server instance stops running, its server entry becomes available. An available server entry should be reused for a new instance of that server by providing the existing entry's name for a new server instance to use with the export operation. If any existing server entries are unavailable, a new server instance requires a new server entry name.

  For a discussion of when a server instance should remove the binding information from its server entry, see the **rpc_ns_binding_unexport(3rpc)** reference page in the *OSF DCE Application Development Reference*.

15.2.2 Using Groups

When a server is first installed on a system, the server or the installer creates one or more server entries for the server. Also, when installing the first instance of the server within a cell, the installer usually creates one or more groups for the application. For any application, the local system and directory service administrators can create site-specific groups whose members are server entries, groups, or both. Typically, a server adds a server entry to at least one group.

Design decisions for defining groups may reflect a number of possible factors. Typical factors that help define effective groups include the proximity of services or resources to clients, the types of any resources offered by servers, the uses of UUIDs, and the types of users that require a specific server.

For example, for a print server, proximity to the clients and the type of supported file formats are both relevant. These factors may affect print servers as follows:

- Proximity

  If the proximity of a server is important to clients, assign servers to groups according to their locations. For example, print servers that are located together can use their own group (for example, print servers in building 1 use the group **bldg_1_print_servers**). Each server instance can add its own entry to the group, or a system administrator can add server entries using the RPC control program.

  To select randomly among servers in a given location, a client imports using the name of a group that corresponds to those servers (or of a profile that refers to that group).

  Note: If proximity is the key factor in selecting among servers, name each server entry for the server's location; for example, **bldg_1_pole_27_print_server**.

- Object types

  When accessing specific classes of resources is important to clients, you can group server instances based on the type of object they offer.

  For servers that advertise resources in server entries, groups often use subsets for server entries according to the resources they advertise. For example, print servers can be grouped according to supported file formats. In this case, an administrator creates a group entry for each file format; for example, **post_printers**, **sixel_printers**, and **ascii_printers**. Each print server entry is a member of one or more groups.

  Users that specify a group for a file format must find the printer that processes the print command. To help the user find the printer, the client can obtain the name of the server entry that supplied the server binding information by calling **rpc_ns_binding_inq_entry_name**( ), and then display the name for the user. If the server entry name indicates the location of the print server (for example, **floor_3_room_45A_print_server**), the user can then find the printer.

An application can set up groups according to different factors for different purposes. For example, the print server application can set up groups of neighboring print servers and a group of print servers for each of the file formats. The same server is a member of at least one group of each kind. Clients requires users to specify the name of a directory service entry as a command-line argument of remote print commands. The user specifies the name of the appropriate group.

Note: If a user wants a specific print server and knows the name of its server entry, the user can specify that name to the client instead of a group.

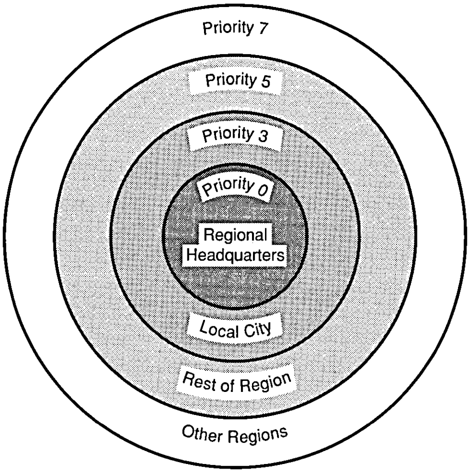
15.2.3 Using Profiles

Profiles are tools for managing NSI searches (performed by **import_next** or **lookup_next** operations). Often profiles are set up as public profiles for the users of a particular environment, such as a directory service cell, a system, a specific application, or an organization. For example, the administrator of the local directory service cell should set up a cell profile for all RPC applications using the cell, and the administrator of each system in the distributed computing environment should set up a system profile for local servers.

For each application, a directory service administrator or the owner of an application should add profile elements to the public profiles that serve the general user population; for example, a cell profile, a system profile, or a profile of an organization. Each profile element associates a profile member (represented in the member field of an element as the global name of a directory service entry) with an interface identifier, access priority, and optional annotation. A candidate for membership in a cell profile is a group or another profile; for example, a group that refers, directly or indirectly, to the servers of an application installed in the local cell or an application-specific profile.

An application may benefit from an application-specific profile. For example, an administrator at a specific location, such as a company's regional headquarters, can assign priorities to profile elements based on the proximity of servers to the headquarters, as illustrated by Figure 15-5.

Figure 15–5. Priorities Assigned on Proximity of Members



An individual user can have a personalized user profile that contains elements for interfaces the user uses regularly and a default element that specifies a public profile, such as the cell profile, as the default profile. NSI searches use the default profile when a client needs an RPC interface that lacks an element in the user profile.

15.3 Models for Defining RPC Servers

The NSI operations accommodate two distinct models for defining servers: the service model and the resource model. These models express different views of how clients use servers and how servers can present themselves in the directory service database. The models are not mutually exclusive, and an application may need to implement both models to meet diverse goals. By evaluating these models before designing an RPC application, you can make informed decisions about whether and how to use object UUIDs, how many server entries to use per server, how to distinguish among instances of a server on a system, whether and how to use groups or profiles or both, and so forth. The two models are as follows:

- Service model

  This model views a server exclusively as a distributed service composed of one or more application-defined interfaces that meet a common goal independently of specific resources.

- Resource model

  This model views servers and clients as manipulating resources. A server and its clients use object UUIDs to identify specific resources.

  With the resource model, any resource an application's servers and clients manipulate using an object UUID is considered an RPC resource. Typically, an RPC resource is a physical resource such as a database. However, an RPC resource may be abstract; for example, a print format such as ASCII. Note that an application that uses the resource model for one context may use the service model for another.

15.3.1 Service Model

The service model is used by applications whose servers offer an identical service and whose clients do not request an RPC resource when importing an interface. Often, with the service model, all the server instances of an application are equivalent and are viewed as interchangeable. However, the service model can accommodate applications that view each server instance as unique. The implications of whether server instances are viewed as interchangeable or unique are significant, so the following subsections address these alternatives separately.

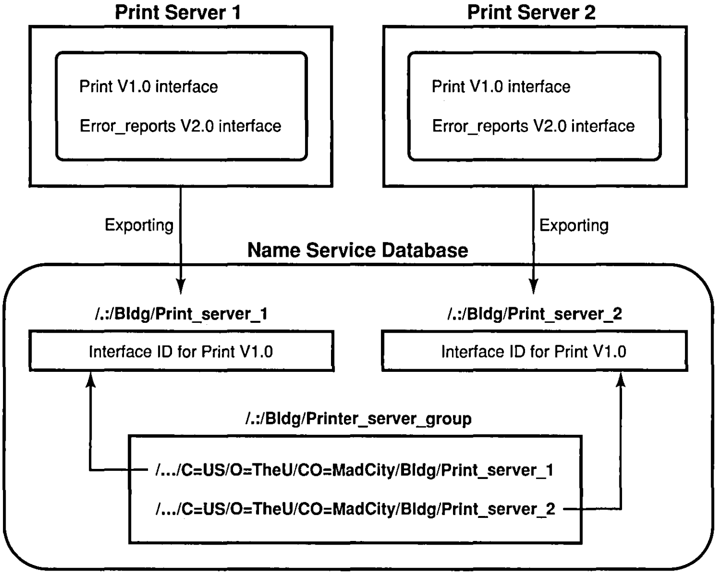
15.3.1.1 Interchangeable Server Instances

With the service model, servers offer an identical service that operates the same way on all host systems. For example, an application that uses the service model is a collection of equivalent print servers that support an identical set of file formats, and that are installed on printers in a single location. The print servers in any location can be segregated from printer servers elsewhere by using a location-specific group.

Figure 15-6 shows interchangeable print servers offering an identical print service on different hosts. To access this service, clients request the Print V1.0 interface and specify the nil object UUID. In this illustration, the starting entry for the NSI search is a group corresponding to local print servers. Note that a client may be able to reach this print server group by starting from a profile or another group.

Note: To simplify the illustrations of the usage models, the contents of server entries are represented without listing any binding information.

Figure 15–6. Service Model: Interchangeable Instances on Two Hosts


Print Server 1
Print V1.0 interface
Error_reports V2.0 interface
Print Server 2
Print V1.0 interface
Error_reports V2.0 interface
Exporting
Exporting
Name Service Database
/.:/Bldg/Print_server_1
Interface ID for Print V1.0
/.:/Bldg/Print_server_2
Interface ID for Print V1.0
/.:/Bldg/Printer_server_group
/.../C=US/O=TheU/CO=MadCity/Bldg/Print_server_1
/.../C=US/O=TheU/CO=MadCity/Bldg/Print_server_2

Search Requirements

Target Interface: Printer V1.0

Target Object: None

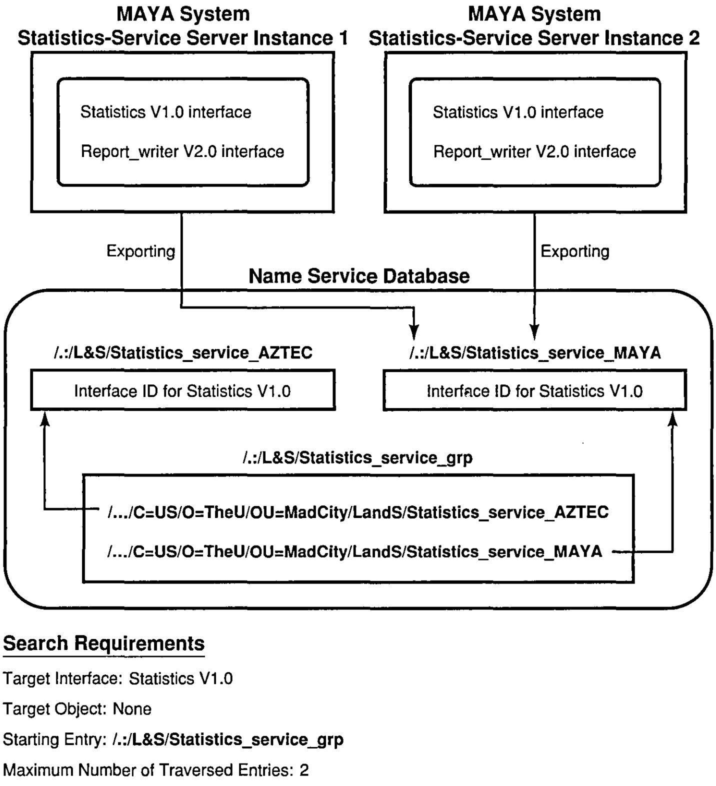
Starting Entry: **/.:/Bldg/Print_server_group**

Maximum Number of Traversed Entries: 2

Note: The number of entries traversed by a search operation is unrelated to the number of binding handles it returns.

Figure 15-7 shows interchangeable service instances offering an identical statistics service on a single host. To access this service, clients request the Statistics V1.0 interface and specify the nil object UUID. The starting entry for the NSI search is a group corresponding to local servers that offer the service (or a profile that refers to that group).

Figure 15–7. Service Model: Interchangeable Instances on One Host


MAYA System
Statistics-Service Server Instance 1
MAYA System
Statistics-Service Server Instance 2
Statistics V1.0 interface
Report_writer V2.0 interface
Statistics V1.0 interface
Report_writer V2.0 interface
Exporting
Exporting
Name Service Database
/.:/L&S/Statistics_service_AZTEC
Interface ID for Statistics V1.0
/.:/L&S/Statistics_service_MAYA
Interface ID for Statistics V1.0
/.:/L&S/Statistics_service_grp
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_AZTEC
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_MAYA
Search Requirements
Target Interface: Statistics V1.0
Target Object: None
Starting Entry: /.:/L&S/Statistics_service_grp
Maximum Number of Traversed Entries: 2

Note that if an application with interchangeable server instances uses the connectionless RPC protocol, the default behavior of the endpoint map service is to always return the endpoint from the first map element for that set of server instances. To avoid having all clients using only one of the instances, before making a remote procedure call to the server, each client must inquire for an endpoint. For a random selection, a client calls the **rpc_ep_resolve_binding**( ) routine. Alternatively, a client can call the **rpc_mgmt_ep_elt_inq_...**( ) routines to obtain all the map elements for compatible server instances, and then use an application-specific selection algorithm to select one of the returned elements.

15.3.1.2 Distinct Service Instances on a Single Host

With the service model, when multiple server instances on a given host are somehow unique, each instance must export to a separate server entry. The exported binding information must contain one or more instance-specific, well-known endpoints or an instance UUID. Well-known endpoints and instance UUIDs are used under the following circumstances:

- Well-known endpoints

  An instance-specific, well-known endpoint must be provided to a server instance as part of its installation; for example, as a command-line argument. Before calling the export operation, the server instance tells the RPC runtime to use each of its well-known endpoints; it does this by calling **rpc_server_use_protseq_ep( )**. The runtime includes these endpoints in the instance's binding information, which the runtime makes available to the instance via a list of server binding handles. The server instance uses this list of binding handles to export its binding information, including the well-known endpoints. The server also uses this list of binding handles to export its well-known endpoint with the local endpoint map; it does this by calling **rpc_ep_register( )** or **rpc_ep_register_no_replace( )**. Remote calls made using an imported well-known endpoint from a server entry are guaranteed by the RPC runtime to go only to the server instance that exported the endpoint to that entry.

  Note: Only one server instance per system can use a well-known endpoint obtained from a given interface specification.

- Instance UUID

  Create an instance UUID only for a new server entry. Generating a new instance UUID each time a server instance exports to a server entry will result in many instance UUIDs that are difficult to manage and may affect performance as new instance UUIDs are constantly added to server entries. If a new server instance inherits a currently unused server entry left behind by an earlier instance, before exporting, the new server instance should inquire for an instance UUID in the server entry; this is done by calling the **rpc_ns_entry_object_inq_{begin,next,done}( )** routines. If the inherited entry contains an instance UUID, the server uses it for an instance UUID, rather than creating and exporting a new instance

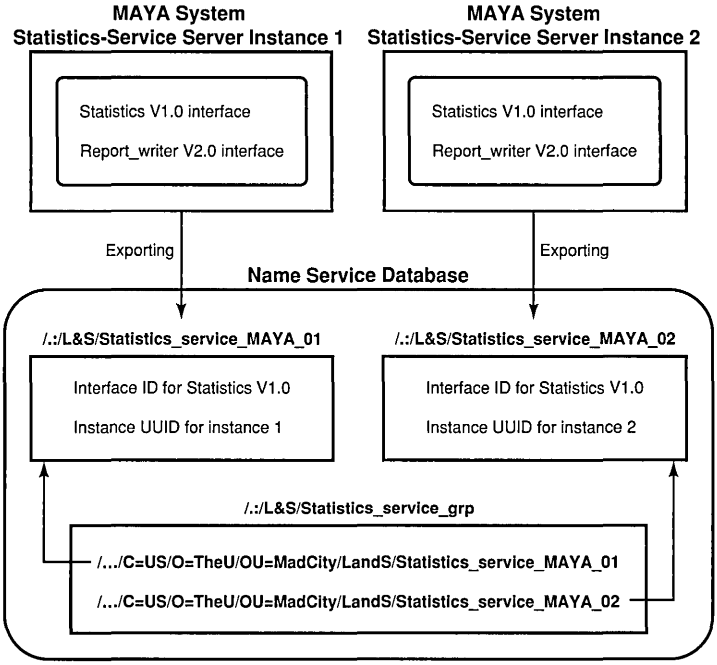
UUID. If an inherited entry lacks an instance UUID, however, the server must create a UUID and export it to the server entry. Note that every server instance must register its instance UUID along with its endpoints in the local endpoint map.

Note: Using an instance UUID precludes any other use of object UUIDs for the application.

Figure 15-8 shows distinct instances of a statistics-service server on the same host. Each server instance uses an instance UUID to identify itself to clients. The instance UUID is the only object UUID a server instance exports to its server entry. Starting at the statistics-service group, clients import the statistics interface.

After finding a server entry with compatible binding information for the statistics interface, the import operation returns an instance UUID along with binding information. Every remote procedure call made with that binding information goes to the server instance that exported the instance UUID.

Figure 15–8. Service Model: Distinct Instances on One Host


MAYA System
Statistics-Service Server Instance 1
Statistics V1.0 interface
Report_writer V2.0 interface
MAYA System
Statistics-Service Server Instance 2
Statistics V1.0 interface
Report_writer V2.0 interface
Exporting
Exporting
Name Service Database
/.:/L&S/Statistics_service_MAYA_01
Interface ID for Statistics V1.0
Instance UUID for instance 1
/.:/L&S/Statistics_service_MAYA_02
Interface ID for Statistics V1.0
Instance UUID for instance 2
/.:/L&S/Statistics_service_grp
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_MAYA_01
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_MAYA_02

Search Requirements

Target Interface: Statistics V1.0

Target Object: None

Starting Entry: **/.:/L&S/Statistics_service_grp**

Maximum Number of Traversed Entries: 2

15.3.2 Resource Model

Applications in which a client requests a server to operate on a particular RPC resource use the resource model. Each server accesses one or more resources, such as print servers or databases. Applications use object UUIDs to refer to resources as follows:

1. Servers offer resources by assigning an object UUID to each specific resource.
2. Clients obtain those object UUIDs and use them to learn about a server that offers a given resource.
3. When making a remote procedure call, a client requests a resource by passing its UUID as part of the server binding information.

Each RPC resource or type of resource requires its own object UUID. A calendar server, for example, may require a distinct UUID to identify each calendar.

RPC interfaces can be defined to operate with different types of resources and can be implemented separately for each type; for example, a print server application that supports PostScript, sixel, and ASCII file formats. When using different implementations of an interface (different managers), servers must associate the object UUID of a resource, such as an ASCII file format and its manager, by assigning them a single type UUID. To request the resource, a client specifies its object UUID in the server binding information. When a print server receives the remote procedure call, it looks up the corresponding type UUID and selects the associated manager.

Some RPC resources belong exclusively to a single server instance; for example, print queues. Some resources can be shared among server instances; for example, a file format or an airline reservation database. For server instances on the same system, sharing a resource means that its object UUID cannot distinguish between the two instances. For a print server, this is unlikely to be a problem, assuming that each printer runs only one instance of the print server. In contrast, an application with a widely accessed database, such as an airline reservation application, may need to ensure that clients can distinguish server instances from each other. An application can distinguish itself by supplying its clients with instance-specific information; for example, a well-known endpoint or an instance UUID.

Note: Multiple server instances that access the same set of resources can introduce concurrency control problems, such as two instances accessing a tape drive at the same time. Also, where the system provides concurrency control, servers may compete and have to wait for resources such as databases. Dealing with delayed access to shared resources may require an application-specific mechanism, such as queuing access requests.

15.3.2.1 Guidelines for Defining and Using RPC Resources

When developing an RPC application, you need to decide whether to use object UUIDs to identify RPC resources and, if so, what sorts of resources receive UUIDs that servers export to the namespace. When making these decisions, consider the following questions:

- Will users need to select a server entry from the namespace based on what object UUIDs the entry contains (and what the client needs)?

  If yes, then a client must specify an object UUID to the import operation.

- Does the type of resource you are using last for a long time (months or years), so you can advertise object UUIDs efficiently in the namespace?

  The information kept in a namespace should be static or rarely change. For example, print queues are appropriate RPC resources. In contrast, quickly changing information, such as the jobs queued for the printer, owners of the jobs, or the time the job was added to the queue, should not be viewed as RPC resources. Such short-lived data may be viewed as local objects, which are stored and managed at a specific server. Programming with local objects is in the area of regular object-oriented programming and is independent of an application's use of RPC resources.

- Is the number of objects belonging to the type of resource bounded in order to avoid placing high demands on the directory service?

- Will the server implement an interface for different types of a resource, such as different forms of calendar databases or different types of queues?

If yes, then the server must classify objects into types. For each type, the server generates a non-nil UUID for the type UUID, sets the type UUID for every object of the type, and specifies that type as the manager type when registering the interface. When making a remote procedure call to the interface, a client must supply an object UUID to specify an RPC resource.

- Is control over specific resources an important factor for distinguishing among server instances on a host?

  If yes, then each server must generate an object UUID for each of its resources.

For some applications, such as those accessing a database that many people use, shared access to one or more objects may be essential. However, not all objects accommodate such shared access.

15.3.2.2 Using Objects and Groups Together

Servers can associate object UUIDs with a group. Each server exports one or more object UUIDs (without exporting any binding information) to the directory service entry of the group. This involves specifying the **NULL** interface identifier to the export operation along with the list of object UUIDs. The object UUIDs reside in the directory service entry of the group. If a server stops offering an advertised object, a server must unexport its object UUID from the group entry in order to keep its object list up to date.

Clients use objects in a group entry as follows:

1. The client inquires for an object UUID from the group entry by calling the **rpc_ns_entry_object_inq_{begin,next,done}()** routines. This routine selects one object UUID at random and returns it to the client.

2. The client imports binding information for the returned object UUID (and the interface of the called remote procedure), specifying the group for the start of the search.

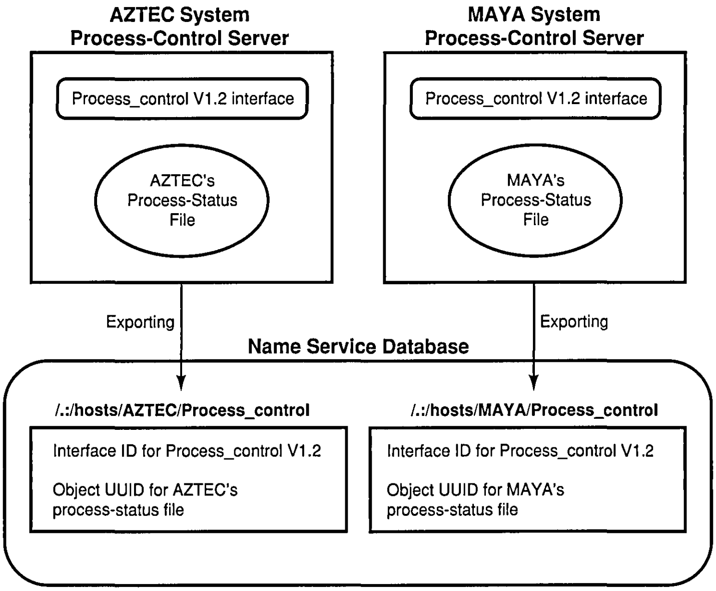
3. The import operation returns a binding handle that refers to the requested object UUID and binding information for a server that offers the corresponding object.

4. The client issues the remote procedure call using that binding handle.
5. The server looks up the type of the requested object.
6. The server assigns the remote procedure call to the manager that implements the called remote procedure for that type of object.

15.3.2.3 System-Specific Applications

For some applications, the clients need to import an RPC resource that belongs to a specific system, and the clients can specify a server entry name to learn about a server on that system. For example, a process server that allows clients to monitor and control processes on a remote machine is useful only to that machine. Figure 15-9 illustrates this type of system-specific interpretation of the resource model.

Figure 15–9. Resource Model: A System-Specific Application


AZTEC System
Process-Control Server
Process_control V1.2 interface
AZTEC's Process-Status File
MAYA System
Process-Control Server
Process_control V1.2 interface
MAYA's Process-Status File
Exporting
Exporting
Name Service Database
/.:/hosts/AZTEC/Process_control
Interface ID for Process_control V1.2
Object UUID for AZTEC's process-status file
/.:/hosts/MAYA/Process_control
Interface ID for Process_control V1.2
Object UUID for MAYA's process-status file

Search Requirements

Target Interface: Process_control V1.2

Target Object: Process-status file of MAYA system

Starting Entry: **/.:/hosts/MAYA/Process_control**

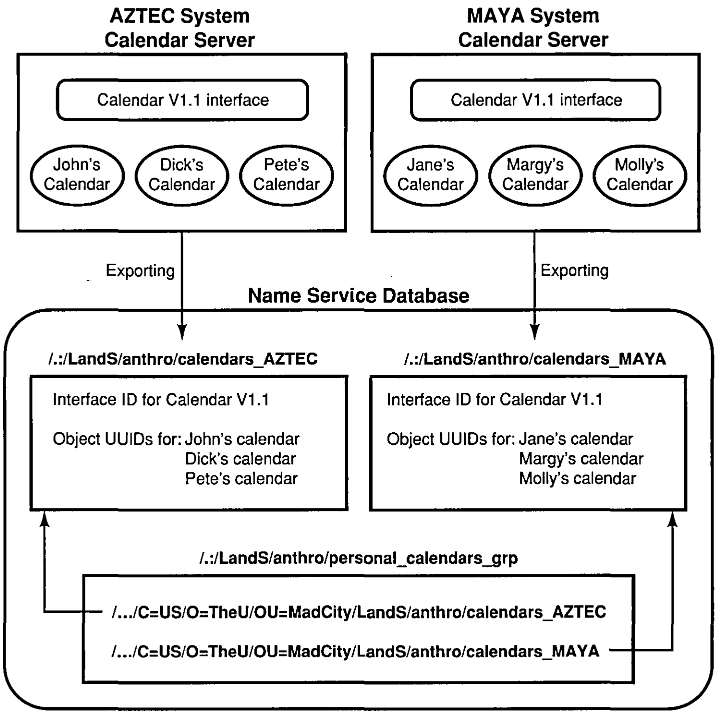
Maximum Number of Traversed Entries: 1

Because clients usually find a system-specific server by specifying its server entry to the import operation, groups are usually not part of the NSI search path for system-specific applications. However, groups are a management tool for such applications. A group containing the names of the server entries of all the current servers can act as an accounting database. Also, a group for the servers on each set of related systems, such as the members of a local area network or an administrative grouping, permits a client to sequentially use the application on every system in the set. An application with system-specific servers should *not* use profiles.

15.3.2.4 Exporting Multiple Object UUIDs to a Single Server Entry

Often a single server offers more than one resource, or it offers several types of resources. In cases where a server instance has a large number of object UUIDs, the application should usually place multiple object UUIDs into a single server entry. Typically, an application places all its object UUIDs into one server entry; however, it may need to segregate them into several server entries according to factors such as object type, location, or who uses the different types of objects. Note that when you are subsetting resources, try to assign each resource to a single set so that its object UUID is exported to only one server entry. Figure 15-10 illustrates a single server entry implementation for each server for the resource model.

Figure 15–10. Resource Model: A Single Server Entry for Each Server


AZTEC System
Calendar Server
Calendar V1.1 interface
John's Calendar
Dick's Calendar
Pete's Calendar
MAYA System
Calendar Server
Calendar V1.1 interface
Jane's Calendar
Margy's Calendar
Molly's Calendar
Exporting
Exporting
Name Service Database
/.:/LandS/anthro/calendars_AZTEC
Interface ID for Calendar V1.1
Object UUIDs for: John's calendar
Dick's calendar
Pete's calendar
/.:/LandS/anthro/calendars_MAYA
Interface ID for Calendar V1.1
Object UUIDs for: Jane's calendar
Margy's calendar
Molly's calendar
/.:/LandS/anthro/personal_calendars_grp
/.../C=US/O=TheU/OU=MadCity/LandS/anthro/calendars_AZTEC
/.../C=US/O=TheU/OU=MadCity/LandS/anthro/calendars_MAYA

Search Requirements

Target Interface: Calendar V1.1

Target Object: A specific personal calendar

Starting Entry: **/.:/LandS/anthro/personal_calendars_grp**

Maximum Number of Traversed Entries: 3

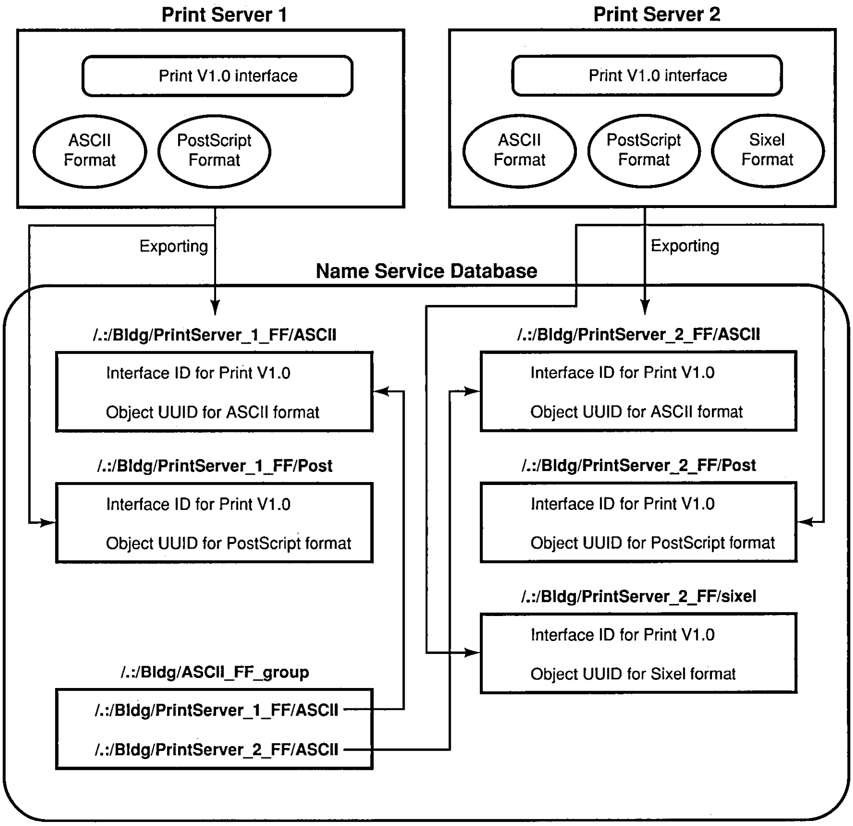
15.3.2.5 Exporting Every Object UUID to a Separate Server Entry

For some applications, exporting each object UUID to a separate server entry is a practical strategy. To avoid excessive demands on directory service resources, however, this strategy requires that the set of objects remain small. Applications with many RPC resources should usually have each server create a single server entry for itself and export the object UUIDs of the resources it offers to that server entry. For example, an application that accesses a different personal calendar for every member of an organization needs to avoid using a separate server entry for each calendar.

For some applications, however, you can use a separate server entry for each object UUID; for example, a print server application that supports a small number of file formats. Each server can create a separate server entry for each supported file format and export its object UUID to that server entry. The server entries for a file format are members of a distinct group.

To import binding information for a server that supports a required file format, a client specifies the nil UUID as the object UUID and the group for that format as the starting entry. The import operation selects a group member at random and goes to the corresponding server entry. Along with binding information, the operation returns the server's object UUID for the requested file format from the server entry. When the client issues a remote procedure call to the server, the imported object UUID correctly identifies the file format the client needs. Figure 15-11 illustrates this use of object UUIDs.

Figure 15–11. Resource Model: A Separate Server Entry for Each Object


Print Server 1
Print V1.0 interface
ASCII Format
PostScript Format
Print Server 2
Print V1.0 interface
ASCII Format
PostScript Format
Sixel Format
Exporting
Exporting
Name Service Database
/.:/Bldg/PrintServer_1_FF/ASCII
Interface ID for Print V1.0
Object UUID for ASCII format
/.:/Bldg/PrintServer_1_FF/Post
Interface ID for Print V1.0
Object UUID for PostScript format
/.:/Bldg/PrintServer_2_FF/ASCII
Interface ID for Print V1.0
Object UUID for ASCII format
/.:/Bldg/PrintServer_2_FF/Post
Interface ID for Print V1.0
Object UUID for PostScript format
/.:/Bldg/PrintServer_2_FF/sixel
Interface ID for Print V1.0
Object UUID for Sixel format
/.:/Bldg/ASCII_FF_group
/.:/Bldg/PrintServer_1_FF/ASCII
/.:/Bldg/PrintServer_2_FF/ASCII

Search Requirements

Target Interface: Print V1.0

Target Object: ASCII file format (client specifies nil object UUID)

Starting Entry: **/.:/Bldg/ASCII_FF_group**

Maximum Number of Traversed Entries: 2

Applications that use a separate entry for each object UUID need to use groups cautiously. Keeping groups small when clients are requesting a specific object is essential because an NSI search looks up the group members in random order. Therefore, the members of a group form a localized flat NSI search path rather than the hierarchical path. Flat search paths are inefficient because the average search will look at half the members. Small groups are not a problem. For example, if a group contains only 4 members, each of whom refers to a server entry that advertises a distinct set of RPC resources, the average number of server entries accessed in each search is 2 entries and the maximum is only 4. The larger the group, however, the more inefficient the resulting search path. For example, for a group containing 12 members, each of whom refers to a server entry that advertises a distinct set of object UUIDs, the average search accesses 6 entries and some searches access all 12 server entries.

Chapter 16

Guidelines for Error Handling

During a remote procedure call, server and communications errors may occur. These errors can be handled using any or all of the following methods:

- Writing exception handler code to recover from the error or to exit the application
- Using the **fault_status** attribute in the ACF to report an RPC server failure
- Using the **comm_status** attribute in the ACF to report a communications failure

Use of exceptions, where the procedure exits the program due to an error, tends to improve code quality. It does this by making errors obvious because the program exits at that point, and by lessening the amount of code needed to detect error conditions and handle them. When you use the **fault_status** attribute, an exception that occurs on the server is not reported to the client as an exception. The variable to which the **comm_status** attribute is attached contains error codes that report errors that would not have occurred if the application were not distributed over a communications network. The **comm_status** attribute provides a method of handling RPC errors without using an exception handler.

16.1 Exceptions

Exceptions report errors, either RPC errors or errors in application code, when **comm_status** or **fault_status** or both are not present in the ACF. Exceptions have the following characteristics:

- You do not have to adjust procedure declarations between local and distributed code.
- You can distribute existing interfaces without changing code.
- You do not have to check for failures. This results in more robust code because errors are reported even if they are not checked.
- Your code is more efficient when there is no recovery coded for failures.
- You can use a simpler coding style.
- Exceptions work well for coarse-grained exception handling.
- If your application does not contain any exception handlers and the application thread gets an error, the application thread is terminated and a system-dependent error message from the threads package is printed.

Note: RPC exceptions are equivalent to RPC status codes. To identify the status code that corresponds to a given exception, replace the **_x_** string of the exception with the string **_s_**. For example, the exception **rpc_x_comm_failure** is equivalent to the status code **rpc_s_comm_failure**. The RPC exceptions are defined in the **dce/rpcexc.h** header file.

The **rpc_status_codes(7rpc)** reference page documents the RPC status codes in alphabetical order. The documentation for each status code includes the message text, the explanation, and the suggested user action. This reference page is in the *OSF DCE Application Development Reference*.

For more information about using exceptions to handle errors, see Part 2 of this guide.

16.2 The fault_status Attribute

The **fault_status** attribute requests that errors occurring on the server due to incorrectly specified parameter values, resource constraints, or coding errors be reported by an additional status parameter instead of by an exception. The **fault_status** attribute has the following characteristics:

- Occurs where you do not want transparent local/remote behavior
- Occurs where you expect that you may be passing incorrect data to the server or the server is not coded robustly, or both
- Works well for fine-grained error handling
- Requires that you adjust procedure declarations between local and distributed code
- Controls the reporting only of errors that come from the server and that are reported via a fault packet

For more information on the **fault_status** attribute, see Chapter 18.

16.3 The comm_status Attribute

The **comm_status** attribute requests that RPC communications failures be reported via an additional status parameter instead of by an exception. The **comm_status** attribute has the following characteristics:

- Occurs where you expect communications to fail routinely; for instance, no server is available, the server has no resources, and so on
- Works well for fine-grained error handling; for example, trying a procedure many times until it succeeds
- Requires that you adjust procedure declarations between local and distributed code to add the new status parameter
- Controls the reporting of errors only from RPC runtime error status codes

For more information on the **comm_status** attribute, see Chapter 18.

16.4 Determining Which Method to Use for Handling Exceptions

Some conditions are better for using the **comm_status** or **fault_status** attribute on an operation, rather than the default approach of handling exceptions.

The **comm_status** attribute is useful only if the call to the operation has a specific recovery action to perform for one or more communications failures; for example, **rpc_s_comm_failure** or **rpc_s_no_more_bindings**. The **comm_status** attribute is recommended only when the application knows that it is calling a remote operation.

If you expect communications to fail often because the server does not have enough resources to execute the call, you can use this attribute to allow the call to be retried several times.

If you are using an implict or explicit binding, you can use the **comm_status** attribute if you want to try another server because the operation cannot be performed on the one you are currently using.

You can also use an exception handler for each of the two previous instances. In general, the advantange of using **comm_status** if the recovery is local to the routine is that the overhead is less. The disadvantage of **comm_status** is that the operation is different between the local and distributed case. Also, if all of the recovery cannot be done locally (where the call is made), there must be a way to pass the status to outer layers of code to process it.

The **fault_status** attribute is useful only if the call to the operation has a specific recovery action to perform for one or more server faults; for example, **rpc_s_invalid_tag**, **rpc_s_fault_pipe_comm_error**, **rpc_s_fault_int_overflow**, or **rpc_s_fault_remote_no_memory**. Use **fault_status** only when the application calls a remote operation and wants different behavior than if it calls the same operation locally.

If you are requesting an operation on a large data set you can use this attribute to trap **rpc_s_fault_remote_no_memory** and retry the operation to a different server, or you may break your data set into two smaller sections.

You can also handle the previous case with exception handlers. The advantange of using **fault_status** if the recovery is local is that the overhead is less. The disadvantage of **fault_status** is that the operation is different between the local and distributed case. Also, if all of the recovery cannot be done locally, there must be a way to pass the status to outer layers of code to process it.

16.5 Examples of Error Handling

The following subsections present two examples of error handling. The first example assumes that the **comm_status** attribute is in use in the ACF. The second example assumes that the **comm_status** attribute is not in use.

16.5.1 The Matrix Math Server Example

Assume that you have an existing local interface that provides matrix math operations. Since it is local, errors such as floating-point overflow or divide by zero are returned to the caller of a matrix operation as exceptions. It is likely that these exceptions are caused by providing data to the operation in an improper form.

In this case, the exceptions are part of the interface, so **fault_status** changes the way the application calls the matrix interface and probably is undesirable. Depending on the environment, finding a server may not be difficult (if the network is relatively stable and has enough resources), and adding **comm_status** serves only to introduce differences between the local and distributed applications.

If a decision as to what action to take is based upon a communications failure, then you may try to add the conditional code **comm_status** requires. Otherwise, using **auto_handle** allows an attempt on each available server. If no server is available, the application terminates because it cannot proceed. You can add an exception handler to the main program to report the error in a user-friendly manner.

16.5.2 The Stock Quote Application Example

Assume that you have a windows application that reads from stock quote servers and displays graphs of the data. Since you do not expect to get server failures because it is a commercial-quality server, you are not interested in writing code to handle values returned from **fault_status**. If high availability and robustness is important, you may have a list of recovery plans to make sure a stock analyst can get the necessary information as quickly as possible. For example:

```
retry_count = 10;
repeat
    query_stock_quote(h, ...,&st);
    switch (st)          /* st parameter can be used because */
    {                    /* [comm_status] is in the ACF */
        case rpc_s_ok:
            break;
        case rpc_s_comm_failure:
            retry_count -= 1;
            break;
        case rpc_s_network_unreachable:
            h = some_other_handle;
            break;
        case
            .
            .
            .
        default:
            retry_count -= 1;
    }
until ((st == rpc_s_ok) || (retry_count <= 0))
```

If this is not a critical application, you may only report that the server is currently unavailable. Depending upon the design of the application, there may be several places to put the exception handler to report the failure but continue processing. For example:

```
TRY
    update_a_quote(...);
CATCH_ALL
    display_message("Stock quote not currently available");
ENDTRY
```

This example assumes that **update_a_quote**() eventually calls the remote operation **query_stock_quote**() and that this call may raise an exception that is detected and reported here.

The advantage of using exceptions in this case is that all of the work done in **update_a_quote**() has the same error recovery and it does not need to be repeated at every call to a remote operation. Another advantage is that if one of the remote operations does have a recovery for one exception, it can handle that one exception and allow the rest to propagate to the more general handler in an outer layer of the code.

Part 3B

Language Syntax and Usage

Part 3B contains reference information for the Interface Definition Language and the Attribute Configuration Language.

Chapter 17

Interface Definition Language

This chapter describes how to construct an Interface Definition Language (IDL) file. First, it describes the IDL syntax notation conventions and lexical elements. It then describes the interface definition structure and the individual language elements supported by the IDL compiler.

17.1 The Interface Definition Language File

The Interface Definition Language (IDL) file defines all aspects of an interface that affect data passed over the network between a caller (client) and a callee (server). An interface definition file has the suffix **.idl**. In order for a caller and callee to interoperate, they both need to incorporate the same interface definition.

17.2 Syntax Notation Conventions

In addition to the documentation conventions described in the Preface of this guide, the IDL syntax uses the special notation described in the following subsections.

17.2.1 Typography

IDL documentation uses the following typefaces:

Bold
: **Bold** typeface indicates a literal item. Keywords and literal punctuation are represented in bold typeface. Identifiers used in a particular example are represented in bold typeface when mentioned in the text.

Italic
: *Italic* typeface indicates a symbolic item for which you need to substitute a particular value. In IDL syntax descriptions, all identifiers that are not keywords are represented in italic typeface.

`Constant width`
: `Constant width` typeface is used for source code examples (in IDL or in C) that are displayed separately from regular text.

17.2.2 Special Symbols

IDL documentation uses the following symbolic notations:

[item]
: *Italic* brackets surrounding an item indicate that the item is optional.

[*item*]
: Brackets shown in regular typeface are a required part of the syntax.

item ...
: Ellipsis points following an item indicate that the item may occur one or more times.

item , ... If an item is followed by a literal punctuation character and then by ellipsis points, the item may occur either once without the punctuation character, or more than once with the punctuation character separating each instance.

... If ellipsis points are shown on a line by themselves, the item or set of items in the preceding line may occur any number of additional times.

item | *item* If several items are shown separated by vertical bars, exactly one of those items must occur.

17.3 IDL Lexical Elements

The following subsections describe these IDL lexical elements:

- Identifiers
- Keywords
- Punctuation characters
- White space
- Case sensitivity

17.3.1 Identifiers

The character set for IDL identifiers comprises the alphabetic characters A to Z and a to z, the digits 0 to 9, and the _ (underscore) character. An identifier must start with an alphabetic character.

No IDL identifier can exceed 31 characters. In some cases an identifier has a shorter maximum length because the IDL compiler uses the identifier as a base from which to construct other identifiers; we identify such cases as they occur.

17.3.2 Keywords

IDL reserves some identifiers as keywords. In the text of this chapter, keywords are represented in **bold** typeface, and identifiers chosen by application developers are represented in *italic* typeface.

17.3.3 Punctuation Characters

IDL uses the following graphic characters:

" ' ( ) * , . / : ; | = [\ ] { }

The { (left brace) and } (right brace) characters are national replacement set characters that may not be available on all keyboards. Wherever IDL specifies a left brace, the **??<** trigraph may be substituted. Wherever IDL specifies a right brace, the **??>** trigraph may be substituted.

Use of these trigraph sequences adds the following punctuation characters to the set in the preceding list:

< > ?

17.3.4 White Space

White space is used to delimit other constructs. IDL defines the following white space constructs:

- A space
- A carriage return
- A horizontal tab
- A form feed at the beginning of a line
- A comment
- A sequence of one or more of the preceding white space constructs

A keyword, identifier, or number not preceded by a punctuation character must be preceded by white space. A keyword, identifier, or number not followed by a punctuation character must be followed by white space. Unless we note otherwise, any punctuation character may be preceded and/or followed by white space.

When enclosed in “ ” (double quotes) or ’ ’ (single quotes), white space constructs are treated literally. Otherwise, they serve only to separate other lexical elements and are ignored.

The character sequence /* (slash and asterisk) begins a comment, and the character sequence */ (asterisk and slash) ends a comment. For example:

```
/* all natural */
import "potato.idl";  /* no preservatives */
```

Comments do not nest.

17.3.5 Case Sensitivity

The IDL compiler does not force the case of identifiers in the generated code.

The only case sensitivity issue that you have to be aware of is the implications involved in calling generated stubs from languages other than C.

17.4 IDL Versus C

IDL resembles a subset of ANSI C. The major difference between IDL and C is that there are no executable statements in IDL.

17.4.1 Declarations

An interface definition specifies how operations are called, not how they are implemented. IDL is therefore a purely declarative language.

17.4.2 Data Types

To support applications written in languages other than C, IDL defines some data types that do not exist in C and extends some data types that do exist in C. For example, IDL defines a Boolean data type.

Some C data types are supported by IDL only with modifications or restrictions. For example, unions must be discriminated, and all arrays must be accompanied by bounds information.

17.4.3 Attributes

The stub modules that are generated from an interface definition require more information about the interface than can be expressed in C. For example, stubs must know whether an operation parameter is an input or an output.

The additional information required to define a network interface is specified via IDL attributes. IDL attributes can apply to types, to structure members, to operations, to operation parameters, or to the interface as a whole. Some attributes are legal in only one of the preceding contexts; others are legal in more than one context. An attribute is always represented in [ ] (brackets) before the item to which it applies. For example, in an operation declaration, inputs of the operation are preceded by the **in** attribute and outputs are preceded by the **out** attribute:

```
void arith_add (
     [in] long a,
     [in] long b,
     [out] long *c,
     );
```

17.5 Interface Definition Structure

An interface definition has the following structure:

[*interface_attribute*, ...] **interface** *interface_name*
{
declarations
}

The portion of an interface definition that precedes the { (left brace) is the interface header. The remainder of the definition is the interface body. Interface header syntax and interface body syntax are described separately in the following two subsections.

17.5.1 Interface Definition Header

The interface header comprises a list of interface attributes enclosed in [ ] (brackets), the keyword **interface**, and the interface name:

[*interface_attribute*, ...] **interface** *interface_name*

Interface names, together with major and minor version numbers, are used by the IDL compiler to construct identifiers for interface specifiers, entry point vectors, and entry point vector types. If the major and minor version numbers are single digits, the interface name can be up to 17 characters long.

17.5.2 Interface Definition Body

The *declarations* in an interface definition body are one or more of the following:

import_declaration
constant_declaration
type_declaration
operation_declaration

A ; (semicolon) terminates each declaration, and { } (braces) enclose the entire body.

Import declarations must precede other declarations in the interface body. Import declarations specify the names of other IDL interfaces that define types and constants used by the importing interface.

Constant, type, and operation declarations specify the constants, types, and operations that the interface exports. These declarations can be coded in any order, provided any constant or type is defined before it is used.

17.6 Overview of IDL Attributes

Table 17-1 lists the attributes allowed in interface definition files and specifies the declarations in which they can occur.

Table 17–1. IDL Attributes

Attribute	**Where Used**
uuid **version** **endpoint** **pointer_default** **local**	Interface definition headers
broadcast **maybe** **idempotent**	Operations
in **out**	Parameters
ignore	Structures

Attribute	Where Used
max_is **size_is** **first_is** **last_is** **length_is**	Arrays
string	Arrays
ptr **ref**	Pointers
handle	Customized handles
context_handle	Context handles
transmit_as	Type declarations
v1_array **v1_enum** **v1_string** **v1_struct**	Migration

17.7 Interface Definition Header Attributes

The following subsections describe in detail the usage and semantics of the IDL attributes that can be used in interface definition headers. The attributes provided for interface definition headers are as follows:

- **uuid**
- **version**
- **endpoint**
- **pointer_default**
- **local**

17.7.1 The uuid Attribute

The **uuid** attribute specifies the Universal Unique Identifier (UUID) that is assigned to an interface. The **uuid** attribute takes the form:

uuid (*uuid_string*)

A *uuid_string* is the string representation of a UUID. This string is typically generated as part of a skeletal interface definition by the utility **uuidgen**. A *uuid_string* contains one group of 8 hexadecimal digits, three groups of 4 hexadecimal digits, and one group of 12 hexadecimal digits, with hyphens separating the groups, as in the following example:

```
01234567-89ab-cdef-0123-456789abcdef
```

A new UUID should be generated for any new interface. If several versions of one interface exist, all versions should have the same interface UUID but different version numbers. A client and a server cannot communicate unless the interface imported by the client and the interface exported by the server have the same UUID. The client and server stubs in an application must be generated from the same interface definition or from interface definitions with identical **uuid** attributes.

Any remote interface must have the **uuid** attribute. An interface must have either the **uuid** attribute or the **local** attribute, but cannot have both.

The following example illustrates use of the **uuid** attribute:

```
uuid(4ca7b4dc-d000-0d00-0218-cb0123ed9876)
```

17.7.2 The version Attribute

The **version** attribute specifies a particular version of a remote interface. The **version** attribute takes the form:

version (*major [***.** *minor]*)

A version number can be either a pair of integers (the major and minor version numbers) or a single integer (the major version number). If both major and minor version numbers are supplied, the integers should be separated by a period without white space. If no minor version number is supplied, 0 (zero) is assumed.

The following examples illustrate use of the **version** attribute:

```
version (1.1)   /* major and minor version numbers */

version (3)     /* major version number only */
```

The **version** attribute can be omitted altogether, in which case the interface is assigned 0.0 as the default version number.

A client and a server can communicate only if the following requirements are met:

- The interface imported by the client and the interface exported by the server have the same major version number.
- The interface imported by the client has a minor version number less than or equal to that of the interface exported by the server.

You must increase either the minor version number or the major version number when you make any compatible change to an interface definition. You cannot decrease the minor version number unless you simultaneously increase the major version number.

You must increase the major version number when you make any incompatible change to an interface definition. (See the definition of compatible changes that follows.) You cannot decrease the major version number.

The following are considered compatible changes to an interface definition:

- Adding operations to the interface, if and only if the new operations are declared after all existing operation declarations in the interface definition.
- Adding type and constant declarations, if the new types and constants are used only by operations added at the same time or later. Existing operation declarations cannot have their signatures modified.

The *major* and *minor* integers in the **version** attribute can range from 0 to 65,535, inclusive. However, these typically are small integers and are increased in increments of one.

17.7.3 The endpoint Attribute

The **endpoint** attribute specifies the well-known endpoint or endpoints on which servers that export the interface will listen. The **endpoint** attribute takes the form:

endpoint (*endpoint_spec*, ...)

Each *endpoint_spec* is a string in the following form:

" *family* : [*endpoint*] "

The *family* identifies a protocol family. The following are accepted values for *family*:

- **ncacn_ip_tcp**: NCA Connection over Internet Protocol: Transmission Control Protocol (TCP/IP)
- **ncadg_ip_udp**: NCA Datagram over Internet Protocol: User Datagram Protocol (UDP/IP)

The *endpoint* identifies a well-known endpoint for the specified *family*. The values accepted for *endpoint* depend on the *family* but typically are integers within a limited range. IDL does not define valid *endpoint* values.

Well-known endpoint values are typically assigned by the central authority that ‘‘owns’’ a protocol. For example, the Internet Assigned Numbers Authority assigns well-known endpoint values for the IP protocol family.

At compile time, the IDL compiler checks each *endpoint_spec* only for gross syntax. At runtime, stubs pass the *family* and *endpoint* strings to the RPC runtime, which validates and interprets them.

Most applications should not use well-known endpoints and should instead use dynamically assigned opaque endpoints. Most interfaces designed for use by applications should therefore not have the **endpoint** attribute.

The following example illustrates use of the **endpoint** attribute:

```
endpoint ("ncacn_ip_tcp:[1025]", "ncadg_ip_udp:[6677]")
```

17.7.4 The pointer_default Attribute

IDL supports two kinds of pointer semantics. The **pointer_default** attribute specifies the default semantics for pointers that are declared in the interface definition. The **pointer_default** attribute takes the form:

pointer_default (*pointer_attribute*)

Possible values for *pointer_attribute* are **ref** and **ptr**.

The default semantics established by the **pointer_default** attribute apply to the following usages of pointers:

- A pointer that occurs in the declaration of a member of a structure or a union
- A pointer that is at other than the top level of an operation parameter declared with more than one pointer operator

Note that the **pointer_default** attribute does not apply to a pointer that is the return value of an operation because this is always a full pointer.

The default semantics can be overridden by pointer attributes in the declaration of a particular pointer. If an interface definition does not specify **pointer_default** and contains a declaration that requires default pointer semantics, the IDL compiler will issue an error.

For additional information on pointer semantics, refer to Section 17.14.7.1.

17.7.5 The local Attribute

The **local** attribute indicates that an interface definition does not declare any remote operations and that the IDL compiler should therefore generate only header files, not stub files. The **local** attribute takes the form:

local

An interface containing operation definitions must have either the **local** attribute or the **uuid** attribute. No interface can have both.

17.7.6 Rules for Using Interface Definition Header Attributes

An interface cannot have both the **local** attribute and the **uuid** attribute. In an interface definition that contains any operation declarations, either **local** or **uuid** must be specified. In an interface definition that contains no operation declarations, both **local** and **uuid** can be omitted.

The **local**, **uuid**, and **version** attributes cannot be coded more than once. If the **endpoint** or the **pointer_default** attribute is coded more than once, the IDL compiler issues a warning, and where conflicts exist, the IDL compiler accepts the last value specified.

17.7.7 Examples of Interface Definition Header Attributes

The following example uses the **uuid** and **version** attributes:

```
[uuid(df961f80-2d24-11c9-be74-08002b0ecef1), version(1.1)]
interface my_interface_name
```

The following example uses the **uuid**, **endpoint**, and **version** attributes:

```
[uuid(0bb1a080-2d25-11c9-8d6e-08002b0ecef1),
endpoint("ncacn_ip_tcp:[1025]", "ncacn_ip_tcp:[6677]"),
version(3.2)]
interface my_interface_name
```

17.8 Import Declarations

The IDL *import_declaration* specifies interface definition files that declare types and constants used by the importing interface. It takes the following form:

import *file*,... ;

The *file* is the pathname, enclosed in " " (double quotes), of the interface definition you are importing. This pathname can be relative; the **-I** option of the IDL compiler allows you to specify a parent directory from which to resolve import pathnames.

The effect of an import declaration is as if all constant, type, and import declarations from the imported file occurred in the importing file at the point where the import declaration occurs. Operation declarations are not imported.

For example, suppose that the interface definition **aioli.idl** contains a declaration to import the definitions for the **garlic** and **oil** interfaces:

```
import "garlic.idl", "oil.idl";
```

The IDL compiler will generate a C header file named **aioli.h** that contains the following **#include** directives:

```
#include "garlic.h"
#include "oil.h"
```

The stub files that the compiler generates will not contain code for any **garlic** and **oil** operations.

Importing an interface many times has the same effect as importing it once.

17.9 Constant Declarations

The IDL *constant_declaration* can take the following forms:

const *integer_type_spec identifier* = *integer* | *value* | *integer_const_expression*;
const boolean *identifier* = **TRUE** | **FALSE** | *value*;
const char *identifier* = *character* | *value*;
const char* *identifier* = *string* | *value*;
const void* *identifier* = **NULL** | *value*;

The *integer_type_spec* is the data type of the integer constant you are declaring. The *identifier* is the name of the constant. The *integer*, *integer_const_expression*, *character*, *string*, or *value* specifies the value to be assigned to the constant. A *value* can be any previously defined constant.

IDL provides only integer, Boolean, character, string, and null pointer constants.

Following are examples of constant declarations:

```
const short TEN = 10;
const boolean FAUX = FALSE;
const char* DSCH = "Dmitri Shostakovich";
```

17.9.1 Integer Constants

An *integer_type_spec* is a *type_specifier* for an integer, except that the *int_size* for an integer constant cannot be **hyper**.

An *integer* is the decimal representation of an integer. IDL also supports the C notation for hexadecimal, octal, and long integer constants.

You can specify any previously defined integer constant as the *value* of an integer constant.

You can specify any arithmetic expression as the *integer_const_expression* that evaluates to an integer constant.

17.9.2 Boolean Constants

A Boolean constant can take one of two values: TRUE or FALSE.

You can specify any previously defined Boolean constant as the *value* of a Boolean constant.

17.9.3 Character Constants

A *character* is an ASCII character enclosed in ' ' (single quotes). A white space character is interpreted literally. The \ (backslash) character introduces an escape sequence, as defined in the ANSI C standard. The ' (single quote) character can be coded as the *character* only if it is escaped by a backslash.

You can specify any previously defined character constant as the *value* of a character constant.

17.9.4 String Constants

A *string* is a sequence of ASCII characters enclosed in " " (double quotes). White space characters are interpreted literally. The \ (backslash) character introduces an escape sequence, as defined in the ANSI C standard. The " (double quote) character can be coded in a *string* only if it is escaped by a backslash.

You can specify any previously defined string constant as the *value* of a string constant.

17.9.5 NULL Constants

A **void*** constant can take only one literal value: **NULL.**

You can specify any previously defined **void*** constant as the *value* of a **void*** constant.

17.10 Type Declarations

The IDL *type_declaration* enables you to associate a name with a data type and to specify attributes of the data type. It takes the following form:

typedef *[[type_attribute*, ...*]] type_specifier type_declarator*, ... ;

A *type_attribute* specifies characteristics of the type being declared.

The *type_specifier* can specify a base type, a constructed type, a predefined type, or a named type.

Each *type_declarator* is a name for the type being defined. Note, though, that a *type_declarator* can also be preceded by an * (asterisk), followed by [ ] (brackets), and can have ( ) (parentheses) for proper grouping.

17.10.1 Type Attributes

A *type_attribute* can be any of the following:

- **handle**: The type being declared is a user-defined, customized-handle type.
- **context_handle**: The type being declared is a context-handle type.
- **transmit_as**: The type being declared is a ''presented type.'' When it is passed in remote procedure calls, it is converted to a specified ''transmitted type.''
- **ref**: The type being declared is a reference pointer.
- **ptr**: The type being declared is a full pointer.
- **string**: The array type being declared is a string type.
- **v1_struct**: This attribute specifies an alternate data alignment for the network representation of a structure type.
- **v1_array**: This attribute specifies alternate network representation for arrays.
- **v1_string**: This attribute specifies alternate network representation for strings.

- **v1_enum**: This attribute specifies alternate network representation for enumerations.

17.10.2 Base Type Specifiers

IDL base types include integers, floating-point numbers, characters, a **boolean** type, a **byte** type, a **void** type, and a primitive handle type.

Table 17-2 lists the IDL base data type specifiers. Where applicable, the table shows the size of the corresponding transmittable type and the type macro emitted by the IDL compiler for resulting declarations.

Table 17–2. Base Data Type Specifiers

	Specifier			Type Macro
(sign)	(size)	(type)	Size	Emitted by idl
	small	int	8 bits	idl_small_int
	short	int	16 bits	idl_short_int
	long	int	32 bits	idl_long_int
	hyper	int	64 bits	idl_hyper_int
unsigned	small	int	8 bits	idl_usmall_int
unsigned	short	int	16 bits	idl_ushort_int
unsigned	long	int	32 bits	idl_ulong_int
unsigned	hyper	int	64 bits	idl_uhyper_int
		float	32 bits	idl_short_float
		double	64 bits	idl_long_float
		char	8 bits	idl_char
		boolean	8 bits	idl_boolean
		byte	8 bits	idl_byte
		void	—	idl_void_p_t
		handle_t	—	—

The base types are described individually later in this chapter.

Note that you can use the **idl_** macros in the code you write for an application to ensure that your type declarations are consistent with those in the stubs, even when the application is ported to another platform. The **idl_** macros are especially useful when passing constant values to RPC calls. For maximum portability, all constants passed to RPC calls declared in your network interfaces should be cast to the appropriate type because the size of integer constants (like the size of the **int** data type) is ambiguous in the C language.

The **idl_** macros are defined in **dce/idlbase.h**, which is included by header files that the IDL compiler generates.

17.10.3 Constructed Type Specifiers

IDL constructed types include structures, unions, enumerations, pipes, arrays, and pointers. (In IDL, as in C, arrays and pointers are specified via declarator constructs rather than type specifiers.) Following are the keywords used to declare constructed type specifiers:

struct
union
enum
pipe

Constructed types are described in detail later in this chapter.

17.10.4 Predefined Type Specifiers

While IDL per se does not have any predefined types, DCE RPC IDL implicitly imports **nbase.idl**, which does predefine some types. Specifically, **nbase.idl** predefines an error status type, several international character data types, and many other types. Following are the keywords used to declare these ‘‘predefined’’ type specifiers:

error_status_t
ISO_LATIN_1
ISO_MULTI_LINGUAL
ISO_UCS

The error status type and international characters are described in detail later in this chapter.

17.10.5 Type Declarator

An IDL *type_declarator* can be either a simple declarator or a complex declarator.

A simple declarator is just an identifier.

A complex declarator is an identifier that specifies an array, a function pointer, or a pointer.

17.11 Operation Declarations

The IDL *operation_declaration* can take the following forms:

[[*operation_attribute*, ...]*]* *type_specifier operation_identifier*
(*parameter_declaration*, ...);
[[*operation_attribute*, ...]*]* *type_specifier operation_identifier*
(*[***void***]*);

Use the first form for an operation that has one or more parameters; use the second form for an operation that has no parameters.

An *operation_attribute* can take the following forms:

- **idempotent**: The operation is idempotent.
- **broadcast**: The operation is always to be broadcast.
- **maybe**: The caller of the operation does not require and will not receive any response.
- **ptr**: The operation returns a full pointer. This attribute must be supplied if the operation returns a pointer result and reference pointers are the default for the interface.

- **context_handle**: The operation returns a context handle.
- **string**: The operation returns a string.

The *type_specifier* in an operation declaration specifies the data type that the operation returns, if any. This type must be either a scalar type or a previously defined type. If the operation does not return a result, its *type_specifier* must be **void**.

For information on the semantics of pointers as operation return values, refer to the discussion of pointers in section 17.4.7.

The *operation_identifier* in an operation declaration is an identifier that names the operation.

Each *parameter_declaration* in an operation declaration declares a parameter of the operation. A *parameter_declaration* takes the following form:

[*parameter_attribute*, ...] *type_specifier parameter_declarator*

Parameter declarations and the parameter attributes are described separately in the following sections.

17.11.1 Operation Attributes

Operation attributes determine the semantics to be applied by the RPC client and server protocol when an operation is called.

17.11.2 Operation Attributes: idempotent, broadcast, and maybe

The **idempotent** attribute specifies that an operation is idempotent; that is, it can safely be executed more than once.

The **broadcast** attribute specifies that an operation is to be broadcast to all hosts on the local network each time the operation is called. The client receives output arguments from the first reply to return successfully, and all subsequent replies are discarded.

An operation with the **broadcast** attribute is implicitly idempotent.

The **maybe** attribute specifies that the caller of an operation does not expect any response. An operation with the **maybe** attribute cannot have any output parameters and cannot return anything. Delivery of the call is not guaranteed.

An operation with the **maybe** attribute is implicitly idempotent.

17.12 Parameter Declarations

A *parameter_declaration* is used in an operation declaration to declare a parameter of the operation. A *parameter_declaration* takes the form:

[*parameter_attribute*, ...] *type_specifier parameter_declarator*

If an interface does not use implicit handles or use interface-based binding, the first parameter must be an explicit handle that gives the object UUID and location. The handle parameter can be of a primitive handle type, **handle_t**, or a nonprimitive user-defined handle type.

A *parameter_attribute* can be any of the following:

- *array_attribute*: One of several attributes that specifies the characteristics of arrays.
- **in**: The parameter is an input attribute.
- **out**: The parameter is an output attribute.
- **ref**: The parameter is a reference pointer: it cannot be **NULL** and cannot be an alias.
- **ptr**: The parameter is a full pointer; it can be **NULL** and can be an alias.
- **string**: The parameter is a string.
- **context_handle**: The parameter is a context handle.
- **v1_array**: This attribute specifies an alternate wire representation for arrays.

- **v1_string**: This attribute specifies an alternate wire representation for strings.
- **v1_struct**: This attribute specifies an alternate wire representation for structure types.
- **v1_enum**: This attribute specifies an alternate wire representation for enumerations.

The directional attributes **in** and **out** specify the directions in which a parameter is to be passed. The **in** attribute specifies that the parameter is passed from the caller to the callee. The **out** attribute specifies that the parameter is passed from the callee to the caller.

An output parameter must be passed by reference and therefore must be declared with an explicit *. (Note that an array is implicitly passed by reference and so an output array does not require an an explicit *.) At least one directional attribute must be specified for each parameter of an operation.

An explicit handle parameter must have at least the **in** attribute.

The **ref** and **ptr** attributes are described later in Section 17.14.7. The **string** attribute is described in Section 17.14.6. The **context_handle** attribute is described in Section 17.14.9. The **v1_array** and **v1_string** attributes are described in Section 17.16.

The *type_specifier* in a parameter declaration specifies the data type of the parameter.

The *declarator* in a parameter declaration can be any simple or complex declarator.

A parameter with the **out** attribute must be either an array or an explicitly declared pointer. An explicitly declared pointer is declared by a *pointer_declarator*, rather than by a *simple_declarator* with a named pointer type as its *type_specifier*.

For information on the semantics of pointers as operation parameters, refer to the discussion of pointers in Section 17.14.7.

17.13 Basic Data Types

The following subsections describe the basic data types provided by IDL and the treatment of international characters within IDL. The basic data types are as follows:

- Integer types
- Floating-point types
- The **char** type
- The **boolean** type
- The **byte** type
- The **void** type
- The **handle_t** type
- The **error_status_t** type

Section 17.14 describes the constructed data types that are built on the basic data types.

17.13.1 Integer Types

IDL provides four sizes of signed and unsigned integer data types, specified as follows:

*int_size [***int***]*
unsigned *int_size [***int***]*
int_size **unsigned** *[***int***]*

The *int_size* can take the following values:

hyper
long
short
small

The **hyper** types are represented in 64 bits. A **long** is 32 bits. A **short** is 16 bits. A **small** is 8 bits.

The keyword **int** is optional and has no effect. The keyword **unsigned** denotes an unsigned integer type; it can occur either before or after the size keyword.

17.13.2 Floating-Point Types

IDL provides two sizes of floating-point data types, specified as follows:

float
double

A **float** is represented in 32 bits. A **double** is represented in 64 bits.

17.13.3 The char Type

The IDL character type is specified as follows:

*[***unsigned***]* **char**

A **char** is unsigned and is represented in 8 bits.

The keyword **unsigned** is optional and has no effect. IDL does not support a signed character type. IDL provides the **small** data type for representing signed 8-bit integers.

17.13.4 The boolean Type

The IDL **boolean** type is specified as follows:

boolean

A **boolean** is represented in 8 bits. A **boolean** is a logical quantity that assumes one of two values: TRUE or FALSE. Zero is FALSE and any nonzero value is TRUE.

17.13.5 The byte Type

The IDL **byte** type is specified as follows:

byte

A **byte** is represented in 8 bits. The data representation format of **byte** data is guaranteed not to change when the data is transmitted by the RPC mechanism.

The IDL integer, character, and floating-point types (and hence any types constructed from these) are all subject to format conversion when they are transmitted between hosts that use different data representation formats. You can protect data of any type from format conversion by transmitting that type as an array of **byte**.

17.13.6 The void Type

The IDL **void** type is specified as follows:

void

The **void** type may be used to do the following:

- Specify the type of an operation that does not return a value
- Specify the type of a context handle parameter, which must be **void***
- Specify the type of a **NULL** pointer constant, which must be **void***

17.13.7 The handle_t Type

The IDL primitive handle type is specified as follows:

handle_t

A **handle_t** is a primitive handle type that is opaque to application programs but meaningful to the RPC runtime library. Section 17.14.8 discusses primitive and nonprimitive handle types.

17.13.8 The error_status_t Type

IDL provides the following predefined data type to hold RPC communications status information:

error_status_t

The values that can be contained in the **error_status_t** data type are compatib!e with the **unsigned long** and **unsigned32** IDL data types. These data types are used for status values in the DCE. The **error_status_t** data type contains an additional semantic to indicate that this particular **unsigned long** contains a DCE format error status value. This additional semantic enables the IDL compiler to perform any necessary translation when moving the status value between systems with differing hardware architectures and software operating systems. If you are using status codes that are not in the DCE error status format or if you do not require such conversion, use an **unsigned long** instead of **error_status_t**.

17.13.9 International Characters

The implicitly imported **nbase.idl** provides predefined data types to support present and emerging international standards for the representation of characters and strings:

ISO_LATIN_1
ISO_MULTI_LINGUAL
ISO_UCS

Data of type **char** is subject to ASCII-EBCDIC conversion when transmitted by the RPC mechanism. The predefined international character types are constructed from the base type **byte** and are thereby protected from data representation format conversion.

The **ISO_LATIN_1** type is represented in 8 bits and is predefined as follows:

```
typedef byte ISO_LATIN_1;
```

The **ISO_MULTI_LINGUAL** type is represented in 16 bits and is predefined as follows:

```
typedef struct {
     byte row, column;
     } ISO_MULTI_LINGUAL;
```

The **ISO_UCS** type is represented in 32 bits and is predefined as follows:

```
typedef struct {
     byte group, plane, row, column;
     } ISO_UCS;
```

17.14 Constructed Data Types

The following subsections describe the constructed data types that are provided by IDL. The constructed types are built on the basic data types, which are described in Section 17.13. The constructed data types are as follows:

- Structures
- Unions
- Enumerations
- Pipes
- Arrays
- Strings

In IDL, as in C, arrays and pointers are specified via declarator constructs. The other constructed types are specified via type specifiers.

17.14.1 Structures

The *type_specifier* for a structure type can take the following forms:

struct *[tag]*
{
struct_member;
...
}

struct *tag*

A *tag*, if supplied in a specifier of the first form, becomes a shorthand form for the set of member declarations that follows it. Such a *tag* can subsequently be used in a specifier of the second form.

A *struct_member* takes the following form:

[[struct_member_attribute, ...]] type_specifier declarator, ...;

A *struct_member_attribute* can be any of the following:

- *array_attribute*: One of several attributes that specify characteristics of arrays.
- **ignore**: An attribute indicating that the pointer member being declared is not to be transmitted in remote procedure calls.
- **ref**: An attribute indicating that the pointer member being declared is a reference pointer: it cannot be **NULL** and cannot be an alias.
- **ptr**: An attribute indicating that the pointer member being declared is a full pointer: it can be **NULL** and can be an alias.
- **string**: An attribute indicating that the array member being declared is a string.
- **v1_array**: An attribute specifying an alternate wire representation for arrays.
- **v1_string**: An attribute specifying an alternate wire representation for strings.
- **v1_struct**: An attribute specifying an alternate wire representation for structure types.
- **v1_enum**: An attribute specifying an alternate wire representation for enumerations.

A structure can contain a conformant array only as its last member. Such a structure can be contained by another structure only as its last member. This requirement iterates through any other embedding structures. A structure that contains a conformant array (a conformant structure) cannot be returned by an operation as its value and cannot be simply an **out** parameter.

A structure cannot contain a pipe.

Note that the **ignore** attribute can be applied only to a pointer that is a member of a structure. This attribute specifies that the pointer is not to be transmitted in remote procedure calls.

17.14.2 Unions

An IDL union must be discriminated. The *type_specifier* for an IDL union can take the following forms:

union *[tag]* **switch** (*disc_type_spec discriminator*) *[union_name]*
{
case
...
[default_case]
}

union *tag*

A *tag*, if supplied in a specifier of the first form, becomes a shorthand form for the **switch** construct, union name, and set of cases that follow it. Such a *tag* can subsequently be used in a specifier of the second form.

The *disc_type_spec* indicates the type of the *discriminator*, which can be an integer, a character, a Boolean, or an enumeration.

The *union_name* specifies a name to be used in C code generated by the IDL compiler. When the IDL compiler generates C code to represent an IDL union, it embeds the union and its discriminator in a C structure. The name of the IDL union becomes the name of the C structure. If you supply a *union_name* in your type declaration, the compiler assigns this name to the embedded C union; otherwise, the compiler assigns the generic name **tagged_union**.

A *case* contains one or more labels and may contain a member declaration:

case *constant*:
...
[union_member];

Each label in a *case* specifies a constant. The *constant* can take any of the forms accepted in an integer, character, or Boolean constant declaration, each of which is described earlier in this chapter.

A *default_case* can be coded anywhere in the list of cases:

default:
[union_member];

A *union_member* takes the following form:

[[union_member_attribute, ...]] type_specifier declarator;

A *union_member_attribute* can be any of the following:

- **ptr**: An attribute indicating that the pointer member being declared is a full pointer: it can be **NULL** and can be an alias.
- **string**: An attribute indicating that the character array member being declared is a string.
- **v1_array**: An attribute specifying alternate semantics for arrays.
- **v1_string**: An attribute specifying alternate semantics for strings.

In any union, the type of the discriminator and the type of all constants in all case labels must resolve to the same type. At the time the union is used, the value of the discriminator selects a member, as follows:

- If the value of the discriminator matches the constant in any label, the member associated with the label is selected.
- If there is no label whose constant matches the value of the discriminator and there is a default case, the default member is selected.
- If there is no label whose constant matches the value of the discriminator and there is no default case, no member is selected and the exception **rpc_x_invalid_tag** is raised.

Note that IDL prohibits duplicate constant label values.

A *union_member* can contain only one declarator. If no *union_member* is supplied, the member is **NULL**, and if that member is selected when the union is used, no data is passed. Note, however, that the tag is always passed.

A union cannot contain a pipe, a conformant array, a varying array, or any structure that contains a conformant or varying array. A union also cannot contain a **ref** pointer or any structure that contains a **ref** pointer.

17.14.3 Enumerations

An IDL enumeration provides names for integers. It is specified as follows:

enum {*identifier*, ...}

Each *identifier* in an enumeration is assigned an integer value. The first identifier is assigned 0 (zero), the second is assigned 1, and so on.

An enumeration can have up to 32,767 identifiers.

17.14.4 Pipes

IDL supports pipes as a mechanism for transferring large quantities of typed data. An IDL pipe is an open-ended sequence of elements of one type. It is specified as follows:

pipe *type_specifier*

The *type_specifier* specifies the type for the elements of the pipe. This type cannot be a pointer type and cannot be a type that contains a pointer.

A pipe type can be used to declare only the type of an operation parameter. IDL recognizes three kinds of pipes, based on the three operation parameters:

- An **in** pipe is for transferring data from a client to a server. It allows the callee (server) to ''pull'' an open-ended stream of typed data from the caller (client).
- An **out** pipe is for transferring data from a server to a client. It allows the callee (server) to ''push'' the stream of data to the caller (client).
- An **in,out** pipe provides for two-way data transfer between a client and server by combining the behavior of **in** and **out** pipes.

A pipe can be defined only through a **typedef** declararation. Anonymous pipe types are not supported.

At the user code to stub call interface (for both the caller and callee), a pipe appears as a simple callback mechanism. To the user code, the processing of a pipe parameter appears to be synchronous. The IDL implementation of pipes in the RPC stub and runtime allows the apparent callbacks to occur without requiring actual remote callbacks. As a result, pipes provide an efficient transfer mechanism for large amounts of data.

17.14.4.1 IDL Pipes Example

To illustrate the IDL implementation of pipes, consider the following IDL fragment:

```
typedef
   pipe base_t pipe_t;
```

When the code containing this fragment is compiled, the IDL compiler will generate the following declarations in the derived header file:

```
typedef
   struct pipe_t {
      /*
      ** pointer to routine callback to pull
      ** the next chunk from the pipe
      */
      void (*pull) (
         char *state, /* in: pipe's state pointer */
         element_t *buf, /* in: buffer in which to place a chunk */
         idl_ulong_int esize, /* in: buffer size (# of elements) */
         idl_ulong_int *ecount /* out: size of chunk (# of elements) */
      );
      /*
      ** pointer to routine callback to push
      ** the next chunk into the pipe
      */
      void (*push) (
         char *state, /* in: pipe's state pointer */
         element_t *buf, /* in: buffer from which to copy chunk */
         idl_ulong_int *ecount /* in: size of chunk (# of elements) */
      );
      /*
      ** pointer to allocate callback to get buffer --
      ** used only on caller side
```

```
        */
        void (*alloc) (
            char *state, /* in: pipe's state pointer */
            idl_ulong_int bsize, /* in: requested size (# of *bytes*) */
            element_t **buf, /* out: pointer to allocated buffer */
            idl_ulong_int *bcount /* out: size of buffer (# of *bytes*) */
        );
        /*
        ** pointer to arbitrary storage for use by
        ** push, pull, and alloc
        */
        char *state;
    } pipe_t;
```

The **pipe_t.alloc** routine allocates memory from which pipe data can be marshalled or into which pipe data can be marshalled. The *bsize* parameter on this routine indicates the preferred size of the buffer, in bytes. The *bcount* and *buf* parameters describe the actual memory that is allocated. If **pipe_t.alloc** allocates less memory than requested, the RPC runtime uses the smaller memory and makes more callbacks to the user. If the routine allocates more memory than requested, the excess memory is not used.

The **pipe_t.state** structure member is provided as a way to help coordinate the activities of the pull, push, and allocate routines. It is available to the implementor of the client and to the implementor of the server manager. It is described in more detail in the subsections that follow.

17.14.4.2 Processing of Pipes

Using the IDL pipes example described in the previous subsection, this subsection describes the client and server responsibilities for processing pipes. The client and server responsibilities are described separately for **in** pipes, **out** pipes, and **in,out** pipes.

The in Pipes: Client Side

For an **in** pipe, the client (caller) must do the following:

1. Allocate the **pipe_t** structure.
2. Initialize the **pipe_t.pull**, **pipe_t.alloc**, and **pipe_t.state** fields.
3. Include code where appropriate for checking the **pipe_t.state** field.
4. Pass the structure as the pipe parameter. The structure can be passed either by value or by reference, as indicated by the signature of the operation that contains the pipe parameter.

To summarize, the client application code must supply **pull** and **alloc** routines. These routines must work together to produce a sequence of pointers to chunks, of which only the last is empty. The client stub does not modify the pipe state information.

To transmit a large amount of data that is already in the proper form in memory (that is, the data is already an array of **base_t**), the client application code can have the **alloc** routine allocate a buffer that already has the information in it. In this case, the **pull** routine becomes a null routine.

The in Pipes: Server Side

The manager reads from the pipe by making calls of the form:

```
#define DESIRED_NUM_ELEMENTS ...
long count;
base_t buf [DESIRED_NUM_ELEMENTS];
        .
        .
        .
do {
        .
        .
        .
    (* (pipe->pull)) (
       pipe->state,
       buf,
       DESIRED_NUM_ELEMENTS,
       &count
    );
} while (count > 0);
```

Using the buffer supplied by the manager, the **pipe->pull** routine unmarshals an amount of data that is nonzero, but not more than the buffer can hold. There is no guarantee that the buffer will be filled. The actual amount of data in the buffer is indicated by the *count* parameter to the **pipe->pull** routine.

The **pipe->pull** routine signals the end of data in the pipe by returning a chunk whose length is 0 (zero). Any attempt to pull data from the pipe after the zero-length chunk has been encountered will cause an exception to be raised. The **in** pipes must be processed in the order in which they occur in the operation signature. Attempting to pull data from an **in** pipe before End-of-Data on any preceding **in** pipe has been encountered will result in an exception being raised. If the manager code attempts to write to an **out** pipe or return control to the server stub before End-of-Data has been encountered on the last **in** pipe, an exception will be raised. Note that there is no guarantee that chunks seen by the manager will match the chunks supplied by the client's **pull** routine.

The out Pipes: Client Side

For an **out** pipe, the client (caller) must do the following:

1. Allocate the **pipe_t** structure.
2. Initialize the **pipe_t.push** and **pipe_t.state** fields.
3. Pass the structure as the pipe parameter, either by value or by reference.

The client stub unmarshals chunks of the pipe into a buffer and calls back to the application, passing a reference to the buffer. To allow the application code to manage its memory usage, and possibly avoid unnecessary copying, the client stub first calls back to the application's **pipe->alloc** routine to get a buffer. In some cases, this may result in the **pipe->push** routine's not having any work to do.

The client stub may go through more than one (**pipe->alloc**, **pipe->push**) cycle in order to unmarshal data that the server marshalled as a single chunk. Note that there is no guarantee that chunks seen by the client stub will match the chunks supplied by the server's push routine.

The out Pipes: Server Side

The stub enforces well-behaved pipe filling by the manager by raising exceptions as necessary. The **out** pipes must be completely filled, in order, after all **in** pipes have been drained completely.

The manager calls the stub-supplied **push** routine with code similar to the following:

```
#define DESIRED_NUM_ELEMENTS ...
long count;
base_t buf [DESIRED_NUM_ELEMENTS];
        .
        .
        .
while (more_pipe_data) {
        .
        .
        .
    (* (pipe->push)) (
       pipe->state,
       &buf,
       count
    );
   };
   (* (pipe->push)) (
      pipe->state,
      &buf,
      0
   );
```

The in,out Pipes: Client Side

For an **in,out** pipe, the client application code provides the **pipe_t** structure. Both the **pull** routine (for the **in** direction) and the **push** routine (for the **out** direction) must be initialized, as well as the **alloc** routine and the state.

During the last **pull** call (when it will return a zero count to indicate that the pipe is drained), the application's **pull** routine must reinitialize the pipe state so that the pipe can be used by the **push** routine correctly.

The in,out Pipes: Server Side

For an **in,out** pipe, the server provides the **pipe_t** structure.

17.14.4.3 Rules for Using Pipes

Observe the following rules when defining pipes in IDL:

- Pipe types can only be parameters. In other words, a pipe type cannot be the base type of an array or a pipe, or a member of a structure or union.
- A pipe cannot be a function result.
- The base type of a pipe cannot be a pointer or contain a pointer.
- The base type of a pipe cannot be a **context_handle** or **handle_t** type.
- A pipe type cannot be used in the definition of another type. For example, the following code fragment is illegal:

```
typedef
   pipe char pipe_t;

typedef
   pipe_t * pipe_p_t;
```

- A pipe type cannot have the **transmit_as** attribute.
- The base type of a pipe cannot have the **transmit_as** attribute.
- A pipe parameter can be passed by value or by reference. A pipe that is passed by reference (that is, has an * (asterisk)) cannot have the **ptr** or **unique** parameter attributes.
- Pipes that pass data from the client to the server must be processed in the order in which they occur in an operation's signature. All such pipes must be processed before data is sent from the server to the client.
- Pipes that pass data from the server to the client must be processed in the order in which they occur in an operation's signature. No such pipes must be processed until all data has been sent from the client to the server.

- An operation that has one or more pipe parameters cannot have the **idempotent** attribute.
- Manager routines must reraise RPC pipe and communications exceptions so that client stub code and server stub code continue to execute properly.

  For example, consider an interface that has an **out** pipe along with other **out** parameters. Suppose that the following sequence of events occurs:

  — The manager routine closes the pipe by writing an empty chunk whose length is 0 (zero).

  — The manager routine attempts to write another chunk of data to the pipe.

  — The generated **push** routine raises the exception **rpc_x_fault_pipe_closed**.

  — The manager routine catches the exception and does not reraise it.

  — The manager routine exits normally.

  — The server stub attempts to marshall the **out** parameters.

  After this sequence, neither the server stub nor the client stub can continue to execute properly.

  To avoid this situation, you *must* reraise the exception.
- A pipe cannot be used in an operation that has the **broadcast** attribute.
- The base type of a pipe cannot be a conformant structure.

17.14.5 Arrays

IDL supports the following types of arrays:

- Fixed: The size of the array is defined in the IDL and all of the data in the array is transferred during the call.
- Conformant: The size of the array is determined at runtime by the value of the field or parameter referenced by a **max_is** or **size_is** attribute. All of the data in the array is transferred during the call.

- Varying: The size of the array is defined in the IDL but the part of its contents transferred during the call is determined by the values of fields or parameters named in one or more data limit attributes. The data limit attributes are **first_is**, **length_is**, and **last_is**.

An array can also be both conformant and varying (or, as it is sometimes termed "open"). In this case, the size of the array is determined at runtime by the value of the field or parameter referenced by the **max_is** or **size_is** attribute. The part of its contents transferred during the call is determined by the values of fields or parameters named in one or more of the data limit attributes.

Note: IDL supports conformance and variance only on the first major dimension of an arrray. It also supports only 0 (zero) as the lower array bound.

An IDL array is declared via an *array_declarator* construct whose syntax is as follows:

array_identifier array_bounds_declarator ...

An *array_bounds_declarator* must be specified for each dimension of an array.

17.14.5.1 Array Bounds

The *array_bounds_declarator* for the first dimension of an array can take any of the following forms:

[*lower* .. *upper*] The lower bound is *lower*. The upper bound is *upper*.

[*size*] The lower bound is 0 (zero). The upper bound is *size* - 1.

[*lower* .. *] The lower bound is *lower*. The upper bound is unspecified. An instance of this array type must have either the **max_is** attribute or the **size_is** attribute, but not both.

[*] The lower bound is 0 (zero). The upper bound is unspecified. An instance of this array type must have either the **max_is** attribute or the **size_is** attribute, but not both.

[ ] Same as the preceding explanation.

In IDL, the only legal value for *lower* is 0 (zero).

If an array is multidimensional, all dimensions other than the first must have fixed bounds. The *array_bounds_declarator* for each of these dimensions can therefore take only the following forms:

[*lower* .. *upper*] The lower bound is *lower*. The upper bound is *upper*.

[*size*] The lower bound is 0 (zero). The upper bound is *size* - 1.

In all forms of *array_bounds_declarator* the *lower* and *upper* must resolve to integer constants.

17.14.5.2 Array Attributes

Array attributes specify the size of an array or the part of an array that is to be transferred during a call. An array attribute specifies a variable that is either a field in the same structure as the array or a parameter in the same operation as the array.

An *array_attribute* can take the following forms:

max_is ([*] *variable*)
size_is ([*] *variable*)
last_is ([*] *variable*)
first_is ([*] *variable*)
length_is ([*] *variable*)

A *variable* specifies a variable whose value at runtime will determine the bound or element count for the associated dimension. A pointer variable is indicated by preceding the variable name with an * (asterisk).

If the array is a member of a structure, any referenced variables must be members of the same structure. If the array is a parameter of an operation, any referenced variables must be parameters of the same operation.

Only the **..._is**(*variable*) form is allowed when the array is a field of a structure. In this case, the **..._is**(**variable*) form is not allowed.

Note that an array with an array attribute (that is, a conformant or varying array) is not allowed to have the **transmit_as** attribute.

The max_is Attribute

The **max_is** attribute allows the maximum possible upper bound for the major dimension of an array to be determined at runtime. When an array with the **max_is** attribute is used at runtime, the value in the identified variable specifies the maximum array index in the dimension.

If the major dimension of an array has an unspecified upper bound, the array must have either the **max_is** attribute or the **size_is** attribute, but not both. A variable must be identified for this dimension.

The **max_is** attribute is for use with conformant arrays. Following are some examples of the **max_is** attribute:

```
/*
** Assume the following value for the referenced variable:
**  long a = 10;
*/

[max_is(a)] long f1[];                    /* f1[0..10] */
[max_is(a)] long f2[][4];                 /* f2[0..10][0..3] */
```

The size_is Attribute

The **size_is** attribute allows the maximum possible element count for the major dimension of an array to be determined at runtime. When an array with the **size_is** attribute is used at runtime, the value in the identified variable specifies the number of elements in the dimension.

If the major dimension of an array has unspecified upper bounds, the array must have either the **max_is** attribute or the **size_is** attribute, but not both. A variable must be identified for this dimension.

The size of a dimension is defined as the upper bound, minus the lower bound, + 1.

The **size_is** attribute is for use with conformant arrays. Following is an example of the **size_is** attribute:

```
/*
** Assume the following value for the referenced variable:
**    long x2 = 12;
*/

[size_is(x2)] long g1[][6];                    /* g1[0..11][0..6] */
```

The last_is Attribute

The **last_is** attribute allows the amount of data in the first dimension of an array that is to be transmitted to be determined at runtime. When an array with the **last_is** attribute is used, the value in the identified variable specifies the upper data limit.

An array can have either the **last_is** attribute or the **length_is** attribute, but not both.

When an array with the **last_is** attribute is used in a remote procedure call, the elements actually passed in the call can be a subset of the maximum possible.

The **last_is** attribute is for use with varying arrays. Following is an example of the **last_is** attribute:

```
/*
** Assume the following value for the referenced variable:
**    long a = 1;
*/
                                         /* What is transmitted */
[last_is(a) long x1[10][20];            /* x1[0..1][0..19] */
```

The first_is Attribute

The **first_is** attribute allows the amount of data in the first dimension of an array that is to be transmitted to be determined at runtime. When an array with the **first_is** attribute is used, the value in the identified variable specifies the lower data limit.

When an array with the **first_is** attribute is used in a remote procedure call, the elements actually passed in the call can be a subset of the maximum possible.

The **first_is** attribute is for use with varying arrays. Following is an example of the **first_is** attribute:

```
/*
** Assume the following value for the referenced variable:
**     long q = 3;
*/
                                          /* What is transmitted */
[first_is(q)] long x1[0..10];             /* x1[3..10] */
```

The length_is Attribute

The **length_is** attribute allows the amount of data in the first dimension of an array that is to be transmitted to be determined at runtime. When an array with the **length_is** attribute is used, the value in the identified variable specifies the actual number of elements for the dimension.

An array can have either the **last_is** attribute or the **length_is** attribute, but not both.

When an array with the **length_is** attribute is used in a remote procedure call, the elements actually passed in the call can be a subset of the maximum possible.

The **length_is** attribute is for use with varying arrays. Following is an example of the **length_is** attribute:

```
/*
** Assume the following values for the referenced variables:
**     long a = 12;
**     long b = 25;
*/
                                          /* What is transmitted */
[length_is(a),max_is(b)] long t1[][20];   /* t1[0..11][0..19] */
```

17.14.5.3 Rules for Using Arrays

Observe the following rules when defining arrays in IDL:

- A structure can contain only one conformant array, which must be the last member in the structure.
- Conformant arrays are not valid in unions.
- A structure containing a conformant array can be passed only by reference.
- Arrays that have the **transmit_as** attribute cannot be conformant or varying arrays.
- The structure member or parameter referenced in an array attribute cannot be defined to have either the **represent_as** or **transmit_as** attribute.
- Array bounds must be integers. Array attributes can reference only structure members or parameters of integer type.
- A parameter that is referenced by an array attribute on a conformant array must have the **in** attribute.
- Array elements cannot be context handles or pipes.

17.14.6 Strings

IDL implements strings as one-dimensional arrays to which the **string** attribute is assigned. The element type of the array must resolve to one of the following:

- Type **char**
- Type **byte**
- A structure all of whose members are of type **byte** or of a named type that resolves to **byte**
- A named type that resolves to one of the previous three types

An array with the **string** attribute represents a string of characters. The **string** attribute does not specify the format of the string or the mechanism for determining its length. Implementations of IDL provide string formats and mechanisms for determining string lengths that are compatible with the programming languages in which applications are written. For DCE RPC IDL, the number of characters in a **string** array includes the **NULL** terminator, and the entire terminated string is passed between stubs.

The *array_bounds_declarator* for a **string** array determines the maximum number of characters in the array. Note that when you declare a string, you must allocate space for one more than the maximum number of characters the string is to hold. For instance, if a string is to store 80 characters, the string must be declared with a size of 81:

```
/* A string type that holds 80 characters */
typedef
   [string] char string_t [81];
```

If an array has the **string** attribute or if the type of an array has the **string** attribute, the array cannot have the **first_is**, the **last_is**, or the **length_is** attribute.

The **string** and **v1_string** attributes are mutually exclusive. If a parameter or structure member has the **string** attribute, the type of the parameter or member cannot have the **v1_string** attribute and cannot be defined in terms of another type with the **v1_string** attribute. Likewise, no instance of a **string** type can have the **v1_string** attribute.

17.14.7 Pointers

Use the following syntax to declare an IDL pointer:

**[*]...pointer_identifier*

The * (asterisk) is the pointer operator, and multiple asterisks indicate multiple levels of indirection.

17.14.7.1 Pointer Attributes

Pointers are used for several purposes, including implementing a parameter passing mechanism that allows a data value to be returned, and building complex data structures. IDL offers two classes of pointers: reference pointers and full pointers. The attributes that indicate these pointers are as follows:

- **ref**: Indicates reference pointers. This is the default for top-level pointers used in parameters.
- **ptr**: Indicates full pointers.

Pointer attributes are used in parameters, structure and union members, and in type definitions. In some instances, IDL infers the applicable pointer class from its usage. However, most pointer declarations require that you specify a pointer class by using one of the following methods:

- Use the **ref** or **ptr** attribute in the pointer declaration.
- Use the **pointer_default** attribute in the IDL interface heading. The default pointer class is determined by the **pointer_default** attribute.

Pointer attributes are applied only to the top-level pointer within the declaration. If multiple pointers are declared in a single declaration, the **pointer_default** established applies to all but the top-level pointer. (See Section 17.14.7.2, which describes pointer attributes in parameters.)

Examples of pointers are shown at the end of this section.

Reference Pointers

A reference pointer is the less complex form of pointer. The most common use for this class of pointer is as a passing mechanism; for example, passing an integer by reference. Reference pointers have significantly better performance than full pointers, but are restrictive; you cannot create a linked list using a reference pointer because a reference pointer cannot have a **NULL** value, and the list cannot be terminated.

A reference pointer has the following characteristics:

- It always points to valid storage; it can never have a **NULL** value.
- Its value does not change during a call; it always points to the same storage on return from the call as it did when the call was made.

- It does not support aliasing; it cannot point to a storage area that is pointed to by any other pointer used in a parameter of the same operation.

When a manager routine is entered, all the reference pointers in its parameters will point to valid storage, except those reference pointers that point neither to targets whose size can be determined at compile time nor to values that have been received from the client.

In the following example, the size of the targets of the reference pointers can be calculated at compilation time:

```
typedef [ref] long *rpl;

void op1( [in] long f,
          [in] long l,
          [in,first_is(f),last_is(l)] rpl rpla[10] );
```

For this example, when the manager is entered, all the pointers in rpla will point to usable storage, although only *rpla[f] through *rpla[l] will be the values received from the client.

Conversely, the size of the targets of the reference pointers cannot be calculated at compile time in the following example:

```
typedef [ref,string] char *rps;

void op1( [in] long f,
          [in] long l,
          [in,first_is(f),last_is(l)] rps rpsa[10] );
```

In this case, only rpsa[f] through rpsa[l], which point to values received from the client, will point to usable storage.

Full Pointers

A full pointer is the more complex form of pointer. It supports all capabilities associated with pointers. For example, by using a full pointer you can build complex data structures such as linked lists, trees, queues, or arbitrary graphs.

A full pointer has the following characteristics:

- Its value can change during a call; it can change from a **NULL** to non-**NULL** value, non-**NULL** to **NULL**, or from one non-**NULL** value to another non-**NULL** value.
- It supports aliasing; it can point to a storage area that is also pointed to by any other full pointer used in a parameter of the same operation. However, all such pointers must point to the beginning of the structure. There is no support for pointers to substructures or to overlapping storage areas. For example, if the interface definition code contains the following:

  ```
  [uuid(0E256080-587C-11CA-878C-08002B111685), version(1.0)]
  interface overlap
  {
    typedef struct {
            long bill;
            long charlie;
    } foo;
    typedef struct {
            long fred;
            foo ken;
    } bar;

    void op ( [in] foo *f, [in] bar *b );
  }
  ```

  and the client application code includes:

  ```
  bar bb;
    .
    .
  op ( &bb.ken, &bb );
  ```

  then the server stub treats these two separate parameters as distinct, and the manager application code does not see them as overlapping storage.
- It allows dynamically allocated data to be returned from a call.

17.14.7.2 Pointer Attributes in Parameters

A pointer attribute can be applied to a parameter only if the parameter contains an explicit pointer declaration (*).

By default, a single pointer (*) operator in a parameter list of an operation declaration is treated as a reference pointer. To override this, specify a pointer attribute for the parameter. When there is more than one pointer operator, or multiple levels of indirection in the parameter list, the pointer on the right is the top-level pointer; all pointers to the left are of a lower level. This top-level pointer is treated as a reference pointer by default; the other lower-level pointers have the semantics specified by the **pointer_default** attribute in the interface. Any pointer attribute you specify for the parameter applies to the top-level pointer only. Note that unless you specify a pointer attribute, the top-level explicit pointer declaration in a parameter defaults to a reference pointer even if the **pointer_default(ptr)** interface attribute is specified.

Using a reference pointer improves performance but is more restrictive. For example, the pointer declared in the following operation, for the parameter **int_value**, is a reference pointer. An application call to this operation can never specify **NULL** as the value of **int_value**.

```
void op ([in] long *int_value);
```

To pass a **NULL** value, use a full pointer. The following two methods make **int_value** into a full pointer:

- Applying the **ptr** attribute to the declaration of the parameter, **int_value**

  ```
  void op ([in, ptr] long *int_value);
  ```

- Using the **pointer_default (ptr)** attribute in an interface header

  ```
  [uuid(135E7F00-1682-11CA-BF61-08002B111685,
   pointer_default(ptr),
   version(1.0)] interface full_pointer
  {
  typedef long *long_ptr;
  void op ([in] long_ptr int_value);
  }
  ```

17.14.7.3 Pointer Attributes in Function Results

Function results that are pointers are always treated as full pointers. The **ptr** attribute is allowed on function results but it is not mandatory. The **ref** pointer attribute is never allowed on function results.

A function result that is a pointer always indicates new storage. A pointer parameter can reference storage that was allocated before the function was called, but a function result cannot.

17.14.7.4 Pointers in Structure Fields and Union Case

If a pointer is declared in a member of a structure or union, its default is determined by the **pointer_default** attribute you specify for the interface. To override this, specify a pointer attribute for the member.

17.14.7.5 Rules for Using Pointers

Use the following rules when developing code in IDL:

- Do not use the full pointer attribute on the following:
  - — The parameter in the first parameter position, when that parameter is of type **handle_t** or is of a type with the **handle** attribute.
  - — Context handle parameters.
  - — A parameter that has the output attribute (**out**) only.
- The base type of a pipe must not be a pointer or a structure containing a pointer.
- A member of a union or a structure contained in a union cannot contain a reference pointer.
- A reference pointer must point to valid storage at the time the call is made.

- A parameter containing a varying array of reference pointers must have all array elements initialized to point to valid storage even if only a portion of the array is input, since the manager code (the application code supporting an interface on a server) may use the remaining array elements. (Recall that a varying array is one to which any of the array attributes **first_is, last_is, length_is** is applied).

17.14.7.6 Memory Management for Pointed-to Nodes

A full pointer can change its value across a call. Therefore, stubs must be able to manage memory for the pointed-to nodes. Managing memory involves allocating and freeing memory for user data structures.

Allocating and Freeing Memory

Manager code within RPC servers usually uses the **rpc_ss_allocate( )** routine to allocate storage. Storage that is allocated by **rpc_ss_allocate( )** is released by the server stub after any output parameters have been marshalled by the stubs. Storage allocated by other allocators is not released automatically but must be freed by the manager code. When the manager code makes a remote call, the default memory management routines are **rpc_ss_allocate( )** and **rpc_ss_free( )**.

The syntax of the **rpc_ss_allocate( )** routine is as follows:

idl_void_p_t rpc_ss_allocate (idl_size_t *size*);

The *size* parameter specifies the size of the memory allocated.

Note: In ANSI standard C environments, **idl_void_p_t** is defined as **void** * and in other environments is defined as **char** *.

Use **rpc_ss_free( )** to release storage allocated by **rpc_ss_allocate( )**. You can also use the **rpc_ss_free( )** routine to release storage pointed to by a full pointer in an input parameter.

The syntax of the routine is as follows:

void rpc_ss_free (idl_void_p_t *node_to_free***);**

The *node_to_free* parameter specifies the location of the memory to be freed.

Enabling and Disabling Memory Allocation

It may be necessary to call manager routines from different environments; for example, when the application is both a client and a server of the same interface. In this case, the same routine may be called both from server manager code and from client code. The **rpc_ss_allocate**( ) routine, when used by the manager code to allocate memory, must be initialized before its first use. The stub performs the initialization automatically. Code, other than stub code, that calls a routine, which in turn calls **rpc_ss_allocate**( ), first calls the **rpc_ss_enable_allocate**( ) routine.

The syntax of the routine is as follows:

void rpc_ss_enable_allocate (void);

The environment set up by the **rpc_ss_enable_allocate**( ) routine is released by calling the **rpc_ss_disable_allocate**( ) routine. This routine releases all memory allocated by calls to **rpc_ss_allocate**( ) since the call to **rpc_ss_enable_allocate**( ) was made. It also releases memory that was used by the memory management mechanism for internal bookkeeping.

The syntax of the routine is as follows:

void rpc_ss_disable_allocate (void);

17.14.7.7 Advanced Memory Management Support

Memory management may also involve setting and swapping the mechanisms used for allocating and freeing memory. The default memory management routines are **malloc**( ) and **free**( ), except when the remote call occurs within manager code, in which case the default memory management routines are **rpc_ss_allocate**( ) and **rpc_ss_free**( ).

Setting the Client Memory Mechanism

Use the **rpc_ss_set_client_alloc_free()** routine to establish the routines used in allocating and freeing memory.

The syntax of the routine is as follows:

```
void rpc_ss_set_client_alloc_free (
   idl_void_p_t (*p_allocate) (
     idl_size_t size),
    void (*p_free) (
      idl_void_p_t ptr)
    );
```

The *p_allocate* parameter points to a routine that has the same procedure declaration as the **malloc()** routine, and was used by the client stub when performing memory allocation. The *p_free* parameter points to a routine that has the same procedure declaration as the **free()** routine, and was used by the client stub to free memory.

Swapping Client Memory Mechanisms

This routine exchanges the current client allocation and freeing mechanism for one supplied in the call. The primary purpose of this routine is to simplify the writing of modular routine libraries in which RPC calls are made. To preserve modularity, any dynamically allocated memory returned by a modular routine library must be allocated with a specific memory allocator. When dynamically allocated memory is returned by an RPC call that is then returned to the user of the routine library, use **rpc_ss_swap_client_alloc_free()** to make sure the desired memory allocator is used. Prior to returning, the modular routine library calls **rpc_ss_set_client_alloc_free()** to restore the previous memory management mechanism.

The syntax of the routine is as follows:

```
void rpc_ss_swap_client_alloc_free (
    idl_void_p_t (*p_allocate) (
      idl_size_t size),
    void (*p_free) (
      idl_void_p_t ptr),
    idl_void_p_t (**p_p_old_allocate) (
      idl_size_t size),
    void (**p_p_old_free) (
      idl_void_p_t ptr)
    );
```

The *p_allocate* parameter points to a routine that has the same procedure declaration as the **malloc**( ) routine, and was used by the client stub when performing memory allocation. The *p_free* parameter points to a routine that has the same procedure declaration as the **free**( ) routine, and was used by the client stub to free memory. The *p_p_old_allocate* parameter points to a pointer to a routine that has the same procedure declaration as the **malloc**( ) routine, and was used for memory allocation in the client stub. The *p_p_old_free* parameter points to a pointer to a routine that has the same procedure declaration as the **free**( ) routine, and was used for memory release in the client.

17.14.7.8 Use of Thread Handles in Memory Management

There are two situations where control of memory management requires the use of thread handles. The more common situation is when the manager thread spawns additional threads. The less common situation is when a program transitions from being a client to being a server, then reverts to being a client.

Spawning Threads

When a remote procedure call invokes the manager code, the manager code may wish to spawn additional threads to complete the task for which it was called. To spawn additional threads that are able to perform memory management, the manager code must first call the **rpc_ss_get_thread_handle**( ) routine to get its thread handle and then pass that thread handle to each spawned thread. Each spawned thread that uses the **rpc_ss_allocate( )** and **rpc_ss_free( )** routines for memory management first calls the **rpc_ss_set_thread_handle( )** routine by using the handle obtained by the original manager thread.

These routine calls allow the manager and its spawned threads to share a common memory management environment. This common environment enables memory allocated by the spawned threads to be used in returned parameters, and causes all allocations in the common memory management environment to be released when the manager thread returns to the server stub.

The main manager thread must not return control to the server stub before all the threads it spawned complete execution; otherwise, unpredictable results may occur.

Transitioning from Client to Server to Client

Immediately before the program changes from a client to a server, it must obtain a handle on its environment as a client by calling **rpc_ss_get_thread_handle( )**. When it reverts from a server to a client, it must reestablish the client environment by calling the **rpc_ss_set_thread_handle( )** routine, supplying the previously obtained handle as a parameter.

Syntax for Thread Routines

The syntax for the **rpc_ss_get_thread_handle( )** routine is as follows:

```
rpc_ss_thread_handle_t  rpc_ss_get_thread_handle(void);
```

The syntax for the **rpc_ss_set_thread_handle( )** routine is as follows:

```
void rpc_ss_set_thread_handle (
   rpc_ss_thread_handle_t id
   );
```

The **rpc_ss_thread_handle_t( )** value identifies the thread to the RPC stub support library. The *id* parameter indicates the thread handle passed to the spawned thread by its creator, or the thread handle returned by the previous call to **rpc_ss_get_thread_handle( )**.

17.14.7.9 Rules for Using the Memory Management Routines

You can use the **rpc_ss_allocate**( ) routine in the following environments:

- The manager code for an operation that has a full pointer in its argument list
- The manager code for an operation to which the **enable_allocate** ACF attribute is applied
- Code that is not called from a server stub but that has called the **rpc_ss_enable_allocate**( ) routine
- A thread, spawned by code of any of the previous three types, that has made a call to the **rpc_ss_set_thread_handle**( ) routine using a thread handle obtained by this code

17.14.7.10 Examples Using Pointers

The examples in this subsection contain the following files, listed here with the function of each file:

Example	Function
STRING_TREE.IDL	Defines data types and interfaces
CLIENT.C	The user of the interface
MANAGER.C	The server code that implements the procedure
SERVER.C	Declares the server; enables the client code to find the interface it needs
STRING_TREE.OUTPUT	Shows the output

The STRING_TREE.IDL Example

```
[uuid(0144D600-2D28-11C9-A812-08002B0ECEF1), version(0)]
interface string_tree
{
  /*
   * Maximum length of a string in the tree
   */
  const long int st_c_name_len = 32;
  /*
   * Definition of a node in the tree.
   */
  typedef struct node
  {
      [string] char name[0..st_c_name_len];
      [ptr] struct node *left;
      [ptr] struct node *right;
  } st_node_t;


  /*
   * Operation that prunes the left subtree of the specified
   * tree and returns it as the value.
   */
  st_node_t *st_prune_left (
      [in, out] st_node_t *tree /* root of tree by ref */
      );
}
```

The CLIENT.C Example

```
#include <stdio.h>
#include "string_tree.h"

#include <stdlib.h>

/*
** Routine to print a depiction of the tree
*/
void st_print_tree (tree, indent)
  st_node_t *tree;
  int  indent;
{
  int i;
  if (tree == NULL) return;
  for (i = 0; i < indent; i++) printf("    ");
  printf("%s\n",tree->name);
  st_print_tree(tree->left, indent + 1);
  st_print_tree(tree->right, indent + 1);
```

```
}


/*
** Create a tree with a few nodes
*/
st_node_t *st_make_tree()
{
  st_node_t *root = (st_node_t *)malloc(sizeof(st_node_t));
  strcpy(root->name,"Root Node");

  /* left subtree node */
  root->left =  (st_node_t *)malloc(sizeof(st_node_t));
  strcpy(root->left->name,"Left subtree");

  /* left subtree children */
  root->left->right = NULL;
  root->left->left = (st_node_t *)malloc(sizeof(st_node_t));
  strcpy(root->left->left->name,"Child of left subtree");
  root->left->left->left = NULL;
  root->left->left->right = NULL;

  /* right subtree node */
  root->right =  (st_node_t *)malloc(sizeof(st_node_t));
  strcpy(root->right->name,"Right subtree");
  root->right->left = NULL;
  root->right->right = NULL;

  return root;
}

main()
{
  st_node_t *tree;
  st_node_t *subtree;

  /* setup and print original tree */
  tree = st_make_tree();
  printf("Original Tree:\n");
  st_print_tree(tree, 1);
  /* call the prune routine */
  subtree = st_prune_left (tree);

  /* print the resulting trees */
  printf("\nPruned Tree:\n");
  st_print_tree(tree, 1);

  printf("\nPruned subtree:\n");
  st_print_tree(subtree, 1);
  }
```

The MANAGER.C Example

```
#include <stdio.h>
#include "string_tree.h"

/*
** Prune the left subtree of the specified tree and return
** it as the function value.
*/
st_node_t *st_prune_left (tree)
  /* [in,out] */  st_node_t *tree;
{
  st_node_t *left_sub_tree = tree->left;
  tree->left = (st_node_t *)NULL;
  return left_sub_tree;
}
```

The SERVER.C Example

```
#include <stdio.h>
#include "string_tree.h"  /* header created by idl compiler */
#define check_error(s, msg) if(s != rpc_s_ok) \
  {fprintf(stderr, "%s", msg); exit(1);}

main ()
{
  unsigned32                status;          /* error status (nbase.h) */
```

```
rpc_binding_vector_p_t  binding_vector; /* set of binding handles (rpc.h) */

rpc_server_register_if(           /* register interface with RPC runtime */
  string_tree_v0_0_s_ifspec,   /* interface specification (string_tree.h) */
  NULL,
  NULL,
  &status                                               /* error status */
);
check_error(status, "Can't register interface\n");

rpc_server_use_all_protseqs(              /* establish protocol sequences */
  rpc_c_protseq_max_calls_default, /* concurrent calls server takes (rpc.h) */
  &status
);
check_error(status, "Can't establish protocol sequences\n");

rpc_server_inq_bindings(      /* get set of this server's binding handles */
  &binding_vector,
  &status
);
check_error(status, "Can't get binding handles\n");

rpc_ep_register(          /* register addresses in endpoint map database */
  string_tree_v0_0_s_ifspec,   /* interface specification (string_tree.h) */
  binding_vector,                   /* the set of binding handles          */
  NULL,
  "",
  &status
);
check_error(status, "Can't add address to the endpoint database\n");

rpc_ns_binding_export(         /* establish namespace entry   */
  rpc_c_ns_syntax_dce,         /* syntax of the entry name (rpc.h)        */
  "string_tree",               /* entry name in directory service         */
  &string_tree_v0_0_s_ifspec,  /* interface specification (string_tree.h) */
  binding_vector,              /* the set of binding handles              */
  NULL,
  &status
);
check_error(status, "Can't export to directory service\n");
```

```
    rpc_binding_vector_free(                    /* free set of binding handles */
      &binding_vector,
      status
    );
    check_error(status, "Can't free binding handles and vector\n");

    rpc_server_listen(                              /* listen for remote calls */
      rpc_c_listen_max_calls_default, /* concurrent calls server executes (rpc.h) */
      &status
    );
    check_error(status, "rpc listen failed\n");
  }
```

The STRING_TREE.OUTPUT Example

```
Original Tree:
    Root Node
       Left subtree
           Child of left subtree
       Right subtree
Pruned Tree:
    Root Node
        Right subtree
Pruned subtree:
    Left subtree
        Child of left subtree
```

17.14.8 Customized Handles

The **handle** attribute specifies that the type being declared is a user-defined, nonprimitive handle type, and is to be used in place of the predefined primitive handle type **handle_t**. The term ‘‘customized handle’’ is used to denote a nonprimitive handle.

The following example declares a customized handle type **filehandle_t**, a structure containing the textual representations of a host and a pathname:

```
typedef [handle] struct {
    char host[256];
    char path[1024];
    } filehandle_t;
```

To build an application that uses customized handles, you must write custom binding and unbinding routines, and you must link those routines with your application client code. At runtime, each time the client calls an operation that uses a customized handle, the client stub calls the custom binding routine before it sends the remote procedure call request, and the client stub calls the custom unbinding routine after it receives a response.

The following paragraphs specify C prototypes for customized binding and unbinding routines; in these prototypes, *CUSTOM* is the name of the customized handle type.

The custom binding routine *CUSTOM*_**bind** generates a primitive binding handle from a customized handle and returns the primitive binding handle:

handle_t *CUSTOM*_**bind** (*CUSTOM c-handle*)

The custom unbinding routine *CUSTOM*_**unbind** takes two inputs, a customized handle and the primitive binding handle that was generated from it, and has no outputs:

void *CUSTOM*_**unbind** (
CUSTOM c-handle,
handle_t *rpc-handle*)

A custom unbinding routine typically frees the primitive binding handle and any unneeded resources associated with the customized handle, but it is not required to do anything.

Because the **handle** attribute can occur only in a type declaration, a customized handle must have a named type. Because customized handle type names are used to construct custom binding and unbinding routine names, these names cannot exceed 24 characters.

A customized handle can be coded either in a parameter list as an explicit handle or in an interface header as an implicit handle.

17.14.9 Context Handles

Manager code often maintains state information for a client. A handle to this state information is passed to the client in an output parameter or as an operation result. The client passes the unchanged handle-to-the-state information as an input or input/output parameter of a subsequent manager operation that the client calls to manipulate that data structure. This handle-to-the-state information is called a ''context handle.'' A context handle is implemented as an untyped pointer.

The manager causes the untyped pointer to point to the state information it will need the next time the client asks the manager to manipulate the context. For the client, the context handle is an opaque pointer (**idl_void_p_t**). The client receives or supplies the context handle by means of the parameter list, but does not perform any transformations on it.

The RPC runtime maintains the context handle, providing an association between the client and the address space running the manager and the state information within that address space.

If a manager supports multiple interfaces, and a client obtains a context handle by performing an operation from one of these interfaces, the client can then supply the context handle to an operation from another of these interfaces.

No client except the one that obtained a context handle may use that context handle.

17.14.9.1 The context_handle Attribute

Specify a context handle by either of the following methods:

- Use the **context_handle** attribute on a parameter of type **void** *.
- Use the **context_handle** attribute on a type that is defined as **void** *.

For example, in the IDL file, you can define a context handle within a type declaration as follows:

```
typedef [context_handle] void * my_context;
```

or within a parameter declaration as follows:

```
[in, context_handle] void * my_context;
```

The first operation on a context creates a context handle that the server procedure passes to the client. The client then passes the unmodified handle back to the server in a subsequent remote call. The called procedure interprets the context handle. For example, to specify a procedure that a client can use to obtain a context handle, you may define the following:

```
typedef [context_handle] void * my_context;
void op1(
   [in]handle_t h,
   [out] my_context * this_object);
```

To specify a procedure that a client may call to make use of a previously obtained context handle, you can define the following:

```
void op2([in] my_context this_object);
```

To close a context, and to clean the context on the client side, you can define the following:

```
[in, out, context_handle] void * my_context;
```

The resources associated with a context handle are reclaimed when, and only when, the manager changes the value of the **in,out** context handle parameter from non-**NULL** to **NULL**.

17.14.9.2 The Context Rundown Procedure

Some uses of context handles may require you to write a context rundown procedure in the application code for the server. If communications between the client and server are broken while the server is maintaining context for the client, RPC invokes the context rundown procedure on the server to recover the resources represented by the context handle. If you declare a context handle as a named type, you must supply a rundown procedure for that type.

When a context requires a context rundown procedure, you must define a named type that has the **context_handle** attribute. For each different context handle type, you must provide a context rundown procedure as part of the manager code.

The format for the rundown procedure name is as follows:

*context_type_name*_**rundown**

A rundown procedure takes one parameter, the handle of the context to be run down, and delivers no result. For example, if you declare the following:

```
typedef [context_handle] void * my_context;
```

then the rundown procedure is as follows:

```
void my_context_rundown (my_context this_object);
```

17.14.9.3 Creating New Context

When a client makes its first request to the manager to manipulate context, the manager creates context information and returns this information to the client through a parameter of the type **context_handle**. This parameter must be an output parameter or an input/output parameter whose value is **NULL** when the call is made. A context handle can also be a function result.

17.14.9.4 Reclaiming Client Memory Resources for the Context Handle

In the event that a communications error causes the context handle to be unusable, the resources that maintain the context handle must be reclaimed. Use the **rpc_ss_destroy_client_context**( ) routine in the client application to reclaim the client side resources and to set the context handle value to **NULL**.

The syntax of the routine is as follows:

void rpc_ss_destroy_client_context(
  void **p_unusable_context_handle);*

17.14.9.5 Relationship of Context Handles and Binding

For the client, the context handle specifies the state within a server, and also contains binding information. If an operation has an input context handle or input/output context handle that is not **NULL**, it is not necessary to supply any other binding information. A context handle that has only the **in** attribute cannot be **NULL**. If an operation has **in,out** context handle parameters but no **in** context handle parameters, at least one of the **in,out** context handle parameters cannot be **NULL**. However, if the only context handle parameters in an operation are output, they carry no binding information. In this case, you must use another method to bind the client to a server.

If you specify multiple context handles in an operation, all active context handles must map to the same remote address space on the same server or the call fails. (A context handle is active while it represents context information that the server maintains for the client. It is inactive if no context has yet been created, or if the context is no longer in use.)

17.14.9.6 Rules for Using Context Handles

The following rules apply to using context handles:

- A context handle can be a parameter or a function result. You cannot use context handles as an array element, as a structure or union member, or as the base type of a pipe.
- A context handle cannot have the **transmit_as** or **ptr** attributes.
- An input-only context handle cannot be **NULL**.
- A context handle cannot be pointed to, except by a top-level reference pointer.

17.14.9.7 Examples Using Context Handles

The following examples show a sample IDL file that uses context handles and a sample context rundown procedure file.

Example of an IDL File That Uses a Context Handle

```
/*
 * Filename: context_handle.idl
 */
[uuid(F38F5080-2D27-11C9-A96D-08002B0ECEF1),
 pointer_default(ref), version (1.0)]
interface files
{
/* File context handle type */
typedef [context_handle] void * file_handle_t;
/* File specification type */
typedef [string] char * filespec_t;
/* File read buffer type */
typedef [string] char buf_t[*];

 /*
  * The file_open call requires that the client has located a
  * file server interface files and that an RPC handle that is
  * bound to that server be passed as the binding parameter h.
  *
  * Operation to OPEN a file; returns context handle for that file.
  */
file_handle_t file_open
(
 /* RPC handle bound to file server */
    [in] handle_t h,
 /* File specification of file to open */
    [in] filespec_t fs
);

 /*
  * The file_read call is able to use the context handle obtained
  * from the file_open as the binding parameter, thus an RPC
  * handle is not necessary.
  *
```

```
     * Operation to read from an opened file; returns true if not
     * end-of-file
     */
   boolean file_read
   (
    /* Context handle of opened file */
       [in] file_handle_t fh,
    /* Maximum number of characters to read */
       [in] long buf_size,
    /* Actual number of characters of data read */
       [out] long *data_size,
    /* Buffer for characters read */
       [out, size_is(buf_size), length_is(*data_size)] buf_t buffer
   );
    /* Operation to close an opened file */
   void file_close
   (
    /* Valid file context handle goes [in]. On successful close,
     * null is returned.
     */
       [in,out] file_handle_t *fh
   );
   }
```

Example of a Context Rundown Procedure

```
/*
 * fh_rundown.c:  A context rundown procedure.
 */

#include <stdio.h>
#include "context_handle.h"     /* IDL-generated header file */

void file_handle_t_rundown
(
    file_handle_t file_handle   /* Active context handle
                                 * (open file handle) */
)

{
    /*
```

```
        * This procedure is called by the RPC runtime on the SERVER
        * side when communication is broken between the client and
        * server. This gives the server the opportunity to reclaim
        * resources identified by the passed context handle.  In
        * this case, the passed context handle identifies a file,
        * and simply closing the file cleans up the state maintained
        * by the context handle, that is "runs down" the context handle.
        * Note that the file_close manager operation is not used here;
        * perhaps it could be, but it is more efficient to use the
        * underlying file system call to do the close.
        *
        * File handle is void*, it must be cast to FILE*
        */
       fclose((FILE *)file_handle);
}
```

17.15 Associating a Data Type with a Transmitted Type

The **transmit_as** attribute associates a ‘‘transmitted type’’ that stubs pass over the network with a ‘‘presented type’’ that clients and servers manipulate. The specified transmitted type must be a named type defined previously in another type declaration.

There are two primary uses for this attribute:

- To pass complex data types for which the IDL compiler cannot generate marshalling and unmarshalling code.
- To pass data more efficiently. An application can provide routines to convert a data type between a sparse representation (presented to the client and server programs) and a compact one (transmitted over the network).

To build an application that uses presented and transmitted types, you must write routines to perform conversions between the types and to manage storage for the types, and you must link those routines with your application code. At runtime, the client and server stubs call these routines before sending and after receiving data of these types.

The following paragraphs specify C prototypes for generic binding and unbinding routines; in these prototypes, *PRES* is the name of the presented type and *TRANS* is the name of the transmitted type.

The *PRES*_**to_xmit( )** routine allocates storage for the transmitted type and converts from the presented type to the transmitted type:

void *PRES*_**to_xmit** (*PRES *presented*, *TRANS **transmitted*)

The *PRES*_**from_xmit( )** routine converts from the transmitted type to the presented type and allocates any storage referenced by pointers in the presented type:

void *PRES*_**from_xmit** (*TRANS *transmitted*, *PRES *presented*)

The *PRES*_**free_inst( )** routine frees any storage referenced by pointers in the presented type by *PRES*_**from_xmit( )**:

void *PRES*_**free_inst** (*PRES *presented*)

Suppose that the **transmit_as** attribute appears either on the type of a parameter or on a component of a parameter and that the parameter has the **out** or **in,out** attribute. Then, the *PRES*_**free_inst( )** routine will be called automatically for the data item that has the **transmit_as** attribute.

Suppose that the **transmit_as** attribute appears on the type of a parameter and that the parameter has only the **in** attribute. Then, the *PRES*_**free_inst( )** routine will be called automatically.

Finally, suppose that the **transmit_as** attribute appears on a component of a parameter and that the parameter has only the **in** attribute. Then, the *PRES*_**free_inst( )** routine will not be called automatically for the component; the manager application code must release any resources that the component uses, possibly by explicitly calling the *PRES*_**free_inst( )** routine.

The *PRES*_**free_xmit( )** routine frees any storage that has been allocated for the transmitted type by *PRES*_**to_xmit( )**:

void *PRES*_**free_xmit** (*TRANS *transmitted*)

The following types cannot have the **transmit_as** attribute:

- A pipe type
- A pipe element type
- A type with the **context_handle** attribute
- A type of which any instance has the **context_handle** attribute
- A conformant array type
- A varying array type
- A structure type containing a conformant array
- An array type of which any instance is varying

A transmitted type specified by the **transmit_as** attribute must be either a base type, a predefined type, or a named type defined via **typedef**. A transmitted type cannot be a conformant array type or a conformant structure type if any instance of that type is an **in** parameter or an **in, out** parameter.

The following is an example of **transmit_as**. Assuming the following declarations:

```
typedef
   struct tree_node_t {
      data_t data;
      struct tree_node_t * left;
      struct tree_node_t * right;
   } tree_node_t;

typedef
   [transmit_as(tree_xmit_t)] tree_node_t *tree_t;
```

the application code must include routines that match the prototypes:

```
void tree_t_to_xmit ( tree_t *, (tree_xmit_t **) );
void tree_t_from_xmit ( (tree_xmit_t *), (tree_t *) );
void tree_t_free_inst ( tree_t *);
void tree_t_free_xmit ( (tree_xmit_t *) );
```

17.16 Migration Attributes

IDL provides four migration attributes that are compatible for use with existing interfaces written in NCS Version 1 of the Network Interface Definition Language (NIDL). The migration attributes are not intended for use in developing new applications.

Note: NCS Version 1 compatibility is provided only for transitional purposes and will be available for only a limited number of DCE update releases. For new applications, use DCE RPC.

17.16.1 The v1_array Attribute

You can specify the **v1_array** attribute on either of the following:

- A type definition
- A parameter or field definition

A **v1_array** can only have the varying or conformant varying property on its first dimension. The highest data limit that can be specified on this dimension is 65,535. The following table shows the array types and the array attributes that can be used with the **v1_array** attribute. Unlike a standard array, the **v1_array** attribute cannot be applied to a conformant-only array.

To describe array as:	Use v1_array with:
Varying	**last_is** **length_is**
Conformant varying	**max_is** with **last_is** **size_is** with **length_is**

NCS Version 1 does not support conformant-only arrays.

Examples Using the v1_array Attribute

A varying array:

```
[v1_array, last_is (last)] char input_buff [80];
```

Two examples of a conformant varying array:

```
[v1_array, max_is (biggest), last_is (end)] char data_return[];
[v1_array, length_is (longest), size_is (all)] char data_return[];
```

17.16.2 The v1_enum Attribute

You can use the **v1_enum** attribute only on an enumeration. The **v1_enum** attribute specifies that the network representation of the enumeration is an **unsigned long** integer rather than an **unsigned short** integer. This is necessary to operate with an interface written in the NCS Version 1 NIDL C language syntax using the **long enum** data type.

17.16.3 The v1_string Attribute

The **v1_string** attribute applies only to a Version 1.0 array (**v1_array**) of elements whose type resolves to the **char** type. When you use the **v1_string** attribute on a multidimensional array, it applies to the last dimension of the array.

An array that has the **v1_string** attribute must contain a string of characters terminated by a 0 (zero) byte. The bounds for an array containing the **v1_string** attribute specify the maximum number of characters in the array, including the **NULL** terminator.

The **string** and **v1_string** attributes cannot be used together.

17.16.4 The v1_struct Attribute

The **v1_struct** attribute applies only to a structure. It specifies an alternate data alignment for the network representation of the structure.

Chapter 18

Attribute Configuration Language

The Attribute Configuration Language is used for writing an Attribute Configuration File (ACF). Use the attributes in the ACF to modify the interaction between the application code and stubs without changing the IDL file.

18.1 Syntax Notation Conventions

The syntax of the Attribute Configuration Language is similar to the syntax of the Interface Definition Language (IDL). For syntax information, see the syntax notation conventions for the IDL in Chapter 17.

Use of Brackets

The use of [ ] (brackets) can be either a required part of the syntax or can denote that a string is optional to the syntax. To differentiate this, brackets that are required are shown as [ ] (regular type brackets). Brackets that contain optional strings are shown as *[ ]* (italicized brackets).

Use of the Vertical Bar

A | (vertical bar) denotes a logical OR.

18.2 Attribute Configuration File

The ACF changes the way the IDL compiler interprets the interface definition, written in the IDL. The IDL file defines a means of interaction between a client and a server. For new server implementations to be compatible across the network with existing servers, the interaction between the client and server must not be modified. If the interaction between an application and a specific stub needs to change, you must provide an ACF when you build this stub.

The ACF affects only the interaction between the generated stub code and the local application code; it has no effect on the interaction between local and remote stubs. Therefore, client and server writers are likely to have different attribute configuration files that they can modify as desired.

18.2.1 Naming the ACF

To name the ACF, replace the extension of the IDL file (**.idl**) with the extension of the ACF (**.acf**). For example:

The ACF associated with *my_idl_filename*.**idl** is *my_idl_filename*.**acf**.

18.2.2 Compiling the ACF

When you issue the **idl** command, naming the IDL file to compile, the compiler searches for a corresponding ACF and compiles it along with the IDL file. The compiler also searches for any ACF (there can be more than one) associated with any imported IDL files. The stubs that the compiler creates contain the appropriate modifications.

18.2.3 ACF Features

The following list contains the ACF attributes and the features associated with the attributes:

- **include** statement: Includes header files in the generated code
- **auto_handle, explicit_handle, implicit_handle**: Controls binding
- **comm_status, fault_status**: Indicates parameters to hold status conditions occurring in the call
- **code, nocode**: Controls which operations of the IDL file are compiled
- **in_line, out_of_line**: Controls the marshalling of data
- **represent_as**: Controls conversion between local and network data types
- **enable_allocate**: Forces the initialization of the memory management routines
- **heap**: Specifies objects to be allocated from heap memory

18.3 Structure

The structure of the attribute configuration file is as follows:

interface_header
{
interface_body
}

Follow these structural rules when writing an attribute configuration file:

- The basename of the ACF must be the same as the basename of the IDL file although the extensions are different.
- The interface name in the ACF must be the same as the interface name in the corresponding IDL file.

- With a few exceptions, any type, parameter, or operation names in the ACF must be declared in the IDL file, or defined in files included by use of the **include** statement, as the same class of name.
- Except for additional status parameters, any parameter name that occurs within an operation in the ACF must also occur within that operation in the IDL file.

The structure of the ACF in extended Backus-Naur Form (BNF) notation is listed in the Supplemental Information part, Language Grammar Synopsis.

18.3.1 ACF Interface Header

The ACF interface header has the following structure:

[[acf_attribute_list]] **interface** *idl_interface_name*

The *acf_attribute_list* is optional. The interface header attributes can include one or more of the following attributes, entered within brackets. If you use more than one attribute, separate them with commas and include the list within a single pair of brackets. (Note that some of these attributes can be used in the ACF body also. See Section 18.3.2 for more information.)

- **code**
- **nocode**
- **in_line**
- **out_of_line**
- **implicit_handle**(*handle_type handle_name*)
- **auto_handle**
- **explicit_handle**

The following example shows how to use more than one attribute in the ACF interface header:

```
[nocode, auto_handle] interface phone_direct
{
}
```

18.3.2 ACF Interface Body

The ACF interface body can contain the elements in the following list. Note that some of the attributes listed here can also be used in the ACF header, as described in Section 18.3.1. If you use more than one attribute, separate them with commas and include the list within a single pair of brackets.

- An **include** statement:

  include "*filename*" *[,"filename"]* ...;

  Note: Omit the extension of the filename in an **include** statement; the compiler appends the correct extension for the language you are using. For the C language, the compiler appends the **.h** extension.

- A declared type:

  typedef *[***[represent_as** (*local_type_name*)**]** | **[in_line]** |
  [out_of_line] | **[heap]***]* *type_name***;**

- An operation:

  *[***[explicit_handle]** | **[comm_status]** | **[fault_status]** |
  [code] | **[nocode]** | **[enable_allocate]***]* *operation_name* (*[parameter_list]*);

  A *parameter_list* is a list of zero or more parameter names as they appear in the corresponding operation definition of the IDL file. You do not need to use all the parameter names that occur in the IDL operation definition; use only those to which you attach an ACF attribute. If you use more than one parameter name, the names must be separated by commas.

- A parameter within an operation:

  *[***[comm_status]** | **[fault_status]** | **[heap]***]* *parameter_name*

18.3.3 The include Statement

This statement specifies any additional header files you want included in the generated stub code. You can specify more than one header file. Do not specify the directory name or file extension when you use the **include** statement. The compiler appends the correct extension for the language you are using. For C, the compiler appends the **.h** extension. If you want to specify the directory name(s), use the **-cc_opt** or **-I** IDL compiler command arguments.

Use the **include** statement whenever you use the **represent_as** or **implicit_handle** attributes and the specified type is not defined or imported in the IDL file.

The **include** statement has the following syntax. (An example is shown with the **represent_as** example in Section 18.3.10 later in this chapter.)

include "*filename***";**

18.3.4 The auto_handle Attribute

This attribute causes the client stub and RPC runtime to manage the binding to the server by using a directory service. Any operation in the interface that has no parameter containing binding information is bound automatically to a server so the client does not have to specify a binding to a server.

When an operation is automatically bound, the client does not have to specify the server on which an operation executes. If you make a call on an operation without explicit binding information in an interface for which you have specified **auto_handle**, and no client/server binding currently exists, the RPC runtime system selects an available server and establishes a binding. This binding is used for this call and subsequent calls to all operations in the interface that do not include explicit binding information, while the server is still available.

Server termination, network failure, or other problems can cause a break in binding. If this occurs during the execution of an automatically bound operation, RPC issues the call to another server, provided one is available, and the operation is idempotent or the RPC runtime system determines that the call did not start to execute on the server. Similarly, if a communications or server failure occurs between calls, RPC binds to another server for the next call, if a server is available.

If the RPC runtime system is unable to find a server to execute the operation, it reports this by returning a **comm_status** value of **rpc_s_no_more_entries**, or by raising the exception **rpc_x_no_more_entries** if the operation does not use **comm_status** error reporting. Note that if a binding breaks, the RPC runtime starts its search at the directory service entry following the one where the binding broke. This means that even if a server earlier in the list becomes available, it is not treated as a candidate for binding. After the RPC runtime tries each server in the list, it reinitializes the list of server candididates and tries again. If the second attempt is unsuccessful, the RPC runtime reports the status condition, **rpc_s_no_more_entries**. The next call on an operation in the interface starts from the top of the list when looking for a server to bind to.

The **auto_handle** attribute can occur only once in the ACF file.

If an interface uses the **auto_handle** attribute, the presence of a binding handle or context handle parameter in an operation overrides **auto_handle** for that operation.

The **auto_handle** attribute declaration has the following syntax. (See the example at the end of this section.)

For an interface:

[auto_handle] interface *interface_name*

You cannot use **auto_handle** if you use **implicit_handle** or if you use **explicity_handle** in the interface header.

Example Using the auto_handle Attribute

ACF

```
[auto_handle] interface math_1
{
}
```

IDL File

```
[uuid(B3C86900-2D27-11C9-AB09-08002B0ECEF1)]
interface math_1
{
/* This operation has no handle parameter,
 * therefore, uses automatic binding.
 */
long add([in] long a,
         [in] long b);

/*
 * This operation has an explicit handle parameter, h,
 * that overrides the [auto_handle] ACF attribute.
 * Explicit handles also override [implicit_handle].
 */
long subtract ([in] handle_t h,
               [in] long a,
               [in] long b);
}
```

18.3.5 The explicit_handle Attribute

This attribute allows the application program to manage the binding to the server. The **explicit_handle** attribute indicates that a binding handle is passed to the runtime as an operation parameter.

The **explicit_handle** attribute has the following syntax. (See the example at the end of this section.)

For an interface:

[explicit_handle] interface *interface_name*

For an operation:

[explicit_handle] *operation_name* (*[parameter_list]*);

When used as an ACF interface attribute, the **explicit_handle** attribute applies to all operations in the IDL file. When used as an ACF operation attribute, this attribute applies to only the operation you specify.

If you use the **explicit_handle** attribute as an ACF interface attribute, you must not use the **auto_handle** or **implicit_handle** attributes.

Using the **explicit_handle** attribute on an interface or operation has no effect on operations in IDL that have explicit binding information in their parameter lists.

Example Using the explicit_handle Attribute

ACF

```
[explicit_handle] interface math_2
{

  /* This causes the operation, as called by the client, to have the
   * parameter handle_t IDL_handle, at the start of the parameter
   * list, before the parameters specified here in the IDL file.
   */
}
```

IDL File

```
[uuid(41CE5B80-0BA7-11CA-87BA-08002B111685)]
interface math_2
{
long add([in] long a,
         [in] long b);
}
```

18.3.6 The implicit_handle Attribute

This attribute allows the application program to manage the binding to the server. You specify the data type and name of the handle variable as part of the **implicit_handle** attribute. The **implicit_handle** attribute informs the compiler of the name and type of the global variable through which the binding handle is implicitly passed to the client stub. A variable of this type and name is defined in the client stub code and the application initializes the variable before making a call to this interface.

The **implicit_handle** attribute declaration has the following syntax. (See the example at the end of this section.)

For an interface:

[implicit_handle (*handle_type handle_name*)**] interface** *interface_name*

If an interface uses the **implicit_handle** attribute, the presence of a binding handle or **in** or **in,out** context handle parameter in an operation overrides the implicit handle for that operation.

The **implicit_handle** attribute can occur only once in the ACF.

You cannot use the **implicit_handle** attribute if you are using the **auto_handle** attribute or the **explicit_handle** attribute as an interface attribute.

If the type in the **implicit_handle** clause is not **handle_t**, then it is treated as if it has the **handle** attribute. For more information, refer to the description of the **handle** attribute in Chapter 17 of this guide.

The ACF in the following example modifies the **math_3** interface to use an implicit handle.

Example Using the implicit_handle Attribute

ACF

```
[implicit_handle(user_handle_t global_handle)] interface math_3
{
/*
 * Since user_handle_t is not a type defined in the IDL, you
 * must specify an include file that contains the definition
 */
include "user_handle_t_def";
}
```

IDL File

```
[uuid(A01D0280-2D27-11C9-9FD3-08002B0ECEF1)]
interface math_3
{
long add([in] long a,
         [in] long b);
}
```

18.3.7 The comm_status and fault_status Attributes

The **comm_status** and **fault_status** attributes cause the status code of any communications failure or server runtime failure that occurs in a remote procedure call to be stored in a parameter or returned as an operation result, instead of being raised to the client user code as an exception.

The **comm_status** attribute causes communications failures to be reported through a specified parameter. The **fault_status** attribute causes server failures to be reported through a specified parameter. Applying both attributes causes all remote and communications failures to be reported through status. Any local exception caused by an error during marshalling, correctness checking performed by the client stubs, or an error in application routines continues to be returned as an exception.

The **comm_status** and **fault_status** attributes have the following syntax. (See the examples at the end of this section.)

For an operation:

[comm_status | fault_status] *operation_name* (*[parameter_list]*)**;**

For a parameter:

operation_name (**[comm_status | fault_status]** *parameter_name*)**;**

Note: You can apply one of each attribute to the same operation and/or parameter at the same time. Separate the attributes with a comma. (See the example at the end of this section.)

If the status attribute occurs on the operation, the returned value result must be defined as type **error_status_t** in the IDL file. If an error occurs during execution of the operation, the error code is returned as the operation result. If the operation completes successfully, the value returned to the client is the value returned by the manager code.

Note: The **error_status_t** type is equivalent to **unsigned32**, which is the data type used by the RPC runtime for an error status. The status code **error_status_ok** is equivalent to **rpc_s_ok**, which is the RPC runtime success status code.

If the status attribute occurs on a parameter, the parameter name does not have to be defined in the IDL file, although it can be. Note the following:

- If the parameter name is one used in the IDL file, then that parameter must be an output parameter of type **error_status_t**. If the operation completes successfully, the value of this parameter is the value returned by the manager code.

- If the parameter name is different from any name defined within the operation definition in the IDL file, then the IDL compiler creates an extra output parameter of type **error_status_t** in your application code after the last parameter defined in the IDL file. In a successfully completed remote call, the extra parameter has the value **error_status_ok**.

In either case, if an error occurs during the remote call, the error code is returned to the parameter that has the status attribute. (See the *OSF DCE Application Development Reference* for a list of status codes.)

If you define both additional **comm_status** and additional **fault_status** parameters, they are automatically added at the end of the procedure declaration in the order of specification in the ACF.

In the following example, there are three possible uses of the status attributes: as the operation result of **add**, as a parameter of **subtract** as defined in the IDL file, and as an additional parameter of **multiply**.

Example Using the comm_status and fault_status Attributes

ACF

```
[auto_handle] interface math_4
{
[comm_status,fault_status] add();

subtract ([comm_status,fault_status] s);

/*
 * 'sts' is not a parameter in the interface definition of
 * operation 'multiply'. This specifies that the application
 * wants a trailing parameter 'sts' that is of type
 * error_status_t, after the parameters a and b.
 */
multiply ([comm_status] c_sts,[fault_status] f_sts);
}
```

IDL File

```
[uuid(91365000-2D28-11C9-AD5A-08002B0ECEF1)]
interface math_4
{
error_status_t add ([in] double a,
                    [in] double b,
                    [out] double *c);
double subtract ([in] double a,
                [in] double b,
                [out] error_status_t *s);
double multiply ([in] double a,
                 [in] double b);
}
```

18.3.8 The code and nocode Attributes

The **code** and **nocode** attributes allow you to control which operations in the IDL file have client stub code generated for them by the compiler. These attributes affect only the generation of a client stub; they have no effect when generating the server stub.

The **code** and **nocode** attributes have the following syntax. (See the example at the end of this section.)

For an interface:

[code | nocode] interface *interface_name*

For an operation:

[code | nocode] *operation_name* **(***[parameter_list]***);**

When you specify **nocode** as an attribute on an ACF interface, stub code is not generated for the operations in the corresponding IDL interface unless you also specify **code** for the particular operation(s) for which you want stub code generated. Similarly, when you specify **code** (the default) as an attribute on an ACF interface, stub code is generated for the operations in the corresponding IDL interface unless you also specify **nocode** for the particular operations for which you do not want stub code generated.

Do *not* use **nocode** on any of the operations if the compiler is generating only server stub code because it has no effect. Server stubs must always contain generated code for all operations.

In the following example, the IDL compiler generates client stub code for the operations **open**, **read**, and **close**, but not for the operation **write**. An alternative method for specifying the same behavior is to use **[nocode] write( )** in the ACF.

Example Using the code and no_code Attributes

ACF

```
[nocode,auto_handle] interface open_read_close
{
[code] open();
[code] read();
[code] close();
}
```

IDL File

```
[uuid(2166D580-0C69-11CA-811D-08002B111685)]
interface open_read_close
{
void open (...);
void read (...);
void write (...);
void close (...);
}
```

18.3.9 The in_line and out_of_line Attributes

The **in_line** and **out_of_line** attributes control whether marshalling and unmarshalling are performed by inline code or by out-of-line code through a subroutine call. By default, **in_line** code controls marshalling and unmarshalling. You can apply either attribute to an interface or a type. Both attributes apply only to nonscalar types; they do not affect the marshalling of scalars. The **out_of_line** attribute reduces stub size at the expense of reducing execution speed. Note that the procedures necessary for marshalling **out_of_line** types are generated into an auxiliary module. It is necessary to build the client and/or server auxiliary file into your application, along with the client and/or server stub module.

The **in_line** and **out_of_line** attributes have the following syntax. (See the examples at the end of this section.)

For an interface:

[in_line | out_of_line] interface *interface_name*;

For a type:

typedef [in_line | out_of_line] *type_name*;

In the following example, the compiler generates inline marshalling code for data type **my_t**, but out-of-line marshalling code (subroutines) for any other nonscalar types defined in the IDL file.

Example Using the in_line Attribute

ACF

```
[out_of_line] interface move_fields
{
typedef [in_line] my_t;
}
```

IDL File

```
[uuid (2F74E680-2D26-11C9-880E-08002B0ECEF1)]
interface move_fields
{
typedef struct
    {
    long   my_val;
    float  my_float;
    double my_double;
    } my_t;
}
```

In the following example, code space is minimized by having **dir_t** marshalled through a procedure call rather than duplicating the marshalling code in the stub routines for **add, lookup**, and **delete**.

Example Using the out_of_line Attribute

ACF

```
[auto_handle] interface phonedir
{
/*
 * The interface uses dir_t in several places. Save code
 * space that is generated for the stub by making it
 * [out_of_line].
 */
typedef [out_of_line] dir_t;
}
```

IDL File

```
[uuid(06A12100-2D26-11C9-AA24-08002B0ECEF1)]
interface phonedir
{
typedef struct
    {
    short int  area_code;
    long int   phone_num;
    char       last_name[20];
    char       first_name[15];
    char       city[20];
    } dir_t;
void add ([in] dir_t *info);
void lookup ([in] char city[20],
             [in] char last_name[20],
             [in] char first_name[15],
             [out] dir_t *info);
void delete ([in] dir_t *info);
}
```

18.3.10 The represent_as Attribute

This attribute associates a local data type that your application code uses with a data type defined in the IDL file. Use of the **represent_as** attribute means that during marshalling and unmarshalling, conversions occur between the data type used by the application code, and the data type specified in the IDL.

The **represent_as** attribute has the following syntax. (See the example at the end of this section.)

typedef [represent_as (*local_type_name***)]** *net_type_name***;**

The *local_type_name* is the local data type that the application code uses. You can define it in the IDL file or in an application header file. If you do not define it in the IDL file, use the **include** statement in the ACF to make its definition available to the stubs.

The *net_type_name* is the data type that is defined in the IDL file.

If you use the **represent_as** attribute, you must write routines that perform the conversions between the local and network types, and routines that release the memory storage used to hold the converted data. The conversion routines are part of your application code.

The suffix for the routine names, the function of each, and where they are used (client or server) appear in the following list:

- **_from_local**( ): Allocates storage instance of the network type and converts from the local type to the network type (used for client and server).
- **_to_local**( ): Converts from the network type to the local type (used for client and server).
- **_free_inst**( ): Frees storage instance used for the network type (used by client and server).
- **_free_local**( ): Frees storage used by the server for the local type (used in server). This routine frees any object pointed to by its argument, but does not attempt to free the argument itself.

Suppose that the **represent_as** attribute is applied to either the type of a parameter or to a component of a parameter and that the parameter has the **out** or **in,out** attribute. Then, the **_free_local**( ) routine will be called automatically for the data item that has the type to which the **represent_as** attribute was applied.

Suppose that the **represent_as** attribute is applied to the type of a parameter and that the parameter has only the **in** attribute. Then, the **_free_local**( ) routine will be called automatically.

Finally, suppose that the **represent_as** attribute is applied to the type of a component of a parameter and that the parameter has only the **in** attribute. Then, the **_free_local**( ) routine will not be called automatically for the component; the manager application code must release any resources that the component uses, possibly by explicitly calling the **_free_local**( ) routine.

Append the suffix of the routine name to the *net_type_name*. The syntax for these routines is as follows:

void *net_type_name*_**from_local** (
(*local_type_name* *),
(*net_type_name* **))

void *net_type_name*_**to_local** (
(*net_type_name* *),
(*local_type_name* *))

void *net_type_name*_**free_inst** ((*net_type_name* *))

void *net_type_name*_**free_local** ((*local_type_name* *))

Example Using the represent_as Attribute

ACF

```
[auto_handle] interface phonedir
{
/*
 * You must specify an included file that contains the
 * definition of my_dir_t.
 */
include "user_types";

/*
 * The application code wants to pass data type my_dir_t
 * rather than dir_t. The [represent_as] clause allows
 * this, and you must supply routines to convert dir_t
 * to/from my_dir_t.
 */
typedef [represent_as(my_dir_t)] dir_t;
}
```

IDL File

```
[uuid(06A12100-2D26-11C9-AA24-08002B0ECEF1)]
interface phonedir
{
typedef struct
    {
    short int  area_code;
    long int   phone_num;
    char       last_name[20];
    char       first_name[15];
    char       city[20];
    } dir_t;
void add ([in] dir_t *info);
void lookup ([in] char city[20],
             [in] char last_name[20],
             [in] char first_name[15],
             [out] dir_t *info);
void delete ([in] dir_t *info);
}
```

18.3.11 The enable_allocate Attribute

The **enable_allocate** attribute on an operation causes the server stub to initialize the **rpc_ss_allocate**( ) routine. The **rpc_ss_allocate**( ) routine requires initialization of its environment before it can be called. The server stub automatically initializes (enables) **rpc_ss_allocate**( ) if the operation uses either full pointers, or a type with the **represent_as** attribute. If the operation does not meet either of these conditions, but the manager application code needs to make use of the **rpc_ss_allocate**( ) and **rpc_ss_free**( ) routines, then use the **enable_allocate** attribute to force the stub code to enable.

The **enable_allocate** attribute has the following syntax.

For an operation:

[enable_allocate] *operation_name* (*[parameter_list]*);

Example Using the enable_allocate Attribute

ACF

```
[auto_handle] interface phonedir
{
[enable_allocate] lookup ();
}
```

IDL File

```
[uuid(06A12100-2D26-11C9-AA24-08002B0ECEF1)]
interface phonedir
{
typedef struct
    {
    short int  area_code;
    long int   phone_num;
    char       last_name[20];
    char       first_name[15];
    char       city[20];
    } dir_t;
void add ([in] dir_t *info);
void lookup ([in] char city[20],
             [in] char last_name[20],
             [in] char first_name[15],
             [out] dir_t *info);
void delete ([in] dir_t *info);
}
```

18.3.12 The heap Attribute

This attribute specifies that the server stub's copy of a parameter or of all parameters of a specified type is allocated in heap memory, rather than on the stack.

The **heap** attribute has the following syntax. (See the example at the end of this section.)

For a type:

typedef [heap] *type_name*;

For a parameter:

operation_name (**[heap]** *parameter_name*);

Any identifier occurring as a parameter name within an operation declaration in the ACF must also be a parameter name within the corresponding operation declaration in the IDL.

Example Using the heap Attribute

ACF

```
[auto_handle] interface galaxies
{
typedef [heap] big_array;
}
```

IDL File

```
[uuid(E61DE280-0D0B-11CA-6145-08002B111685)]
interface galaxies
{
typedef long big_array[1000];
}
```

18.4 Summary of Attributes

Table 18-1 lists the attributes available for use in the Attribute Configuration File and where in the file the attribute can be used.

Table 18–1. Summary of the ACF Attributes

Attribute	Where Used
auto_handle	Interface header
code	Interface header, operation
comm_status	Operation, parameter
enable_allocate	Operation
explicit_handle	Interface header, operation
fault_status	Operation, parameter
heap	Type, parameter
implicit_handle	Interface header
in_line	Interface header, type
nocode	Interface header, operation
out_of_line	Interface header, type
represent_as	Type

Part 3C

Supplemental Information

Part 3C contains the following:

- Summary lists of the RPC stub-support and runtime routines
- The language grammar synopses of the Interface Definition Language and the Attribute Configuration Language
- A discussion of the issues involved in using NCS Version 1 applications in a DCE RPC environment

Chapter 19

Summary of Runtime Routines

This chapter presents summaries of the DCE RPC runtime routines.

19.1 Summary of RPC Stub-Support Routines

The RPC stub-support routines are shown in the following list.

- **rpc_ss_allocate( )**: Allocates memory within the RPC stub memory management scheme (usually server, possibly client).
- **rpc_ss_client_free( )**: Frees memory returned from a client stub (usually server, possibly client).
- **rpc_ss_destroy_client_context( )**: Reclaims the client memory resources for the context handle, and sets the context handle to **NULL** (client).
- **rpc_ss_disable_allocate( )**: Releases resources and allocated memory (client).
- **rpc_ss_enable_allocate( )**: Enables the allocation of memory by the **rpc_ss_allocate( )** routine when not in manager code (client).

- **rpc_ss_free( )**: Frees memory allocated by the **rpc_ss_allocate( )** routine (usually server, possibly client).
- **rpc_ss_thread_handle( )**: Gets a thread handle for the manager code before it spawns additional threads, or for the client code when it becomes a server (usually server, possibly client).
- **rpc_ss_register_auth_info( )**: Registers authentication and authorization information for an interface (client).
- **rpc_ss_set_client_alloc_free( )**: Sets the memory allocation and freeing mechanism used by the client stubs, thereby overriding the default routines the client stub uses to manage memory for pointed-to nodes (client).
- **rpc_ss_set_thread_handle( )**: Sets the thread handle for either a newly created spawned thread or for a server that was formerly a client and is ready to be a client again (usually server, possibly client).
- **rpc_ss_swap_client_alloc_free( )**: Exchanges the current memory allocation and freeing mechanism used by the client stubs with one supplied by the client (client).

19.2 Summary of RPC Runtime Routines

The RPC nonstub support routines are shown in the following list.

- **dce_error_inq_text( )**: Returns the message text for a DCE component status code (client, server, or management).
- **rpc_binding_copy( )**: Returns a copy of a binding handle (client or server).
- **rpc_binding_free( )**: Releases binding handle resources (client or server).
- **rpc_binding_from_string_binding( )**: Returns a binding handle from a string representation of a binding handle (client or management).
- **rpc_binding_inq_auth_client( )**: Returns authentication and authorization information from the binding handle for an authenticated client (server).

- **rpc_binding_inq_auth_info( )**: Returns authentication and authorization information from a server binding handle (client).
- **rpc_binding_inq_object( )**: Returns the object UUID from a binding handle (client or server).
- **rpc_binding_reset( )**: Resets a server binding handle so the host remains specified, but the server instance on that host is unspecified (client or management).
- **rpc_binding_server_from_client( )**: Converts a client binding handle to a server binding handle (server).
- **rpc_binding_set_auth_info( )**: Sets authentication and authorization information into a server binding handle (client).
- **rpc_binding_set_object( )**: Sets the object UUID value into a server binding handle (client).
- **rpc_binding_to_string_binding( )**: Returns a string representation of a binding handle (client, server, or management).
- **rpc_binding_vector_free( )**: Frees the memory used to store a vector and binding handles (client or server).
- **rpc_ep_register( )**: Adds to, or replaces, server address information in the local endpoint map (server).
- **rpc_ep_register_no_replace( )**: Adds to server address information in the local endpoint map (server).
- **rpc_ep_resolve_binding( )**: Resolves a partially bound server binding handle into a fully bound server binding handle (client or management).
- **rpc_ep_unregister( )**: Removes server address information from the local endpoint map (server).
- **rpc_if_id_vector_free( )**: Frees a vector and the interface identifier structures it contains (client, server, or management).
- **rpc_if_inq_id( )**: Returns the interface identifier for an interface specification (client or server).
- **rpc_mgmt_ep_elt_inq_begin( )**: Creates an inquiry context for viewing the elements in a local or remote endpoint map (management).
- **rpc_mgmt_ep_elt_inq_done( )**: Deletes the inquiry context for viewing the elements in a local or remote endpoint map (management).

- **rpc_mgmt_ep_elt_inq_next( )**: Returns one element at a time from a local or remote endpoint map (management).
- **rpc_mgmt_ep_unregister( )**: Removes server address information from a local or remote endpoint map (management).
- **rpc_mgmt_inq_com_timeout( )**: Returns the communications time-out value in a binding handle (client).
- **rpc_mgmt_inq_dflt_protect_level( )**: Returns the default protection level for an authentication service (client or server).
- **rpc_mgmt_inq_if_ids( )**: Returns a vector of interface identifiers of the interfaces a server offers (client, server, or management).
- **rpc_mgmt_inq_server_princ_name( )**: Returns a server's principal name (client, server, or management).
- **rpc_mgmt_inq_stats( )**: Returns RPC runtime statistics (client, server, or management).
- **rpc_mgmt_is_server_listening( )**: Tells whether a server is listening for remote procedure calls (client, server, or management).
- **rpc_mgmt_set_authorization_fn( )**: Establishes an authorization function for processing remote calls to a server's management routines (server).
- **rpc_mgmt_set_cancel_timeout( )**: Sets the lower bound on the time to wait before timing out after forwarding a cancel (client).
- **rpc_mgmt_set_com_timeout( )**: Sets the communications time-out value in a binding handle (client).
- **rpc_mgmt_set_server_stack_size( )**: Specifies the stack size for each server thread (server).
- **rpc_mgmt_stats_vector_free( )**: Frees a statistics vector (client, server, or management).
- **rpc_mgmt_stop_server_listening( )**: Tells a server to stop listening for remote procedure calls (client, server, or management).
- **rpc_network_inq_protseqs( )**: Returns all protocol sequences supported by both the RPC runtime and the operating system (client or server).

- **rpc_network_is_protseq_valid( )**: Tells whether the specified protocol sequence is supported by both the RPC runtime and the operating system (client or server).
- **rpc_ns_binding_export( )**: Establishes a directory service entry with binding handles or object UUIDs for a server (server).
- **rpc_ns_binding_import_begin( )**: Creates an import context for an interface and an object in the namespace (client).
- **rpc_ns_binding_import_done( )**: Deletes the import context for searching the namespace (client).
- **rpc_ns_binding_import_next( )**: Returns a binding handle of a compatible server (if found) from the namespace (client).
- **rpc_ns_binding_inq_entry_name( )**: Returns the name of an entry in the namespace from which the server binding handle came (client).
- **rpc_ns_binding_lookup_begin( )**: Creates a lookup context for an interface and an object in the namespace (client).
- **rpc_ns_binding_lookup_done( )**: Deletes the lookup context for searching the namespace (client).
- **rpc_ns_binding_lookup_next( )**: Returns a list of binding handles of one or more compatible servers (if found) from the namespace (client).
- **rpc_ns_binding_select( )**: Returns a binding handle from a list of compatible server binding handles (client).
- **rpc_ns_binding_unexport( )**: Removes the binding handles for an interface, or object UUIDs, from an entry in the namespace (server).
- **rpc_ns_entry_expand_name( )**: Expands the name of a directory service entry (client, server, or management).
- **rpc_ns_entry_object_inq_begin( )**: Creates an inquiry context for viewing the objects of an entry in the namespace (client, server, or management).
- **rpc_ns_entry_object_inq_done( )**: Deletes the inquiry context for viewing the objects of an entry in the namespace (client, server, or management).
- **rpc_ns_entry_object_inq_next( )**: Returns one object at a time from an entry in the namespace (client, server, or management).

- **rpc_ns_group_delete( )**: Deletes a group attribute (client, server, or management).
- **rpc_ns_group_mbr_add( )**: Adds an entry name to a group; if necessary, creates the entry (client, server, or management).
- **rpc_ns_group_mbr_inq_begin( )**: Creates an inquiry context for viewing group members (client, server, or management).
- **rpc_ns_group_mbr_inq_done( )**: Deletes the inquiry context for a group (client, server, or management).
- **rpc_ns_group_mbr_inq_next( )**: Returns one member name at a time from a group (client, server, or management).
- **rpc_ns_group_mbr_remove( )**: Removes an entry name from a group (client, server, or management).
- **rpc_ns_mgmt_binding_unexport( )**: Removes multiple binding handles, or object UUIDs, from an entry in the namespace (management).
- **rpc_ns_mgmt_entry_create( )**: Creates an entry in the namespace (management).
- **rpc_ns_mgmt_entry_delete( )**: Deletes an entry from the namespace (management).
- **rpc_ns_mgmt_entry_inq_if_ids( )**: Returns the list of interfaces exported to an entry in the namespace (client, server, or management).
- **rpc_ns_mgmt_handle_set_exp_age( )**: Sets a handle's expiration age for local copies of directory service data (client, server, or management).
- **rpc_ns_mgmt_inq_exp_age( )**: Returns the application's global expiration age for local copies of directory service data (client, server, or management).
- **rpc_ns_mgmt_set_exp_age( )**: Modifies the application's global expiration age for local copies of directory service data (client, server, or management).
- **rpc_ns_profile_delete( )**: Deletes a profile attribute (client, server, or management).
- **rpc_ns_profile_elt_add( )**: Adds an element to a profile. If necessary, creates the entry (client, server, or management).

- **rpc_ns_profile_elt_inq_begin( )**: Creates an inquiry context for viewing the elements in a profile (client, server, or management).
- **rpc_ns_profile_elt_inq_done( )**: Deletes the inquiry context for a profile (client, server, or management).
- **rpc_ns_profile_elt_inq_next( )**: Returns one element at a time from a profile (client, server, or management).
- **rpc_ns_profile_elt_remove( )**: Removes an element from a profile (client, server, or management).
- **rpc_ns_set_authn( )**: Turns authentication on and off for RPC directory service routines.
- **rpc_object_inq_type( )**: Returns the type of an object (server).
- **rpc_object_set_inq_fn( )**: Registers an object inquiry function (server).
- **rpc_object_set_type( )**: Assigns the type of an object (server).
- **rpc_protseq_vector_free( )**: Frees the memory used by a vector and its protocol sequences (client or server).
- **rpc_server_inq_bindings( )**: Returns binding handles for communications with a server (server).
- **rpc_server_inq_if( )**: Returns the manager entry point vector registered for an interface (server).
- **rpc_server_listen( )**: Tells the RPC runtime to listen for remote procedure calls (server).
- **rpc_server_register_auth_info( )**: Registers authentication information with the RPC runtime (server).
- **rpc_server_register_if( )**: Registers an interface with the RPC runtime (server).
- **rpc_server_unregister_if( )**: Unregisters an interface from the RPC runtime (server).
- **rpc_server_use_all_protseqs( )**: Tells the RPC runtime to use all supported protocol sequences for receiving remote procedure calls (server).
- **rpc_server_use_all_protseqs_if( )**: Tells the RPC runtime to use all the protocol sequences and endpoints specified in the interface specification for receiving remote procedure calls (server).

- **rpc_server_use_protseq( )**: Tells the RPC runtime to use the specified protocol sequence for receiving remote procedure calls (server).
- **rpc_server_use_protseq_ep( )**: Tells the RPC runtime to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls (server).
- **rpc_server_use_protseq_if( )**: Tells the RPC runtime to use the specified protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls (server).
- **rpc_string_binding_compose( )**: Combines the components of a string binding into a string binding (client or server).
- **rpc_string_binding_parse( )**: Returns, as separate strings, the components of a string binding (client or server).
- **rpc_string_free( )**: Frees a character string allocated by the runtime (client, server, or management).
- **uuid_compare( )**: Compares two UUIDs and determines their order (client, server, or management).
- **uuid_create( )**: Creates a new UUID (client, server, or management).
- **uuid_create_nil( )**: Creates a nil UUID (client, server, or management).
- **uuid_equal( )**: Determines if two UUIDs are equal (client, server, or management).
- **uuid_from_string( )**: Converts a string UUID to its binary representation (client, server, or management).
- **uuid_hash( )**: Creates a hash value for a UUID (client, server, or management).
- **uuid_is_nil( )**: Determines if a UUID is nil (client, server, or management).
- **uuid_to_string( )**: Converts a UUID from a binary representation to a string representation (client, server, or management).

Chapter 20

Language Grammar Synopsis

This chapter summarizes the syntax for both IDL and ACF. In each case, the syntax is presented in extended Backus-Naur Format (BNF) notation.

20.1 Interface Definition Language

This section lists the syntax for the Interface Definition Language (IDL).

```
<interface> ::= <interface_header> "{" <interface_body> "}"

<interface_header> ::=
 "[" <interface_attributes> "]" "interface" <identifier>

<interface_attributes> ::=
 <interface_attribute> [ "," <interface_attribute> ] ...

<interface_attribute> ::= "uuid" "(" <uuid_rep> ")"
 | "version" "(" <major> [ "." <minor> ] ")"
 | "endpoint" "(" <endpoint_spec> [ "," <endpoint_spec> ] ... ")"
 | "pointer_default" "(" <pointer_attribute> ")"
 | "local"

<major> ::= <integer>

<minor> ::= <integer>

<endpoint_spec> ::=
 """ <family_string> ":" "[" <endpoint_string> "]" """

<family_string> ::= <identifier>

<endpoint_string> ::= <identifier>

<interface_body> ::= [ <import> ] ... [ <export> ] ...

<export> ::= <const_declaration> ";"
 | <type_declaration> ";"
 | <op_declaration> ";"

<import> ::= import <import_files> ";"

<import_files> ::= <filename> [ "," <filename> ] ... ";"

<filename> ::= """ <character> ... """

<const_declaration> ::=
 "const" <const_type_spec> <identifier> "=" <const_exp>

<const_type_spec> ::=
 <integer_type> | "char" | "char" "*" | "boolean" | "void" "*"

<const_exp> ::=
 <integer_const_exp> | <character_const> | <string_const>
 | <identifier> | "TRUE" | "FALSE" | "NULL"

<integer_const_exp> ::= <conditional_exp>

<conditional_exp> ::= <logical_or_exp>
 | <logical_or_exp> "?" <integer_const_exp> ":" <conditional_exp>

<logical_or_exp> ::= <logical_and_exp>
```

```
 | <logical_or_exp> "||" <logical_and_exp>

<logical_and_exp> ::= <inclusive_or_exp>
 | <logical_and_exp> "&&" <inclusive_or_exp>

<inclusive_or_exp> ::= <exclusive_or_exp>
 | <inclusive_or_exp> "|" <exclusive_or_exp>

<exclusive_or_exp> ::= <and_exp>
 | <and_exp> "^" <and_exp>

<and_exp> ::= <equality_exp>
 | <and_exp> "&" <equality_exp>

<equality_exp> ::= <relational_exp>
 | <equality_exp> "==" <relational_exp>
 | <equality_exp> "!=" <relational_exp>

<relational_exp> ::= <shift_exp>
 | <relational_exp> "<" <shift_exp>
 | <relational_exp> ">" <shift_exp>
 | <relational_exp> "<=" <shift_exp>
 | <relational_exp> ">=" <shift_exp>

<shift_exp> ::= <additive_exp>
 | <shift_exp> "<<" <additive_exp>
 | <shift_exp> ">>" <additive_exp>

<additive_exp> ::= <multiplicative_exp>
 | <additive_exp> "+" <multiplicative_exp>
 | <additive_exp> "-" <multiplicative_exp>

<multiplicative_exp> ::= <unary_exp>
 | <multiplicative_exp> "*" <unary_exp>
 | <multiplicative_exp> "/" <unary_exp>
 | <multiplicative_exp> "%" <unary_exp>

<unary_exp> ::= <primary_exp>
 | "+" <primary_exp>
 | "-" <primary_exp>
 | "~" <primary_exp>
 | "!" <primary_exp>

<primary_exp> ::= <integer_literal>
 | <identifier>

<character_const> ::= "'" <character> "'"

<string_const> ::= """ [ <character> ] ... """

<type_declaration> ::=
 "typedef" [ <type_attributes> ] <type_spec> <declarators>

<type_spec> ::= <simple_type_spec>
 | <constructed_type_spec>

<simple_type_spec> ::= <base_type_spec>
 | <predefined_type_spec>
 | <identifier>

<declarators> ::= <declarator> [ "," <declarator> ] ...

<declarator> ::= <simple_declarator>
 | <complex_declarator>

<simple_declarator> ::= <identifier>

<complex_declarator> ::= <array_declarator>
 | <function_ptr_declarator>
 | <ptr_declarator>

<tagged_declarator> ::= <tagged_struct_declarator>
 | <tagged_union_declarator>

<base_type_spec> ::= <integer_type>
 | <floating_type>
 | <char_type>
 | <boolean_type>
 | <byte_type>
 | <void_type>
 | <handle_type>
```

```
<floating_type> ::= "float" | "double"

<integer_type> ::= <signed_int> | <unsigned_int>

<signed_int> ::= <int_size> [ "int" ]

<unsigned_int> ::= <int_size> "unsigned" [ "int" ]
 | "unsigned" <int_size> [ "int" ]

<int_size> ::= "hyper" | "long" | "short" | "small"

<char_type> ::= [ "unsigned" ] "char"

<boolean_type> ::= "boolean"

<byte_type> ::= "byte"

<void_type> ::= "void"

<handle_type> ::= "handle_t"

<constructed_type_spec> ::= <struct_type>
 | <union_type>
 | <tagged_declarator>
 | <enumeration_type>
 | <pipe_type>

<tagged_struct_declarator> ::= "struct" <tag>
 | <tagged_struct>

<struct_type> ::= "struct" "{" <member_list> "}"

<tagged_struct> ::= "struct" <tag> "{" <member_list> "}"

<tag> ::= <identifier>

<member_list> ::= <member> [ <member> ] ...

<member> ::= <field_declarator> ";"

<field_declarator> ::= [ <field_attribute_list> ]
```

```
 <type_spec> <declarators>

<field_attribute_list> ::= "[" <field_attribute> [ ","
 <field_attribute>] ... "]"

<tagged_union_declarator> ::= "union" <tag>
 | <tagged_union>

<union_type> ::= "union" <union_switch> "{" <union_body> "}"
 | "union" "{" <union_body_n_e> "}"

<union_switch> ::= "switch" "(" <switch_type_spec> <identifier> ")"
 [ <union_name> ]

<switch_type_spec> ::= <integer_type>
 | <char_type>
 | <boolean_type>
 | <enumeration_type>

<tagged_union> ::= "union" <tag> <union_switch> "{ <union_body> "}"
 | "union" <tag> "{ <union_body_n_e> "}"

<union_name> ::= <identifier>

<union_body> ::= <union_case> [ <union_case> ] ... [ <default_case> ]

<union_body_ne> ::= <union_case_n_e> [ <union_case_n_e> ] ...
 [ <default_case_n_e> ]

<union_case> ::= <union_case_label> [ <union_case_label> ] ...  <union_arm>

<union_case_n_e> ::= <union_case_label_n_e> <union_arm>

<union_case_label> ::= "case" <const_exp> ":"

<union_case_label_n_e> ::= "[" "case" "(" <const_exp> [ ","
 <const_exp> ] ... ")" "]"

<default_case> ::= "default" ":" <union_arm>

<default_case_n_e> ::= "[" "default" "]" <union_arm>
```

```
<union_arm> ::= [ <field_declarator> ] ";"

<union_type_switch_attr> ::= "switch_type" "(" <switch_type_spec> ")"

<union_instance_switch_attr> ::= "switch_is" "(" <attr_var> ")"

<enumeration_type> ::=
 "enum" "{" <identifier> [ "," <identifier> ] ... "}"

<pipe_type> ::= "pipe" <type_spec> <pipe_declarators>

<array_declarator> ::= <identifier> <array_bounds_list>

<array_bounds_list> ::= <array_bounds_declarator>
 [ <array_bounds_declarator> ] ...

<array_bounds_declarator> ::= "[" [ <array_bound> ] "]"
 | "[" <array_bounds_pair> "]"

<array_bounds_pair> ::= <array_bound> ".." <array_bound>

<array_bound> ::= "*"
 | <integer_literal>
 | <identifier>

<type_attribute> ::= "transmit_as" "(" <xmit_type> ")"
 | "handle"
 | "align" "(" <int_size> ")"
 | "v1_struct"
 | "v1_enum"
 | <usage_attribute>
 | <union_type_switch_attr>
 | <ptr_attr>

<usage_attribute> ::= "string"
 | "v1_string"
 | "context_handle"

<xmit_type> ::= <simple_type_spec>

<field_attribute> ::= "first_is" "(" <attr_var_list> ")"
```

```
 | "last_is" "(" <attr_var_list> ")"
 | "length_is" "(" <attr_var_list> ")"
 | "max_is" "(" <attr_var_list> ")"
 | "size_is" "(" <attr_var_list> ")"
 | "v1_array"
 | <usage_attribute>
 | <union_instance_switch_attr>
 | "ignore"
 | <ptr_attr>

<attr_var_list> ::= <attr_var> [ "," <attr_var> ] ...

<attr_var> ::= [ ["*"]<identifier> ]

<ptr_declarator> ::= "*"<identifier>

<ptr_attr> ::= "ref"
 | "unique"
 | "full"

<op_declarator> ::= [ <operation_attributes> ]
 <simple_type_spec> <identifier> <parameter_declarators>

<operation_attributes> ::=  "[" <operation_attribute>
 [ "," <operation_attribute> ] ... "]"

<operation_attribute> ::= "idempotent"
 | "broadcast"
 | "maybe"
 | <usage_attribute>
 | <ptr_attr>

<param_declarators> ::= "(" "void" ")"
 | "(" [ <param_declarator> [ "," <param_declarator> ] ... ] ")"

<param_declarator> ::= <param_attributes> <type_spec> <declarator>

<param_attributes> ::=
 "[" <param_attribute> [ "," <param_attribute> ] ... "]"

<param_attribute> ::= <directional_attribute>
```

```
 | <field_attribute>

<directional_attribute> ::= "in" [ "(" "shape" ")" ]
 | "out" [ "(" "shape" ")" ]

<function_ptr_declarator> ::= <simple_type_spec>
 "(" "*"<identifier> ")" <param_declarators>

<predefined_type_spec> ::= "error_status_t"
 | <international_character_type>

<international_character_type> ::= ISO_LATIN_1
 | ISO_MULTI_LINGUAL
 | ISO_UCS

<pipe_declarators> ::= <pipe_declarator> [ "," <pipe_declarator> ] ...

<pipe_declarator> ::= <simple_declarator>
 | <ptr_declarator>
```

20.2 Attribute Configuration Language

The syntax description in this section uses an extended Backus-Naur Form (BNF) to represent ACF grammar. The following lists the symbols used in this section and their meanings:

```
<acf_interface> ::=
   <acf_interface_header> "{" <acf_interface_body> "}"

<acf_interface_header> ::=
   [ <acf_interface_attr_list> ] "interface" <idl_interface_name>

<acf_interface_attr_list> ::= "[" <acf_interface_attrs> "]"

<acf_interface_attrs> ::=
   <acf_interface_attr> [ "," <acf_interface_attr> ] ...
```

```
<acf_interface_attr> ::= <acf_code_attr>
   | <acf_nocode_attr>
   | <acf_in_line_attr>
   | <acf_out_of_line_attr>
   | <acf_auto_handle_attr>
   | <acf_explicit_handle_attr>
   | <acf_implicit_handle_attr>

<acf_auto_handle_attr> ::= "auto_handle"

<acf_explicit_handle_attr> ::= "explicit_handle"

<acf_implicit_handle_attr> ::=
   "implicit_handle" "(" <acf_named_type> <Identifier> ")"

<acf_interface_name> ::= <Identifier>

<acf_interface_body> ::= [ <acf_body_element> ] ...

<acf_body_element> ::= <acf_include> ";"
   | <acf_type_declaration> ";"
   | <acf_operation> ";"

<acf_include> ::= "include" <acf_include_list>

<acf_include_list> ::= <acf_include_name> [ "," <acf_include_name> ] ...

<acf_include_name> ::= """ <filename> """

<acf_type_declaration> ::= typedef [ <acf_type_attr_list> ] <acf_named_type>

<acf_named_type> ::= <Identifier>

<acf_type_attr_list> ::= "[" <acf_type_attrs> "]"

<acf_type_attrs> ::= <acf_type_attr> [ "," <acf_type_attr> ] ...

<acf_type_attr> ::= <acf_represent_attr>
   | <acf_in_line_attr>
   | <acf_out_of_line_attr>
   | <acf_heap_attr>
```

```
<acf_represent_attr> ::= "represent_as" "(" <acf_repr_type> ")"

<acf_repr_type> ::= <acf_named_type>

<acf_operation> ::= [ <acf_op_attr_list> ] <Identifier> "("
   [ <acf_parameters> ] ")"

<acf_op_attr_list> ::= "[" <acf_op_attrs> "]"

<acf_op_attrs> ::= <acf_op_attr> [ "," <acf_op_attr> ] ...

<acf_op_attr> ::= <acf_explicit_handle_attr>
   | <acf_comm_status_attr>
   | <acf_fault_status_attr>
   | <acf_code_attr>
   | <acf_nocode_attr>
   | <acf_enable_allocate_attr>

<acf_parameters> ::= <acf_parameter> [ "," <acf_parameter> ] ...

<acf_parameter> ::= [ <acf_param_attr_list> ] <Identifier>

<acf_param_attr_list> ::= "[" <acf_param_attrs> "]"

<acf_param_attrs> ::= <acf_param_attr> [ "," <acf_param_attr> ] ...

<acf_param_attr> ::= <acf_comm_status_attr>
   | <acf_fault_status_attr>
   | <acf_heap_attr>

<acf_code_attr> ::= "code"

<acf_nocode_attr> ::= "nocode"

<acf_in_line_attr> ::= "in_line"

<acf_out_of_line_attr> ::= "out_of_line"

<acf_comm_status_attr> ::= "comm_status"

<acf_fault_status_attr> ::= "fault_status"
```

```
<acf_enable_allocate_attr> ::= "enable_allocate"

<acf_heap_attr> ::= "heap"
```

Chapter 21

Using NCS in a DCE RPC Environment

Two alternatives exist for using NCS Version 1 applications (which include DECrpc Version 1.0 applications) in a DCE RPC Version 1.0 environment.

- Use the compatibility features of DCE RPC.
- Migrate your existing application to a DCE RPC application.

Note: NCS Version 1 compatibility is provided only for transitional purposes and is available for only a limited number of DCE update releases. For new applications, you must use DCE RPC.

21.1 Using Compatibility Features

For DCE RPC, compatibility is the ability of NCS Version 1 applications to operate in a DCE RPC environment. If you do not need to enhance an existing application, compatibility is the better short-term alternative. However, in a future version of DCE RPC the compatibility features will disappear; therefore, if you have a stable NCS application that you want to continue using, you will have to migrate it to DCE RPC.

Compatibility encompasses the following kinds of configurations:

- An NCS Version 1 application can run on a DCE RPC system.

  NCS Version 1 servers and clients can run in the DCE RPC runtime environment. The DCE RPC runtime contains a compatibility library that enables NCS Version 1 applications to use the DCE RPC runtime. Porting an NCS Version 1 application to the DCE RPC environment involves only linking the NCS object code to the DCE RPC runtime library. DCE RPC provides the NCS Local and Global Location Brokers for NCS Version 1 applications that use the DCE RPC runtime.

  Alternatively, NCS Version 1 and DCE RPC software (the RPC runtimes and so on) can coexist on the same system and run independently.

- An NCS client can call a DCE RPC server, or vice versa.

  The data types, operations, and so on, of an NIDL (Network Interface Definition Language) interface can be reproduced in an IDL interface, making the IDL and NIDL interfaces compatible. NCS Version 1 and DCE RPC Version 1.0 applications that use compatible NIDL and IDL interfaces can interoperate.

  To create a compatible IDL interface for an NIDL interface, run the NIDL-to-IDL translator on an NIDL interface definition, which produces a compatible IDL interface definition file. Compile the new IDL file with the DCE IDL compiler, which creates an IDL-based stub that is compatible with NIDL-based stubs for the interface.

  Note: For information on the NIDL-to-IDL translator, see Section 21.2.

- An NCS Version 1 application running on an NCS system can interoperate with an application using a compatible interface on a DCE RPC system.

  For the subset of DCE RPC features that correspond to NCS Version 1 features, DCE RPC Version 1.0 provides wire interoperability. Wire interoperability ensures that NIDL data types are correctly transmitted between NCS and DCE RPC systems. An NCS Version 1 client running on an NCS system should be able to interoperate with a compatible NCS Version 1 or DCE RPC Version 1.0 server running on a DCE RPC system. Conversely, an NCS Version 1 or DCE RPC Version 1.0 client running on a DCE RPC system should be able to interoperate with a compatible NCS server running on an NCS system.

DCE RPC and Version 1.5.1 NCS use different formats and string representations for UUIDs. As a compatibility feature, the DCE RPC **uuid_to_string()** routine converts a UUID into the corresponding string representation. For example, if you specify an NCS Version 1.5.1 UUID to the DCE RPC **uuid_to_string()** routine, it returns an NCS Version 1.5.1 string representation of the UUID. In addition, the DCE **uuid_from_string()** routine converts a string representation into the corresponding form of UUID.

21.2 Migrating an Application to DCE RPC

For DCE RPC, migration is upgrading an NCS Version 1 application to a DCE RPC Version 1.0 application. To enhance an application; for example, to use IDL data types or to use a global directory service, migrating an application is worthwhile. Even if enhancements are independent of any extensions to NCS Version 1, migration is preferable to further development effort using NCS.

DCE RPC migration is divided into two steps:

1. Translating the NIDL interface definition into an IDL interface definition

2. Updating the NCS Version 1 runtime calls into DCE RPC runtime calls

The following subsections discuss these two steps.

21.2.1 Translating an Interface Definition from NIDL to IDL

DCE RPC provides a translator to convert interface definitions written in the Network Interface Definition Language (NIDL) syntax to the DCE Interface Definition Language (IDL) syntax. Invoke this translator with the **nidl_to_idl** command. The translator creates an IDL interface definition file and an Attribute Configuration File (ACF), if needed. Compiling the output files from the translator with the DCE IDL compiler generates files that are compatible with the DCE RPC runtime library.

Before you run the translator, you must have an input interface definition file that was compiled successfully by the NIDL compiler. Most NIDL data types translate directly into IDL data types. However, IDL defines enhanced forms of some data types, such as arrays and structures. For these data types, the translator generates IDL migration attributes (the **v1_*** attributes) that tell the DCE IDL compiler to use the NIDL forms of the data types, rather than the corresponding IDL forms.

The translator cannot handle some NIDL features because no direct translation exists. If a feature cannot be translated, the translator issues a warning and you must manually correct the DCE IDL output file. You will also have to make corresponding changes in the application code that uses the feature.

When your code translates without error, the automated translation is complete, and you can then use the DCE IDL compiler to compile the translated output.

For additional information on the NCS migration attributes, see Chapter 17 of this guide.

For information about the **nidl_to_idl** command, filenames, and messages, see the *OSF DCE Application Development Reference*.

21.2.2 Updating Runtime Calls

The DCE RPC environment is more extensive than the NCS environment, and unlike NCS, DCE RPC runtime calls are designed for system and transport independence. Therefore, migrating an application into the DCE RPC environment requires that you understand the DCE RPC runtime operations. You must migrate runtime calls manually.

The correspondence between NCS and DCE RPC runtime routines varies from close correspondence to no correspondence. Many DCE RPC runtime routines lack any corresponding NCS routines; for example, the DCE RPC **rpc_ns...**( ) routines and the **rpc_string_...**( ) routines. Even where direct correspondence to NCS Version 1 routines exists, as for most of the NCS **uuid_$...**( ) routines, the NCS routine names have been modified to conform to the standard DCE routine syntax, which disallows the $ (dollar sign). Many functional sets of NCS routines have been replaced by a different set of DCE RPC routines that perform equivalent functions but only partially correspond. For example, some NCS communications routines, such as the

NCS **rpc_$inq_...()** routines, have counterparts among the DCE RPC communications routines; others, such as the NCS **socket_$...()** routines, lack any DCE counterparts. DCE Threads routines have completely replaced the NCS **pfm_$...()** routines.

Part 4

DCE Directory Service

Chapter 22

DCE Directory Service Overview

This chapter provides an overview of the DCE Directory Service for application programmers. The chapter begins with a description of Part 4 of this guide. It then introduces DCE Directory Service concepts, following which the structure of DCE names and the DCE namespace are described. The chapter then provides an overview of the programming interfaces used to access the DCE Directory Service.

22.1 Introduction to Part 4

Part 4 of this guide describes how application developers can access the DCE Directory Service. From the application programmer's perspective, the Directory Service has three main parts: the DCE Cell Directory Service (CDS), the DCE Global Directory Service (GDS), and the X/Open Directory Service (XDS) and X/Open OSI-Abstract-Data Manipulation (XOM) programming interfaces. This is reflected in the organization of Part 4:

- Part 4A. CDS Application Programming
- Part 4B. GDS Application Programming

- Part 4C. XDS/XOM Supplementary Information

Parts 4A and 4B contain conceptual material on CDS and GDS with descriptions of programming tasks, including the use of programming interfaces. The final chapter in each of these parts (Chapter 24 of Part 4A and Chapter 28 of Part 4B) contains annotated source code for sample applications.

Part 4C consists mostly of tables of values for the data structures used by the XDS and XOM application interfaces, which are the interfaces used to directly access the DCE Directory Service. These chapters supplement the reference pages for the XDS and XOM function calls, which are located in the *OSF DCE Application Development Reference*.

22.1.1 Part 4 Document Usage

Before reading this guide, you should read the *Introduction to OSF DCE*. It contains overviews, along with illustrations, of all the DCE components and of DCE as a whole. Many concepts and details are explained in the *Introduction to OSF DCE* that are necessary to a full understanding of what is described here. Next, read this chapter in its entirety.

Determine whether you will be programming primarily in the CDS namespace or the GDS namespace and read Part 4A or Part 4B accordingly. At this point, you are ready to begin programming and should proceed to Part 4C. The main purpose of Part 4C is to provide a convenient location to look up the details of object values and structures needed when writing code.

If you do not find the information you need either in this guide or in the *OSF DCE Application Development Reference*, see the *OSF DCE Administration Guide* and the *OSF DCE Administration Reference*. For example, information about the DCE Cell Directory Service as a separate component is found in the administration documentation. Although the DCE Security Service is documented in the application development books, some information of interest to programmers (such as adding new principals to the registry database) is also found in the administration books.

22.1.2 Directory Service Tools

Both CDS and GDS have commands that allow system administrators to inspect and alter the contents of the directory. This can be useful when developing applications that access the DCE namespace.

For information on the CDS Control Program (**cdscp**), see the *OSF DCE Administration Guide*. For information on the CDS Browser (**cdsbrowser**), which is a Motif-based utility that allows you to inspect the CDS namespace, see the *OSF DCE User's Guide and Reference*.

For information on the GDS system administration commands, **gdssysadm**, **gdsdirinfo**, **gdsditadm**, and **gdscacheadm**, see the *OSF DCE Administration Guide*.

22.2 Using the DCE Directory Service

The DCE Directory Service can be used in many ways. It is used by the DCE services themselves to support the DCE environment. For example, cells are registered in the global part of the Directory Service, enabling users from different cells to share information and resources.

The DCE Directory Service is also useful to DCE applications. The client and server sides of an application can use the Directory Service to find each other's locations. The Directory Service can also be used to store information that needs to be made available in a globally accessible, well-known place.

For example, one DCE application could be a print service consisting of a client side application that makes requests for jobs to be printed, and a server side that prints jobs on an available printer. The Directory Service could be used as a central place where the print clients could look up the location of a print server. Furthermore, the Directory Service could be used to store information about printers; for example, what type of jobs a printer can accept, whether the printer is currently up or down, and whether it is currently lightly or heavily loaded.

In some ways, a Directory Service can be used in the same way as a file system has traditionally been used; that is, for containing globally accessible information in a well-known place. An example is the use of configuration information stored in files in a UNIX **/etc** directory. However, the Directory Service differs in important ways. It can be replicated so that information is available even if one server goes down. Replicas can be kept automatically up-to-date, so that, unlike multiple copies of a file on different machines, the information in the replicas of the Directory Service can be kept current without manual intervention.

The Directory Service can also provide security for data that is kept in a globally accessible place. It supports Access Control Lists (ACLs) that provide fine-grained control over who is able to read, modify, create, and perform other operations on its data.

As you learn about the DCE Directory Service and how to access it, think about the ways in which your application can best take advantage of the services it provides.

22.3 DCE Directory Service Concepts

This section provides a description of DCE Directory Service concepts that are important to application developers. Concepts that are specific to GDS are covered in more detail in Chapter 29. The following concepts are intended to convey general definitions that are applicable to the DCE Directory Service as a whole rather than specific to a particular Directory Service component. For more detailed definitions, see the Glossary in the *Introduction to OSF DCE.*

- DCE Namespace

  The DCE namespace is the collection of names in a DCE environment. It can be made up of several domains, in which different types of servers own the names in different parts of the namespace (see Section 22.5). Typically, there are two high-level, or global, domains to a DCE namespace: the GDS namespace and the Domain Name Service (DNS) namespace. At the next level is the CDS namespace, with names contained in the cell's CDS Server. A DCE environment always contains a cell namespace, which is implemented by CDS. Parts of the DCE namespace may not be contained in any of the Directory Services; for example, the DFS namespace, also called the filespace, contains the names of files and directories in DFS, and the Security namespace contains principals and groups contained in the Security Server.

The term ''DCE namespace'' is used when referring to names, but not the information associated with them. For example, it would include the name of a printer in the Directory Service, but not its associated location attribute, and it would include the name of a DFS file, but not its contents.

- Cell Namespace

  All of the names found in a single DCE cell comprise the cell's namespace. This includes names managed by the cell's CDS Server and Security Server, names in the cell's DFS if it has one, and any other names that reside within a particular cell.

- Hierarchy

  The DCE namespace is organized into a hierarchy; that is, each name except the global root has a parent node and may itself have child nodes or leaves. The leaves are called objects or entries, and in the CDS and DFS namespace, the nodes are called directories.

- Directory

  The word ''directory'' has two meanings, which can be differentiated by their context. The first is the node of a hierarchy as mentioned in the previous definition. The second is a collection of objects managed by a directory service.

- Directory Service

  A directory service is software that manages names and their associated attributes. A directory service can store information, be queried about information, and be requested to change information. DCE contains two different directory services: CDS and GDS. It also interacts with a third directory service, DNS, which is not part of DCE.

- Junction

  A junction is a point in the DCE namespace where two domains meet. For example, the point where the DFS entries are ''mounted'' into a CDS namespace is a junction. DCE also has junctions between the global directory services and CDS, and between CDS and the Security Service.

- Object

  The word ''object'' can have two meanings, depending on the context. Sometimes it means an entry in a directory service. Sometimes it means a real object that an entry in a directory service describes, such as a printer. In the context of XDS/XOM, the requested data is returned to the application in one or more *interface objects*, which are data structures that the application can manipulate.

- Entry

  An entry is a unit of information in a directory service. It consists of a name and associated attributes. For example, an entry could consist of the name of a printer, its capabilities, and its network address.

  — Class

  In GDS, each entry has a class associated with it. The class determines what type of entry it is and what attributes may be associated with it.

  — Link

  A link is one type of object class. This type of object is a pointer to another object; it is similar to a soft link in a UNIX file system. A CDS link is similar to a GDS alias.

- Attribute

  If an object is like a complex data structure, then its attributes are analogous to the separate member fields within that structure. Some of an object's attributes may be of significance only to the directory service that manages it. With attributes such as these, a directory service implements objects that contain various kinds of data about the directory itself, thus enabling the service to organize the entries into a meaningful structure. For example, directory objects can contain attributes whose values are other directory objects (called child directories or subdirectories) in the directory. Or link objects can contain attributes whose values are the names and internal identifiers of other directory entries, making a link object's entry name an alias of the other object to which its attributes indirectly refer.

  — Type

  Every attribute is characterized as being of a certain type. The attribute is used to hold a certain kind of data, such as a zip code or the name of a cat. Entries can also be classified by type; for entries, the term used is ''class.''

- — Value

  An attribute can have one or more values.

- Object Identifier

  Directory attributes are uniquely identified by Object Identifiers (OIDs), which are administered by the International Organization for Standards (ISO). In GDS, OIDs are also used to identify object classes. When it creates new attribute types, an application is responsible for tagging them with new, properly allocated OIDs (see your Directory Service administrator for OID assignments). In CDS, attribute types are identified by strings, which can be representations of OIDs.

- Name

  A DCE name corresponds to an entry in some service participating in the DCE namespace, usually a directory service (see Section 22.4).

  — Global Name

  A global name is a name that contains a path through one of the global namespaces (GDS or DNS).

  — Local Name

  A local name is a name that uses the cell prefix **/.:** to indicate the cell name and therefore does not have a specific path through a global namespace. The entry for a local name is always contained in the local cell.

- Access Control List

  Access to DCE namespace entries is determined by lists of entities that are attached through the DCE Security Service to both the entries and the objects when they are created. The lists, called Access Control Lists (ACLs), specify the privileges that an entity or group of entities has for the entry the ACL is associated with. The DCE Security Service provides servers with authenticated identification of every entity that contacts them; it is then the server's responsibility to check the ACL attached to the object that the potential client wants to access, and perform or refuse to perform the requested operation on the basis of what it finds there. The ACLs are checked using Security Service library routines.

Objects in the GDS namespace have ACLs associated with them, but they are not DCE Security Service ACLs.

- Replication

  The DCE Directory Service can keep replicas (copies) of its data on different servers. This means that if one server is unavailable, clients can still obtain information from another server.

- Caching

  Both the CDS and GDS components of the DCE Directory Service support caching of data on the client machine. When a client requests a piece of data from the Directory Service for the first time, the information must be obtained over the network from a server. However, the data can then be cached (stored) on the local machine, and subsequent requests for the same data can be satisfied more quickly by looking in the local cache instead of sending a request over the network. Programmers need to be aware of caching because in some cases you will want to bypass the cache to ensure that the data you obtain is as up-to-date as possible.

22.4 Structure of DCE Names

The following subsections describe the structure of the names found in a DCE environment. DCE names can consist of several different parts, which reflect the federated nature of the DCE namespace (see Section 22.5). A DCE name has some combination of the following elements. They must occur in this order, but most elements are optional.

- Prefix
- GDS cell name or DNS cell name
- GDS name or CDS name
- Junction
- Application name

A DCE name can be represented by a string that is a readable description of a specific entry in the DCE namespace. The name is a string consisting of a series of elements separated by / (slashes). The elements are read from left to right. Each consecutive element adds further specificity to the entry being described, until finally one arrives at the rightmost element, which is the simple name of the entry itself. Thus, in appearance DCE names are similar to UNIX filenames.

In the discussion that follows, a DCE name *element* is the single piece of a name string enclosed between a consecutive pair of slashes. For example, in the following string:

/.../C=US/O=OSF/OU=DCE/hosts/abc/self

the substring

O=OSF

is an element, and so is

abc

and the entire name contains (counting the **...** element) a total of seven elements.

In GDS, an element is called a Relative Distinguished Name (RDN) and the entire name is called a Distinguished Name (DN). In the preceding example, the attribute type **O** stands for the Organization type OID, which is 2.5.4.10.

22.4.1 DCE Name Prefixes

The leftmost element of any valid DCE name is a root prefix. The appearance and meaning of each is as follows:

/... This is the *global root*. It signifies that the immediately following elements form the name of a global namespace entry. Usually, the entry's contents consist of binding information for a DCE cell (more specifically, for some CDS server in the cell), and the name of the global entry is the name of the cell.

/.: This is the *cell root*. It is an alias for the global prefix plus the name of the local cell; that is, the cell in which the prefix is being used. It signifies that all of the following elements taken together form the name of a cell namespace entry.

/: This is the *filespace root*. It is an alias for the global prefix, the name of the local cell, and the DFS junction.

DCE also supports a junction into the Security Service namespace, but there is no alias for this junction.

A prefix by itself is a valid DCE name. For example, you can list the contents of the **/.:** directory to see the top-level entries in the CDS namespace, and you can use a file system command to list the contents of the **/:** directory to see the top-level entries in the filespace.

22.4.2 Names of Cells

After the global root prefix, the next section of a DCE name contains the name of the cell in which the object's name resides. The name of a cell can be expressed as either a GDS name or a DNS name, depending on which global directory service (GDS or DNS) the cell is registered in. The following subsections provide examples.

22.4.2.1 GDS Cell Names

GDS elements always consist of a substring in which an abbreviation or acronym in capital letters is followed by an = (equal sign), which is followed by a string value. As you will learn in more detail in Chapter 23, these substrings represent pairs of attribute types and atttribute values.

For example, in the global DCE name

/.../C=DE/O=SNI/OU=DCE/subsys/druecker/docs

the *attribute=value* form of the leftmost elements after the **/...** indicates that the global part of the name is a GDS namespace entry, and that it ends after the **OU=DCE** element; therefore, the rest of the name is in the **/.../C=DE/O=SNI/OU=DCE** cell.

22.4.2.2 DNS Cell Names

If DNS is used as the global directory, a global name has a form like the following:

/.../cs.univ.edu/subsys/printers/docs

where the single element

cs.univ.edu

is the cell name; that is, the cell's name in the DNS namespace. The DNS name consists of up to four domain names (depending on the name assigned to the cell), separated by dots.

22.4.2.3 Discovering Your Local Cell's Name

A DCE cell consists of the machines that are configured into it; each DCE machine belongs to one DCE cell. Your local cell is therefore the cell to which the machine you are using belongs. Depending on the DCE name you are using, you can access your own cell or other (foreign) cells. If the name you are accessing is global, then its cell is explicitly named. If the name begins with the local cell prefix, then you are accessing a name within your local cell. You can find out what cell you are in by calling the **dce_cf_get_cell_name**( ) function.

Using the global directory services, applications can access resources and services in foreign cells; however, applications most frequently use resources from their local cell. If this is not the case, the cell boundaries may not have been well chosen.

22.4.3 CDS Names

After the cell name or cell root prefix, the next part of a DCE name is often a CDS name. For example, in the name

/.../C=DE/O=SNI/OU=DCE/subsys/druecker/docs

the CDS part is

/subsys/druecker/docs

Or in the name

/.../cs.univ.edu/subsys/printers/docs

the CDS part is

/subsys/printers/docs

The following strings show equivalent names using the cell root prefix, assuming that the name is used from within the **/.../C=DE/O=SNI/OU=DCE** and **/.../cs.univ.edu** cells, respectively:

/.:/subsys/druecker/docs
/.:/subsys/printers/docs

22.4.4 GDS Names

Some names fall entirely in the GDS namespace. These names are ‘‘pure’’ X.500 (and therefore GDS) names, since each element consists of a type and an attribute. The entries for these names are contained in a GDS server. The following is an example of a pure GDS name:

/.../C=US/L=Cambridge/CN=Kilroy

22.4.5 Junctions in DCE Names

Some junctions are implied in a DCE name; others can be seen. There is an implied junction between the global prefix and either GDS or DNS. It occurs after the global prefix. The junction between either of the global namespaces and the local cell namespace is also implied. It occurs after the cell name. The junction between the local cell namespace and either the DFS namespace or the security namespace is shown by the symbol **/fs** or **/sec**, respectively. The following are examples of visible junctions in DCE names:

/.:/fs/usr/snowpaws
/.../dce.osf.org/sec/principal/ziggy

22.4.6 Application Names

The part of a DCE name that occurs after a junction into a DCE application is the application name. DFS and Security names are the currently supported examples; in the future, application programmers may also be able to create junctions in the namespace.

DFS names occur after the DFS junction. They are typeless and resemble UNIX file system names. After the global and CDS parts of a DFS name have been resolved by the appropriate directory services, the rest of the DFS name is handled within the Distributed File Service. In the equivalent examples that follow, **/usr/snowpaws** is the DFS part of the DCE name:

/.../dce.osf.org/fs/usr/snowpaws
/.:/fs/usr/snowpaws
/:/usr/snowpaws

Security names are similar to DFS names; first the parts of the name within the DCE Directory Service are resolved, then the rest of the name is handled by the Security Service. The entry is contained in the Security registry database. In the following example:

/.:/sec/principal/ziggy

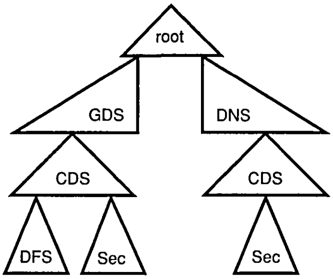
the Security part of the DCE name is **/principal/ziggy**.

22.5 The Federated DCE Namespace

The DCE namespace is a single hierarchy of names, but the names can be contained in many different services. Because several services cooperate to make the DCE namespace, it is a federated namespace.

Figure 22-1 shows a typical DCE namespace and the different services in which names reside.

Figure 22–1. A Federated DCE Namespace



The following sections describe the different domains of the DCE namespace.

22.5.1 The GDS Namespace

This section provides a brief overview of the main characteristics of the GDS namespace regarded apart from the XDS interface used to access it. More detailed information about GDS and XDS can be found in Part 4B and Part 4C, respectively.

In a GDS name such as

/.../C=US/O=OSF/OU=DCE

the **C=US** and **O=OSF** elements do not refer to directory entries that are fundamentally different from the one represented by **OU=DCE**, unlike in CDS or the UNIX file system.

Thus, in the name string

/C=US/O=OSF/OU=DCE

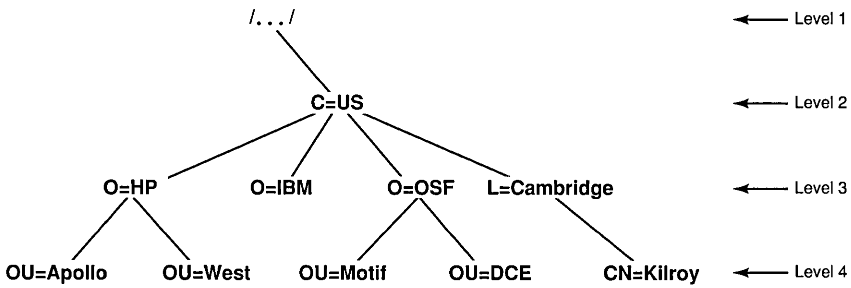
the element **C=US** refers to a one-level-down Country entry whose value is **US**, then to a two-levels-down Organization entry whose value is **OSF**, and then to a three-levels-down Organization Unit entry whose value is **DCE**. Concatenating these elements results in a valid path of entries from the directory root to the **DCE** entry. The entry itself is the namespace sign to a GDS directory object that contains binding information for the **/.../C=US/O=OSF/OU=DCE** cell.

22.5.1.1 An Example GDS Namespace

Figure 22-2 shows what a part of the DCE global namespace could look like. Levels in the tree of entries are numbered; the global root is at Level 0. The GDS structure rules as defined for DCE allow only country name entries at the next level under the root; organization name and locality name entries can exist at the level below a country name. An organizational unit name can be a child of an organizational name entry, and a common name can be a child of a locality name. The details of the GDS rules for the valid types and locations of entries in the directory tree can be found in the *OSF DCE Administration Guide*.

The object entry **/C=US/O=OSF/OU=DCE** belongs to the Organizational Unit class. One of the object's values is the CDS server binding information that is used to reach the cell from other DCE cells. The entire name is an attribute of the object that it refers to, as is the CDS server binding information that it contains.

Figure 22–2. GDS Namespace Entries and Directory Objects



22.5.1.2 The GDS Schema

The schema defines the shape and format of entries in the GDS directory. It contains four types of rules, which describe the following:

- The legal hierarchy of entries. What entries may be subordinate of other entries. These rules are what prevents, for example, countries from being subordinate to states.
- The allowable object classes, the mandatory and optional attributes of entries, and which attributes are the naming attributes.
- The allowable attribute types, associating a unique OID and an attribute syntax with each attribute type.
- The syntaxes of attributes that describe what attribute values look like, such as strings, numbers, or OIDs.

By installing the proper schema, an entry of any particular object class can have the two attributes needed to identify a cell. See the *OSF DCE Administration Guide* for a full description of how to set up a cell entry using either GDS or DNS.

22.5.2 The CDS Namespace

The CDS namespace is the part of the DCE namespace that resides in the local cell's CDS. DCE itself is made up of components that, like the applications that use them, are distributed client/server applications. These components rely on the Cell Directory Service to make themselves available as services to DCE applications. This requires that the structure of the cell namespace be stable, known, and have parts that are not alterable by casual users or applications.

22.5.2.1 The CDS "Schema"

The cell namespace's hierarchy model is different from the GDS model, and the CDS rules do not enforce any particular model; CDS allows entries containing any kind of data to be created anywhere in the namespace. Thus, CDS offers a free-form namespace in which entries and directories can be organized as desired, and in which any entry or directory can contain any attributes. The CDS administrator can create additional directories, and applications can add name entries as needed; applications *cannot* create CDS directory entries. Because of this, and because the cell namespace is so important to the operation of the cell, application developers and system administrators have more responsibility in planning and regulating their use of it.

The cell namespace has a structure similar to that of a UNIX file system. The CDS namespace is a tree of entries that grows from the root downward. The name entries are organized under directory entries, which can themselves be subentries of other directories. The cell root (represented by the prefix **/.:**) can be thought of as the location you get when you dereference the cell's global name. New directories and new entries within the directories can be added anywhere in the tree, subject to the restrictions mentioned previously.

22.5.2.2 CDS Entries and CDS Attributes

There are three different kinds of CDS entries that are of significance to application programmers:

- Object
- Soft link
- Directory

The object entries are the most primitive form. These are where data is stored. Directory entries contain other entries (that is, can have children) just like UNIX file system directories. Soft link entries are essentially alias names for other directory or object entries. Only object entries can be created by applications; soft links and directories have to be created and manipulated with the **cdscp** command.

Thus, any CDS entry is defined as a directory, a soft link, or an object entry by the presence of a certain combination of attributes belonging to that kind of entry. You can use the **cdscp** command to get a display of all the attributes of any CDS entry.

The term ''attribute'' as applied to namespace entry objects has roughly the same meaning in CDS and GDS. The main difference is that CDS does not restrict or control the combinations of attributes attached to entries written in its namespace.

22.5.3 Other Namespaces

For information about names contained in the DFS namespace (the filespace) and the Security namespace, refer to the chapters on those components in this guide.

22.6 Programming Interfaces to the DCE Directory Service

The following two subsections describe two programming interfaces for accessing the DCE Directory Service.

22.6.1 The X/Open Directory Service Interface

The main programming interface to all services within the DCE Directory Service is XDS/XOM, as defined by X/Open. The calls correspond to the X.500 service requests, including Read, List (enumerate children), Search, Add Entry, Modify Entry, Modify RDN, and Remove Entry. XDS uses XOM to define and manipulate data structures (called *objects*) used as the parameters to these calls, and used to describe the directory entries manipulated by the calls. XOM is extremely flexible, but also somewhat complex. The interfaces are used in different ways, depending on which underlying directory service is being addressed. For example, CDS entries are typeless, but GDS entries are typed. This difference is reflected in the use of the interface.

22.6.2 The RPC Name Service Interface

The DCE Remote Procedure Call facility supports an interface to the Directory Service that is specific to RPC and is layered on top of DCE Directory Service interfaces; it is called the Name Service Independent (NSI) interface. NSI can manipulate three object classes: entries, groups, and profiles, which were created to store RPC binding information. NSI data is stored in CDS. Programming using this interface is discussed in Part 1 and Part 3 of this guide.

22.6.3 Namespace Junction Interfaces

For information about programming interfaces to names that occur in namespace junctions, see the documentation for that component. For example, for information about using DFS names, see Part 7 of this guide.

Part 4A

CDS Application Programming

Part 4A describes DCE Directory Service application programming in the CDS namespace. Chapter 23 describes the contents of the CDS namespace, where applications should put their data, and what the valid CDS characters and names are. Chapter 24 describes how to use the XDS programming interface to access data in the CDS namespace.

Chapter 23

Programming in the CDS Namespace

This chapter provides information about writing applications that use the XDS/XOM interface to access the portion of the DCE namespace contained in the Cell Directory Service.

The XDS/XOM interface provides generalized access to CDS. However, if you only need to use CDS to store information related to RPC (for example, storing the location of a server so that clients can find it), you should use the Name Service Independent (NSI) interface of DCE RPC. NSI implements RPC-specific use of the namespace. For information on using RPC NSI, see Chapter 2 and Part 3 of this guide.

For information on the details of accessing the CDS namespace through the XDS/XOM interface, see Chapter 24 and Part 4C. For information about using XDS/XOM to access the GDS portion of the DCE namespace, see Part 4B.

23.1 Initial Cell Namespace Organization

The following subsections describe the organization of a cell's namespace after it has initially been configured. (For more information on configuring a cell, see the *OSF DCE Administration Guide*.)

Every DCE cell is set up at configuration with the basic namespace structure necessary for the other DCE components to be able to find each other and to be accessible to applications. The vital parts of the namespace are protected from being accessed by unauthorized entities by Access Control Lists (ACLs) that are attached to the entries and directories.

Figure 23-1 shows what the cell namespace looks like after a cell has been configured and before any additional directories or entries have been added to it by system administrators or applications. In the figure, ovals represent directories, rectangles represent simple entries, circles represent soft links, and triangles represent namespace junctions.

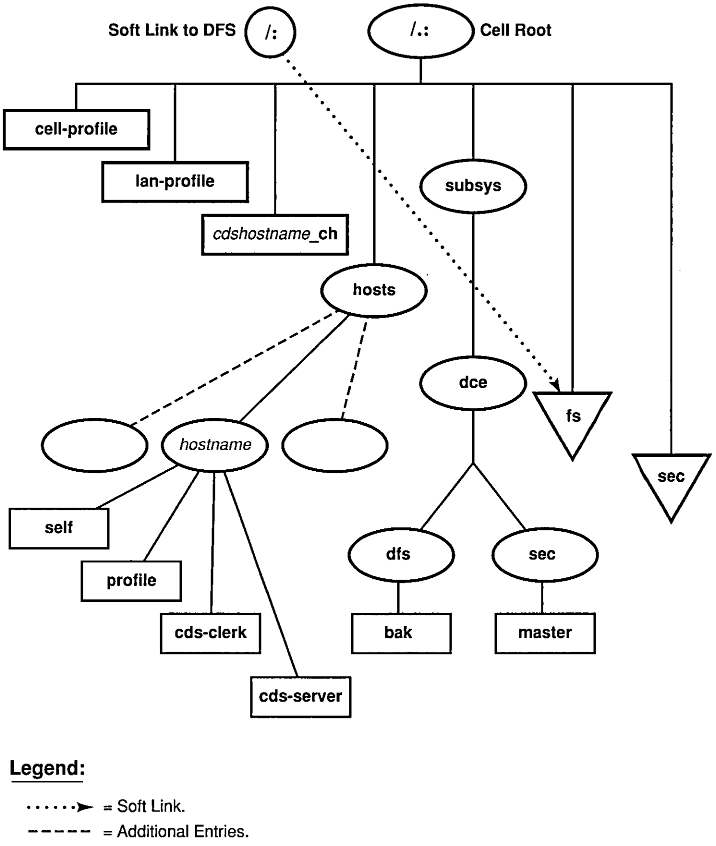
All of the simple entries shown in the figure are created for use with RPC NSI routines; that is, they all contain server-binding information and exist to enable clients to find servers. These are referred to as ''RPC entries.''

Note that only the name entries (those in boxes) and junction entries (those in triangles) are RPC entries. The directories (entries indicated by ovals) are normal CDS directories.

Some of the namespace entries in the figure are intended to be used (if desired) directly by applications; namely, **/.:/cell-profile**, **/.:/lan-profile**, and, through the **/:** soft link alias, **/.:/fs**. The **self** and **profile** name entries under **hosts** also fall into this category. Others, such as those under **/.:/subsys/dce**, are for the internal use of the DCE components themselves.

Each of the entries is explained in detail in the following subsections. See the *OSF DCE Administration Guide* for detailed information on the contents of the initial DCE cell namespace.

Figure 23–1. The Cell Namespace After Configuration



23.1.1 The Cell Profile

The **/.:/cell-profile** entry is an RPC profile entry that contains the default list of namespace entries to be searched by clients trying to bind to certain basic services. An RPC profile is a class of namespace entry used by the RPC NSI routines. When a client imports bindings from such an entry, it imports, through the profile, from an ordered list of RPC entries containing appropriate bindings. The list of entries is keyed by their interface UUIDs so that only bindings to servers offering the interface sought by the client are returned. The entries listed in the profile exist independently in the namespace, and can be separately accessed in the normal way. The profile is simply a way of organizing clients' searches. (For further information, see Part 3 of this guide.)

The main purpose of **cell-profile** is to have a ''path of last resort'' for prospective clients. All other profile entries in the cell namespace are required to have the **cell-profile** entry in *their* entry lists so that if a client exhausts a particular profile's list of entries, it tries the entries in **cell-profile.**

23.1.2 The LAN Profile

The **/.:/lan-profile** entry is a LAN-oriented default list of services' namespace entries that is used when servers' relative positions in the network topography are of importance to their prospective clients.

23.1.3 The CDS Clearinghouse

The */.:/cdshostname*_**ch** entry is the namespace entry for *cdshostname*'s clearinghouse, where *cdshostname* is the name of the host machine on which a CDS server is installed.

A ''clearinghouse'' is the database managed by a CDS server; it is where CDS directory replicas are physically stored. For more information about clearinghouses, see the *OSF DCE Administration Guide.* All clearinghouse namespace entries reside at the cell root, and there must be at least one in a

DCE cell. The first clearinghouse's name must be in the form shown in Figure 23-1, but additional clearinghouses can be named as desired.

23.1.4 The Hosts Directory

The **/.:/hosts** entry is a directory containing entries for all of the host machines in the cell. Each host has a separate directory under **hosts**; its directory has the same name as the host. Four entries are created in each host's directory:

- **self**

  This entry contains bindings to the host's RPC daemon (**rpcd**, also called the endpoint mapper), which is responsible for dynamically resolving the partial bindings that it receives in incoming RPCs from clients attempting to reach servers resident on this host.

- **profile**

  This entry is the default profile entry for the host. This profile contains in its list of entries at least the **/.:/cell-profile** entry described in Section 23.1.1.

- **cds-clerk**

  This entry contains bindings to the host's resident CDS clerk.

- **cds-server**

  This entry contains bindings to a CDS server.

23.1.5 The Subsystems Directory

The **/.:/subsys** entry is the directory for subsystems. Subdirectories below **subsys** are used to hold entries that contain location-independent information about services, particularly RPC binding information for servers.

The **dce** directory is created below **/.:/subsys** at configuration. This directory contains directories for the DCE Security Service and Distributed File Service components. The functional difference between these two directories and the **fs** and **sec** junctions described in Section 23.1.7 is that the latter two entries are the access points for the components' special databases, whereas the directories under **subsys/dce** contain the services' binding information.

Subsystems that are added to DCE should place their system names in directories created beneath the **/.:/subsys** directory.

23.1.6 The /: DFS Alias

The entry **/:** is created and set up as a soft link to the **/.:/fs** entry, which is the DFS database junction. The name **/:** is equivalent to **/.:/fs**. Note, however, that the name **/:** is well known, whereas the name **/.:/fs** is not, so using **/:** makes an application more portable. A CDS soft link entry is an alias to some other CDS entry. A soft link is created through the **cdscp** command. The procedure is described in the *OSF DCE Administration Guide*.

23.1.7 The DFS and Security Service Junctions

The **/.:/fs** entry is the Distributed File Service junction entry. This is the entry for a server that manages the DFS file location database.

The **/.:/sec** entry is the DCE Security Service junction entry. This is the entry for a server that manages the Security Service database (also called the registry database).

The **/.:/fs** and **/.:/sec** root entries in Figure 23-1 are junctions maintained by DCE components. The **/.:/sec** junction is the DCE Security Service's namespace of principal identities and related information. The Distributed File Service's fileset location servers are reached through the **/.:/fs** entry, making **/.:/fs** effectively the entry point into the cell's distributed file system.

Note that **/.:/sec** and **/.:/fs** are both actually RPC group entries (for definitions of the RPC entry types, see Chapter 2 of the present book); the junctions are implemented by the servers whose entries are members of the group entries. (See the *OSF DCE Administration Guide* for further details on the Security Service and DFS junctions.)

23.2 Recommended Use of the CDS Namespace

CDS data is maintained in a loosely consistent manner. This means that when the writeable copy of a replicated name is updated, the read-only copies may not be updated for some period of time, and applications reading from those nonsynchronized copies can receive stale data. This is in contrast to distributed databases, which use multiphase commit protocols that prevent readers from accessing potentially stale or inconsistent data while the writes are being propagated to all copies of the data. It is possible to specifically request data from the master copy, which is guaranteed to be up-to-date, but replication advantages are then lost. This should only be done when it is important to obtain current data.

23.2.1 Storing Data in CDS Entries

Some CDS entries may contain information that is immediately useful or meaningful to applications. Other entries may contain RPC information that enables application clients to reach application servers; that is, binding handles for servers, which are stored and retrieved using the RPC NSI routines. In either case, the entry's name should be a meaningful identification label for the information that the entry contains. This is because the namespace entry names are the main clue that users and applications have to the available set of resources in the DCE cell. Using the CDS namespace to store and retrieve binding information for distributed applications is the function of DCE RPC NSI. (See Part 3 of this guide for information on that aspect of namespace usage.)

In general, applications can store data into CDS object entry attributes in any XDS-expressible form they wish. Tables 24-2 and 24-3 show XDS-to-CDS data type translations. If you add new attributes to the **/opt/dcelocal/etc/cds_attributes** file, together with a meaningful CDS syntax (that is, a data type identifier) and name, then the attribute is displayed by **cdscp show** commands when executed on objects containing instances of that attribute.

There are three main questions to consider when using CDS to store data through application calls to XDS:

1. **Where in the CDS namespace should the new entries be placed?**

   You are free to create new directories as long as you do not disturb the namespace's configured structure. Keep in mind that CDS directories must be created with the **cdscp** command; they cannot be created by applications.

   Only two root-level directories are created at configuration: **hosts** and **subsys**. Applications should not add entries under the **hosts** tree; the host's default profile should instead be set up by a system administrator. The **subsys** directory is intended to be populated by directories (for example, **/.:/subsys/dce**) in which the servers and other components of independent vendors' distributed products are accessed. Thus, the typical cell should usually have a series of root-level CDS directories that represent a reasonable division of categories.

   One obvious division could be between entries intended for RPC use (that is, namespace entries that contain bindings for distributed applications), and entries that contain data of other kinds. On the other hand, it may be very useful to add supplementary data attributes to RPC entries in which various housekeeping or administrative data can be held. In this way, for example, performance data for printers can be associated with the print servers' name entries. You can either add new attributes to the server entries themselves, where, for example,

   /.:/applications/printers/pr1

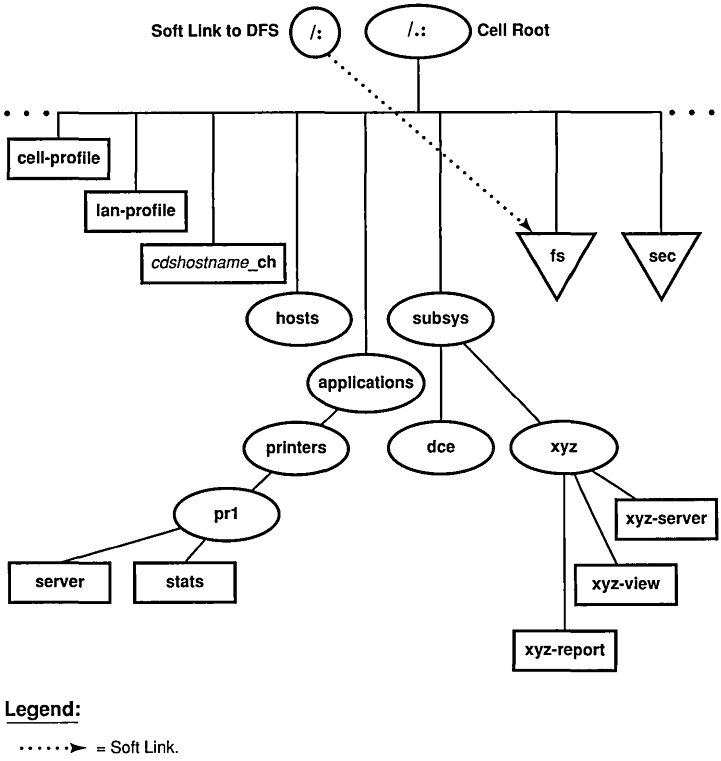
   is the name of a server entry that receives the new attributes. Or you can change the subtree structure so that new *entries* are added to hold the data, the server bindings are still held in separate wholly RPC entries, and each group of entries is located under a directory named for the printer:

   /.:/applications/printers/pr1 —*directory*
   /.:/applications/printers/pr1/server —*server bindings*
   /.:/applications/printers/pr1/stats —*extra data*

In general, the same principals of logic and order that apply to the organization of a file system apply to the organization of a namespace. For example, server entries should *not* be created directly at the namespace root because this is the place for default profiles, clearinghouse entries, and directories.

Figure 23-2 illustrates some of the preceding suggestions, added to the initial configuration namespace structure shown in Figure 23-1.

Figure 23–2. A Possible Namespace Structure



In Figure 23-2, the vendor of the ‘‘xyz’’ subsystem has set up an **xyz** directory under **/.:/subsys** in which the system’s servers are exported. This cell also has an **/.:/applications** directory in which the **printers** directory contains separate directories for each installed printer available on the system; the directory for **pr1** is illustrated in the figure. In the **pr1** directory, **server** is an RPC entry containing exported binding handles, and **stats** is an entry created and maintained through the XDS interface.

2. **How should the entries be constructed?**

   Because CDS allows you to add as many attributes as you wish to an object entry, it is up to you to impose some restraint in doing this. In view of the XDS overhead involved in reading and writing single CDS attributes, it makes sense to combine multiple related attributes under single entries (that is, in the same directory object) where they can be read and written in single calls to **ds_read()** or **ds_modify_entry()**. This way, for example, you only have to create one interface input object (to pass to **ds_read()**) to read all the attributes, which you can do with one call to **ds_read()**. You can then separate out the returned subobjects that you are interested in and ignore the rest. Chapter 24 contains detailed discussions of XDS programming techniques.

   In any case, you should define object types for use in applications so that namespace access operations can be standardized and kept efficient. A CDS object type consists of a specific set of attributes that belong to an object of that type, with no other attributes allowed. Note again that CDS, unlike GDS, does not force you to do things this way. You could theoretically have hundreds of CDS object entries, each of which would contain a different combination of attributes.

3. **Should a directory or an entry be created?**

   When you consider adding information to the namespace, you can choose between creating a new directory, possibly with entries in it, or creating simply one or more entries. When making your decision, take into consideration the following:

   a. Directories cannot be created using XDS; they must be created using administrative commands. Directories are more expensive; they take up more space and take more time to access. However, they can contain entries and can therefore be used to organize information in the namespace.

   b. Entries can be created using XDS and they are cheaper to create and use than directories. However, they must be created in existing directories, and cannot themselves contain other entries.

23.2.2 Access Control for CDS Entries

Each object in the CDS namespace is automatically equipped with a mechanism by which access to it can be regulated by the object's owner or by another authority. For each object, the mechanism is implemented by a separate list of the entities that can access the object in some way; for example, to read it, write to it, delete it, and so on. Associated with each entity in this list is a string that specifies which operations are allowed for that entity on the object. The object's list is automatically checked by CDS whenever any kind of access is attempted on that object by any entity. If the entity can be found in the object's list, and if the kind of access the entity intends is found among its permissions, then the operation is allowed to proceed by CDS; otherwise, it is not allowed.

DCE permission lists are called Access Control Lists (ACLs). ACLs are one of the features of the DCE Security Service used by the Cell Directory Service. ACLs are used to test the entities' (that is, the principals') authorization to do things to the objects they propose to do them to. The authorization mechanism for all CDS objects is handled by CDS itself. All that users of the CDS namespace have to do is make sure that ACLs on the CDS objects that they create are set up with the appropriate permissions.

23.2.2.1 Creation of ACLs

Whenever you create a new entry in the CDS namespace, an ACL is created for it implicitly, and its initial list of entries and their permission sets are determined by the ACL templates associated with the CDS directory in which you create the entry.

Each CDS directory has the following two ACL templates associated with it:

- Initial Container

  This template is used to generate the initial ACL for any directories created within the directory.

- Initial Object

  This template is used to generate ACLs for entries created within the directory.

Every CDS directory also has its own ACL, just like any other CDS object. This ACL is generated from the parent directory's Initial Container template when the child directory is created. The Initial Container template also serves as a template for the child directories' own Initial Container templates.

23.2.2.2 Manipulating ACLs

There are two ways to manipulate ACLs: either through the **acl_edit** command, which is documented in the *OSF DCE User's Guide and Reference*, or through the DCE ACL application interface, which consists of routines documented in the *OSF DCE Application Development Reference*. (These routines have names in the form of **sec_acl_...( )**.)

23.2.2.3 Initializing ACLs

After creating a CDS directory using the **cdscp** command, your first step is usually to run the **acl_edit** command to set up the new directory's ACLs the way you want them. (The new directory will have inherited its ACLs and its templates from the directory in which it was created, as explained in Section 23.2.2.1.) You may want to modify not only the directory's own ACLs, but also its two templates. To edit the latter, you can specify the **-ic** option (for the Initial Container template) or the **-io** option (for the Initial Object template); otherwise, you will edit the object ACL.

You can modify a directory's ACL templates from an application, assuming that you have control permission for the object, with the same combination of **sec_acl_lookup( )** and **sec_acl_replace( )** calls as for the object ACL. An option to these routines lets you specify which of the three possible ACLs on a directory object you want the call applied to. The ACLs themselves are in identical format.

The **-e** (entry) option to **acl_edit** can be used to make sure that you get the ACL for the specified namespace entry object, and not the ACL (if any) for the object that is *referenced by* the entry. This distinction has to be made clear to **acl_edit** because it finds the object (and hence the ACL) in question by looking it up in the namespace and binding to its ACL manager. Essentially, the **-e** option tells **acl_edit** whether it should bind to the CDS

ACL manager (if the entry ACL is wanted), or to the manager responsible for the referenced object's ACL. This latter manager would be a part of the server application whose binding information the entry contained.

An example of such an ambiguous name would be a CDS clearinghouse entry, such as the *cdshostname*_**ch** entry discussed previously. With the **-e** option

acl_edit -e */.:/cdshostname*_**ch**

you would edit the ACL on the namespace entry; without the **-e** option you would edit the ACL on the clearinghouse itself, which you presumably do *not* want to do.

Similarly, there is a *bind_to_entry* parameter by which the caller of **sec_acl_bind()** can indicate whether the entry object's ACL or the ACL to which the entry refers is desired. For further details on the pitfalls of binding ambiguity, see Chapter 2 of this guide.

23.2.2.4 Namespace ACLs at Cell Configuration

The ACLs attached to the CDS namespace at configuration are described in *OSF DCE Administration Guide*. The following ACL permissions are defined for CDS objects. The single letter in parentheses for each item represents the DCE notation for that permission. These single letters are identical to the untokenized forms returned by **sec_acl_get_printstring()**.

- read (**r**)

  This permission allows a principal to look up an object entry and view its attribute values.

- write (**w**)

  This permission allows a principal to change an object's modifiable attributes, except for its ACLs.

- insert (**i**)

  This permission allows a principal to create new entries in a CDS directory. It is used with directory entries only.

- delete (**d**)

  This permission allows a principal to delete a name entry from the namespace.

- test (**t**)

  This permission allows a principal to test whether an attribute of an object has a particular value, but does not permit it actually to see any of the attribute values (in other words, read permission for the object is not granted). The test permission allows an application to verify a particular CDS attribute's value without reading it.

- control (**c**)

  This permission allows a principal to modify the entries in the object's ACL. The control permission is automatically granted to the creator of a CDS object.

- administer (**a**)

  This permission allows a principal to issue **cdscp** commands that control the replication of directories. It is used with directory entries only.

Detailed instructions on the mechanics of setting up ACLs on CDS objects can be found in the *OSF DCE Administration Guide*.

For CDS directories, read and test permissions are sufficient to allow ordinary principals to access the directory and to read and test entries therein. Principals who you want to be able to add entries in a CDS directory should have insert permission for that directory. Entries created by the RPC NSI routines (for example, when a server exports bindings for the first time) are automatically set up with the correct permissions. However, if you are creating new CDS directories for RPC use, you should be sure to grant prospective user principals insert permission to the directory so that servers can create entries when they export their bindings. A general list of the permissions required for the various RPC NSI operations can be found in the **intro(3rpc)** reference page in the *OSF DCE Application Development Reference*, and the reference pages for the RPC NSI routines (all of whose names are in the form **rpc_ns_...( )**) in the same manual.

Note that CDS names do not behave the same way as file system names. A principal does not have to have access to an entire entry name path in order to have access to an entry at the end of that path. For example, a principal can be granted read access to the following entry:

/.:/applications/utilities/pr2

and yet not have read access to the **utilities** directory itself.

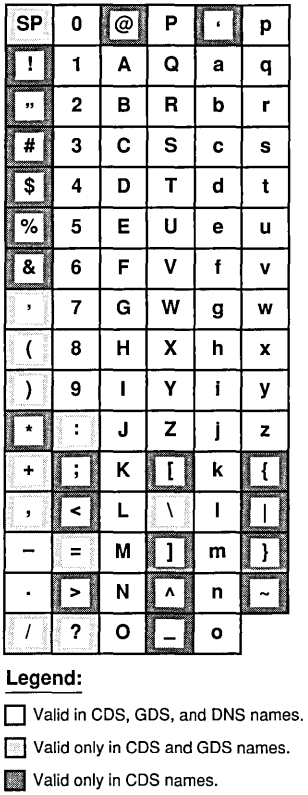
23.3 Valid Characters and Naming Rules for CDS

The following subsections discuss the valid character sets for DCE Directory Service names as used by CDS interfaces. They also explain some characters that have special meaning and describe some restrictions and rules regarding case matching, syntax, and size limits.

The use of names in DCE often involves more than one directory service. For example, CDS interacts with either GDS or DNS to find names outside the local cell.

Figure 23-3 details the valid characters in CDS names, and the valid characters in GDS and DNS names as used by CDS interfaces.

Figure 23–3. Valid Characters in CDS, GDS, and DNS Names



SP	0	@	P	‘	p
!	1	A	Q	a	q
”	2	B	R	b	r
#	3	C	S	c	s
$	4	D	T	d	t
%	5	E	U	e	u
&	6	F	V	f	v
’	7	G	W	g	w
(	8	H	X	h	x
)	9	I	Y	i	y
*	:	J	Z	j	z
+	;	K	[	k	{
,	<	L	\	l	|
–	=	M	]	m	}
.	>	N	^	n	~
/	?	O	_	o	

Legend:

□ Valid in CDS, GDS, and DNS names.

▨ Valid only in CDS and GDS names.

■ Valid only in CDS names.

Note: Because CDS, GDS, and DNS all have their own valid character sets and syntax rules, the best way to avoid problems is to keep names short and simple, consisting of a minimal set of characters common to all three services. The recommended set is the letters A to Z, a to z, and the digits 0 to 9. In addition to making directory service interoperations easier, use of this subset decreases the probability that users in a heterogeneous hardware and software environment will encounter problems creating and using names.

Although spaces are valid in both CDS and GDS names, a CDS simple name containing a space must be enclosed in " " (double quotes) when you enter it through the CDS control program. Additional interface-specific rules are documented in the modules where they apply.

23.3.1 Metacharacters

Certain characters have special meaning to the directory services; these are known as metacharacters. Table 23-1 lists and explains the CDS, GDS, and DNS metacharacters.

Table 23–1. Metacharacters and Their Meaning

Directory Service	Character	Meaning
CDS	/	Separates elements of a name (simple names).
	*	When used in the rightmost simple name of a name entered in a **cdscp show** or **list** command, acts as a wildcard, matching zero or more characters.
	?	When used in the rightmost simple name of a name entered in a **cdscp show** or **list** command, acts as a wildcard, matching exactly one character.
	\	Used where necessary in front of a / (slash), a \ (backslash), an * (asterisk), or a ? (question mark) to escape the character (indicates that the following character is not a metacharacter).
GDS	/	Separates Relative Distinguished Names (RDNs).
	,	Separates multiple attribute type/value pairs (attribute value assertions) within an RDN.
	=	Separates an attribute type and value in an attribute value assertion.
	\	Used in front of a / (slash), a , (comma), or an = (equal sign) to escape the character (indicates that the following character is not a metacharacter).
DNS	.	Separates elements of a name.

Some metacharacters are not permitted as normal characters within a name. For example, a \ (backslash) cannot be used as anything but an escape character in GDS. You can use other metacharacters as normal characters in a name, provided that you escape them with the backslash metacharacter.

23.3.2 Additional Rules

Table 23-2 summarizes major points to remember about CDS, GDS, and DNS character sets, metacharacters, restrictions, case-matching rules, internal storage of data, and ordering of elements in a name. For additional details, see the documentation for each technology.

Table 23–2. Summary of CDS, GDS, and DNS Characteristics

Characteristic	**CDS**	**GDS**	**DNS**
Character Set	a to z, A to Z, 0 to 9 plus space and special characters shown in Figure 23-3	a to z, A to Z, 0 to 9 plus . : , ' + - =) ? / and space	a to z, A to Z, 0 to 9 plus . -
Metacharacters	/ * ? \	/ , = \	.
Restrictions	Simple names cannot begin or end with a / (slash). The first simple name following the global cell name (or /.: prefix) cannot contain an = (equal sign). When entering a name as part of a **cdscp show** or **list** command, you must use a \ (backslash) to escape any * (asterisk) or ? (question mark) character in the rightmost simple name. Otherwise, the character is interpreted as a wildcard.	Relative distinguished names cannot begin or end with a / (slash). Attribute types must begin with an alphabetic character, can contain only alphanumerics, and cannot contain spaces. An alternate method of specifying attribute types is by object identifier, a sequence of digits separated by . (dots). You must use a \ (backslash) to escape a / (slash), a , (comma), and an = (equal sign) when using them as anything other than metacharacters. Multiple consecutive unescaped occurrences of / (slashes), , (commas), = (equal signs) and \ (back-slashes) are not allowed. Each attribute value assertion contains exactly one unescaped = (equal sign).	The first character must be alphabetic. The first and last characters cannot be a . (dot) or a - (dash). Cell names in DNS must contain at least one . (dot); they must be more than one level deep.
Case-Matching Rules	Case exact	Attribute types are matched case insensitive. The case-matching rule for an attribute value can be case exact or case insensitive, depending on the rule defined for its type at the DSA.	Case insensitive
Internal Representation	Case exact	Depends on the case-matching rule defined at DSA. If the rule says case insensitive, alphabetic characters are converted to all lowercase characters. Spaces are removed regardless of the case-matching rule.	Alphabetic characters are converted to all lowercase characters.
Ordering of Name Elements	Big endian (left to right from root to lower-level names).	Big endian (left to right from root to lower-level names).	Little endian (right to left from root to lower-level names).

23.3.3 Maximum Name Sizes

Table 23-3 lists the maximum sizes for Directory Service names. Note that the limits are implementation specific, not architectural.

Table 23–3. Maximum Sizes of Directory Service Names

Name Type	**Maximum Size (characters)**
CDS simple name (character string between two slashes)	254
CDS full name (including global or local prefix, cell name, and slashes separating simple names)	1023
GDS relative distinguished name	64
GDS distinguished name	1024
DNS relative name (character string between two dots)	64
DNS fully qualified name (sum of all relative names)	255

23.3.3.1 Valid Characters for GDS Attributes

This section describes the valid character sets for the GDS attributes.

The values of the country attributes are restricted to the ISO 3166 Alpha-2 code representation of country names. (For more information, see the *OSF DCE Administration Guide*.)

The character set for all other naming attributes is the T61 graphical character set. It is described in the next section.

23.3.3.2 T61 Syntax

Table 23-4 shows the T61 graphical character set.

Note: The 1) entry in the table indicates that it is not recommended that you use the codes in Column 2, Row 3 and Column 2, Row 4. Instead, use the appropriate code in Column A.

Table 23–4. T61 Syntax

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P		p				°			Ω	K
1			!	1	A	Q	a	q			¡	±	`		Æ	æ
2			’’	2	B	R	b	r			¢	²	´		Đ	đ
3			1)	3	C	S	c	s			£	³	^		ª	ð
4			1)	4	D	T	d	t			$	x	~		Ħ	ħ
5			%	5	E	U	e	u			¥	µ	–			ı
6			&	6	F	V	f	v			#	¶	˘		Ĳ	ĳ
7			'	7	G	W	g	w			§	•	˙		L•	l•
8			(	8	H	X	h	x			¤	+	¨		Ł	ł
9			)	9	I	Y	i	y							Ø	ø
A			*	:	J	Z	j	z					˚		Œ	œ
B			+	;	K	[	k				<<	>>	¸		º	ß
C			,	<	L		l	|				¼	_		Þ	þ
D			–	=	M	]	m					½	’’		Ŧ	ŧ
E			·	>	N		n					¾	¸		η	η
F			/	?	O	_	o					¿	ˇ		’n	

The administration interface supports only characters smaller than 0x7e for names. The X/Open Directory Service (XDS) Application Programming Interface (API) supports the full T61 range as indicated in the preceding table.

Some T61 alphabetical characters have a 2-byte representation. For example, a lowercase letter "a" with an acute accent is represented by 0xc2 (the code for an acute accent) followed by 0x61 (the code for a lowercase "a").

Only certain combinations of diacritical characters and basic letters are valid. They are shown in Table 23-5.

Table 23–5. Combinations of Diacritical Characters and Basic Letters

Name	Repr.	Code	Valid Basic Letters Following
grave accent	`	0xc1	a, A, e, E, i, I, o, O, u, U
acute accent	´	0xc2	a,A,c,C,e,E,g,i,I,l,L,n,N,o,O,r,R, s,S,u,U,y,Y,z,Z
circumflex accent	^	0xc3	a,A,c,C,e,E,g,G,h,H,i,I,j,J,o,O,s,S, u,U,w,W,y,Y
tilde	˜	0xc4	a,A,i,I,n,N,o,O,u,U
macron	¯	0xc5	a,A,e,E,i,I,o,O,u,U
breve	˘	0xc6	a, A, g, G, u, U
dot above	˙	0xc7	c,C,e,E,g,G,I,z,Z
umlaut	¨	0xc8	a,A,e,E,i,I,o,O,u,U,y,Y
ring	˚	0xca	a, A, u, U
cedilla	¸	0xcb	c,C,G,k,K,l,L,n,N,r,R,s,S,t,T
double accent	˝	0xcd	o, O, u, U
ogonek	˛	0xce	a, A, e, E, i, I, u, U
caron	ˇ	0xcf	c,C,d,D,e,E,l,L,n,N,r,R,s,S,t,T,z,Z

The nonspacing underline (code 0xcc) must be followed by a Latin alphabetical character; that is, a basic letter (a to z or A to Z), or a valid diacritical combination.

23.4 Use of Object Identifiers

Object Identifiers (OIDs) are not seen by applications that restrict themselves to using only the RPC NSI routines (**rpc_ns_...()**), but these identifiers are important for applications that use the XDS interface to read entries directly or to create new attributes for use with namespace entries.

RPC makes use of only four different entry attributes in various application-specified or administrator-specified combinations. CDS, however, contains definitions for many more than these, which can be added by applications to RPC entries through the XDS interface. Attributes that already exist are already properly identified so applications that use these attributes do not have to concern themselves with the OIDs, except to the extent of making sure that they handle them properly.

Unlike UUIDs, OIDs are not generated by command or function call. They originate from the International Organization for Standards (ISO), which allocates them in hierarchically organized blocks to recipients. Each recipient, typically some organization, is then responsible for ensuring that the OIDs it receives are used uniquely.

For example, the OID block

1.3.22

was allocated to OSF by ISO. OSF can now generate, for example, the OID

1.3.22.1.1.4

and allocate it to identify some DCE directory object. (The OID **1.3.22.1.1.4** identifies the RPC profile entry object attribute.) OSF is responsible for making sure that **1.3.22.1.1.4** is not used to identify any other attribute. Thus, as long as all OIDs are generated only from within each owner's properly obtained block, and as long as each block owner makes sure that the OIDs generated within its block are properly used, each OID will always be a universally valid identifier for its associated value.

OIDs are encoded and internally represented as strings of hexadecimal digits, and comparisons of OIDs have to be performed as hexadecimal string comparisons (not as comparisons on **NULL**-terminated strings since OIDs can have **NULL** bytes as part of their value).

When applications have occasion to handle OIDs, they do so directly because the numbers do not change and should not be reused. However, for users' convenience, CDS also maintains a file (called **cds_attributes**, found in **/opt/dcelocal/etc**) that lists string equivalents for all the OIDs in use in a cell in entries like the following one:

```
1.3.22.1.1.4    RPC_Profile    byte
```

This allows users to see **RPC_Profile** in output, rather than the meaningless string **1.3.22.1.1.4**. Further details about the **cds_attributes** file and OIDs can be found in the *OSF DCE Administration Guide*.

In summary, the procedure you should follow to create new attributes on CDS entries consists of three steps:

1. Request and receive from your locally designated authority the OIDs for the attributes you intend to create.
2. Update the **cds_attributes** file with the new attributes' OIDs and labels if you want your application to be able to use string name representations for OIDs in output.
3. Using XDS, write the routines to create, add, and access the attributes.

Your cell administrator should be able to provide you with a name and an OID. The *name* is a guaranteed-unique series of values for a global directory entry name. If the directory is GDS, the *name* is a series of type/value pairs, such as:

```
C=US O=OSF
```

The cell administrator can also obtain an OID block. From this OID space, the administrator can assign you the OIDs you need for your application.

Note that there is no need for new OIDs in connection with cell names. The OIDs for Country Name and Organization Name are part of the X.500 standard implemented in GDS; only the values associated with the OIDs (the values of the objects) change from entry name to entry name. Instead, being able to generate new OIDs gives you the ability to invent and add new details to the directory itself. For example, you can create new kinds of CDS entry attributes by generating new OIDs to identify them. The same thing can be done to GDS, although the procedure is more complicated because it involves altering the directory schema.

Chapter 24

XDS and the DCE Cell Namespace

This chapter describes the use of the XDS programming interface when accessing the CDS namespace. The first section provides an introduction to using XDS in the CDS namespace. Section 24.2 describes XDS objects and how they are used to access CDS data. Section 24.3 provides a step-by-step procedure for writing an XDS program to access CDS. Section 24.4 provides examples of using the XOM interface to manipulate objects. Section 24.5 provides details of the structure of XDS/CDS objects. Finally, Section 24.6 provides translation tables between XDS and CDS for attributes and data types.

24.1 Introduction to Accessing CDS with XDS

Outside of the DCE cells and their separate namespaces is the global namespace in which the cell names themselves are entered, and where all intercell references are resolved. Two directory services participate in the global namespace. The first is the X.500-compliant Global Directory Service (GDS) supplied with DCE. The second is the Domain Name Service (DNS), which DCE interacts with, but is not a part of DCE.

The global and cell directory services are accessed implicitly by RPC applications using the NSI interface. GDS and CDS can also be accessed explicitly using the X/Open Directory Service (XDS) interface. With XDS, application programmers can gain access to GDS, a powerful general-purpose distributed database service, which can be used for many other things besides intercell binding. XDS can also be used to access the *cell* namespace directly, as this chapter describes.

An XDS application looks very different from the RPC-based DCE applications. This is partly because there is no dependency in XDS on the DCE RPC interface, although you can use both interfaces in the same application. Also, XDS is a generalized directory interface, oriented more toward performing large database operations than toward fine-tuning the contents of RPC entries. Nevertheless, XDS can be used as a general access mechanism on the CDS namespace.

24.1.1 Using the Reference Material in this Chapter

Complete descriptions of all the XDS and XOM functions used in CDS operations can be found in the the *OSF DCE Application Development Reference*, which you should have beside you as you read through the examples in this chapter. Definitive descriptions of all XDS and XOM class types can be found in Part 4C of this guide. In particular, refer to those chapters for information about XDS error objects, which are not discussed in this chapter.

Complete descriptions for all objects required as *input* parameters by XDS functions when accessing a CDS namespace can be found in Section 24.5. Abbreviated definitions for these same objects can be found with all the others in Part 4C. XOM functions are general-purpose utility routines that operate on objects of any class, and take the rest of their input in conventional form.

Slightly less detailed descriptions of the *output* objects you can expect to receive when accessing CDS through XDS are also given in Section 24.5. You do not have to construct objects of these classes yourself; you just have to know their general structure so that you can disassemble them using XOM routines.

No information is given in this chapter about the **DS_status** error objects that are returned by unsuccessful XDS functions; a description of all the subclasses of **DS_status** requires a chapter in itself. Code for a rudimentary general-purpose **DS_status**-handling routine can be found in the **teldir.c** XDS sample program in Chapter 28 of this guide.

24.1.2 What You Cannot Do with XDS

XDS allows you to perform general operations on CDS entry attributes, something which you cannot do through the DCE RPC NSI interface. However, there are certain things you cannot do to cell directory entries even through XDS:

- You cannot create or modify directory entries; the **ds_modify_rdn( )** function does not work in a CDS namespace. These operations must be performed through the CDS **cdscp** command. For more information, see the *OSF DCE Administration Reference*.
- You cannot perform XDS searches on the cell namespace; the XDS function **ds_search( )** does not work. This is mainly because the CDS has no concept of a hierarchy of entry attributes, such as the X.500 schema. The **ds_compare( )** function, however, does work.

24.1.3 What Must Be Set Up

If you are planning to use XDS to access the cell namespace in a one-cell environment (that is, your cell does not need to communicate with other DCE cells), you do not need to set up a cell entry in GDS for your cell because the XDS functions simply call the appropriate statically linked CDS routines to access the cell namespace. In these circumstances, XDS always tries to access the local cell when given an untyped (non-X.500) name.

For XDS to be able to access any nonlocal cell namespace, that cell must be registered (that is, have an entry) in the global namespace.

For information on setting up your cell name, see the *OSF DCE Administration Guide*.

24.2 XDS Objects

The XDS interface differs from the other DCE component interfaces in that it is ''object oriented.'' The following subsections explain two things: first, what object-oriented programming means in terms of using XDS; and second, how to use XDS to access the Cell Directory Service.

Imagine a generalized data structure that always has the same data *type*, and yet can contain any kind of data, and any amount of it. Functions could pass these structures back and forth in the same way all the time, and yet they could use the same structures for any kind of data they wanted to store or transfer. Such a data structure, if it existed, would be a true *object*. Programming language constructs allow interfaces to pretend that they use objects, although the realities of implementation are not usually so simple.

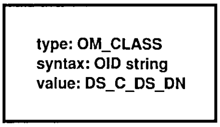
XDS is such an interface. For the most part, XDS functions neither accept nor return values in any form but as objects. The objects themselves are indeed always the same data type; namely, pointers to arrays of *object descriptor* (C **struct**) elements. Contained within these **OM_descriptor** element structures are unions that can actually accommodate all the different kinds of values an object can be called on to hold. In order to allow the interface to make sense of the unions, each **OM_descriptor** also contains a **syntax** field, which indicates the data type of that descriptor's union. There is also a record of what the descriptor's value actually is; for example, whether it is a name, a number, an address, a list, and so on. This information is held in the descriptor's **type** field.

These **OM_descriptor** elements, which are referred to as ''object descriptors'' or ''descriptors,'' are the basic building blocks of *all* XDS objects; every actual XDS object reduces to arrays of them. Each descriptor contains three items:

- A **type** field, which identifies the descriptor's value
- A **syntax** field, which indicates the data type of the **value** field
- The **value** field, which is a union

Figure 24-1 illustrates one such object descriptor.

Figure 24–1. One Object Descriptor



type: OM_CLASS
syntax: OID string
value: DS_C_DS_DN

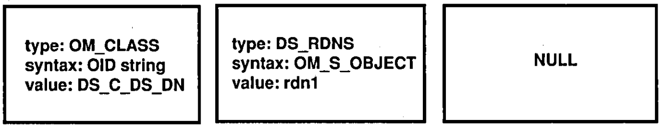
Note that, from an abstract point of view, **syntax** is just an implementation detail. The scheme really consists only of a type/value pair. The **type** gives an identity to the object (something like CDS entry attribute, CDS entry name, or DUA access point), and the **value** is some data associated with that identity, just as a variable has a name that gives meaning to the value it holds, and the value itself.

In order to make the representation of objects as logical and as flexible as possible, these two logical components of every object, **type** and **value**, are themselves each represented by separate object descriptors. Thus, the first element of every complete object descriptor array is a descriptor whose **type** field identifies its **value** field as containing the name of the kind (or *class*) of this object, and the **syntax** field indicates how that name **value** should be read. Next is usually one (or more, if the object is multivalued) object descriptor whose **type** field identifies its **value** field as containing some value appropriate for this class of object. Finally, every complete object descriptor array ends with a descriptor element that is identified by its fields as being a **NULL**-terminating element.

Thus, a minimum-size descriptor array consists of just two elements: the first contains its class identity, and the second is a **NULL** (it is legitimate for objects not to have values). When an object does have a value, it is held in the **value** field of a descriptor whose **type** field communicates the value's meaning.

Figure 24-2 illustrates the arrangement of a complete object descriptor array.

Figure 24–2. A Complete Object Represented



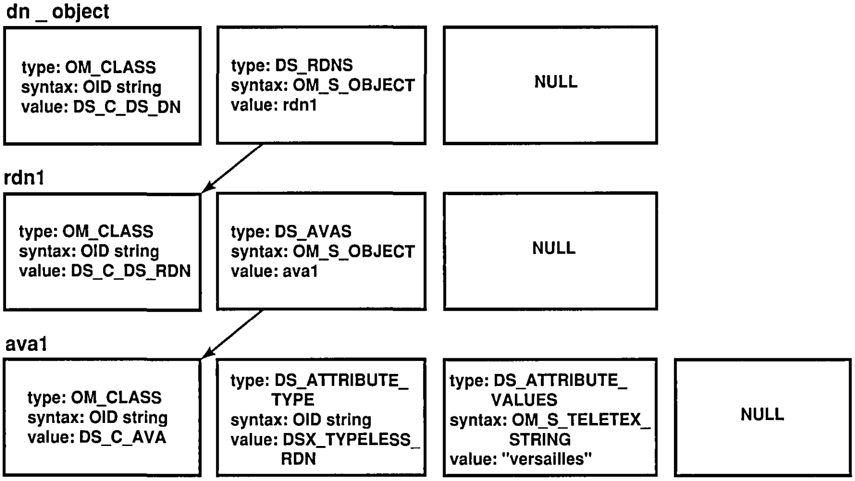
24.2.1 Object Attributes

The generic term for any object value is *attribute*. In this sense, an object is nothing but a collection of attributes, and every object descriptor describes one attribute. The first attribute's value identifies the object's class, and this determines all the other attributes the object is supposed to have. One or more other attributes follow, which contain the object's working values. The **NULL** object descriptor at the end is an implementation detail, and is not a part of the object.

Note that, depending on the attribute it represents, a descriptor's **value** field can contain a pointer to another array of object descriptors. In other words, an object's value can be another object.

Figure 24-3 shows a three-layer compound object: the top-level superobject, **dn_object**, contains the subobject **rdn1**, which in turn contains the subobject **ava1**.

Figure 24–3. A Three-Layer Compound Object



24.2.2 Interface Objects and Directory Objects

GDS is comprised of objects; these are directory objects, and reflect the X.500 design. The XDS interface also works with objects. However, there is a big difference between directory and XDS objects. Programmers do not work directly with the directory objects; they are composed of attributes that make up the directory service's implementation of entries.

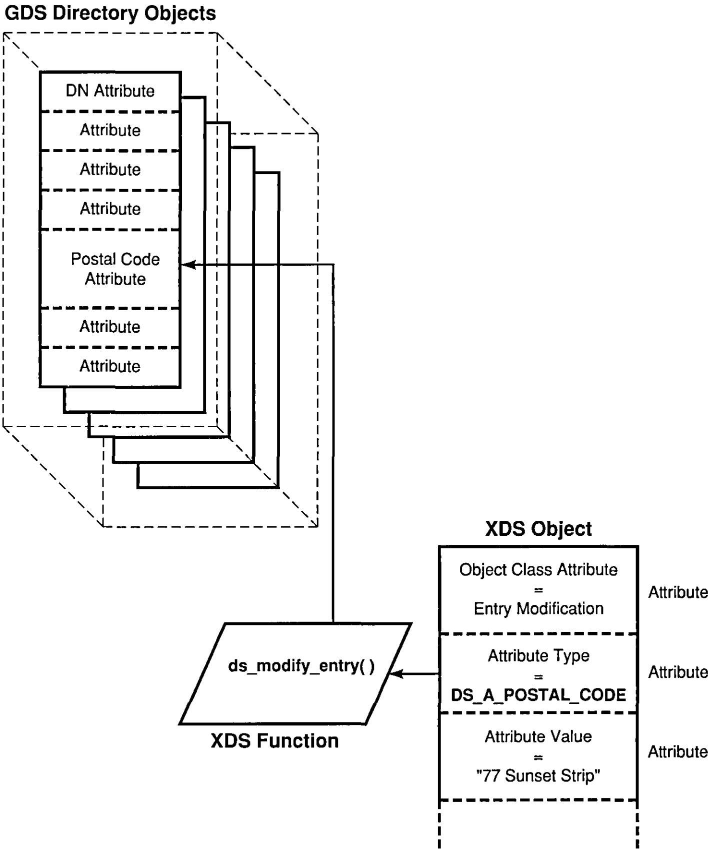
Programmers work with XDS objects. XDS objects have explicit data representations that can be directly manipulated with programming language operators. Some of these techniques are described in this chapter; others can be found in Chapter 28.

XDS and GDS terminology sometimes suggests that XDS objects are somehow direct representations of the directory objects to which they communicate information. This is not the case, however. You never directly see or manipulate the directory objects; the XDS interface objects are used only to pass parameters to the XDS calls, which in turn request GDS (or CDS) to perform operations on the directory objects. The XDS objects are therefore somewhat arbitrary structures defined by the interface.

Figure 24-4 illustrates the relationship between XDS (also called *interface*) objects and directory objects. The figure shows an application passing several properly initialized XDS objects to some XDS function; it then takes some action, which affects the attribute contents of certain directory objects. The application never works with the directory objects; it works with the XDS interface objects.

A side effect of the existence of a separate XDS interface and GDS or CDS directory objects is the existence of attributes for both kinds of objects as well. Because the purpose of XDS objects is to feed data into and extract data from directory objects, programmers work with XDS objects whose attributes have *directory* object attributes as their values. You should keep in mind the distinction between directory objects and interface objects.

Figure 24–4. Directory Objects and XDS Interface Objects

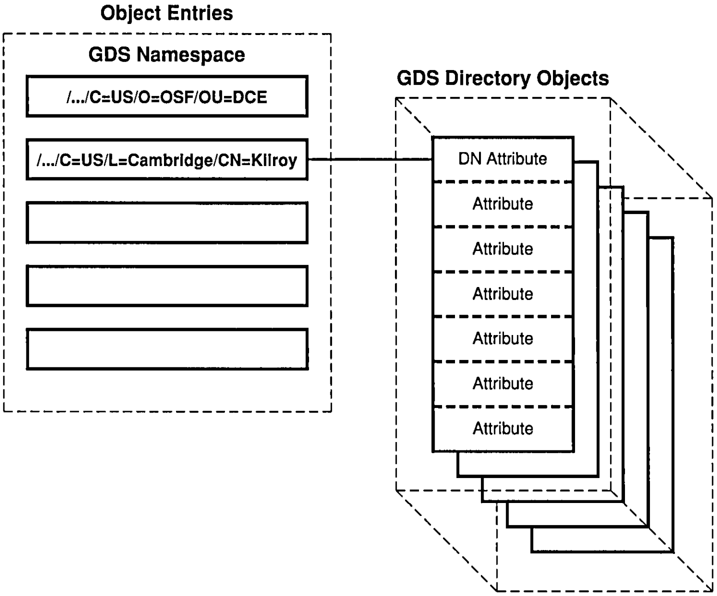


24.2.3 Directory Objects and Namespace Entries

The GDS namespace is a hierarchical collection of entries. The name of each of these entries is an attribute of a directory object. The object is accessed through XDS by stating its name attribute.

Figure 24-5 shows the relationship of entry names in the GDS namespace to the GDS directory objects to which they refer.

Figure 24–5. Directory Objects and Namespace Entries



24.2.4 Values That an Object Can Contain

There are many different classes of objects defined for the XDS interface; still more are defined by the X.500 standard for general directory use. But only a small number of classes are needed for XDS/CDS operations, and only those classes are discussed in this chapter. Information about other classes can be found in Part 4B of this guide.

The class that an object belongs to determines what sort of information the object can contain. Each object class consists of a list of attributes that objects must have. For example, you would expect an object in the directory entry name class to be required to have an attribute to hold the entry name string. However, it is not sufficient to simply place a string like:

/.../C=US/O=OSF/OU=DCE/hosts/tamburlaine/self

into an object descriptor.

A full directory entry name such as the preceding one is called in XDS a Distinguished Name (DN), meaning that the entry name is fully qualified (distinct) from root to entry name. To properly represent the entry name in an object, you must look up the definition of the XDS distinguished name object class and build an object that has the set of attributes that the definition prescribes.

24.2.5 Building a Name Object

Complete definitions for all the object classes required as input for XDS functions can be found in Section 24.5. Among them is the class for distinguished name objects, called **DS_C_DS_DN**. There you will learn that this class of object has two attributes: its class attribute, which identifies it as a **DS_C_DS_DN** object, and a second attribute, which occurs multiple times in the object. Each instance of this attribute contains as its value one piece of the full name; for example, the directory name **hosts**.

The **DS_C_DS_DN** attribute that holds the entry name piece, or Relative Distinguished Name, is defined by the class rules to hold, not a string, but another object of the Relative Distinguished Name class (**DS_C_DS_RDN**).

Thus, a static declaration of the descriptor array representing the **DS_C_DS_DN** object would look like the following:

```
static OM_descriptor    Full_Entry_Name_Object[] = {

     OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
/*   ^^^^^^^^^^^                                               */
/*   Macro to put an "OID string" in a descrip-                */
/*     tor's type field and fill its other                     */
/*     fields with appropriate values.                         */


     {DS_RDNS, OM_S_OBJECT, {0, Country_RDN}},
/*    ^^^^^^^  ^^^^^^^^^^^      ^^^^^^^^^^^                    */
/*     type      syntax            value                       */
/*                                                             */
/*     (the "value" union is in fact here a                    */
/*      structure; the 0 fills a pad field in                  */
/*      that structure.)                                         */


     {DS_RDNS, OM_S_OBJECT, {0, Organization_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Org_Unit_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Hosts_Dir_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Tamburlaine_Dir_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Self_Entry_RDN}},


     OM_NULL_DESCRIPTOR
/*   ^^^^^^^^^^^^^^^^^^                                          */
/*    Macro to fill a descriptor with proper                     */
/*      NULL values.                                             */


};
```

The use of the **OM_OID_DESC** and **OM_NULL_DESCRIPTOR** macros slightly obscures the layout of this declaration. However, each line contains code to initialize exactly one **OM_descriptor** object; the array consists of eight objects.

The names (such as **Country_RDN**) in the descriptors' **value** fields refer to the other descriptor arrays, which separately represent the relative name objects. (The order of the C declaration in the source file is opposite to the order described here.) Since **DS_C_DS_RDN** objects are now called for, the next step is to look at what attributes that class requires.

The definition for **DS_C_DS_RDN** can be found in Section 24.5.2.6. This class object is defined, like **DS_C_DS_DN**, to have only one attribute (with the exception of the **OM_Object** attribute, which is mandatory for all objects). The one attribute, **DS_AVAS**, holds the value of one relative name. The syntax of this value is **OM_S_OBJECT**, meaning that **DS_AVAS**'s value is a pointer to yet another object descriptor array:

```
static OM_descriptor    Country_RDN[] = {

     OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
        {DS_AVAS, OM_S_OBJECT, {0, Country_Value}},

        OM_NULL_DESCRIPTOR
    };
```

Note that there should also be five other similar declarations, one for each of the other **DS_C_DS_RDN** objects held in the **DS_C_DS_DN**.

The declarations have the same meanings as they did in the previous example. **Country_Value** is the name of the descriptor array that represents the object of class **DS_C_AVA**, which we are now about to look up.

The rules for the **DS_C_AVA** class can be found in this chapter just after **DS_C_DS_RDN**. They tell us that **DS_C_AVA** objects have two attributes aside from the omnipresent **OM_Object**; namely:

- **DS_ATTRIBUTE_VALUES**

  This attribute holds the object's value.

- **DS_ATTRIBUTE_TYPE**

  This attribute gives the meaning of the object's value.

In this instance, the meaning of the string **US** is that it is a country name. There is a particular directory service attribute value for this; it is identified by an OID that is associated with the label **DS_A_COUNTRY_NAME** (the OIDs held in objects are represented in string form). Accordingly, we make that OID the value of **DS_ATTRIBUTE_TYPE**, and we make the name string itself the value of **DS_ATTRIBUTE_VALUES**:

```
static OM_descriptor    Country_Value[] = {

    OM_OID_DESC(OM_CLASS, DS_C_AVA),

    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),

    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("US")},
/*                                              ^^^^^^^^^^^^^^^    */
/*                                                Macro to properly */
/*          fill the "value" union with the NULL-terminated string. */
```

```
        OM_NULL_DESCRIPTOR
    };
```

There are also five other **DS_C_AVA** declarations, one for each of the five other separate name piece objects referred to in the **DS_C_DS_RDN** superobjects.

24.2.6 A Complete Object

The previous sections described how an object is created: you look up the rules for the object class you require, and then add the attributes called for in the definition. Whenever some attribute is defined to have an object as its value, you have to look up the class rules for the new object and declare a further descriptor array for it. In this way you continue working down through layers of subobjects until you reach an object class that contains no subobjects as values; at that point, you are finished.

Normally, you do not statically declare objects in real applications. The steps outlined in the preceding text are given as a method for determining what an object looks like. Once you have done that, you can then write routines to create the objects dynamically. An example of how to do this can be found in the **teldir.c** example application in Chapter 28 of this guide.

The process of object building is somewhat easier than it sounds. There are only five different object classes needed for input to XDS functions when accessing CDS, and only one of those, the **DS_C_DS_DN** class, has more than one level of subobjects. The rules for all five of these classes can be found in Part 4B of this guide. In order to use the GDS references, you should know a few things about class hierarchy.

24.2.7 Class Hierarchy

Object classes are hierarchically organized so that some classes may be located above some classes in the hierarchy and below others in the hierarchy. In any such system of subordinate classes, each next lower class inherits all the attributes prescribed for the class immediately above it, plus whatever attributes are defined peculiarly for it alone. If the hierarchy continues further down, cumulative collection of attributes continues to accumulate. If there were a class for every letter of the alphabet, starting at the highest level with A and continuing down to the lowest level with Z, and if each succeeding letter was a subclass of its predecessor, the Z class would possess all the attributes of all the other letters, as well as its own, while the A class would possess only the A class attributes.

XDS/XOM classes are seldom nested more than two or at most three layers. All inherited attributes are explicitly listed in the object descriptions that follow, so you do not have to worry about class hierarchies here. However, the complete descriptions of XDS/XOM objects in Part 4C of this guide rely on statements of class inheritance to fill out their attribute lists for the different classes. Refer to Part 4C for information about the classes of objects that can be returned by XDS calls in order to be able to handle those returned objects.

24.2.8 Class Hierarchy and Object Structure

Note that *class hierarchy* is different from *object structure*. Object structure is the layering of object arrays that was previously described in the **DS_C_DS_DN** declaration in Section 24.2.5. It occurs when one object contains another object as the value of one or more of its attributes.

This is what is meant by recursive objects: one object can point to another object as one of its attribute values. The layering of subobjects below superobjects in this way is described repeatedly in Section 24.5.

The only practical significance of class hierarchy is in determining all the attributes a certain object class must have. Once you have done this, you may find that a certain attribute requires as its value some other object. The result is a compound object, but this is completely determined by the attributes for the particular class you are looking at.

24.2.9 Public and Private Objects and XOM

In Section 24.2.5, you saw how a multilevel XDS object can be statically declared in C code. Now imagine that you have written an application that contains such a static **DS_C_DS_DN** object declaration. From the point of view of your application, that object is nothing but a series of arrays, and you can manipulate them with all the normal programming operators, just as you can any other data type. Nevertheless, the object is syntactically perfectly acceptable to any XDS (or XOM) function that is prepared to receive a **DS_C_DS_DN** object.

Objects are also created by the XDS functions themselves; this is the way they usually return information to callers. However, there is a difference between objects generated by the XDS interface and objects that are explicitly declared by the application: you cannot access the former, *private*, objects in the direct way that you can the latter, *public*, objects.

These two kinds of objects are the same as far as their classes and attributes are concerned. The only difference between them is in the way they are accessed. The public objects that an application explicitly creates or declares in its own memory area are just as accessible as any of the other data storage it uses. However, private objects are created and held in the XDS interface's own system memory. Applications get handles to private objects, and in order to access the private objects' contents, they have to pass the handles to Object Management functions. The Object Management (XOM) functions make up a sort of all-purpose companion interface to XDS. Whereas XDS functions typically require some specific class object as input, the XOM functions accept objects of any class and perform useful operations on them.

If a private object needs to be manipulated, one of the XOM functions, **om_get**( ), can be called to make a public copy of the private object. Then, calling the XOM **om_create**( ) function allows applications to generate private objects manipulable by **om_get**( ). The main significance of private as opposed to public objects is that they do not have to be explicitly operated on; instead, you can access them cleanly through the XOM interface and let it do most of the work. You still have to know something about the objects' logical representation, however, to use XOM.

Except for a few more details, which will be mentioned as needed, this is practically all there is to XDS object representation.

24.2.10 XOM Objects and XDS Library Functions

To call an XDS library function, do the following:

1. Decide what input parameters you must supply to the function.
2. Bundle up these parameters in objects (that is, arrays of object descriptors) in an XDS format.

Almost all data returned to you by an XDS function is enclosed in objects, which you must parse to recover the information that you want. This task is made almost automatic by a library function supplied with the companion X/Open OSI-Abstract-Data Manipulation (XOM) interface.

With XDS, the programmer has to perform a lot of call parameter management, but in other respects the interface is easy to use. The XDS functions' dependence on objects makes them easy to call, once you have the objects themselves correctly set up.

24.3 Accessing CDS Using the XDS Step-by-Step Procedure

You now know all that you need to know to work with a cell namespace through XDS. The following subsections provide a walk-through of the steps of some typical XDS/CDS operations. They describe what is involved in using XDS to access existing CDS attributes. They then describe how you can create and access new CDS entry attributes.

24.3.1 Reading and Writing Existing CDS Entry Attributes Using XDS

Suppose that you want to use XDS to read some information from the following CDS entry:

/.../C=US/O=OSF/OU=DCE/hosts/tamburlaine/self

As explained in the *OSF DCE Administration Guide*, the **/.:/hosts/***hostname***/self** entry, which is created at the time of cell configuration, contains binding information for the machine *hostname*. Since this is a simple RPC NSI entry, there is not very much in the entry that is interesting to read, but this entry is used as an example anyway as a simple demonstration.

Following are the header inclusions and general data declarations.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <xom.h>
#include <xds.h>
#include <xdsbdcp.h>
#include <xdscds.h>
```

Note that the **xom.h** and **xds.h** header files must be included in the order shown in the preceding example. Also note that the **xdscds.h** header file is brought in for the sake of **DSX_TYPELESS_RDN**. This file is where the CDS-significant OIDs are defined. The **xdsbdcp.h** file contains information necessary to the Basic Directory Contents Package, which is the basic version of the XDS interface you can use in this program.

The XDS/XOM interface defines numerous object identifier string constants, which are used to identify the many object classes, parts, and pieces (among other things) that it needs to know about. In order to make sure that these OID constants do not collide with any other constants, the interface refers to them with the string **OMP_O_** prefixed to the user-visible form; for example, **DS_C_DS_DN** becomes **OMP_O_DS_C_DS_DN** internally. In order to make application instances consistent with the internal form, use **OM_EXPORT** to import *all* XDS-defined or XOM-defined OID constants used in your application.

```
OM_EXPORT( DS_A_COUNTRY_NAME )
OM_EXPORT( DS_A_OBJECT_CLASS )
OM_EXPORT( DS_A_ORG_UNIT_NAME )
OM_EXPORT( DS_A_ORG_NAME )
OM_EXPORT( DS_C_ATTRIBUTE )
OM_EXPORT( DS_C_ATTRIBUTE_LIST )
OM_EXPORT( DS_C_AVA )
OM_EXPORT( DS_C_DS_DN )
OM_EXPORT( DS_C_DS_RDN )
OM_EXPORT( DS_C_ENTRY_INFO_SELECTION )
OM_EXPORT( DSX_TYPELESS_RDN )
  /* ...Special OID for an untyped (i.e., non-X.500) "Relative */
  /*  Distinguished Name". Defined in xdscds.h header.        */
```

A further important effect of **OM_EXPORT** is that it builds an **OM_string** structure to hold the exported Object Identifier hexadecimal string. As explained in the previous chapter, OIDs are not numeric values, but strings. Comparisons and similar operations on OIDs must access them as strings. Once an OID has been exported, you can access it using its declared name. For example, the hexadecimal string representation of **DS_C_ATTRIBUTE** is contained in **DS_C_ATTRIBUTE.elements**, and the length of this string is contained in **DS_C_ATTRIBUTE.length**.

24.3.1.1 Significance of Typed and Untyped Entry Names

Next are the static declarations for the lowest layer of objects that make up the global name (Distinguished Name) of the CDS directory entry you want to read. These lowest-level objects contain the string values for each part of the name. Remember that the first three parts of the name (excluding the global prefix **/.../**, which is not represented):

/C=US/O=OSF/OU=DCE/

constitute the cell name. In this example, assume that GDS is being used as the cell's global directory service, so the cell name is represented in X.500 format, and each part of it is typed in the object representation; for example, **DS_A_COUNTRY_NAME** is the **DS_ATTRIBUTE_TYPE** in the **Country_String_Object**. If you were using DNS, and the cell name were something like:

osf.org.dce

then the entire string **osf.org.dce** would be held in a single object whose **DS_ATTRIBUTE_TYPE** would be **DSX_TYPELESS_RDN**.

DSX_TYPELESS_RDN is a special type that marks a name piece as not residing in an X.500 namespace. If the object resides under a typed X.500 name, as is the case in the declared object structures, then it serves as a delimiter for the end of the cell name GDS looks up, and the beginning of the name that is passed to a CDS server in that cell, assuming that the cell has access to GDS; if not, such a name cannot be resolved. If the untyped portion of the name is at the beginning, as would be the case with the name:

/.../osf.org.dce/hosts/zenocrate/self

then the name is passed immediately by XDS via the local CDS (and the GDA) to DNS for resolution of the cell name. Thus, the typing of entry names determines which directory service a global directory entry name is sent to for resolution.

24.3.1.2 Static Declarations

The following are the static declarations you need:

```
/*****************************************************************/
/* Here are the objects that contain the string values for each  */
/*  part of the CDS entry's global name...                        */

static OM_descriptor    Country_String_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("US")},
 OM_NULL_DESCRIPTOR
};


static OM_descriptor    Organization_String_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("OSF")},
 OM_NULL_DESCRIPTOR
```

```
    };


    static OM_descriptor    Org_Unit_String_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_AVA),
     OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_UNIT_NAME),
     {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("DCE")},
     OM_NULL_DESCRIPTOR
    };


    static OM_descriptor    Hosts_Dir_String_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_AVA),
     OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
     {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("hosts")},
     OM_NULL_DESCRIPTOR
    };


    static OM_descriptor    Tamburlaine_Dir_String_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_AVA),
     OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
     {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("tamburlaine")},
     OM_NULL_DESCRIPTOR
    };


static OM_descriptor    Self_Entry_String_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("self")},
 OM_NULL_DESCRIPTOR
};
```

The string objects are contained by a next-higher level of objects that identify the strings as being pieces (RDNs) of a fully qualified directory entry name (DN). Thus, the **Country_RDN** object contains **Country_String_Object** as the value of its **DS_AVAS** attribute; **Organization_RDN** contains **Organization_String_Object**, and so on.

```
/*****************************************************************/
/* Here are the "Relative Distinguished Name" objects.

static OM_descriptor    Country_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Country_String_Object}},
 OM_NULL_DESCRIPTOR
};


static OM_descriptor    Organization_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Organization_String_Object}},
 OM_NULL_DESCRIPTOR
};


static OM_descriptor    Org_Unit_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Org_Unit_String_Object}},
 OM_NULL_DESCRIPTOR
};


static OM_descriptor    Hosts_Dir_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Hosts_Dir_String_Object}},
 OM_NULL_DESCRIPTOR
};


static OM_descriptor    Tamburlaine_Dir_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Tamburlaine_Dir_String_Object}},
 OM_NULL_DESCRIPTOR
};


static OM_descriptor    Self_Entry_RDN[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Self_Entry_String_Object}},
 OM_NULL_DESCRIPTOR
};
```

At the highest level, all the subobjects are gathered together in the DN object named **Full_Entry_Name_Object**.

```
/****************************************************************/

static OM_descriptor    Full_Entry_Name_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
     {DS_RDNS, OM_S_OBJECT, {0, Country_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Organization_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Org_Unit_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Hosts_Dir_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Tamburlaine_Dir_RDN}},
     {DS_RDNS, OM_S_OBJECT, {0, Self_Entry_RDN}},
     OM_NULL_DESCRIPTOR
};
```

24.3.1.3 Other Necessary Objects for ds_read( )

The **ds_read**( ) procedure takes requests in the form of a **DS_C_ENTRY_INFO_SELECTION** class object. However, if you refer to the recipe for this object class in Section 24.5, you will find that it is much simpler than the name object; it contains no subobjects, and its declaration is straightforward.

The value of the **DS_ALL_ATTRIBUTES** attribute specifies that all attributes be read from the CDS entry, which is specified in the **Full_Entry_Name_Object** variable.

Note that the term ''attribute'' is used slightly differently in CDS and XDS contexts. In XDS, attributes describe the values that can be held by various object classes; they can be thought of as ''object fields.'' In CDS, attributes describe the values that can be associated with a directory entry. The following code fragment shows the definition of a **DS_C_ENTRY_INFO_SELECTION** object.

```
static OM_descriptor    Entry_Info_Select_Object[] = {

     OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
     {DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_TRUE},
     {DS_INFO_TYPE, OM_S_ENUMERATION, DS_TYPES_AND_VALUES},
     OM_NULL_DESCRIPTOR
};
```

24.3.1.4 Miscellaneous Declarations

The following are declarations for miscellaneous variables:

```
OM_workspace            xdsWorkspace;
     /* ...will contain handle to our "workspace"  */

DS_feature featureList[] = {

     { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
     { 0 }
};
     /* ...list of service "packages" we will want from XDS  */

OM_private_object     session;
     /* ...will contain handle to a bound-to directory session  */


DS_status               dsStatus;
     /* ...status return from XDS calls  */

OM_return_code          omStatus;
     /* ...status return from XOM calls  */

OM_sint                 dummy;
     /* ...for unsupported ds_read() argument  */

OM_private_object     readResultObject;
     /* ...to receive entry information read from CDS by "ds_read()" */
```

```
OM_type I_want_entry_object[] = {DS_ENTRY, OM_NO_MORE_TYPES};
OM_type I_want_attribute_list[] = {DS_ATTRIBUTES, OM_NO_MORE_TYPES};
OM_type I_want_attribute_value[] = {DS_ATTRIBUTE_VALUES, OM_NO_MORE_TYPES};
     /* ...arrays to pass to "om_get()" to extract subobjects */
     /*  from the result object returned by "ds_read()"       */

OM_value_position number_of_descriptors;
     /* ...to hold number of attribute descriptors returned   */
     /*  by "om_get()                                         */

OM_public_object entry;
     /* ...to hold public object returned by "om_get()"       */
```

24.3.1.5 The Main Program

This section describes the main program. Three calls usually precede any use of XDS.

First, **ds_initialize**( ) is called to set up a *workspace*. A workspace is a memory area in which XDS can generate objects that will be used to pass information to the application. If the call is successful, it returns a handle that must be saved for the **ds_shutdown**( ) call. If the call is unsuccessful, it returns **NULL**, but this example does not check for errors.

```
xdsWorkspace = ds_initialize();
```

If GDS is being used as the global directory service, the service packages are specified next. Packages consist of groups of objects, together with the associated supporting interface functionality, designed to be used for some specific end. For example, to access the (X.500) Global Directory, specify **DSX_GDS_PKG**. This example uses the basic XDS service so **DS_BASIC_DIR_CONTENTS_PKG** is specified. The *featureList* parameter to **ds_version**( ) is an array, not an object, since packages are not being handled yet:

```
dsStatus = ds_version(featureList, xdsWorkspace);
```

Note that if you are *not* using GDS as your global directory service (in other words, if you are using XDS by itself), then you should *not* call **ds_version()**.

From this point on, status is returned by XDS functions via a **DS_status** variable. **DS_status** is a handle to a private object, whose value is **DS_SUCCESS** (that is, **NULL**) if the call was successful. If something went wrong, the information in the (possibly complex) private error object has to be analyzed through calls to **om_get()**, which is one of the general-purpose object management functions that belongs to XDS's companion interface XOM. Usage of **om_get()** is demonstrated later on in this program, but return status is not checked in this example.

The third necessary call is to **ds_bind()**. This call brings up the directory service, which binds to a Directory System Agent (DSA), the GDS server, through a Directory User Agent (DUA), the GDS client. The **DS_DEFAULT_SESSION** parameter calls for a default session. The alternative is to build and fill out your own **DS_C_SESSION** object, specifying such things as DSA addresses, and pass that. The default is used in this example:

```
dsStatus = ds_bind(DS_DEFAULT_SESSION, xdsWorkspace, &session);
```

24.3.1.6 Reading a CDS Attribute

At this point, you can read a set of object attributes from the cell namespace entry. Call **ds_read()** with the two objects that specify the entry to be read and the specific entry attribute you want:

```
dsStatus = ds_read(session, DS_DEFAULT_CONTEXT, Full_Entry_Name_Object,
                   Entry_Info_Select_Object, &readResultObject, &dummy);
```

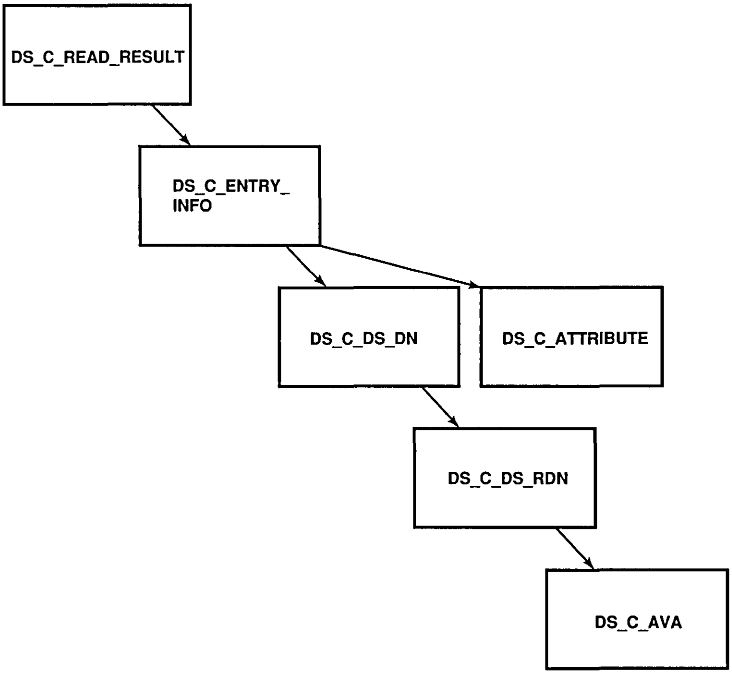
The **DS_DEFAULT_CONTEXT** parameter could be substituted with a **DS_C_CONTEXT** object, which would typically be reused during a series of related XDS calls. This object specifies and records how GDS should perform the operation, how much progress has been made in resolving a name, and so on.

If the call succeeds, the private object **readResultObject** contains a series of **DS_C_ATTRIBUTE** subobjects, one for each attribute read from the cell name entry. A complete recipe for the **DS_C_READ_RESULT** object can be found in Chapter 30, but the following is a skeletal outline of the object's structure:

```
DS_C_READ_RESULT
        DS_ENTRY: object(DS_C_ENTRY_INFO)
        DS_ALIAS_DEREFERENCED: OM_S_BOOLEAN
        DS_PERFORMER: object(DS_C_NAME)

    DS_C_ENTRY_INFO
            DS_FROM_ENTRY: OM_S_BOOLEAN
            DS_OBJECT_NAME: object(DS_C_NAME)
            DS_ATTRIBUTES: one or more object(DS_C_ATTRIBUTE)

        DS_C_NAME == DS_C_DS_DN
                DS_RDNS: object(DS_C_DS_RDN)

            DS_C_DS_RDN
                    DS_AVAS: object(DS_C_AVA)

                DS_C_AVA
                        DS_ATTRIBUTE_TYPE: OID string
                        DS_ATTRIBUTE_VALUES: anything

        DS_C_ATTRIBUTE —one for each attribute read
       DS_ATTRIBUTE_TYPE: OID string
                DS_ATTRIBUTE_VALUES: anything

        DS_C_ATTRIBUTE
                DS_ATTRIBUTE_TYPE: OID string
                DS_ATTRIBUTE_VALUES: anything
```

Figure 24-6 illustrates the general object structure of a **DS_C_READ_RESULT**, showing only the object-valued attributes, and only one **DS_C_ATTRIBUTE** subobject.

Figure 24-6. The DS_C_READ_RESULT Object Structure



24.3.1.7 Handling the Result Object

The next goal is to extract the instances of the **DS_C_ATTRIBUTE** subsubclass, one for each attribute read, from the returned object. The first step is to make a public copy of **readResultObject**, which is a *private* object, and therefore does not allow access to the object descriptors themselves. Using the XOM **om_get( )** function, you can make a public copy of **readResultObject**, and at the same time specify that only the relevant parts of it be preserved in the copy. Then with a couple of calls to **om_get( )**, you can reduce the object to manageable size, leaving a superobject whose immediate subobjects are fairly easily accessed.

The **om_get( )** function takes as its third input parameter an **OM_type_list**, which is an array of **OM_type**. Possible parameters are **DS_ENTRY**, **DS_ATTRIBUTES**, **DS_ATTRIBUTE_VALUES**, and anything that can legitimately appear in an object descriptor's **type** field. The types specified in this parameter are interpreted according to the options specified in the preceding parameter. For example, the relevent attribute from the read result is **DS_ENTRY**. It contains the **DS_C_ENTRY_INFO** object, which in turn contains the **DS_C_ATTRIBUTE** objects. The **DS_C_ATTRIBUTE** objects contain the data read from the cell directory name entry. Therefore, you should specify the **OM_EXCLUDE_ALL_BUT_THESE_TYPES** option, which has the effect of excluding everything but the contents of the object's **DS_ENTRY** type attribute.

The **OM_EXCLUDE_SUBOBJECTS** option is also ORed into the parameter. Why would you not preserve the subobjects of **DS_C_ENTRY_INFO**? Because **om_get()** works only on private, not on public, objects. If you were to use **om_get()** on the entire object substructure, you would not be able to continue getting the subobjects, and instead you would have to follow the object pointers down to the **DS_C_ATTRIBUTE**s. However, when **om_get()** excludes subobjects from a copy, it does not really leave them out; it merely leaves the subobjects private, with a handle to the private objects where pointers would have been. This allows you to continue to call **om_get()** as long as there are more subobjects.

The following is the first call:

```
/* The DS_C_READ_RESULT object that ds_read() returns has    */
/*  one subobject, DS_C_ENTRY_INFO; it in turn has two sub- */
/*  objects, i.e. a DS_C_NAME which holds the object's di-  */
/*  stinguished name (which we don't care about here), and  */
/*  a DS_C_ATTRIBUTE which contains the attribute info we    */
/*  read; that one we want. So we climb down to it...       */
/* This om_get() will "return" the entry-info object...      */

omStatus = om_get(readResultObject,
                  OM_EXCLUDE_ALL_BUT_THESE_TYPES +
                  OM_EXCLUDE_SUBOBJECTS,
                  I_want_entry_object,
                  OM_FALSE,
                  OM_ALL_VALUES,
                  OM_ALL_VALUES,
                  &entry,
                  &number_of_descriptors);
```

The **number_of_descriptors** parameter contains the number of attribute descriptors returned in the public copy, not in any excluded subobjects.

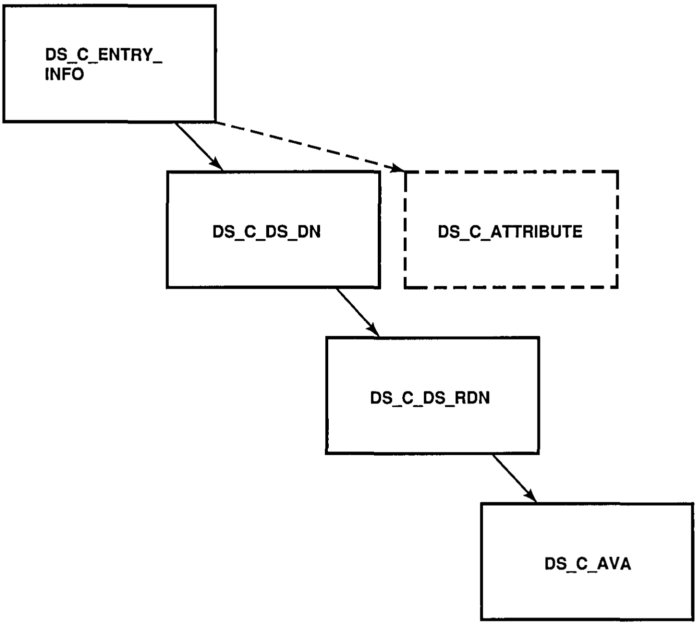
If an XOM function is successful, it returns an **OM_SUCCESS** code. Unsuccessful calls to XOM functions do not return error objects, but rather return simple error codes. The interface assumes that if the XOM function does not accept your object, then you will not be able to get much information from any further objects. The return status is not checked in this example.

The return parameter **entry** should now contain a pointer to the **DS_C_ENTRY_INFO** object with the following immediate structure. (The number of instances of **DS_ATTRIBUTES** depends on the number of attributes read from the entry.)

```
DS_C_ENTRY_INFO
    DS_FROM_ENTRY: OM_S_BOOLEAN
    DS_OBJECT_NAME: object(DS_C_NAME)
    DS_ATTRIBUTES: object(DS_C_ATTRIBUTE)
                                DS_C_ATTRIBUTE
                                    DS_ATTRIBUTE_TYPE: OID string
                                    DS_ATTRIBUTE_VALUES: anything

    DS_ATTRIBUTES: object(DS_C_ATTRIBUTE)
                         object(DS_C_ATTRIBUTE)
                                DS_C_ATTRIBUTE
                                    DS_ATTRIBUTE_TYPE: OID string
                                    DS_ATTRIBUTE_VALUES: anything
```

The italics indicate private subobjects. Figure 24-7 shows the **DS_C_ENTRY_INFO** object. Only one instance of a **DS_C_ATTRIBUTE** subobject is shown in the figure; usually there are several such subobjects, all at the same level, each containing information about one of the attributes read from the entry. These subobjects are represented in **DS_C_ENTRY_INFO** as a series of descriptors of type **DS_ATTRIBUTES**, each of which has as its value a separate **DS_C_ATTRIBUTE** subobject.

Figure 24–7. The DS_ENTRY_INFO Object Structure



Now extract the separate attribute values of the entry that was read. These were returned as separate object values of **DS_ATTRIBUTES**; each one has an object class of **DS_C_ATTRIBUTE**. To return any one of these subobjects, a second call to **om_get**( ) is necessary, as follows.

```
/* The second om_get() returns one selected sub-object      */
/*  from the DS_C_ENTRY_INFO subobject we just got. The     */
/*  contents of "entry" as we enter this call is the pri-   */
/*  vate subobject which is the value of DS_ATTRIBUTES. If  */
/*  we were to make the following call with the             */
/*  OM_EXCLUDE_SUBOBJECTS and without the                   */
/*  OM_EXCLUDE_ALL_BUT_THESE_VALUES flags, we would get     */
/*  back an object consisting of six private subobjects,    */
/*  one for each of the attributes returned. Note the val-  */
/*  ues for initial and limiting position: "2" specifies    */
/*  that we want only the third DS_C_ATTRIBUTE subobject    */
/*  to be gotten (the subobjects are numbered from 0, not   */
/*  from one), and the "3" specifies that we want no more   */
/*  than that-- in other words, the limiting value must al- */
/*  ways be one more than the initial value if the latter   */
/*  is to have any effect. OM_EXCLUDE_ALL_BUT_THESE_VALUES  */
/*  is likewise required for the initial and limiting val-  */
/*  ues to have any effect...                               */

omStatus = om_get(entry->value.object.object,
                  OM_EXCLUDE_ALL_BUT_THESE_TYPES
                  + OM_EXCLUDE_SUBOBJECTS
                  + OM_EXCLUDE_ALL_BUT_THESE_VALUES,
                  I_want_attribute_list,
                  OM_FALSE,
                  ((OM_value_position) 2),
                  ((OM_value_position) 3),
                  &entry,
                  &number_of_descriptors);
```

Note the value that is passed as the first parameter. Since **om_get()** does not work on public objects, pass it the handle of the private subobject explicitly. To do this you have to know the arrangement of the descriptor's value union, which is defined in **xom.h**.

24.3.1.8 Representation of Object Values

The following is the layout of the **object** field in a descriptor's **value** union:

```
typedef struct {
        OM_uint32       padding;
        OM_object       object;
} OM_padded_object;
```

The following is the layout of the **value** union itself:

```
typedef union OM_value_union {
        OM_string       string;
        OM_boolean      boolean;
        OM_enumeration  enumeration;
        OM_integer      integer;
        OM_padded_object        object;
} OM_value;
```

The following is the layout of the descriptor itself:

```
typedef struct OM_descriptor_struct {
        OM_type                 type;
        OM_syntax               syntax;
        union OM_value_union    value;
} OM_descriptor;
```

Thus, if **entry** is a pointer to the **DS_C_ENTRY_INFO** object, then **entry->value.object.object** is the private handle to the **DS_C_ATTRIBUTE** object that you want next.

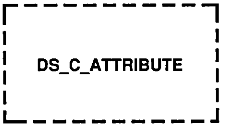
24.3.1.9 Extracting an Attribute Value

The last call yielded one separate **DS_C_ATTRIBUTE** subsubobject from the original returned result object:

```
DS_C_ATTRIBUTE
        DS_ATTRIBUTE_TYPE: OID string
        DS_ATTRIBUTE_VALUES: anything
```

Figure 24-8 illustrates what is left.

Figure 24–8. The DS_C_ATTRIBUTE Object Structure



A final call to **om_get( )** returns the single object descriptor that contains the actual value of the single attribute you selected from the returned object:

```
omStatus = om_get(entry->value.object.object,
                  OM_EXCLUDE_ALL_BUT_THESE_TYPES,
                  I_want_attribute_value,
                  OM_FALSE,
                  OM_ALL_VALUES,
                  OM_ALL_VALUES,
                  &entry,
                  &number_of_descriptors);
```

At this point, the value of **entry** is the base address of an object descriptor whose **entry->type** is **DS_ATTRIBUTE_VALUES**. Depending on the value found in **entry->syntax**, the value of the attribute can be read from **entry->value.string**, **entry->value.integer**, **entry->value.boolean**, or **entry->value.enumeration**. For example, suppose the value of **entry->syntax** is **OM_S_OCTET_STRING**. The attribute value, represented as an octet string (*not* terminated by a **NULL**), is found in **entry->value.string.elements**; its length is found in **entry->value.string.length**.

You can check any attribute value against the value you get from the **cdscp** command by entering:

cdscp show object /.:/hosts/tamburlaine/self

For further information on **cdscp**, see the *OSF DCE Administration Reference*.

Note that you can always call **om_get()** to get the *entire* returned object from an XDS call. This yields a full structure of object descriptors that you can manipulate like any other data structure. To do this with the **ds_read()** return object would have required the following call:

```
/* make a public copy of ENTIRE object...   */

omStatus = om_get(readResultObject,
                  OM_NO_EXCLUSIONS,
                  ((OM_type_list) 0),
                  OM_FALSE,
                  ((OM_value_position) 0),
                  ((OM_value_position) 0),
                  &entry,
                  &number_of_descriptors);
```

At the end of every XDS session you have to unbind from the GDS, and then deallocate the XDS and XOM structures and other storage. You must also explicitly deallocate any service-generated objects, whether public or private, with calls to **om_delete()**, as follows:

```
/* delete service-generated public or private objects... */

omStatus = om_delete(readResultObject);
omStatus = om_delete(entry);

/* unbind from the GDS...  */
dsStatus = ds_unbind(session);

/* close down the workspace... */
dsStatus = ds_shutdown(xdsWorkspace);

exit();
```

24.3.2 Creating New CDS Entry Attributes

The following subsections provide the procedure and some code examples for creating new CDS entry attributes.

24.3.2.1 Procedure for Creating New Attributes

To create new attributes of your own on cell namespace entries, you must do the following:

1. Allocate a new ISO Object Identifier (OID) for the new attribute. For information on how to do this, see Chapter 2 of this guide, and the *OSF DCE Administration Guide*.

2. Enter the new attribute's name and OID in the **/.:/opt/dcelocal/etc/cds_attributes** file. This text file contains OID-to-readable string mappings that are used, for example, by the CDS administration command **cdscp** when it displays CDS entry attributes. Each entry also gives a syntax for reading the information in the entry itself. This should be congruent with the format of the data you intend to write in the attribute. For more information about the **cds_attributes** file, see the *OSF DCE Administration Guide*.

3. In the **xdscds.h** header file, define an appropriate OID string constant to represent the new attribute.

   For example, the following shows the **xdscds.h** definition for the CDS **CDS_Class** attribute:

   ```
   #define OMP_O_DSX_A_CDS_Class       "\x2B\x16\x01\x03\x0F"
   ```

   Note the XDS internal form of the name. This is what **DSX_A_CDS_Class** looks like when it has been exported using **OM_EXPORT** in an application, as all OIDs must be. Thus, if you wanted to create a CDS attribute called **CDS_Brave_New_Attrib**, you would obtain an OID from your administrator and add the following line to **xdscds.h**:

   ```
   #define OMP_O_DSX_A_CDS_Brave_New_Attrib "your_OID"
   ```

4. In an application, call the XDS **ds_modify_entry()** routine to add the attribute to the cell namespace entry of your choice.

24.3.2.2 Coding Examples

In the following code fragments a set of declarations similar to those in the previous examples is assumed.

The **ds_modify_entry()** function, which is called to add new attributes to an entry or to write new values into existing attributes, requires a **DS_C_ENTRY_MOD_LIST** input object whose contents specify the attributes and values to be written to the entry. The name, as always, is specified in a **DS_C_DS_DN** object.

The following is a static declaration of such a list, which consists of two attributes:

```
static OM_descriptor    Entry_Modification_Object_1[] = {
      OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD),
      OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_Brave_New_Attrib),
      {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING,
              OM_STRING("O brave new attribute")},
      {DS_MOD_TYPE, OM_S_ENUMERATION, DS_ADD_ATTRIBUTE},
      OM_NULL_DESCRIPTOR
};

static OM_descriptor    Entry_Modification_Object_2[] = {
      OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD),
      OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_Class),
      {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("Miscellaneous")},
      {DS_MOD_TYPE, OM_S_ENUMERATION, DS_ADD_ATTRIBUTE},
      OM_NULL_DESCRIPTOR
};


static OM_descriptor    Entry_Modification_List_Object[] = {
      OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD_LIST),
      {DS_CHANGES, OM_S_OBJECT, {0, Entry_Modification_Object_1}},
      {DS_CHANGES, OM_S_OBJECT, {0, Entry_Modification_Object_2}},
      OM_NULL_DESCRIPTOR
};
```

A full description of this object can be found in Section 24.5. There could be any number of additional attribute changes in the list; this would mean additional **DS_C_ENTRY_MOD** objects declared, and an additional **DS_CHANGES** descriptor declared and initialized in the **DS_C_ENTRY_MOD_LIST** object.

With the **DS_C_ENTRY_MOD_LIST** class object having been declared as shown previously, the following code fragment illustrates how to call XDS to write a new attribute value (actually two new values since two attributes are contained in the list object). Note that any of the attributes may be new, although the entry itself must already exist.

```
dsStatus = ds_modify_entry(session,  /* Directory session from "ds_bind()"  */
                           DS_DEFAULT_CONTEXT, /* Usual directory context   */
                           Full_Entry_Name_Object, /* Entry name object     */
                           Entry_Modification_List_Object, /* Entry Modifi- */
                                                   /*  cation object        */
                           &dummy);                /* Unsupported argument  */
```

If the entire entry is new, you must call **ds_add_entry**( ). This function requires an input object of class **DS_C_ATTRIBUTE_LIST**, whose contents specify the attributes (and values) to be attached to the new entry. Following is the static declaration for an attribute list that contains three attributes:

```
static OM_descriptor    Class_Attribute_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
     OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_Class),
     {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("Printer")},
     OM_NULL_DESCRIPTOR
};

static OM_descriptor    ClassVersion_Attribute_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
     OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_ClassVersion),
     {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("1.0")},
     OM_NULL_DESCRIPTOR
};

static OM_descriptor    My_Own_Attribute_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
     OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_My_OwnAttribute),
     {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("zorro")},
     OM_NULL_DESCRIPTOR
};
```

```
static OM_descriptor    Attribute_List_Object[] = {
     OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
     {DS_ATTRIBUTES, OM_S_OBJECT, {0, Class_Attribute_Object}},
     {DS_ATTRIBUTES, OM_S_OBJECT, {0, ClassVersion_Attribute_Object}},
     {DS_ATTRIBUTES, OM_S_OBJECT, {0, My_Own_Attribute_Object}},
     OM_NULL_DESCRIPTOR
};
```

The **ds_add_entry**( ) function also requires a **DS_C_DS_DN** class object containing the new entry's full name, for example:

/.../osf.org.dce/subsys/doc/my_book

where every member of the name exists except for the last one, **my_book**. Assuming that **Full_Entry_Name_Object** is a **DS_C_DS_DN** object, the following code shows what the call would look like:

```
dsStatus = ds_add_entry(session,      /* Directory session from "ds_bind()"  */
                        DS_DEFAULT_CONTEXT,  /* Usual directory context      */
                        Full_Entry_Name_Object, /* Name of new entry         */
                        Attribute_List_Object, /* Attributes to be attached */
                                     /*  to new entry, with values           */
                        &dummy);              /* Unsupported argument        */
```

24.4 Object-Handling Techniques

The following subsections describe the use of XOM and discuss dynamic object creation.

24.4.1 Using XOM to Access CDS

The following code fragments demonstrate an alternative way to set up the entry modification object for a **ds_modify_entry( )** call, mainly for the sake of showing how the **om_put( )** and **om_write( )** functions are used.

The following technique is used to initialize the modification object:

1. The **om_create( )** function is called to generate a private object of a specified class.
2. The **om_put( )** function is called to copy statically declared attributes into a declared private object.
3. The **om_write( )** function is called to write the value string, which is to be assigned to the attribute, into the private object.
4. The **om_get( )** function is called to make the private object public.
5. The object is now public, and its address is inserted into the **DS_C_ENTRY_MOD_LIST** object's **DS_CHANGES** attribute.

The following new declarations are necessary:

```
OM_private_object newAttributeMod_priv;
     /* ...handle to a private object to "om_put()" to           */

OM_public_object newAttributeMod_pub;
     /* ...to hold public object from "om_get()"                 */

OM_type types_to_include[] = {DS_ATTRIBUTE_TYPE, DS_ATTRIBUTE_VALUES,
                              DS_MOD_TYPE, OM_NO_MORE_TYPES};
     /* ...i.e., all attribute values of the Entry Modification  */
     /*  object. For "om_put()" and "om_get()"                   */

char *my_string = "O brave new attribute";
     /* ...value I want to write into attribute                  */

OM_value_position number_of_descriptors;
     /* ...to hold value returned by "om_get()"                  */
```

First, use XOM to generate a private object of the desired class:

```
omStatus = om_create(DS_C_ENTRY_MOD,  /* Class of object                */
                     OM_TRUE,    /* Initialize attributes per defaults  */
                     xdsWorkspace,  /* Our workspace handle             */
                     &newAttributeMod_priv);  /* Created object handle */
```

Next, copy the public object's attributes into the private object:

```
omStatus = om_put(newAttributeMod_priv, /* Private object to copy      */
                                        /*  attributes into            */
                 OM_REPLACE_ALL, /* Which attributes to replace in    */
                                /*  destination object                 */
                 Entry_Modification_Object, /* Source object to copy  */
                                           /*  attributes from         */
                 types_to_include, /* List of attribute types we want */
                                  /*  copied                           */
                 0, 0); /* Start-stop index for multivalued attri-    */
                       /*  butes; ignored with OM_REPLACE_ALL          */
```

Since **om_put**( ) ignores the class of the source object (the object from which attributes are being copied), it is not necessary to declare class descriptors for the source objects. In other words, the static declarations could have omitted the **OM_CLASS** initializations if this technique were being used, for example:

```
static OM_descriptor    Entry_Modification_Object_2[] = {
/*     OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD),           */
/*     Not needed for "om_put()"...                     */

     OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_Class),
     {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("Miscellaneous")},
     {DS_MOD_TYPE, OM_S_ENUMERATION, DS_ADD_ATTRIBUTE},
     OM_NULL_DESCRIPTOR
};
```

The **OM_CLASS** was already properly initialized by **om_create( )**.

Next, write the attribute value string into the private object:

```
omStatus = om_write(newAttributeMod_priv,/* Private object to write to */
                    DS_ATTRIBUTE_VALUES, /* Attribute type whose value */
                                         /*  we're writing             */
                    0, /* Descriptor index if attribute is multivalued */
                    OM_S_PRINTABLE_STRING, /* Syntax of value          */
                    0, /* Offset in source string to write from        */
                    my_string); /* Source string to write from         */
```

Now make the whole thing public again:

```
omStatus = om_get(newAttributeMod_priv, /* Private object to get       */
                  0,                    /* Get everything              */
                  types_to_include,     /* All attribute types         */
                  0,                    /* Unsupported argument        */
                  0, 0, /* Start-stop descriptor index for multi-val-  */
                        /*  ued attributes; ignored in this case       */
                  &newAttributeMod_pub, /* Pointer to returned copy    */
                  &number_of_descriptors); /* Number of attribute      */
                                           /*  descriptors returned    */
```

Finally, insert the address of the subobject into its superobject:

```
Entry_Modification_List_Object[1].value.object.object = newAttributeMod_pub;
```

24.4.2 Dynamic Creation of Objects

Objects can be completely dynamically allocated and initialized; however, you have to implement the routines to do this yourself. The examples in this section are code fragments; for complete examples, see Chapter 28.

Initialization of object structures can be automated by declaring macros or functions to do this. For example, the following macro initializes one object descriptor with a full set of appropriate values:

```
    /* Put a C-style (NULL-terminated) string into an object, and    */
    /* set all the other descriptor fields to requested values...   */
#define FILL_OMD_STRING( desc, index, typ, syntx, val ) \
        desc[index].type = typ; \
        desc[index].syntax = syntx; \
        desc[index].value.string.length = (OM_element_position)strlen(val); \
        desc[index].value.string.elements = val;
```

When generating objects, use **malloc**( ) to allocate space for the number of objects desired, and then use macros (or functions) such as the preceding one to initialize the descriptors. The following code fragment shows how this can be done for the top-level object of a **DS_C_DS_DN**, such as the one described near the beginning of this chapter. Recall that the **DS_C_DS_DN** has a separate **DS_RDNS** descriptor for each name piece in the full name.

```
/* Calculate number of "DS_RDNS" attributes there should be... */
numberOfPieces = number_of_name_pieces;

/* Allocate space for that many descriptors, plus one for the  */
/*  object class at the front, and a NULL descriptor at the    */
/*  back...                                                     */
Name_Object = (OM_object)malloc((numberOfPieces + 2) * sizeof(OM_descriptor));
if(Name_Object == NULL)                      /* "malloc()" failed */
return OM_MEMORY_INSUFFICIENT;

/* Initialize it as a DS_C_DS_DN object by placing that class  */
/*  identifier in the first position...                         */
```

```
FILL_OMD_XOM_STRING(Name_Object, 0, OM_CLASS,
    OM_S_OBJECT_IDENTIFIER_STRING, DS_C_DS_DN)
```

Note that all of these steps would have to be repeated for each of the **DS_C_DS_RDN** objects required as attribute values of the **DS_C_DS_DN**. Then a tier of **DS_C_AVA** objects would have to be created in the same way, since each of the **DS_C_DS_RDN**s requires one of them as *its* attribute value.

You could now use **om_create**( ) and **om_put**( ) to generate a private copy of this object, if so desired.

The application is responsible for managing the memory it allocates for such dynamic object creation.

24.5 XDS/CDS Object Recipes

The following subsections contain shorthand for object classes. For example, if you look at the reference pages for the **ds_...**( ) functions, you will see that an object of class **DS_C_NAME** is required to hold entry names you want to pass to the call, *not* **DS_C_DS_DN** as is stated in this chapter. However, **DS_C_NAME** is in fact an abstract class with only one subclass, **DS_C_DS_DN**, so in this chapter, **DS_C_DS_DN** is used.

24.5.1 Input XDS/CDS Object Recipes

In general, the objects you work with in an XDS/CDS application fall into two categories:

- Objects you have to supply as *input parameters* to XDS functions
- Objects returned to you as *output* by XDS functions

This section describes only the first category, since you have to construct these input objects yourself.

Table 24-1 shows XDS functions and the objects given to them as input parameters.

Only items significant to CDS are listed in the table. **DS_C_SESSION** and **DS_C_CONTEXT** are ignored. **DS_C_SESSION** is returned by **ds_bind**( ), which usually receives the **DS_DEFAULT_SESSION** constant as input. **DS_C_CONTEXT** is usually substituted by the **DS_DEFAULT_CONTEXT** constant.

Note: **DS_C_NAME** is an abstract class that has the single subclass **DS_C_DS_DN**. Therefore, **DS_C_NAME** is practically the same thing as **DS_C_DS_DN**.

Table 24–1. Directory Service Functions with their Required Input Objects

Function	**Input Object**
ds_add_entry( )	**DS_C_NAME** **DS_C_ATTRIBUTE_LIST**
ds_bind( )	None
ds_compare( )	**DS_C_NAME** **DS_C_AVA**
ds_initialize( )	None
ds_list( )	**DS_C_NAME**
ds_modify_entry( )	**DS_C_NAME** **DS_C_ENTRY_MOD_LIST**
ds_read( )	**DS_C_NAME** **DS_C_ENTRY_INFO_SELECTION**
ds_remove_entry( )	**DS_C_NAME**
ds_shutdown( )	None
ds_unbind( )	None
ds_version( )	None

24.5.2 Input Object Classes for XDS/CDS Operations

The following subsections contain information about all the object types required as input to any of the XDS functions that can be used to access the CDS. In order to use these functions successfully, you must be able to construct and modify the objects that the functions expect as their input parameters. XDS functions require most of their input parameters to be wrapped in a nested series of data structures that represent objects, and these functions deliver their output returns to callers in the same object form.

Objects that are returned to you by the interface are not difficult to manipulate because the **om_get( )** function allows you to go through them and retrieve only the value parts you are interested in, and discard the parts of data structures you are not interested in. Some examples of how to do this are given in Section 26.7.2.2. However, any objects you are required to supply as *input* to an XDS or XOM function are another matter: you must build and initialize these object structures yourself.

The basics of object building have already been explained earlier in this chapter. Each object described in the following subsections is accompanied by a static declaration in C of a very simple instance of that object class. The objects in an application are usually built dynamically (this technique was demonstrated earlier in this chapter). The static declarations that follow give a simple example of what the objects look like.

An object's properties, such as what sort of values it can hold, how many of them it can hold, and so on, are determined by the *class* the object belongs to. Each class consists of one or more *attributes* that an object can have. The attributes hold whatever values the object contains. Thus, the objects are data structures that all look the same (and can be handled in the same way) from the outside, but whose specific data fields are determined by the class each object belongs to. At the abstract level, objects consist of attributes, just as structures consist fields.

24.5.2.1 XDS/CDS Object Types

Following is a list of all the object types that are described in the following subsections. Most of these objects are object structures; that is, compounds consisting of superobjects that contain subobjects as some of their values. These latter may in turn contain other objects, and so on. Subobjects are indicated by indentation. A **DS_C_DS_DN** object contains at least one **DS_C_DS_RDN** object, and each of the latter contains one **DS_C_AVA** object. Note that subobjects can, and often do, exist by themselves, depending on what object class is called for by a given function.

This list contains all the possible kinds of objects that can be required as input for any XDS/CDS operation:

- **DS_C_ATTRIBUTE_LIST**
  - **DS_C_ATTRIBUTE**
- **DS_C_DS_DN**
  - **DS_C_DS_RDN**
    - **DS_C_AVA**
- **DS_C_ENTRY_MOD_LIST**
  - **DS_C_ENTRY_MOD**
- **DS_C_ENTRY_INFO_SELECTION**

In each section, information is provided for the described object's attributes. All of its attributes are listed.

The illustrations in the following sections can be compared to the same object classes' tabular definitions in Chapter 30.

24.5.2.2 The DS_C_ATTRIBUTE_LIST Object

A **DS_C_ATTRIBUTE_LIST** class object is required as input to **ds_add_entry**( ). The object contains a list of the directory attributes you want associated with the entry that is to be added.

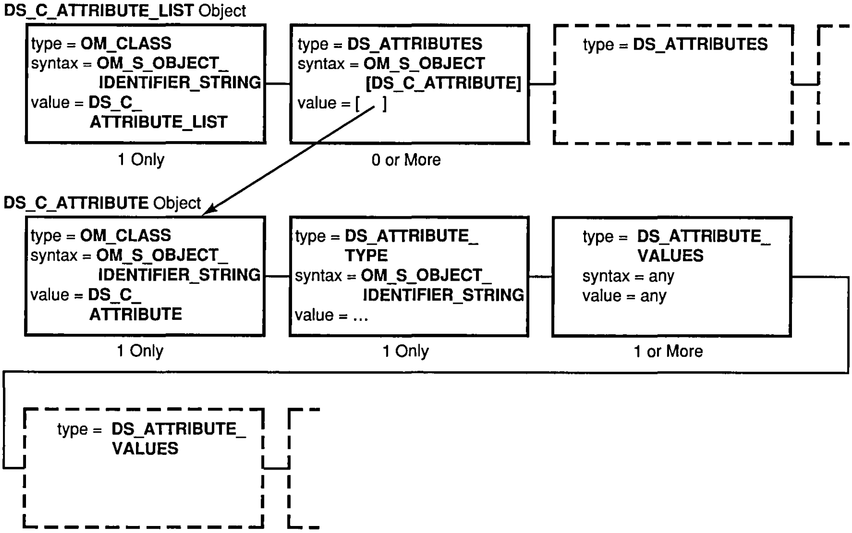
Its general structure is as follows:

- Attribute List class type attribute
- Zero or more Attribute objects:
  - Attribute class type attribute
  - Attribute Type attribute
  - Zero or more Attribute Value(s)

Thus, a **DS_C_ATTRIBUTE_LIST** object containing one attribute consists of two object descriptor arrays because each additional attribute in the list requires an additional descriptor array to represent it. The subobject arrays' names (that is, addresses) are the contents of the value fields in the **DS_ATTRIBUTES** object descriptors.

Figure 24-9 shows the attributes of the **DS_C_ATTRIBUTE_LIST** object.

Figure 24–9. The DS_C_ATTRIBUTE_LIST Object


DS_C_ATTRIBUTE_LIST Object
type = OM_CLASS
syntax = OM_S_OBJECT_IDENTIFIER_STRING
value = DS_C_ATTRIBUTE_LIST
1 Only
type = DS_ATTRIBUTES
syntax = OM_S_OBJECT [DS_C_ATTRIBUTE]
value = [ ]
0 or More
type = DS_ATTRIBUTES
DS_C_ATTRIBUTE Object
type = OM_CLASS
syntax = OM_S_OBJECT_IDENTIFIER_STRING
value = DS_C_ATTRIBUTE
1 Only
type = DS_ATTRIBUTE_TYPE
syntax = OM_S_OBJECT_IDENTIFIER_STRING
value = ...
1 Only
type = DS_ATTRIBUTE_VALUES
syntax = any
value = any
1 or More
type = DS_ATTRIBUTE_VALUES

- **OM_CLASS**
  The value of this attribute is an OID string that identifies the object's class; its value is always **DS_C_ATTRIBUTE_LIST**.

- **DS_ATTRIBUTES**
  This is an attribute whose value is another object of class **DS_C_ATTRIBUTE** (see Section 24.5.2.3). The attribute is defined by a separate array of object descriptors whose base address is the value of the **DS_ATTRIBUTES** attribute. Note that there can be any number of instances of this attribute, and therefore any number of subobjects.

24.5.2.3 The DS_C_ATTRIBUTE Object

An object of this class can be an attribute of a **DS_C_ATTRIBUTE_LIST** object (see Section 24.5.2.2).

- **OM_CLASS**
  The value of this attribute is an OID string that identifies the object's class; its value is always **DS_C_ATTRIBUTE.**
- **DS_ATTRIBUTE_TYPE**
  The value of this attribute, which is an OID string, identifies the directory attribute whose value is contained in this object.
- **DS_ATTRIBUTE_VALUES**
  These are the actual values for the directory attribute represented by this **DS_C_ATTRIBUTE** object. Both the value syntax and the number of values depend on what directory attribute this is; that is, they depend on the value of **DS_ATTRIBUTE_VALUE.**

24.5.2.4 Example Definition of a DS_C_ATTRIBUTE_LIST Object

The following code fragment is a definition of a **DS_C_ATTRIBUTE_LIST** object.

```
static OM_descriptor    Single_Attribute_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_CDS_Class),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("Printer")},
 OM_NULL_DESCRIPTOR
};


static OM_descriptor    Attribute_List_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, Single_Attribute_Object}},
 OM_NULL_DESCRIPTOR
};
```

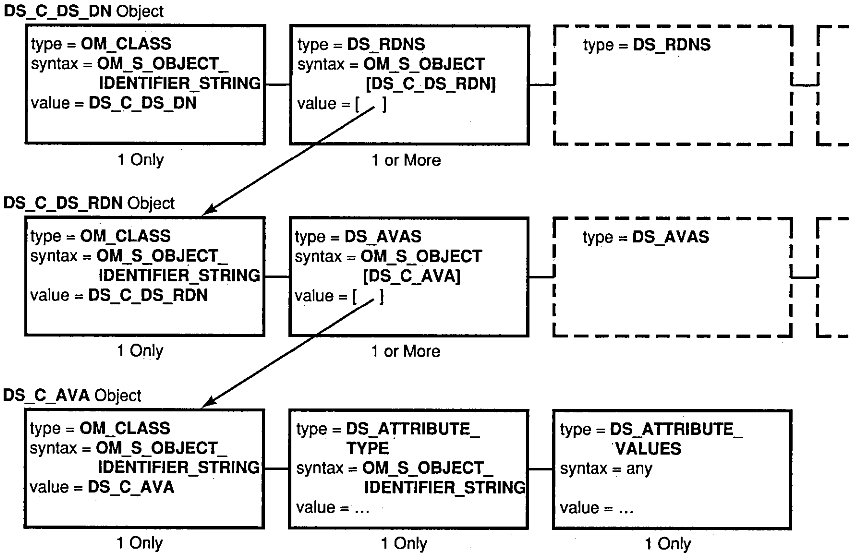
24.5.2.5 The DS_C_DS_DN Object

DS_C_DS_DN class objects are used to hold the full names of directory entries (Distinguished Names). You need an object of this class to pass directory entry names to the following XDS functions:

- **ds_add_entry( )**
- **ds_compare( )**
- **ds_list( )**
- **ds_modify_entry( )**
- **ds_read( )**
- **ds_remove_entry( )**

Figure 24-10 shows the attributes of a **DS_C_DS_DN** object.

Figure 24–10. DS_C_DS_DN Object Attributes



- **OM_CLASS**
  The value of this attribute is an OID string that identifies the object's class; its value is **DS_C_DS_DN**.

- **DS_RDNS**
  This is an attribute whose value is another object of class **DS_C_DS_RDN** (see Section 24.5.2.6). The **DS_C_DS_RDN** object is defined by a separate array of object descriptors whose base address is the value of the **DS_RDNS** attribute.

  There are as many **DS_RDNS** attributes in a **DS_C_DS_DN** object as there are separate name components in the full directory entry name. For example, to represent the following CDS entry name:

  /.../C=US/O=OSF/OU=DCE/hosts/brazil/self

  a total of six instances of the **DS_RDNS** attribute are required in the **DS_C_DS_DN** object. The **/.../** (global root prefix) is not represented. This means that another six object descriptor arrays are required to hold the Relative Distinguished Name objects, as well as six object descriptors in the present object, one to hold (as the value of a **DS_RDNS** attribute) a pointer to each array.

  Note that the order of these **DS_RDNS** attributes is significant; that is, the first **DS_RDNS** should contain as its value a pointer to the array representing the **C=US** part of the name; the next **DS_RDNS** should contain as its value a pointer to the array representing the **O=OSF** part, and so on. The root part of the name is not represented at all.

24.5.2.6 The DS_C_DS_RDN Object

DS_C_DS_RDN class objects are required as values for the **DS_RDNS** attributes of **DS_C_DS_DN** objects. (For an illustration of its structure, see Figure 24-10.) **RDN** refers to the X.500 term Relative Distinguished Name that is used to signify a part of a full entry name. Separate objects of this class are not usually required as input to XDS functions.

The standard permits multiple AVAs in an RDN, but the DCE Directory and XDS API restrict an RDN to one AVA.

- **OM_CLASS**
  The value of this attribute is an OID string that identifies the object's class; its value is always **DS_C_DS_RDN**.

- **DS_AVAS**
  This is an attribute whose value is yet another object of class **DS_C_AVA** (see Section 24.5.2.7). The **DS_C_AVA** object is defined by a separate array of object descriptors whose base address is the value of the **DS_AVAS** attribute.

  Note that there can only be one instance of this attribute in the **DS_C_RDN** object. The object descriptor array describing this object always consists of three object descriptor structures: the first describes the object's class, the second describes the **DS_AVAS** attribute, and the third descriptor is the terminating **NULL**.

24.5.2.7 The DS_C_AVA Object

The **DS_C_AVA** class object is used to hold an actual value. The value is usually in the form of one of the many different XOM string types. (For an illustration of its structure, see Figure 24-10.)

In calls to **ds_compare**( ), an object of this type is required to hold the type and value of the attribute that you want compared with those in the entry you specify. It holds the type and value in a separate **DS_C_DS_DN** object.

DS_C_AVA is also included here because it is a required subsubobject of **DS_C_DS_DN** itself. **DS_C_AVA** is the subobject in which the name part's actual literal value is held.

- **OM_CLASS**
  The value of this attribute is an OID string that identifies the object's class; its value is always **DS_C_AVA**.

- **DS_ATTRIBUTE_TYPE**
  The value of this attribute, which is an OID string, identifies the directory attribute whose value is contained in this object.

- **DS_ATTRIBUTE_VALUES**
  This is the literal value of what is represented by this **DS_C_AVA** object.

If the **DS_C_AVA** object is a subobject of **DS_C_DS_RDN** (and therefore also of **DS_C_DS_DN**), then the value is a string representing the part of the directory entry name represented by this object. For example, if the **DS_C_DS_RDN** object contains the **O=OSF** part of an entry name, then the string **OSF** is the value of the **DS_ATTRIBUTE_VALUES** attribute, and **DS_A_COUNTRY_NAME** is the value of the **DS_ATTRIBUTE_TYPE** attribute.

On the other hand, if **DS_C_AVA** contains an entry attribute type and value to be passed to **ds_compare( )**, then **DS_ATTRIBUTE_TYPE** identifies the type of the attribute, and **DS_ATTRIBUTE_VALUES** contains a value, which is appropriate for the attribute type, to be compared with the entry value.

For example, suppose you wanted to compare a certain value with a CDS entry's **CDS_Class** attribute's value. The identifiers for all the valid CDS entry attributes are found in the **/.:/opt/dcelocal/etc/cds_attributes** file. The value of **DS_ATTRIBUTE_TYPE** would be **CDS_Class**, which is the label of an object identifier string, and **DS_ATTRIBUTE_VALUES** would contain some desired value, in the correct syntax for **CDS_Class**. The syntax also is found in the **cds_attributes** file; for **CDS_Class** it is **byte**; that is, a character string.

24.5.2.8 Example Definition of a DS_C_DS_DN Object

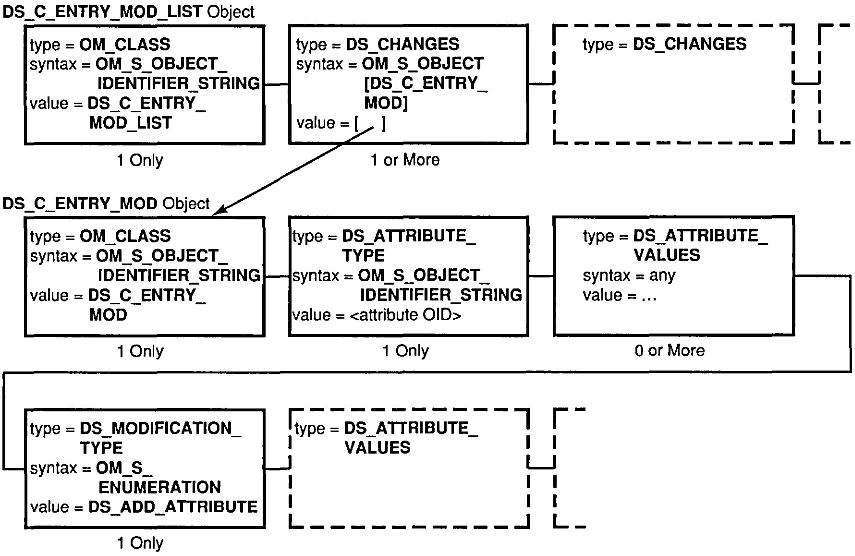
The following code fragment shows an example definition for a **DS_C_DS_DN** object.

```
static OM_descriptor    Entry_String_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_TYPELESS_RDN),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("brazil")},
 OM_NULL_DESCRIPTOR
};


static OM_descriptor    Entry_Part_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 {DS_AVAS, OM_S_OBJECT, {0, Entry_String_Object}},
 OM_NULL_DESCRIPTOR
};


static OM_descriptor    Entry_Name_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
 {DS_RDNS, OM_S_OBJECT, {0, Entry_Part_Object}},
 OM_NULL_DESCRIPTOR
};
```

24.5.2.9 The DS_C_ENTRY_MOD_LIST Object

DS_C_ENTRY_MOD_LIST class objects, which contain a list of changes to be made to some directory entry, must be passed to **ds_modify_entry()**. **DS_C_ENTRY_MOD_LIST** objects have the attributes shown in Figure 24-11.

Figure 24–11. The DS_C_ENTRY_MOD_LIST Object



DS_C_ENTRY_MOD_LIST Object
type = OM_CLASS
syntax = OM_S_OBJECT_IDENTIFIER_STRING
value = DS_C_ENTRY_MOD_LIST
1 Only
type = DS_CHANGES
syntax = OM_S_OBJECT [DS_C_ENTRY_MOD]
value = [ ]
1 or More
type = DS_CHANGES
DS_C_ENTRY_MOD Object
type = OM_CLASS
syntax = OM_S_OBJECT_IDENTIFIER_STRING
value = DS_C_ENTRY_MOD
1 Only
type = DS_ATTRIBUTE_TYPE
syntax = OM_S_OBJECT_IDENTIFIER_STRING
value = <attribute OID>
1 Only
type = DS_ATTRIBUTE_VALUES
syntax = any
value = ...
0 or More
type = DS_MODIFICATION_TYPE
syntax = OM_S_ENUMERATION
value = DS_ADD_ATTRIBUTE
1 Only
type = DS_ATTRIBUTE_VALUES

- **OM_CLASS**
  The value of this attribute is an OID string that identifies the object's class; its value is always **DS_C_ENTRY_MOD_LIST.**

- **DS_CHANGES**
  This is an attribute whose value is another object of class **DS_C_ENTRY_MOD** (see Section 24.5.2.10). The **DS_C_ENTRY_MOD** object is defined by a separate array of object descriptors whose base address is the value of the **DS_CHANGES** attribute.

  Note that there can be one or more instances of this attribute in the object, which is why it is called **_LIST**. Each attribute contains one separate entry modification. To learn how the modification itself is specified, see Section 24.5.2.10. The order of multiple instances of this attribute is significant because if more than one modification is specified, the modifications are performed by **ds_modify_entry**( ) in the order in which the **DS_CHANGES** attributes appear in the **DS_C_ENTRY_MOD_LIST** object.

24.5.2.10 The DS_C_ENTRY_MOD Object

The **DS_C_ENTRY_MOD** class object holds the information associated with a directory entry modification. (For an illustration of its structure, see Figure 24-11.) Each **DS_C_ENTRY_MOD** object describes one modification. To create a list of modifications suitable to be passed to a **ds_modify_entry**( ) call, describe each modification in a separate **DS_C_ENTRY_MOD** object, and then insert these objects as multiple instances of the **DS_CHANGES** attribute in a **DS_C_ENTRY_MOD_LIST** object (see Section 24.5.2.9).

- **OM_CLASS**
  The value of this attribute is an OID string that identifies the object's class; its value is always **DS_C_ENTRY_MOD.**

- **DS_ATTRIBUTE_TYPE**
  The value of this attribute, which is an OID string, identifies the directory attribute whose modification is described in this object.

- **DS_ATTRIBUTE_VALUES**
  These are the values required for the entry modification; their type and number depend on both the entry type and the modification requested.

- **DS_MOD_TYPE**
  The value of this attribute identifies the kind of modification requested. It can be one of the following:

  — **DS_ADD_ATTRIBUTE**
  The attribute specified by **DS_ATTRIBUTE_TYPE** is not currently in the entry. It should be added, along with the value(s) specified by **DS_ATTRIBUTE_VALUES**, to the entry. The entry itself is specified in a separate **DS_C_DS_DN** object, which is also passed to **ds_modify_entry**( ).

  — **DS_ADD_VALUES**
  The specified attribute is currently in the entry. The value(s) specified by **DS_ATTRIBUTE_VALUES** should be added to it.

- **DS_REMOVE_ATTRIBUTE**
  The specified attribute is currently in the entry and should be deleted from the entry. Any values specified by **DS_ATTRIBUTE_VALUES** are ignored.

- **DS_REMOVE_VALUES**
  The specified attribute is currently in the entry. One or more values, specified by **DS_ATTRIBUTE_VALUES**, should be removed from it.

24.5.2.11 Example Definition of a DS_C_ENTRY_MOD_LIST Object

The following code fragment is an example definition of a **DS_C_ENTRY_MOD_LIST** object.

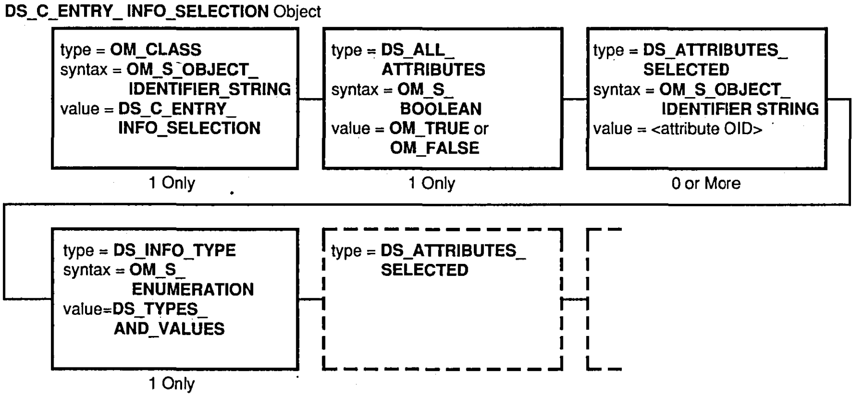
```
OM_string my_uuid;


static OM_descriptor    Entry_Mod_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_UUID),
 {DS_ATTRIBUTE_VALUES, OM_S_OCTET_STRING, my_uuid},
 {DS_MOD_TYPE, OM_S_ENUMERATION, DS_ADD_ATTRIBUTE},
 OM_NULL_DESCRIPTOR
};



static OM_descriptor    Entry_Mod_List_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ENTRY_MOD_LIST),
 {DS_CHANGES, OM_S_OBJECT, {0, Entry_Mod_Object}},
 OM_NULL_DESCRIPTOR
};
```

24.5.2.12 The DS_C_ENTRY_INFO_SELECTION Object

When you call **ds_read**( ) to read one or more attributes from a CDS entry, you specify in the **DS_C_ENTRY_INFO_SELECTION** object the entry attributes you want to read.

The **DS_C_ENTRY_INFO_SELECTION** obrt contains the attributes shown in Figure 24-12.

Figure 24–12. The DS_C_ENTRY_INFO_SELECTION Object


DS_C_ENTRY_ INFO_SELECTION Object
type = OM_CLASS
syntax = OM_S_OBJECT_ IDENTIFIER_STRING
value = DS_C_ENTRY_ INFO_SELECTION
1 Only
type = DS_ALL_ ATTRIBUTES
syntax = OM_S_ BOOLEAN
value = OM_TRUE or OM_FALSE
1 Only
type = DS_ATTRIBUTES_ SELECTED
syntax = OM_S_OBJECT_ IDENTIFIER STRING
value = <attribute OID>
0 or More
type = DS_INFO_TYPE
syntax = OM_S_ ENUMERATION
value=DS_TYPES_ AND_VALUES
1 Only
type = DS_ATTRIBUTES_ SELECTED

Note that this object class has no subobjects.

- **OM_CLASS**
  The value of this attribute is an OID string that identifies the object's class; its value is always **DS_C_ENTRY_INFO_SELECTION.**

- **DS_ALL_ATTRIBUTES**
  This attribute is a simple Boolean option whose value indicates whether all the entry's attributes are to be read, or only some of them. Its possible values are as follows:

  — **OM_TRUE**, meaning that all attributes in the directory entry should be read. Any values specified by the **DS_ATTRIBUTES_SELECTED** attribute are ignored.

  — **OM_FALSE**, meaning that only some of the entry attributes should be read; namely, those specified by the **DS_ATTRIBUTES_SELECTED** attribute.

- **DS_ATTRIBUTES_SELECTED**
  The value of this attribute, which is an OID string, identifies the entry attribute to be read. Note that this attribute's value has meaning only if the value of **DS_ALL_ATTRIBUTES** is **OM_FALSE**; if it is **OM_TRUE**, the value of **DS_ATTRIBUTES_SELECTED** is ignored.

  Note also that there are multiple instances of this attribute if more than one attribute, but not all of them, is to be selected for reading. Each separate instance of **DS_ATTRIBUTES_SELECTED** has as its value an OID string that identifies one directory entry attribute to be read. If **DS_ATTRIBUTES_SELECTED** is present but does not have a value, **ds_read**( ) reads the entry but does not return any attribute data; this technique can be used to verify the existence of a directory entry.

- **DS_INFO_TYPE**
  The value of this attribute specifies what information is to be read from each attribute specified by **DS_ATTRIBUTES_SELECTED**. The two possible values are as follows:

  — **DS_TYPES_ONLY**, meaning that only the attribute types of the selected attributes should be read.

  — **DS_TYPES_AND_VALUES**, meaning that both the attribute types and the attribute values of the selected attributes should be read.

24.5.2.13 Example Definition of a DS_C_ENTRY_INFO_SELECTION Object

The following code fragment provides an example definition of a **DS_C_ENTRY_INFO_SELECTION** object.

```
static OM_descriptor    Entry_Info_Select_Object[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
 OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DSX_A_CDS_Class),
 {DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_FALSE},
 {DS_INFO_TYPE, OM_S_ENUMERATION, DS_TYPES_AND_VALUES},
 OM_NULL_DESCRIPTOR
};
```

24.6 Attribute and Data Type Translation

This section provides translations between CDS and XDS for attributes and data types. Table 24-2 provides the OM syntax for CDS attributes. Table 24-3 provides the OM syntax for CDS data types.

Table 24–2. CDS Attributes to OM Syntax Translation

CDS Attribute	OM Syntax
CDS_CTS	OM_S_OCTET_STRING
CDS_UTS	OM_S_OCTET_STRING
CDS_Class	OM_S_OCTET_STRING
CDS_ClassVersion	OM_S_INTEGER
CDS_ObjectUID	OM_S_OCTET_STRING
CDS_AllUpTo	OM_S_OCTET_STRING
CDS_Convergence	OM_S_INTEGER
CDS_InCHName	OM_S_INTEGER
CDS_DirectoryVersion	OM_S_INTEGER
CDS_UpgradeTo	OM_S_INTEGER
CDS_LinkTimeout	OM_S_INTEGER
CDS_Towers	OM_S_OCTET_STRING

Table 24–3. OM Syntax to CDS Data Types Translation

OM Syntax	CDS Data Type
OM_S_TELETEX_STRING	dns_char
OM_S_OBJECT_IDENTIFIER_STRING	dns_byte
OM_S_OCTET_STRING	dns_byte
OM_S_PRINTABLE_STRING	dns_char
OM_S_NUMERIC_STRING	dns_char
OM_S_BOOLEAN	dns_long
OM_S_INTEGER	dns_long
OM_S_UTC_TIME_STRING	dns_char
OM_S_ENCODING	dns_byte

Part 4B

GDS Application Programming

Part 4B provides an overview of programming GDS using XDS. Chapter 25 discusses GDS concepts and provides an overview of GDS programming. Chapter 26 describes XOM programming and Chapter 27 describes XDS programming. Chapter 28 provides programming examples.

Chapter 25

GDS API: Concepts and Overview

The Global Directory Service (GDS) is a distributed, replicated directory service. It is distributed because information is stored in different places in the network. Requests for information may be routed by the Global Directory Service to directory servers throughout the network. It is replicated because information can be stored in more than one location for easier and more efficient access by its users.

The Global Directory Service is based on the CCITT X.500/ISO 9594 (1988) international standard. The aim of this standard, also referred to as the OSI Directory standard, is to provide a global directory that supports network users and applications with information required for communications. The Directory plays a significant role in allowing the interconnection of information processing systems from different manufacturers, under different managements, of different levels of complexity, and of different ages.

GDS is the DCE implementation of the OSI Directory standard. Together with the Cell Directory Service (CDS) it provides its users with a centralized place to store information required for communications, which can be retrieved from anywhere in a distributed system. GDS maintains information describing objects such as people, organizations, applications, distribution lists, network hardware, and other distributed services dispersed over a large geographical area.

The Cell Directory Service stores names and attributes of resources located in a DCE cell. A DCE cell consists of various combinations of DCE machines connected by a network. Each DCE cell contains its own Cell Directory Server, which provides access to local resource information. The Cell Directory Service is optimized for local information access by its users. For a more detailed description of cells and their resource services, see the *Introduction to OSF DCE.*

The Global Directory Service serves as a general-purpose information repository. It provides information about resources outside a DCE cell. It ties together the various cells by helping to find remote cells. A detailed discussion of the DCE namespace, its various servers and their interaction is provided in Chapter 22.

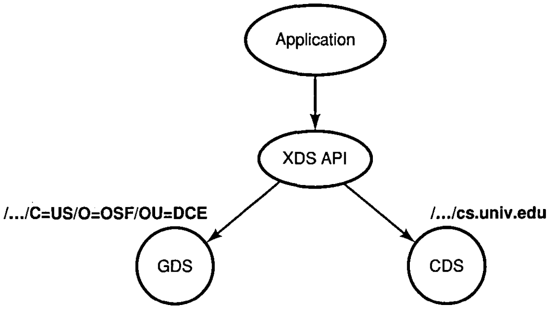
25.1 Directory Service Interfaces

X/Open Directory Service (XDS) and X/Open OSI-Abstract-Data Manipulation (XOM) are application programming interfaces. XOM and XDS application interfaces are based on X/Open standards specifications. Together these interfaces provide the application programmer with a library of functions with which to develop applications that access the Directory Service.

The XOM Application Programming Interface (XOM API) is an interface for creating, deleting, and accessing information objects. The XOM API defines an object-oriented information model. Objects belong to classes and have attributes associated with them. The XOM API also defines basic data types, such as Boolean, string, object, and so on. The representation of these objects are transparent to the programmer. Objects can only be manipulated through the XOM interface, not directly.

DCE programmers use the XDS API to make Directory Service calls. In DCE, XDS API directs the calls it receives to either the Global Directory Service or the Cell Directory Service by examing the names of the information objects to be looked up as shown in Figure 25-1. It uses the XOM interface for defining and handling information objects. These objects are passed as parameters and return values to the XDS routines. The XDS API contains functions for managing connections with a Directory Server: reading, comparing, adding, removing, modifying, listing, and searching for directory entries. The Global Directory Service Package provides additional information objects that provide for security and cache management when using GDS.

Figure 25–1. XDS: Interface to GDS and CDS



25.2 The X.500 Directory Information Model

This section describes the directory information model of X.500, which GDS is based on. A directory is a collection of information about some part of the world. The most familiar type of directory is the list of names and numbers that make up a city telephone directory. A name is provided with some information about the named object, such as an address and telephone number. The ISO and CCITT standards define a *directory information model* that defines the abstract structure of directory information, services, and protocols for a computer network environment, such as DCE.

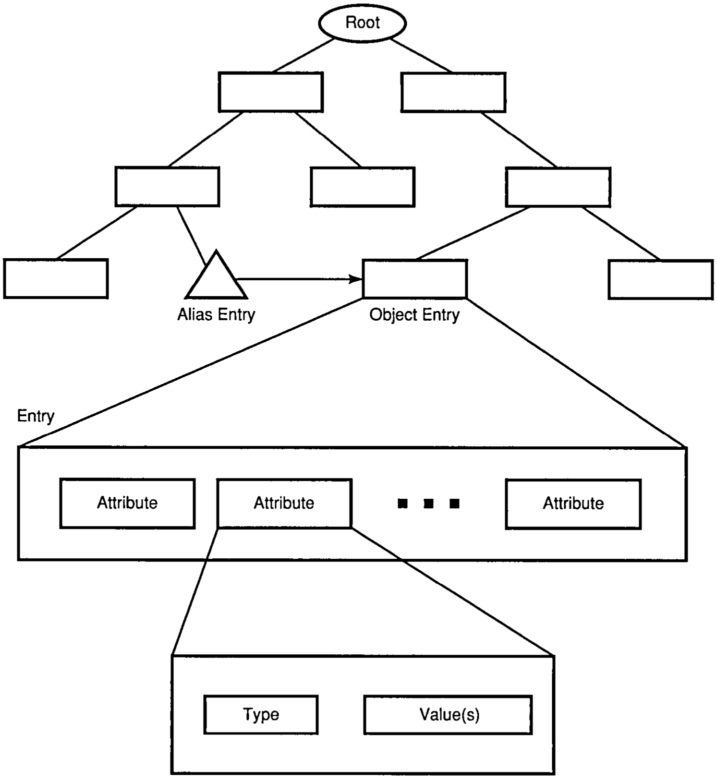
25.2.1 Directory Objects

The Directory contains information about objects. The standard defines an object very broadly as ‘‘anything in some ‘world,’ generally the world of telecommunications and information processing or some part thereof, which is identifiable (can be named).’’ Some examples of objects include people, corporations, and application processes.

Each object known to the Directory is represented by an entry. The set of all entries is called the Directory Information Base (DIB), which is a hierarchical tree. Each entry consists of a set of attributes representing specific information about the object. Each attribute, in turn, has a type and one or more values of that type. Attributes with more than one value are referred to as *multivalued* or *recurring* attributes.

Figure 25-2 shows the structure of the DIB.

Figure 25–2. The Structure of the DIB



The attributes that constitute a single entry may be of various types. For example, an entry for a person may contain that person's name, address, and phone number. If the person has a second telephone number, the attribute of type telephone number may have two values, one for each telephone number.

Object entries are composed of mandatory and optional attributes. Mandatory and optional attributes are discussed in Section 25.4.3.

25.2.2 Attribute Types

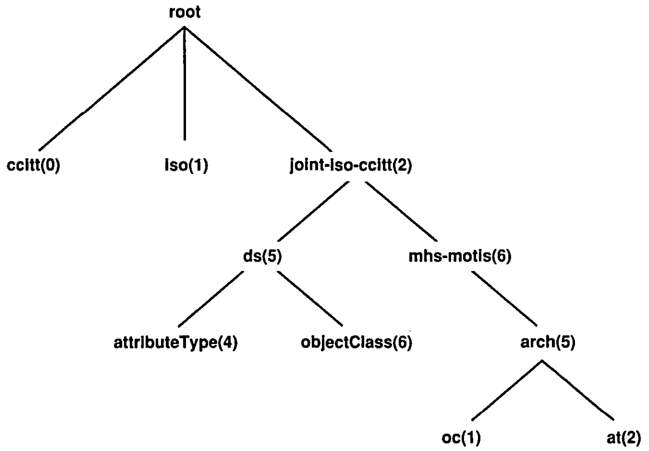
All attributes in a particular entry must be of different attribute types. Each attribute type is assigned a unique object identifier value. The Directory standard assigns object identifiers for several commonly used attribute types, including surname, country name, telephone number, and presentation address. Other international standards may define additional attribute types. For example, the X.400 Message Handling standard defines mail specific attributes like O/R address. It is expected that various national and private organizations will also define attribute types of their own. The CDS attributes (defined in the **xdscds.h** header file) and the Global Directory Service Package attributes (defined in the **xdsgds.h** header file) are examples of additional attribute definitions.

25.2.3 Object Identifiers

Objects in a network environment, such as DCE, require unique names to distinguish them from one another. To provide these names, object identifiers are allocated by an administrative organization, such as a standards body. An object identifier is a hierarchical sequence of numbers uniquely identifying an object. Associated with each object identifier is a character string to make it easier to document.

The possible values of object identifiers are defined in a tree. Part of this tree is shown in Figure 25-3. It begins with three numbered branches coming from the root: branch 0, assigned to CCITT, branch 1, assigned to ISO, and branch 2, a joint ISO-CCITT branch. Below each of these branches are other numbered branches assigned to various standards such as the Directory Service (**ds(5)**) and Electronic Mail Service (**mhs-motis(6)**) with each ending in a named object. Thus, the name of any of these objects is a series of integers describing a path down this tree to the leaf node.

Figure 25–3. Object Identifiers



The object identifier associated with the XDS Directory Service Package is defined as follows:

{iso(1) identified-organization(3) icd-ecma(12) member-company(2) dec(1011) xopen(28) dsp(0)}.

All object classes and object attributes in the Directory Service Package have these numbered branches associated with them. The classes and attributes, in turn, have their own unique numbers. These object identifiers are defined in header files included as part of the XDS and XOM API software. For example, the attribute type **Common Name** is identified by the object identifier 2.5.4.3.

Table 25-1 contains a sample list of object identifiers for selected attributes. The complete list is provided in Chapter 32.

Table 25–1. Object Identifiers for Selected Attribute Types

Attribute Type	Object Identifier
Aliased Object Name	2.5.4.1
Business Category	2.5.4.15
Common Name	2.5.4.3
Country Name	2.5.4.6
Description	2.5.4.13

Note: The object identifiers in Table 25-1 stem from the root **{joint-iso-ccitt(2) ds(5) attributeType(4)}**.

25.2.4 Object Entries

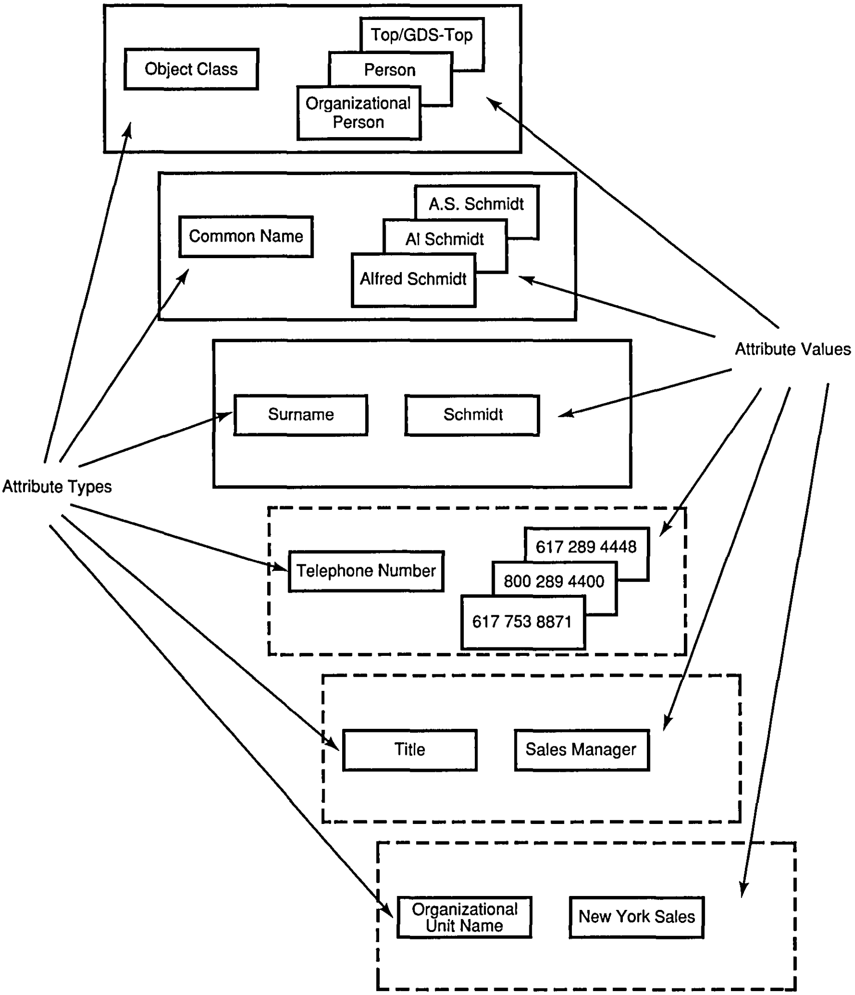
Entries are grouped into generic object classes based on the type of object they represent. Examples of object classes are Country, Organizational Person, and Application Entity. All entries contain a special attribute, the object class attribute, indicating to which object class (or classes) they belong.

Entries that model a certain object and contain information about the object in terms of attributes are called *object entries*. The Directory contains a second type of entry, which is a pointer to an object entry called an *alias entry*. Alias entries are discussed in Section 25.3.4.

In summary (as shown previously in Figure 25-2), the DIB is made up of entries, each of which contains information about objects. Entries consist of attributes; each attribute has a type and one or more values.

Section 25.3 describes how objects are organized in the DIB using the Directory Information Tree (DIT). Figure 25-4 shows an example of an entry describing Organizational Person.

Figure 25–4. A Directory Entry Describing Organizational Person



25.3 X.500 Naming Concepts

Large amounts of information need to be organized in some way to make efficient retrieval possible and ensure that names are unique. Information in the DIB is organized into a hierarchical structure known as the Directory Information Tree (DIT). The structure and naming of the nodes in the DIT are specified by registration authorities for a standardized set of X.500 names and by implementors of the directory service (such as OSF) for implementation-specific names. The DIT hierarchy is described by a schema. Schemas are described in more detail in Section 25.4.

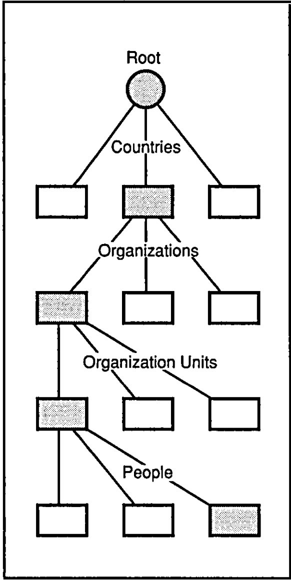
Although the X.500 standard does not mandate a specific schema, it does make general recommendations. For example, countries and organizations should be named close to the root of the DIT; people, applications, and devices should be named further down in the hierarchy. GDS supplies a default schema that complies with these recommendations.

25.3.1 Distinguished Names

A hierarchical path exists from the root of the DIT to any entry in the DIB. To access information stored in an entry, a name that uniquely describes that entry must be given. An RDN distinguishes an entry from other entries with the same superior node in the DIT. A sequence of RDNs, starting from the root of the tree, can identify a unique path down the tree, and thus a unique entry. This sequence of RDNs, each of which indentifies a particular entry, is the distinguished name of that entry. Each entry in the DIB can be referenced by giving its distinguished name.

Figure 25-5 shows an example of a distinguished name. The shaded boxes in the DIT represent the entries that are named in the column labeled RDN (Relative Distinguished Name). The schema dictates that countries are named directly below the root, followed by organizations, organization units, and people.

Figure 25–5. A Distinguished Name in a Directory Information Tree

DIT	RDN	Distinguished Name
		{}
	C =US	{C =US}
	O =Acme Enterprises	{C =US O =Acme Enterprises}

	OU =New York Sales	{C=US
		O=Acme Enterprises
		OU=New York Sales}

	CN =Alfred Schmidt	{C =US
		O =Acme Enterprises
		OU =New York Sales
		CN =Alfred Schmidt}

Every entry in the DIB has a distinguished name, not just the leaf nodes. For example, the entry for the Organization, Acme Enterprises (shown in Figure 25-5) is represented by the shaded box in the Organizations subtree. Its distinguished name is the concatenation of the distinguished name of the entry above with its relative distinguished name. The entry for People, Alfred Schmidt, is represented by the shaded box in the People subtree.

25.3.2 Relative Distinguished Names and Attribute Value Assertions

Each entry has a unique Relative Distinguished Name (RDN), which distinguishes it from all other entries with a particular immediate superior in the DIT.

An RDN consists of one or more assertions of the type and value of an attribute. A pair consisting of an attribute type and a value of that type is known as an Attribute Value Assertion (AVA). All attribute types in an

RDN must be different. The attribute value of an attribute in an RDN's AVA is called the distinguished value of that attribute, as opposed to the other possible values of that attribute.

The assertion is TRUE if the entry contains an attribute of the specified type, and if one of that attribute's values matches the AVA's distinguished attribute value. An entry commonly has an RDN that consists of a single AVA. In some cases, however, more than one AVA may be required to distinguish an entry. (Multiple AVAs are discussed in Section 25.3.3.)

The entry shown in Figure 25-4 contains the RDN: **Common Name = Alfred Schmidt**. The attribute consists of three values: Alfred Schmidt, A. Z. Schmidt, and Al Schmidt. The AVA **Common Name = Alfred Schmidt** contains the value Alfred Schmidt, which has been designated as the distinguished value in the AVA.

25.3.3 Multiple AVAs

Frequently, as shown in the previous section, an entry contains a single distinguished value and the RDN therefore comprises a single AVA. However, under certain circumstances additional values (and hence multiple AVAs) may be used.

Figure 25-4 shows the contents of an entry describing Organizational Person. The RDN of an Organizational Person entry is usually composed of a single AVA, such as the Common Name attribute type with a distinguished value (in Figure 25-5, the AVA **CN = Alfred Schmidt**). Depending on the schema, the RDN of an Organizational Person entry may contain more than one AVA. For example, the RDN in Figure 25-5 could contain the AVAs **CN = Alfred Schmidt, OU = New York Sales** with Alfred Schmidt and New York Sales as distinguished values.

In summary:

- A DIT consists of a collection of distinguished names.
- Distinguished names result from a concatenation of the RDNs.
- RDNs consist of an unordered collection of attribute type and value pairs (AVAs).

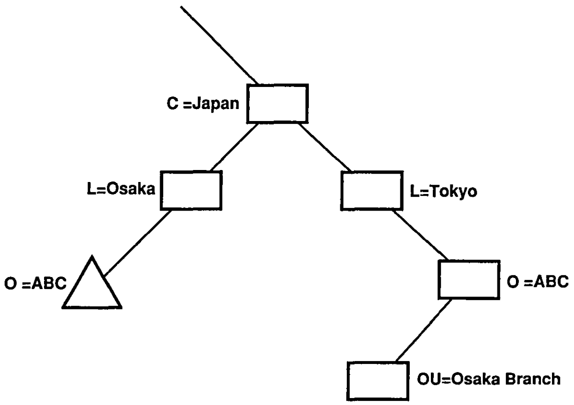
25.3.4 Aliases

An alternative name or alias is supported in the DIT by the use of special pointer entries called alias entries. Alias entries do not contain any other attributes beyond their distinguished attributes, the object class attribute, and the aliased object name attribute; that is, the distinguished name of the aliased object entry. Furthermore, an alias entry has no subordinate entries, making it, by definition, a leaf entry of the DIT as shown in Figure 25-6. Alias entries point to object entries and provide the basis for alternative names for the corresponding objects.

Aliases are used to do such things as provide more user-friendly names, direct the search for a particular entry, reduce the scope of a search, provide for common alternate abbreviations and spellings, or provide continuity after a name change.

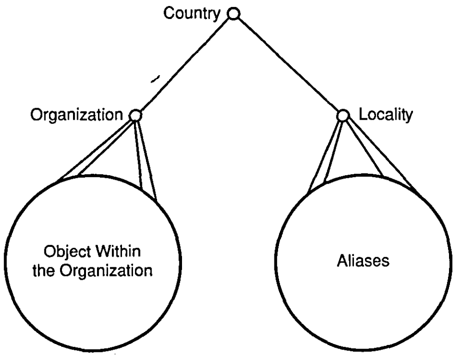
Figure 25-6 demonstrates how an alias name provides continuity after a name change. The ABC company's branch office located originally in Osaka has moved to Tokyo. To make the transition easier for Directory Service users and to guarantee that a search based on the old information finds its target, an alias for **O=ABC** has been added to the directory beneath **L=Osaka**. This alias entry points to the object entry **O=ABC**. A search for ABC under **L=Osaka** in the DIT finds the entry: **/C=Japan/L=Tokyo/O=ABC**.

Figure 25–6. An Alias in the Directory Information Tree



Another use of alias entries is as an alternative to *filtering*; that is, using assertions about particular attributes to search through the DIT. Although this approach does not require any special information to be set up in the DIT, it may be expensive to search where there is a large population of entries and attributes. An alternative approach is to set up special subtrees whose naming structures are designed for ''Yellow Pages'' type searching. Figure 25-7 shows an example of such a subtree populated by alias entries only. In reality, the entries within these subtrees may be a mixture of object and alias entries, so long as there exists only one object entry for each object stored in the directory.

Figure 25–7. A Subtree Populated by Aliases



An object with an entry in the DIT may have zero or more aliases. Several alias entries may point to the same object entry. An alias entry may point to an object that is not a leaf entry. Only object entries may have aliases. Thus, aliases of aliases are not permitted.

25.3.5 Name Verification

A Directory user identifies an entry by supplying an ordered set of RDNs (each of which consists of an unordered set of AVAs) that form a purported name. The purported name is mapped onto the desired entry by the process of name verification, which performs a distributed tree walk through the DIT. When a purported name is a valid name, a distinguished name exists with the same number of RDNs and matching AVAs within the RDNs.

25.4 Schemas

The structure of directory information is governed by a set of rules called a *schema*. Schemas specify rules for the following:

- The structure of the DIT
- The contents of entries in terms of attributes
- The syntax of attribute values and rules for comparing and matching them

25.4.1 The GDS Standard Schema

When the DCE software package is shipped to a customer, it includes a default or ''standard'' schema for GDS. This is the GDS proprietary interpretation of the X.500 schema.

Each attribute in the schema is assigned a unique object identifier and the syntax of its value. In addition, the schema specifies the mechanism by which attributes of this type are compared with one another. Each entry in the DIT belongs to an object class governed by the schema. Object class definitions may be used to derive subclasses, supporting the inheritance and refinement of the attribute types defined for the super-class.

Included with the GDS standard schema are the following tables that define the structure of the Directory.

- Structure Rule Table (SRT)
- Object Class Table (OCT)
- Attribute Table (AT)

25.4.2 The Structure Rule Table

The Structure Rule Table (SRT) specifies the relationship of object classes in the structure of the Directory. The SRT supplied with the GDS standard schema contains the entries shown in Table 25-2.

Table 25–2. Structure Rule Table Entries

Rule Number	Superior Rule Number	Acronym of Naming Attribute	Acronym of Structural Object Class
1	0	**CN**	SCH
2	0	**C**	C
3	2	**O**	ORG
4	3	**OU**	OU
5	4	**CN**	ORP
6	4	**CN, OU**	ORP
7	4	**CN**	ORR
8	4	**CN**	MDL
9	4	**CN**	APP
10	9	**CN**	APE
11	9	**CN**	DSA
12	9	**CN**	MMS
13	9	**CN**	MTA
14	9	**CN**	MUA
15	2	**L**	LOC
16	15	**CN**	REP
17	15	**CN, STA**	REP

The SRT determines how the object classes are laid out in the DIT by assigning rule numbers to each object class. An object class's Superior Rule Number specifies the object class directly above it in the DIT.

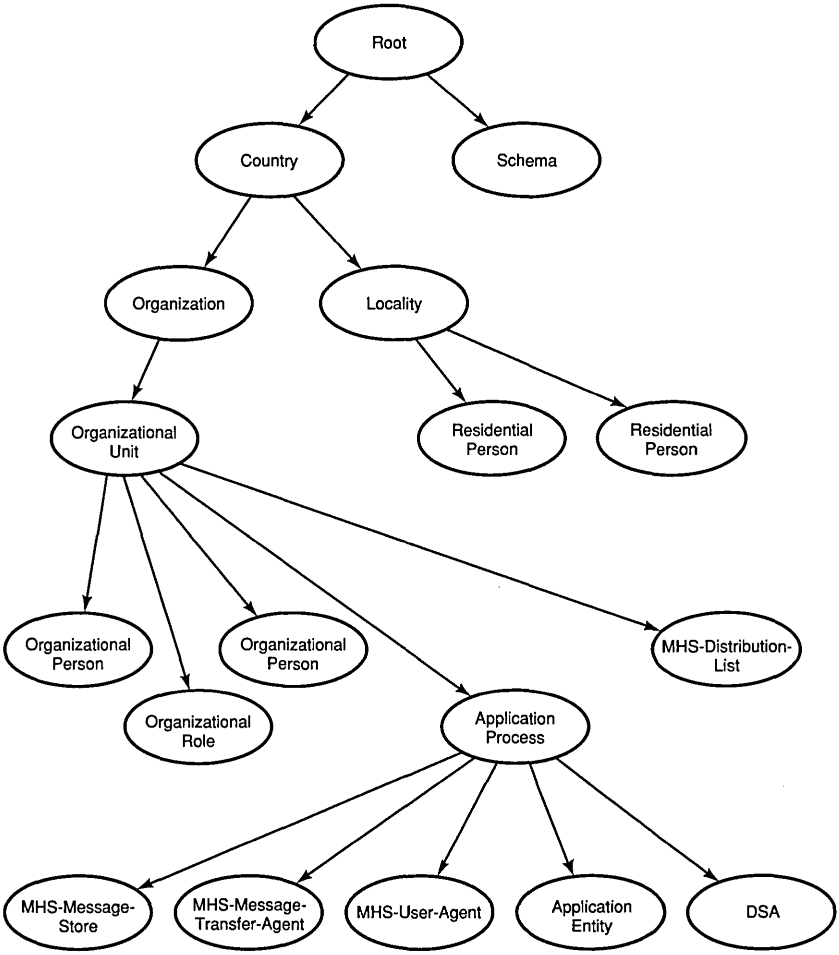
For example, the object class Organization (abbreviated with the acronym ORG in the SRT) has a Superior Rule Number of 2, indicating that it is located in the DIT beneath the object class Country (C), which has a Rule Number of 2. Organization Unit (OU) is located beneath Organization because it has a Superior Rule Number of 3 and so forth.

The SRT only contains structured object classes; that is, classes that form branches in the DIT. Other object classes, such as abstract and alias classes, are not included.

The SRT specifies the attribute(s) used to name entries belonging to each object class. These attributes, called *naming attributes*, are used to define the RDN and therefore the distinguished name of directory entries.

Figure 25-8 shows the structure of the DIT as defined by the SRT of the GDS standard schema.

Figure 25–8. SRT DIT Structure for the GDS Standard Schema



25.4.3 The Object Class Table

The object classes that make up the GDS standard schema are defined in the OCT. Table 25-3 contains a partial listing of the OCT (refer to the *OSF DCE Administration Guide* for a complete listing of the OCT for the GDS standard schema). Each column in Table 25-3 contains information about an object class entry in the schema.

Table 25–3. Object Class Table Entries

Object Class			Super-		File	Mandatory	Optional
Acronym	**Name**	**Kind**	**class**	**OID**	**No.**	**Attributes**	**Attributes**
TOP	Top	Abstract	None	85.6.0	-1	OCL	None
ALI	Alias	Alias	TOP	85.6.1	-1	AON	None
C	Country	Structural	GTP	85.6.2	1	C	DSC SG CDC CDR
LOC	Locality	Structural	GTP	85.6.3	4	None	DSC L SPN STA SEA SG CDC CDR
ORG	Organization	Structural	GTP	85.6.4	1	O	DSC L SPN STA PDO PA PC POB FTN IIN TN TTI TXN X1A PDM DI RA SEA UP BC SG CDC CDR

Note: All these object identifiers stem from the root **{joint-iso-ccitt(2) ds(5) objectClass(6)}**.

Column 4, Superclass acronyms, provides the class from which an object class inherits its attributes. Using the information in Column 4, it is possible to derive a graphical representation of the inheritance properties of object classes in the DIT as shown in Figure 25-9.

The object class Top is the root of the tree, with Alias and GDS-Top as the main branches. Top contains the attribute type object class, which is inherited by all the other object classes.

Do not confuse the information in the OCT with that presented in the SRT. There is no direct relationship between the relative location of branches and leaves in the DIT structure and the inheritance properties of classes with their superclasses and subclasses. For example, when a Directory Service request is made by a directory user, such as a read operation, the SRT is used by the Directory Service to indicate its position in the DIT. The Directory Service uses the information defined in the SRT for tree traversal so that the requested object can be located in the Directory. Figure 25-8 shows the object class Organization located beneath Country in the DIT.

On the other hand, the OCT defines, among other things, the attributes of an object class along with its inherited attributes from its superclass. The superclass, in turn, inherits the attributes from its superclass, and so on until the root, Top, is reached (from which all classes derive their attributes). Figure 25-9 shows the object class Organization as a subclass of GDS-Top. As such, it inherits its attributes from GDS-Top, which in turn inherits from its superclass, Top.

The OCT also contains the unique object identifier of each class in the DIT. These numbers are defined by various standards authorities and in the X.500 standards documents mentioned previously. The AT also contains the predefined object identifiers for each attribute in the Directory. These object identifiers are defined in the header files that are included as part of the GDS API. Table 25-4 shows some examples of object identifiers for directory classes as defined in the X.500 standard.

Figure 25–9. A Partial Representation of the Object Class Table

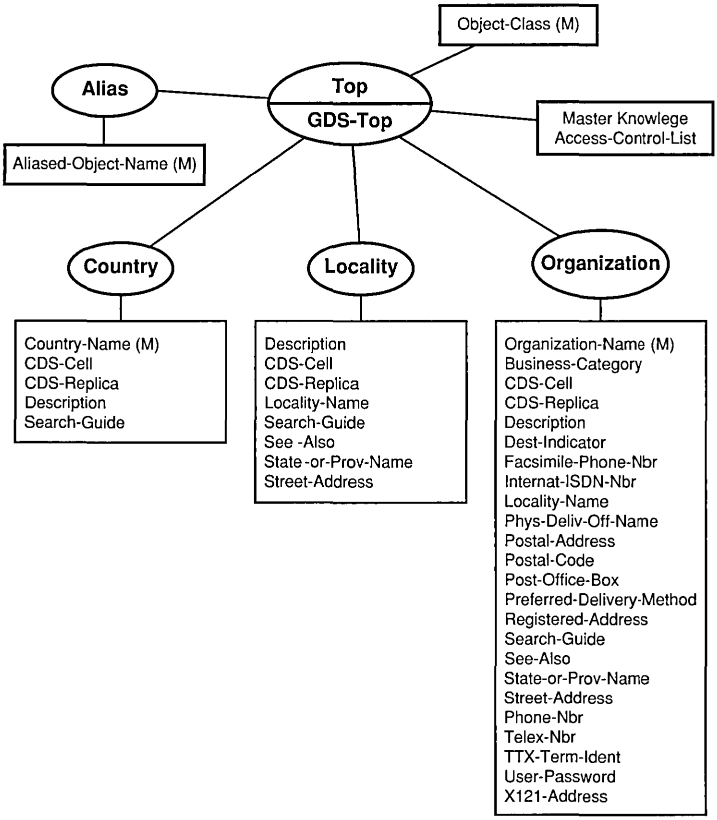


Table 25–4. Object Identifiers for Selected Directory Classes

Object Class Type	Object Identifier
Alias	85.6.1
Application Entity	85.6.12
Application Process	85.6.11
Country	85.6.2
Device	85.6.14
DSA	85.6.13
Group of Names	85.6.9
Locality	85.6.3
Organization	85.6.4
Organizational Person	85.6.7
Organizational Role	85.6.8
Organizational Unit	85.6.5
Person	85.6.6
Residential Person	85.6.10
Top	85.6.0

Note: All these object identifiers stem from the root **{joint-iso-ccitt(2) ds(5) objectClass(6)}**.

Another important feature of the OCT is the distinction made between mandatory and optional attributes for each object class. This distinction is based on recommendations from X.500 standards documents. These documents (Recommendations X.520 and X.521) define selected object classes and associated attribute types using ASN.1 notation. Most object classes have one or more mandatory attributes associated with them for use by implementors who want to comply with the X.500 standards recommendations. In addition, optional attributes are defined.

For example, the following example provides a flavor of ASN.1 notation; it shows how the object class **country** is described in Recommendation X.520 (*The Directory: Selected Object Classes*).

```
country OBJECT-CLASS
     SUBCLASS of top
     MUST CONTAIN {
         countryName}
     MAY CONTAIN {
            description,
            searchGuide}
        ::= {objectClass 2}
```

This ASN.1 definition defines **country** as a subclass of superclass **top**. The class, **country**, must contain the mandatory attribute **countryName** (or **country-name** as defined in the GDS standard schema) and may contain the optional attributes **description** and **searchGuide**. In addition, the DCE implementation adds two more attributes, **CDS-Cell** and **CDS-Replica**, to incorporate other aspects of the DCE environment that are implementation specific.

Country is assigned the object identifier 2.5.6.2. This number distingushes it from the other object classes defined by the standard. The Top superclass is designated as 2.5.6.0. The first three numbers, 2.5.6, identify the object class as a member of a discrete set of object classes defined by X.500. The last number in the object identifier distinguishes objects within that discrete set. Alias, a subclass of Top, is assigned the number 2.5.6.1. Country is assigned the number 2.5.6.2, and so on. GDS-Top has no object identifier because it is implementation specific and thus not identified by the standard.

25.4.4 The Attribute Table

The attributes that make up the entries in the GDS standard schema are defined in the Attribute Table (AT). (Refer to the *OSF DCE Administration Guide* for a complete listing of the AT.) The object identifiers are in the range from 85.4.0 through 85.4.35 as defined by the X.500 standard, 86.5.2.0 through 86.5.2.10 as defined by the X.400 standard, and there are additional object identifiers for GDS specific attributes.

Table 25-5 shows a partial listing of the attribute table for the GDS standard schema.

Note: The access class for every attribute listed in Table 25-5 is 0 (zero).

Table 25–5. Attribute Table Entries

Attr. Acr.	Obj. ID	Name of Attribute	Lower Bound	Upper Bound	Max. No. of Val.	Syntax	Phon. Flag	Index Level
OCL	85.4.0	Object-Class	1	28	0	2	0	0
AON	85.4.1	Aliased-Object-Name	1	1024	1	1	0	0
KNI	85.4.2	Knowledge-Information	1	1024	0	4	0	0
CN	85.4.3	Common-Name	1	64	2	4	1	1
SN	85.4.4	Surname	1	64	2	4	1	0
SER	85.4.5	Serial-Number	1	64	2	5	0	0
C	85.4.6	Country-Name	2	2	1	1010	1	1
L	85.4.7	Locality-Name	1	128	2	4	1	1
SPN	85.4.8	State-or-Province-Name	1	128	2	4	1	0

The columns with the headings Lower Bound and Upper Bound specify the range of the number of bytes (or octets) that the value of an attribute can contain. The schema puts constraints on the number of values that an attribute can contain in the Maximum Number of Values column.

The Syntax column describes how the data is represented and relates to ASN.1 syntax definitions for attributes. For example, a sample of ASN.1 notation for the Common-Name attribute follows:

```
commonName ATTRIBUTE
   WITH ATTRIBUTE-SYNTAX
       caseignoreStringSyntax
           (SIZE(1..ub-common-name))
   ::= (attributeType 3)
```

The **commonName** attribute is defined as case insensitive. The size of the string is from 1 to the upper bound defined by the schema for the **commonName** attribute in the Upper Bound column (in this case, 64 bytes or octets).

Note also that the **commonName** attribute is assigned the number 3 by the standard. This corresponds to the 3 in the object identifier 85.4.3.

The other columns in the AT refer to the phonetic matching flag, security access classes, and index level.

As mentioned previously for object classes, object identifier values specified in the AT are defined as constants in the GDS header files.

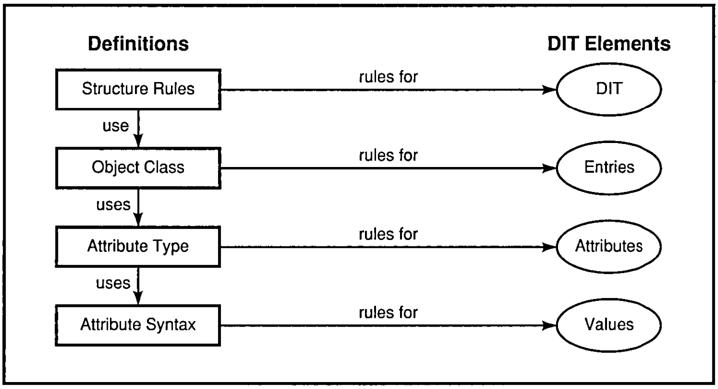
25.4.5 Defining Subclasses

The ability to define subclasses is a powerful feature of the directory. Structure rules govern which object classes may be children of which others in the DIT and therefore determine possible name forms.

The directory standard defines a number of standard attribute types and object classes. For example, the attribute types Common Name and Description, and the object classes Country and Organizational Person are defined. Implementations of the directory standard, such as DCE, define their own schemas following rules stated in the standard with additional attribute types and object classes.

Figure 25-10 shows the relationship between schemas and the directory information model.

Figure 25–10. The Relationship Between Schemas and the DIT



25.5 Abstract Syntax Notation 1

The need for Abstract Syntax Notation 1 (ASN.1) arises because different computer systems represent information in different ways. For example, one computer may use EBCDIC character representation while another may use ASCII. To transfer a file of characters from one system to another, common representation must be used during the transfer. This transfer may be one representation or the other or some mutually agreed upon representation negotiated by the two systems. Similarly, floating-point values, integers, and other types of data may be stored internally in different ways. To exchange information, a common format must be agreed to before information can be exchanged.

The translation of EBCDIC to ASCII characters may seem like a trivial problem, but that leaves the larger issue of mapping between the many diverse representations that may exist within a network environment. To address this need, the ISO standards committee defined Abstract Syntax Notation 1 (ASN.1) and Basic Encoding Rules (BER).

ASN.1 is based on the idea that the aspects of transferred information that are preserved are type, length, and value. Data types are collections of values distinguished for some reason, such as characters, integers, and floating-point values. Records and structure types become more complex when they combine several types into a single structure.

ASN.1 provides a way to group types into abstract syntaxes. An abstract syntax is a named group of types. The standard defines abstract syntax as the notation rules that are independent of the encoding technique used to represent them. Abstract syntax does not specify how to represent values of types, but merely defines the types that make up the group of types.

Abstract syntaxes are not enough to define how values of the data types in a specific abstract syntax are to be represented during communications. For this reason, ISO further defines a transfer syntax for each abstract syntax. A transfer syntax is a set of rules for encoding values of some specified group of types.

25.5.1 ASN.1 Types

ASN.1 is similar to a high-level programming language. Unlike other high-level languages, ASN.1 has no executable statements. It includes only language constructs required to define types and values.

ASN.1 defines a number of built-in types. Users of ASN.1 can then define their own types based on the built-in types provided by the language. The ASN.1 standard defines four categories of types that are commonly used in defining application interfaces such as XOM and XDS:

- ASN.1 Simple Types
- ASN.1 Useful Types
- ASN.1 Character String Types
- ASN.1 Type Constructors

ASN.1 simple types are Bit String, Boolean, Integer, Null, Object Identifier, Octet String, and Real. Table 25-6 shows the relationship of OM syntaxes (syntaxes defined in XOM API) to ASN.1 simple types. (Refer to Chapter 35 for the complete set of tables for the four categories of ASN.1 types.) As shown in the table, for every ASN.1 type except Real, there is an OM syntax that is functionally equivalent to it. The simple types are listed in the first column of the table; the corresponding syntaxes are listed in the second column.

Table 25–6. Syntax for the Simple ASN.1 Types

ASN.1 Type	OM Syntax
Bit String	String(**OM_S_BIT_STRING**)
Boolean	**OM_S_BOOLEAN**
Integer	**OM_S_INTEGER**
Null	**OM_S_NULL**
Object Identifier	String(**OM_S_OBJECT_IDENTIFIER_STRING**)
Octet String	String(**OM_S_OCTET_STRING**)
Real	None[1]

[1] A future edition of XOM may define a syntax corresponding to this type.

An example will illustrate how OM syntaxes are used to define the syntax of values for various attributes. One of the simplest of the ASN.1 types is Boolean. There are only two possible values for a Boolean type: TRUE and FALSE. The **DS_FROM_ENTRY** OM attribute of the **DS_C_ENTRY_INFO** object class has a value syntax of **OM_S_BOOLEAN. OM_S_BOOLEAN** is the C language representation for the OM syntax that corresponds to the ASN.1 Boolean type. The value of the **DS_FROM_ENTRY** OM attribute indicates whether information from the directory was extracted from the specified object's entry (TRUE), or from a copy of the entry (FALSE). The actual C language definition for **OM_S_BOOLEAN** is made in the XOM API header file **xom.h**.

25.5.2 Basic Encoding Rules

It is possible to define a single transfer syntax that is powerful enough to encode values drawn from a number of abstract syntaxes. ISO defines a set of rules for encoding values of many different types for ASN.1. This set of encoding rules is called Basic Encoding Rules (BER). It is so powerful that values from any abstract syntax described using ASN.1 can be encoded using the transfer syntax defined by BER.

Although other transfer syntaxes could be used for representing values from ASN.1, BER is used most often.

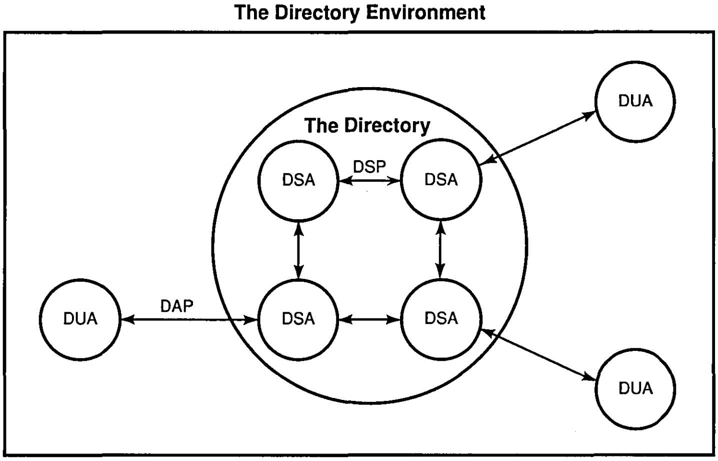
25.6 GDS as a Distributed Service

When present in a DCE cell, GDS can serve two basic functions. First, it can provide a high-level, worldwide directory service by tying together independent DCE cells. Second, it can be used as an additional directory service to CDS for storing object names and attributes in a central place.

The GDS database contains information that can be distibuted over several GDS servers. In addition, copies of information can be stored in multiple GDS servers, and the information can also be cached locally. The unit of replication in GDS is the directory entry; whole subtrees may be also replicated.

The information belonging to the DIB is shared between several Directory Service Agents (DSAs). A DSA is a process that runs on a GDS server machine and manages the GDS database. DSAs cooperate to perform directory service operations with each DSA knowing a fraction of the total directory information, as shown in Figure 25-11. DSAs are a combination of local database functions and a remote interface to the clients of users and other DSAs. DSAs may cooperate to execute operations. This cooperation often involves the navigation of operations through the network.

Figure 25–11. The Relationship Between the DSA and the DUA



Users access the directory via Directory User Agents (DUAs). DUAs make requests of DSAs on behalf of users requesting directory service operations. The manner in which DUAs communicate with DSAs is defined by the X.500 standard. For communications between DUAs and DSAs, the Directory Access Protocol (DAP) is defined. For communications between DSAs in a distributed directory, the standard defines the Directory System Protocol (DSP).

25.6.1 The Directory Access Protocol

The directory standard defines directory functions in the Directory Access Protocol (DAP). The directory functions can be divided into three general categories: read, search, and modify.

Read operations involve the retrieval of information from specific named entries. This allows a general name-to-attributes mapping analogous to the ''White Pages'' phone directory.

Search operations involve the general browsing and relational searching of information. Search operations support human interaction with the Directory Service and is analogous to that of the ''Yellow Pages'' telephone directory.

Modify operations involve the modification of information in the directory.

25.6.2 The Directory System Protocol

The DSA may interact with other DSAs to provide services using the Directory System Protocol (DSP). DSP is a protocol defined by the directory standard to allow DSAs to communicate with one another. DSP provides two methods of distributed request resolution: referral and chaining.

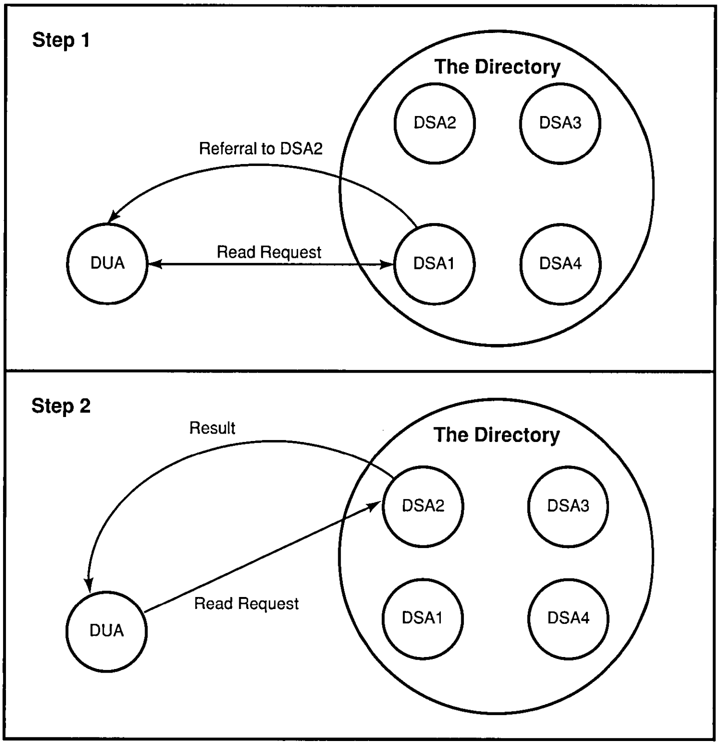
25.6.3 Referral

In some cases, a DSA may not be able to provide service to a DUA because the required information is held elsewhere in the network. A DSA may simply choose to inform the DUA or the calling DSA where the information can be found. This is called *referral* and may occur because of the user's preference or the DSA's circumstances.

Referrals are possible because the distinguished name provided by the DUA identifies where in the DIT the requested entry is located. DSAs use their knowledge of the DIT to inform the DUA of the DSA that holds the requested information.

Figure 25-12 shows an example of a referral. DSA1 passes a referral to DSA2 back to the DUA. The DUA then makes a request to DSA2.

Figure 25–12. An Example of a Referral



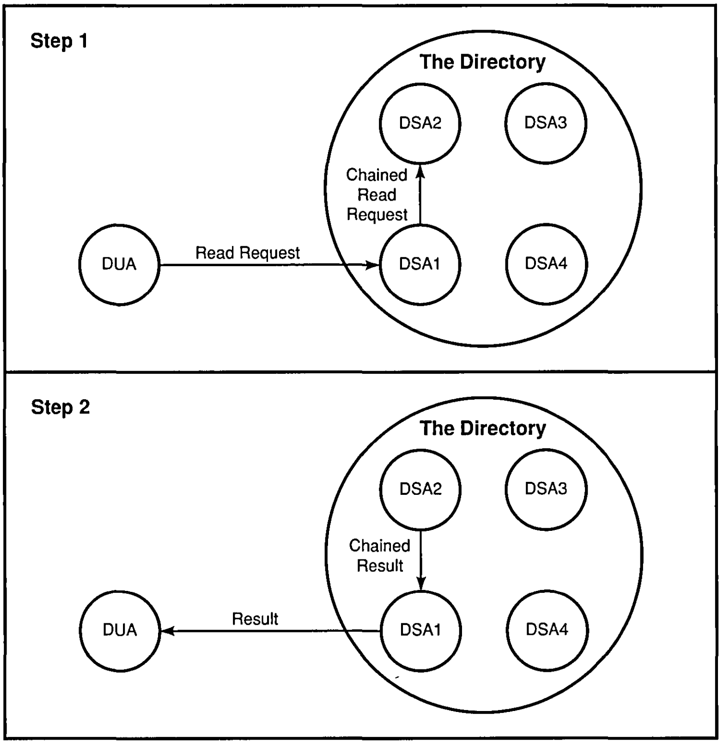
25.6.4 Chaining

If a request received from a DUA cannot be fulfilled by the receiving DSA, that DSA may send a referral back to the initiating DUA over DAP. Alternatively, the DSA may chain the request over DSP, asking another DSA to perform the requested function. That DSA may perform the function or may send back a referral of its own. In either case, the first DSA eventually responds to the originating DUA with either the results of the completed operation or a referral.

Chaining can go deeper than one level. To prevent lengthy searches, a user can request no chaining or specify a limit on the total elapsed time for an operation.

Figure 25-13 shows an example of chaining. The DUA makes a request of DSA1. DSA1 is unable to service the request and passes it to DSA2. DSA2 services the request, passes the result back to DSA1, and DSA1 passes the result back to the DUA.

Figure 25–13. An Example of Chaining



25.6.5 The Directory User Agent Cache

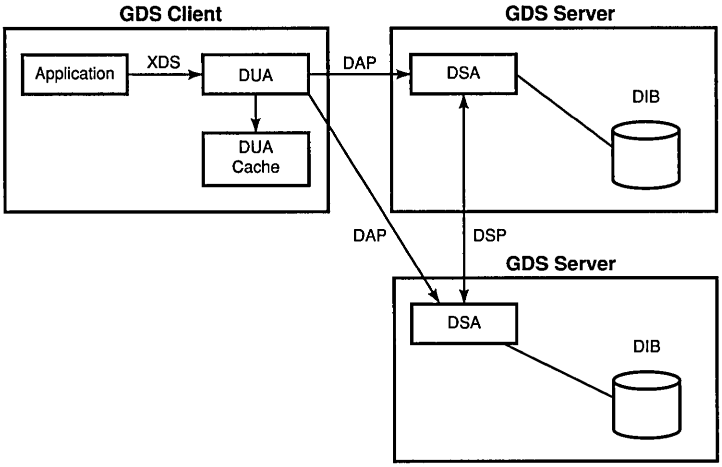
The Directory User Agent Cache is a process that keeps a cache of information obtained from DSAs. One DUA Cache runs on each client machine and is used by all the users on that machine. The DUA Cache contains copies of recently accessed object entries and information about DSAs. The user specifies which information should be cached. It is also possible to bypass the DUA Cache to obtain information directly from a DSA. This is desirable, for example, when the user wants to make sure the information obtained is up-to-date.

The Shadow Update and Cache Update are processes that update replicated information in DSAs and DUA Caches. These processes run as needed and then terminate. The Shadow Update process runs on the GDS server machine; the Cache Update process runs on GDS client machines.

When an application program makes a Directory Service call using XDS API, the call is handed to the DUA library. The DUA first looks in the DUA Cache (if requested by the user) to see if the requested information is already available on the local machine. If it is not, the DUA queries a DSA. The DSA may have the requested information, and if it does, it returns the results to the DUA. If it does not, the query can proceed either by using chaining or a referral. In either case, different DSAs are queried until the information is found. It is cached (if requested by the user) in the DUA Cache and the results are returned to the application program.

Figure 25-14 shows the interaction between an application program, via the XDS interface, and the GDS client and server. The GDS client and server use Directory Access Protocol (DAP) to communicate. The GDS Servers use the Directory Service Protocol (DSP) to communicate with one another. DAP and DSP perform functions similar to the functions that DCE RPC protocols perform in other DCE services.

Figure 25–14. GDS Components



A special object OM class, **DSX_C_GDS_CONTEXT**, is provided in the GDS Package to allow an application program to manage the placement of entries in the local DUA Cache as a result of a directory request.

DSX_C_GDS_CONTEXT inherits the OM attributes of its superclasses **OM_C_OBJECT** and **DS_C_CONTEXT**. To enable caching entries, the **DS_DONT_USE_COPY** OM attribute of **DS_C_CONTEXT** must be set to a value of **OM_FALSE**, indicating that a directory request can access copies of directory entries maintained in other DSAs or copies cached locally.

DSX_C_GDS_CONTEXT has the following private extension OM attributes in addition to the OM attributes inherited from **DS_C_CONTEXT**:

- **DSX_DUAFIRST**
- **DSX_DONT_STORE**
- **DSX_NORMAL_CLASS**
- **DSX_PRIV_CLASS**
- **DSX_RESIDENT_CLASS**

- **DSX_USEDSA**
- **DSX_DUA_CACHE**

DSX_DUAFIRST determines where a query operation, such as a search or list, looks first for an entry. The default value is **OM_FALSE**, indicating that the DSA is searched first. If the entry is not found, then the DUA Cache is searched.

DSX_DONT_STORE determines if information read from the DSAs by a query function also needs to be stored in the DUA Cache. If this OM attribute is set to **OM_TRUE**, nothing is stored in the cache. If this OM attribute is set to **OM_FALSE**, object entries returned from a list or compare operation are stored as distinguished names in the cache without associated attribute information. Object entries returned from a read or search operation are stored with all public attributes, except the ACL attribute.

The three different memory classes that the user can specify for a cached entry are **DSX_NORMAL_CLASS**, **DSX_PRIV_CLASS**, and **DSX_RESIDENT_CLASS**.

DSX_NORMAL_CLASS assigns the entry to the class of normal objects. If the number of entries in this class exceeds a maximum value, the entry that is not accessed for the longest period of time is removed from the DUA Cache.

DSX_PRIV_CLASS assigns the entry to the class of privileged objects. Entries can be removed from the class in the same way as normal objects. However, by setting this area of memory aside to be used sparingly, the user can protect entries from deletion.

DSX_RESIDENT_CLASS assigns the entry to the class of resident objects. An entry in this class is never removed automatically. It must be explicitly removed using an XDS **ds_remove_entry()** applied directly to the cache; that is, **DSX_DUA_CACHE** and **DSX_USEDSA** are set to **OM_TRUE** and **OM_FALSE**, respectively.

Tables 25-7 through 25-9 show the possible conditions that result when **DSX_DUA_CACHE** and **DSX_USEDSA** are set to **OM_TRUE**.

Table 25–7. Cache Attributes: Read Cache First

OM Attribute Type	OM_TRUE	OM_FALSE
DSX_DUA_CACHE	X	
DSX_USEDSA	X	
DS_DONT_USE_COPY		X
DSX_DUAFIRST	X	

In the situation presented in Table 25-7, the cache is read first, then the other DSAs. The requested operation is permitted to use copies of entries.

Table 25–8. Cache Attributes: Read DSA First

OM Attribute Type	OM_TRUE	OM_FALSE
DSX_DUA_CACHE	X	
DSX_USEDSA	X	
DS_DONT_USE_COPY		X
DSX_DUAFIRST		X

In the situation presented in Table 25-8, the DSA is read first, then the cache. The requested operation is permitted to use copies of entries.

Table 25–9. Cache Attributes: Read DSA Only

OM Attribute Type	OM_TRUE	OM_FALSE
DSX_DUA_CACHE	X	
DSX_USEDSA	X	
DS_DONT_USE_COPY	X	
DSX_DUAFIRST	N/A	N/A

In the situation presented in Table 25-9, only the DSA is read. The requested operation is not permitted to use copies of entries.

Tables 25-10 through 25-12 show the possible situations when **DSX_DUA_CACHE** and **DSX_USEDSA** are not both set to **OM_TRUE**.

Table 25–10. Cache Attributes: DSX_USEDSA is OM_FALSE

OM Attribute Type	OM_TRUE	OM_FALSE
DSX_DUA_CACHE	X	
DSX_USEDSA		X

In the situation presented in Table 25-10, the DUA Cache is used exclusively.

Table 25–11. Cache Attributes: DSX_DUA_CACHE is OM_FALSE

OM Attribute Type	OM_TRUE	OM_FALSE
DSX_DUA_CACHE		X
DSX_USEDSA	X	

In the situation presented in Table 25-11, the DSA is used excusively.

Table 25–12. Cache Attributes: Error

OM Attribute Type	OM_TRUE	OM_FALSE
DSX_DUA_CACHE		X
DSX_USEDSA		X

In the situation presented in Table 25-12, neither the DSA or the DUA Cache is used, and an error is returned.

25.6.6 GDS Configurations

A GDS machine can be configured in two ways:

- Client Only

  A node can contain only the client side of GDS. This node can access remote DSAs and cache information in the DUA Cache.

- Client/Server

  A machine can be configured with both the GDS client and server. This is the typical configuration for a machine acting as a GDS server. This configuration can be useful even if a node acts mainly as a client because the DSA can be used as a larger, more permanent cache of information contained in remote DSAs.

Note: When a client and server reside on the same machine, access to the directory is optimized. Communications between the DUA and the DSA are by means of Interprocess Communications (IPC) via shared memory.

25.6.7 GDS Security

To establish a session with a GDS server, an application program must perform a bind operation to a GDS server. This is accomplished by using the XDS **ds_bind( )** function. A bind operation can be performed by the application program with or without user credentials. A bind with credentials is referred to as an *authenticated bind* and allows an application program to require a user to specify a distinguished name password as user credentials. A bind without user credentials only permits access to public information in the directory.

A special OM object class, **DSX_C_GDS_SESSION**, is provided in the GDS Package to accommodate user credentials. In addition to the OM attributes inherited from its superclass **DS_C_SESSION**, this OM class consists of the following OM attributes:

- **DSX_PASSWORD**

  This attribute contains the password for the user credentials.

- **DSX_DIR_ID**

  This attribute contains the identifier for distinguishing between several configurations of the Directory Service within a GDS installation. **DSX_DIR_ID** plays no role in user credentials.

The GDS Package also provides the following special OM classes to support access rights to specific OM attributes by Directory Service users:

- **DSX_C_GDS_ACL**

  This attribute describes up to five categories of rights for one or more directory users.

- **DSX_C_GDS_ACL_ITEM**

  This attribute specifies the user, or subtree of users, to whom an access right applies.

The five categories of rights correspond to the access rights defined for the Directory Service as described in the *OSF DCE Administration Guide*. The categories are as follows:

- Modify Public
- Read Standard
- Modify Standard
- Read Sensitive
- Modify Sensitive

Refer to Chapter 27 for more information on binding with credentials and setting access rights for users. The sample programs in Chapter 28 provide examples of how security features are used in application programs.

Chapter 26

XOM Programming

XOM API defines a general-purpose interface for use in conjunction with other application-specific APIs for OSI services, such as XDS API to Directory Services or X.400 Application API to electronic mail service. It presents the application programmer with a uniform information architecture based on the concept of groups, classes, and similar information objects.

This chapter describes some of the basic concepts required to understand and use the XOM API effectively.

The following names:

- **acl.c** (**acl.h**)
- **example.c** (**example.h**)
- **teldir.c**

refer to the complete XDS example programs, which can be found in Chapter 28.

26.1 OM Objects

The purpose of XOM API is to provide an interface to manage complex information objects. These information objects belong to classes and have attributes associated with them. There are two distinct kinds of classes and attributes that are used throughout Part 4 of this guide: *directory* classes and attributes and *OM* classes and attributes.

The directory classes and attributes defined for XDS API correspond to entries that make up the objects in the directory. These classes and attributes are defined in the X.500 directory standard and by additional GDS extensions created for DCE. Other APIs, such as the X.400 Application Interface, which is the application interface for the industry standard X.400 electronic mail service, define their own set of objects in terms of classes and attributes. OM classes and OM attributes are used to model the objects in the directory.

XOM API provides a common information architecture so that the information objects defined for any API that conforms to this architectural model can be shared. Different application service interfaces can communicate using this common way of defining objects by means of workspaces. A workspace is simply a common work area where objects defined by a service can be accessed and manipulated. In turn, XOM API provides a set of standard functions that perform common operations on these objects in a workspace. Two different APIs can share information by copying data from one workspace to another.

26.1.1 OM Object Attributes

OM objects are composed of OM attributes. OM objects may contain zero or more OM attributes. Every OM attribute has zero or more values. An attribute comprises an integer that indicates the attribute's value. Each value is accompanied by an integer that indicates that value's syntax.

An OM attribute type is a category into which all the values of an OM attribute are placed on the basis of its purpose. Some OM attributes may either have zero, one, or multiple values. The OM attribute type is used as the name of the OM attribute.

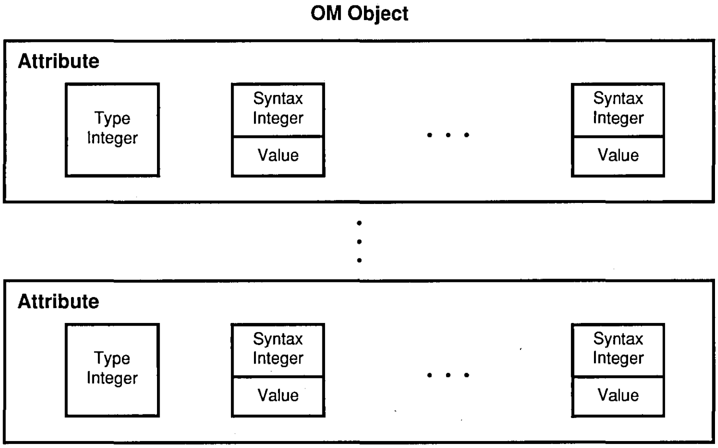
A syntax is a category into which a value is placed on the basis of its form. **OM_S_PRINTABLE_STRING** is an example of a syntax.

An OM attribute value is an information item that can be viewed as a characteristic or property of the OM object of which it is a part.

OM attribute types and syntaxes have integer values and symbolic equivalents assigned to them for ease of use by naming authorities in the various API specifications. The integers that are assigned to the OM attribute type and syntax are fixed, but the attribute values may change. These OM attribute types and syntaxes are defined in the DCE implementation of XDS and XOM APIs in header files that are included with the software along with additional OM attributes specific to the GDS implementation.

Figure 26-1 shows the internal structure of an OM object.

Figure 26–1. The Internal Structure of an OM Object



For example, the tables in Figure 26-2 show the OM attributes, syntax, and values for the OM class **DS_C_ENTRY_INFO_SELECTION**, and how the integer values are mapped to corresponding names in the **xom.h** and **xds.h** header files. The chapters in Part 4C of this guide contain tables for every OM class supported by the Directory Service. Refer to Chapter 30 for a complete description of **DS_C_ENTRY_INFO_SELECTION** and the accompanying table.

DS_C_ENTRY_INFO_SELECTION is a subclass of **OM_C_OBJECT.** This information is supplied in the description of this OM class in Chapter 35. As such, **DS_C_ENTRY_INFO_SELECTION** inherits the OM attributes of **OM_C_OBJECT.** The only OM attribute of **OM_C_OBJECT** is **OM_CLASS. OM_CLASS** identifies the object's OM class, which in this case is **DS_C_ENTRY_INFO_SELECTION. DS_C_ENTRY_INFO_SELECTION** identifies information to be extracted from a directory entry and has the following OM attributes, in addition to those inherited from **OM_C_OBJECT**:

- **DS_ALL_ATTRIBUTES**
- **DS_ATTRIBUTES_SELECTED**
- **DS_INFO_TYPE**

As part of an XDS function call, **DS_ALL_ATTRIBUTES** specifies to the Directory Service whether all the attributes of a directory entry are relevant to the application program. It can take the values **OM_TRUE** or **OM_FALSE.** These values are defined to be of syntax **OM_S_BOOLEAN.** The value **OM_TRUE** indicates that information is requested on all attributes in the directory entry. The value **OM_FALSE** indicates that information is only requested on those attributes that are listed in the OM attribute **DS_ATTRIBUTES_SELECTED.**

DS_ATTRIBUTES_SELECTED lists the types of attributes in the entry from which information is to be extracted. The syntax of the value is specified as **OM_S_OBJECT_IDENTIFIER_STRING.**

OM_S_OBJECT_IDENTIFIER_STRING contains an octet string of integers that are BER encoded object identifiers of the types of OM attributes in the OM attribute list. The value of **DS_ATTRIBUTES_SELECTED** is only significant if the value of **DS_ALL_ATTRIBUTES** is **OM_FALSE,** as described previously.

DS_INFO_TYPE identifies what information is to be extracted from each OM attribute identified. The syntax of the value is specified as Enum(**DS_INFORMATION_TYPE**). **DS_INFORMATION_TYPE** is an enumerated type that has two possible values: **DS_TYPES_ONLY** and **DS_TYPES_AND_VALUES. DS_TYPES_ONLY** indicates that only the attribute types in the entry are returned by the Directory Service operation. **DS_TYPES_AND_VALUES** indicates that both the types and the values of the attributes in the directory entry are returned.

Figure 26–2. Mapping the Class Definition of DS_C_ENTRY_INFO_SELECTION

OM Attributes of an **OM_C_OBJECT**

Attribute	Value Syntax	Value Length	Value Number	Value Initially
OM_CLASS	String **(OM_S_OBJECT_IDENTIFIER_STRING)**	—	1	—

OM Attributes of a **DS_C_ENTRY_INFO_SELECTION**

Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALL_ ATTRIBUTES	**OM_S_BOOLEAN**	—	1	**OM_TRUE**
DS_ATTRIBUTES_ SELECTED	String **(OM_S_OBJECT_IDENTIFIER_STRING)**	—	0 or more	—
DS_INFO_TYPE	Enum(**DS_Information_Type**)	—	1	**DS_TYPES AND_VALUES**

```
#define OM_CLASS ((OM_type)3)

#define OM_S_BOOLEAN ((OM_syntax)2)

#define OM_S_OBJECT_IDENTIFIER_STRING ((OM_syntax)14)

#define OM_S_ENUMERATION ((OM_syntax)4)
```

Sample code from the **xom.h** header file

```
enum DS_Information_Type {

               DS_TYPES_ONLY  = 0,
               DS_TYPES_AND_VALUES = 1

     };

               •
               •
               •

#define DS_ALL_ATTRIBUTES ((OM_type)707)

#define DS_ATTRIBUTES_SELECTED ((OM_type)710)

#define DS_INFO_TYPE ((OM_type)734)
```

Sample code from the **xds.h** header file

A typical Directory Service operation, such as a read operation (**ds_read**( )), requires the *entry_information_selection* parameter to specify to the Directory Service the information to be extracted from the directory entry. This *entry_information_selection* parameter is built by the application program as a public object (Section 26.1.4 describes how to create a public object), and is included as a parameter to the **ds_read**( ) function call, as shown in the following code fragment from **example.c**:

```
/*
 * Public Object ("Descriptor List") for
 * Entry-Information-Selection
 * parameter to ds_read().
 */
 OM_descriptor selection[] = {
 OM_OID_DESC(OM_CLASS,DS_C_ENTRY_INFO_SELECTION),
 { DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, { OM_FALSE, NULL } },
 OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR),
 { DS_INFO_TYPE,OM_S_ENUMERATION,
 { DS_TYPES_AND_VALUES,NULL } },
 OM_NULL_DESCRIPTOR
 };

 CHECK_DS_CALL(ds_read(session, DS_DEFAULT_CONTEXT,
                   name, selection, &result, &invoke_id));
```

26.1.2 Object Identifiers

OM classes are uniquely identifiable by means of ASN.1 object identifiers. OM classes have mandatory and optional OM attributes. Each OM attribute has a type, value, and syntax. OM objects are instances of OM classes that are uniquely identifiable by means of ASN.1 object identifiers. The syntax of values defined for these OM object classes and OM attributes are representations at a higher level of abstraction so that implementors can provide the necessary high-level language binding for their own implementations of the various application interfaces, such as XDS API.

The DCE implementation uses the C language to define the internal representation of OM classes and OM attributes. These definitions are supplied in the header files that are included as part of the XDS and XOM API.

OM classes are defined as symbolic constants that correspond to ASN.1 object identifiers. An ASN.1 object identifier is a sequence of integers that uniquely identifies a specific class. OM attribute type and syntax are defined as integer constants. These standardized definitions provide application programs with a uniform and stable naming environment in which to perform directory operations. Registration authorities are responsible for allocating the unique object identifiers.

The following code fragment from the **xdsbdcp.h** (the Basic Directory Contents Package) header file contains the symbolic constant **OMP_O_DS_A_COUNTRY_NAME:**

```
#ifndef dsP_attributeType /* joint-iso-ccitt(2) ds(5) attributeType(4) */
#define dsP_attributeType(X) ("\x55\xo4" #X)
#endif

#define OMP_O_DS_A_COUNTRY_NAME                 dsp_attributeType(\x06)
```

It resolves to 2.5.4.6, which is the object identifier value for the Country Name attribute type as defined in the directory standard. The symbolic constant for the directory object class Country resolves to 2.5.6.2, the corresponding object identifier in the directory standard. OM classes are defined in the header files in the same manner.

26.1.3 C Naming Conventions

In the DCE implementation of XDS and XOM APIs, all object identifiers start with the letters **ds**, **DS**, **MH**, or **OMP**. Note that the interface reserves *all* identifiers starting with the letters **dsP** and **omP** for internal use by implementations of the interface. It also reserves all identifiers starting with the letters **dsX**, **DSX**, **omX**, and **OMX** for vendor specific extensions of the interface. Applications programmers should not use any identifier starting with these letters.

The C identifiers for interface elements are formed using the following conventions:

- XDS API function names are specified entirely in lowercase letters, and are prefixed by **ds_** (for example, **ds_read( )**).
- XOM API function names are specified entirely in lowercase letters, and are prefixed by **om_** (for example, **om_get( )**).
- C function parameters are derived from the parameter and result names and are specified entirely in lowercase letters. In addition, the names of results have **_return** added as a suffix (for example, **operation_status_return**).
- OM class names are specified entirely in uppercase letters, and are prefixed by **DS_C_** and **MH_C_** (for example, **DS_C_AVA**).
- OM attribute names are specified entirely in uppercase letters, and are prefixed by **DS_** and **MH_** (for example, **DS_RDNS**).
- OM syntax names are specified entirely in uppercase letters, and are prefixed by **OM_S_** (for example, **OM_S_PRINTABLE_STRING**).
- Directory class names are specified entirely in uppercase letters, and are prefixed by **DS_O** (for example, **DS_O_ORG_PERSON**).
- Directory attribute names are specified entirely in uppercase letters, and are prefixed by **DS_A** (for example, **DS_A_COUNTRY_NAME**).
- Errors are treated as a special case. Constants that are the possible values of the OM attribute **DS_PROBLEM** of a subclass of the OM class **DS_C_ERROR** are specified entirely in uppercase letters, and are prefixed by **DS_E_** (for example, **DS_E_BAD_CLASS**).
- The constants in the Value Length and Value Number columns of the OM class definition tables are also assigned identifiers. Where the upper limit in one of these columns is *not* 1, it is given a name that consists of the OM attribute name, prefixed by **DS_VL_** for value length, or **DS_VN_** for value number.
- The sequence of octets for each object identifier is also assigned an identifier for internal use by certain OM macros. These identifiers are all uppercase letters and are prefixed by **OMP_O_**.

Tables 26-1 and 26-2 summarize the XDS and XOM naming conventions.

Table 26–1. C Naming Conventions for XDS

Item	Prefix
Reserved for implementors	**dsP**
Reserved for interface extensions	**dsX**
Reserved for interface extensions	**DSX**
XDS functions	**ds_**
Error problem values	**DS_E_**
OM class names	**DS_C_, MH_C_**
OM attribute names	**DS_, MH_**
OM value length limits	**DS_VL_**
OM value number limits	**DS_VN_**
Other constants	**DS_, MH_**
Attribute type	**DS_A_**
Object class	**DS_O_**

Table 26–2. C Naming Conventions for XOM

Element Type	Prefix
Data type	**OM_**
Data value	**OM_**
Data value (class)	**OM_C_**
Data value (syntax)	**OM_S_**
Data value component (structure member)	None
Function	**om_**
Function parameter	None
Function result	None
Macro	**OM_**
Reserved for use by implementors	**OMP**
Reserved for use by implementors	**omP**
Reserved for proprietary extension	**omX**
Reserved for proprietary extension	**OMX**

26.1.4 Public Objects

The ultimate aim of an application program is access to the directory to perform some operation on the contents of the directory. A user may request the telephone number or electronic mail address of a fellow employee. In order to access this information, the application performs a read operation on the directory so that information is extracted about a target object in the directory and manipulated locally within the application.

XDS functions that perform directory operations, such as **ds_read()**, require *public* and *private* objects as input parameters. Typically, a public object is generated by an application program and contains the information required to access a target directory object. This information includes the AVAs and RDNs that make up a distinguished name of an entry in the directory. However, an application program may also generate a private object. Private objects are described in Section 26.1.5.

A public object is created using OM classes and OM attributes. These OM classes and OM attributes model the target object entry in the directory and provide other information required by the Directory Service to access the directory.

26.1.4.1 Descriptor Lists

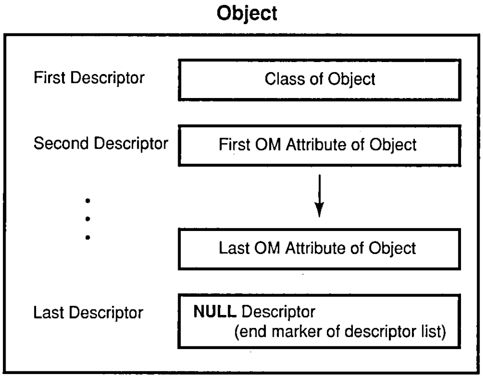
A public object is represented by a sequence of **OM_descriptor** data structures that is built by the application program. A descriptor contains the type, syntax, and value for an OM attribute in a public object.

The data structure **OM_descriptor** is defined in the **xom.h** header file as follows:

```
typedef struct OM_descriptor_struct {
        OM_type                 type;
        OM_syntax               syntax;
        union OM_value_union    value;
}OM_descriptor;
```

Figure 26-3 shows the representation of a public object in a descriptor list. The first descriptor in the list indicates the object's OM class; the last descriptor is a **NULL** descriptor that signals the end of the list of OM attributes. In between the first and the last descriptor are the descriptors for the OM attributes of the object.

Figure 26–3. A Representation of a Public Object Using a Descriptor List


Object
First Descriptor
Class of Object
Second Descriptor
First OM Attribute of Object
Last OM Attribute of Object
Last Descriptor
NULL Descriptor
(end marker of descriptor list)

For example, the following represents the public object **country** in **example.c**:

```
static OM_descriptor        country[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING,OM_STRING("US") },
 OM_NULL_DESCRIPTOR
 };
```

The descriptor list is an array of data type **OM_descriptor** that defines the OM class, OM attribute types, syntax, and values that make up a public object.

The first descriptor gives the OM class of the object. The OM class of the object is defined by the OM attribute type, **OM_CLASS.** The **OM_OID_DESC** macro initializes the syntax and value of an object identifier, in this case to OM class **DS_C_AVA**, with syntax of **OM_S_OBJECT_IDENTIFIER_STRING.**

OM_S_OBJECT_IDENTIFIER_STRING is an OM syntax type that is assigned by definition in the macro to any OM attribute type and value parameters input to it.

The second descriptor defines the first OM attribute type, **DS_ATTRIBUTE_TYPE**, which has as its value

DS_A_COUNTRY_NAME and syntax **OM_S_OBJECT_IDENTIFIER_STRING.**

The third descriptor specifies the AVA of an object entry in the directory. The **OM_OID_DESC** macro is not used here because **OM_OID_DESC** is only used to initialize values having **OM_S_OBJECT_IDENTIFIER_STRING** syntax. The OM attribute type is **DS_ATTRIBUTE_VALUES,** the syntax is **OM_S_PRINTABLE_STRING,** and the value is **US.** The **OM_STRING** macro creates a data value for a string data type (data type **OM_string**), in this case **OM_S_PRINTABLE_STRING.** A string is specified in terms of its length or whether or not it terminates with a **NULL.** (The **OM_STRING** macro is described in Section 26.8.4.2.)

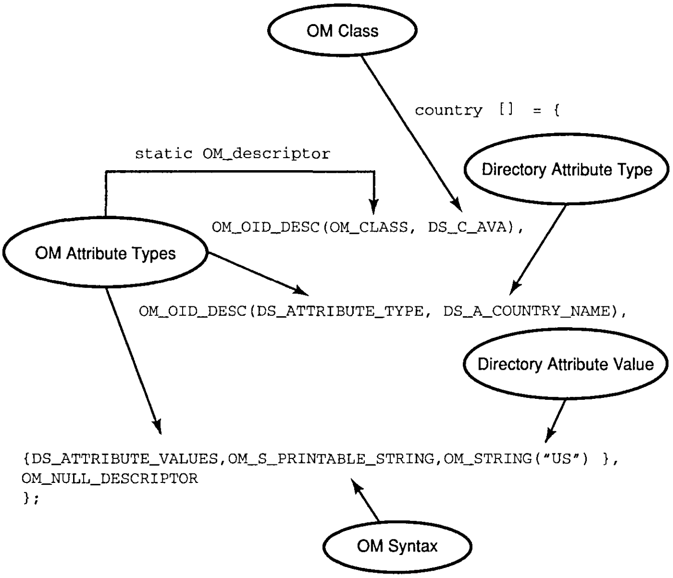
The last descriptor is a **NULL** descriptor that marks the end of the public object definition. It is defined in the **xom.h** header file as follows:

```
#define OM_NULL_DESCRIPTOR
  { OM_NO_MORE_TYPES, OM_S_NO_MORE_SYNTAXES,
  { { 0, OM_ELEMENTS_UNSPECIFIED } } }
```

OM_NULL_DESCRIPTOR is OM attribute type **NO_MORE_TYPES,** syntax **OM_S_NO_MORE_SYNTAXES,** and value **OM_ELEMENTS_UNSPECIFIED.**

Figure 26-4 shows the composition of a descriptor list representing a public object.

Figure 26–4. A Descriptor List for the Public Object: country



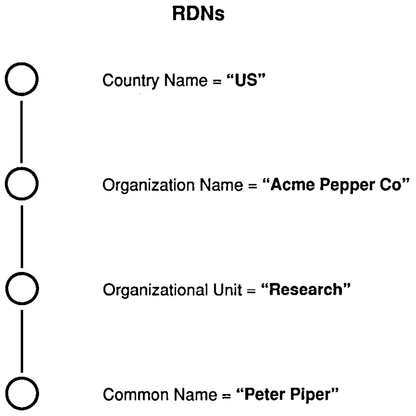
```
static OM_descriptor country [] = {
OM_OID_DESC(OM_CLASS, DS_C_AVA),
OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
{DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING,OM_STRING("US") },
OM_NULL_DESCRIPTOR
};
```

26.1.4.2 Building the Distinguished Name as a Public Object

Recall that RDNs are built from AVAs and a distinguished name is built from a series of RDNs. In a typical application program, several AVAs are defined in descriptor lists as public objects. These public objects are incorporated into descriptor lists that represent corresponding RDNs. Finally, the RDNs are incorporated into one descriptor list that represents the distinguished name of an object in the directory, as shown in Figure 26-5. This descriptor list is included as one of the input parameters to a Directory Service function.

The following code fragment from **example.c** shows how a distinguished name is built as a public object. The public object is the *name* parameter for a subsequent read operation call to the directory. The representation of a distinguished name in the DIT is shown in Figure 26-5.

Figure 26–5. The Distinguished Name of “Peter Piper” in the DIT


RDNs
Country Name = **“US”**
Organization Name = **“Acme Pepper Co”**
Organizational Unit = **“Research”**
Common Name = **“Peter Piper”**

Distinguished Name = **{C=US, O=Acme Pepper Co, OU=Research, CN=Peter Piper}**

The first section of code defines the four AVAs. These AVAs make the assertion to the Directory Service that the attribute values in the distinguished name of **Peter Piper** are valid and can therefore be read from the directory. The country name is **US**, the organization name is **Acme Pepper Co**, the organizational unit name is **Research**, and the common name is **Peter Piper**.

```
/*
 * Public Object ("Descriptor List") for Name parameter to
 * ds_read().
 * Build the Distinguished-Name of Peter Piper
 */

static OM_descriptor        country[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING,OM_STRING("US") },
 OM_NULL_DESCRIPTOR
 };
```

```
static OM_descriptor        organization[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_TELETEX_STRING, OM_STRING("Acme Pepper Co") },
 OM_NULL_DESCRIPTOR
 };
static OM_descriptor        organizational_unit[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_UNIT_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_TELETEX_STRING, OM_STRING("Research") },
 OM_NULL_DESCRIPTOR
 };
static OM_descriptor        common_name[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_TELETEX_STRING, OM_STRING("Peter Piper") },
 OM_NULL_DESCRIPTOR
 };
```

The next section of code is nested one level above the previously defined AVAs. Each RDN has a descriptor with OM attribute type **DS_AVAS** (indicating that it is OM attribute type AVA), a syntax of **OM_S_OBJECT**, and a value of the name of the descriptor array defined in the previous section of code for an AVA. The **rdn1** descriptor contains the descriptor list for the AVA country, the **rdn2** descriptor contains the descriptor list for the AVA organization, and so on.

OM_S_OBJECT is a syntax that indicates that its value is a subobject. For example, the value for **DS_AVAS** is the previously defined object **country**. In this manner a hierarchy of linked objects and subobjects may be constructed.

```
static OM_descriptor        rdn1[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 { DS_AVAS, OM_S_OBJECT, { 0, country } },
 OM_NULL_DESCRIPTOR
 };
static OM_descriptor        rdn2[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 { DS_AVAS, OM_S_OBJECT, { 0, organization } },
 OM_NULL_DESCRIPTOR
 };
```

```
static OM_descriptor        rdn3[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 { DS_AVAS, OM_S_OBJECT, { 0, organizational_unit } },
 OM_NULL_DESCRIPTOR
  };
static OM_descriptor        rdn4[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
 { DS_AVAS, OM_S_OBJECT, { 0, common_name } },
 OM_NULL_DESCRIPTOR
 };
```

The next section of code contains the RDNs that make up the distinguished name, which is stored in the array of descriptors called **name**. It is made up of the OM class **DS_C_DS_DN** (representing a distinguished name) and four RDNs of OM attribute type **DS_RDNS** and syntax **OM_S_OBJECT**.

```
OM_descriptor       name[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
 { DS_RDNS, OM_S_OBJECT, { 0, rdn1 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn2 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn3 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn4 } },
 OM_NULL_DESCRIPTOR
 };
```

In summary, the distinguished name for **Peter Piper** is stored in the array of descriptors called **name**, which is composed of three nested levels of arrays of descriptors (see Figure 26-6). The definitions for the AVAs are at the innermost level, the definitions for RDNs are at the next level up, and the distinguished name is at the top level.

Figure 26–6. Building a Distinguished Name

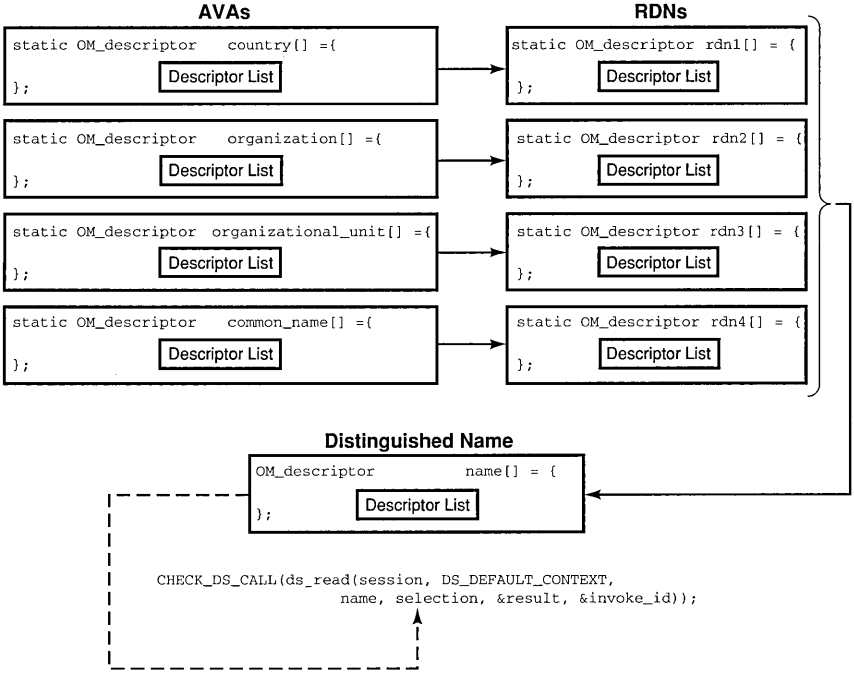
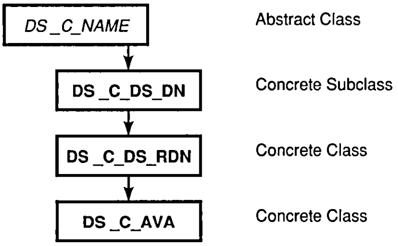

AVAs
RDNs
static OM_descriptor country[] ={
Descriptor List
};
static OM_descriptor organization[] ={
Descriptor List
};
static OM_descriptor organizational_unit[] ={
Descriptor List
};
static OM_descriptor common_name[] ={
Descriptor List
};
static OM_descriptor rdn1[] = {
Descriptor List
};
static OM_descriptor rdn2[] = {
Descriptor List
};
static OM_descriptor rdn3[] = {
Descriptor List
};
static OM_descriptor rdn4[] = {
Descriptor List
};
Distinguished Name
OM_descriptor name[] = {
Descriptor List
};
CHECK_DS_CALL(ds_read(session, DS_DEFAULT_CONTEXT,
name, selection, &result, &invoke_id));

Figure 26-7 shows a more general view of the structure distinguished name.

Figure 26–7. A Simplified View of the Structure of a Distinguished Name



Note: Abstract classes are shown in italics.

The **name** descriptor defines a public object that is provided as the *name* parameter required by the XDS API read function call, **ds_read**(), as follows (XDS API function calls are described in detail in Chapter 27):

```
CHECK_DS_CALL(ds_read(session, DS_DEFAULT_CONTEXT,
                       name, selection, &result, &invoke_id));
```

The result of the **ds_read**() function call is in a private implementation-specific format; it is stored in a workspace and pointed to by **result**. The application program must use XOM function calls (described in Section 26.7) to interpret the data and extract the information. This extraction process involves uncovering the nested data structures in a series of XOM function calls.

26.1.4.3 Client-Generated and Service-Generated Public Objects

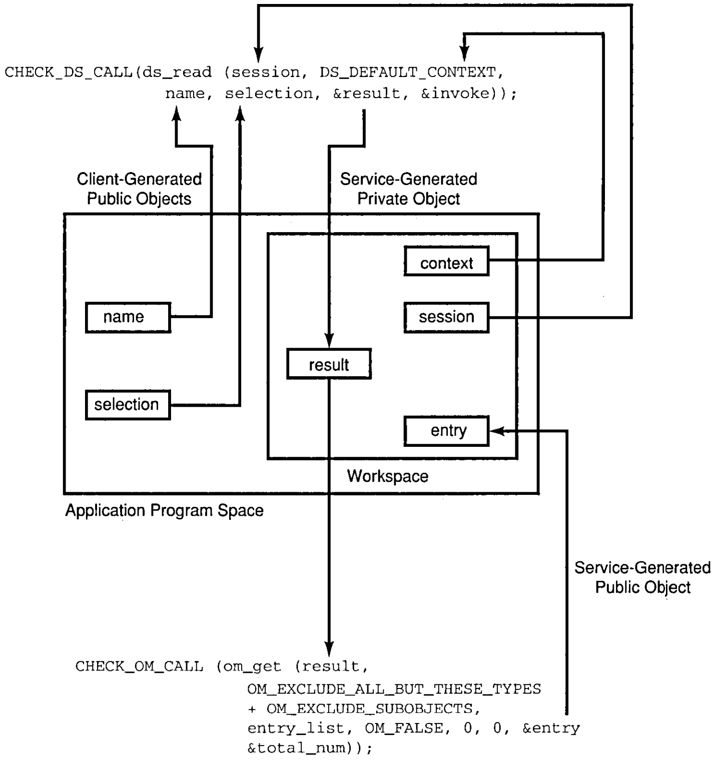
There are two types of public objects: service-generated objects and client-generated objects. The distinguished name object described in the previous section is a client-generated public object because an application program (the client) created the data structure. As the creator of the public object, it is the responsibility of the application program to manage the memory resources allocated for it.

Service-generated public objects are created by the XOM service. Service-generated public objects may be generated as a result of an XOM request.

An XOM API function, such as **om_get( )**, converts a private object into a service-generated public object. This is necessary because XDS may return a pointer to data in private format that can only be interpreted by XOM functions such as **om_get( )**.

For example, Figure 26-8 shows how the read request described in the previous example returns a pointer to an encoded data structure stored in **result**. This encoded data structure, referred to as a *private object* (described in the next section) is one of the input parameters to **om_get( )**. The **om_get( )** function provides a pointer to a public object (in this case, **entry**) as an output parameter. The public object is a data structure that has been interpreted by **om_get( )** and is accessible by the application program (the client). The information requested by the application in the read request is contained in the output parameter **entry**.

Figure 26–8. Client-Generated and Service-Generated Objects



```
CHECK_DS_CALL(ds_read (session, DS_DEFAULT_CONTEXT,
                name, selection, &result, &invoke));
```

```
CHECK_OM_CALL (om_get (result,
                OM_EXCLUDE_ALL_BUT_THESE_TYPES
                + OM_EXCLUDE_SUBOBJECTS,
                entry_list, OM_FALSE, 0, 0, &entry
                &total_num));
```

The application program is responsible for managing the storage (memory) for the service-generated public object. This is an important point because it requires that the application issue a series of **om_delete( )** calls to delete the service-generated public object from memory. Because the data structures involved with Directory Service requests can be very large (often involving large subtrees of the DIT), it is imperative that the application programmer build into any application program the efficient management of memory resources.

The following code fragment from **example.h** demonstrates how storage for public and private objects is released using a series of **om_delete( )** function calls after they are no longer needed by the application program. The data (a list of phone numbers associated with the name **Peter Piper** required by the application program) has already been extracted using a series of **om_get( )** function calls:

```
 /*  We can now safely release all the private objects
 *  and the public objects we no longer need
 */
CHECK_OM_CALL(om_delete(session));
CHECK_OM_CALL(om_delete(result));
CHECK_OM_CALL(om_delete(entry));
CHECK_OM_CALL(om_delete(attributes));
CHECK_DS_CALL(ds_shutdown(workspace));
```

26.1.5 Private Objects

Private objects are created dynamically by the service interface. In Figure 26-8 the **ds_read( )** function returns a pointer to the data structure **result** in the workspace. This service-generated data structure is a private object in a private implementation-specific format, which requires a call to **om_get( )** to interpret the data. A private object is one of the required input parameters to XOM API functions (such as **om_get( )**), as shown in Figure 26-8. Private objects are always service generated.

Table 26-3 compares private and public objects.

Table 26–3. Comparison of Private and Public Objects

Private	Public
Representation is implementation specific	Representation is defined in the API specification
Not directly accessible by the client	Directly accessible by the client
Manipulated by the client using OM functions	Manipulated by the client using programming constructs
Created in storage provided by the service	Is a service-generated object if created by the service Is a client-generated object if created by the client in storage provided by the client
Cannot be modified by the client directly, except through the service interface	If a client-generated object, can be modified directly by the client If a service-generated object, cannot be modified directly by the client, except through the service interface
Storage is allocated and released by the service	If a service-generated object, storage is allocated and released by the service If a client-generated object, storage is allocated and released by the client

Private objects may also be used as input to XOM and XDS API functions to improve program efficiency. For example, the output of a **ds_search()** request may be used as input to a **ds_read()**. The search request returns the name of each entry in the search. If the application program requires the address and telephone number of each name, a **ds_read()** operation can be performed on each name as a private object.

26.1.6 Object Classes

Objects are categorized into OM classes based on their purpose and internal structure. An object is an instance of its OM class. An OM class is characterized by OM attribute types that may appear in its instances. An OM class is uniquely identified by an ASN.1 object identifier.

Later in this section, it will be shown how OM classes are organized into groups of OM classes, called packages, that support some aspect of the Directory Service.

26.1.6.1 OM Class Hierarchy and Inheritance Properties

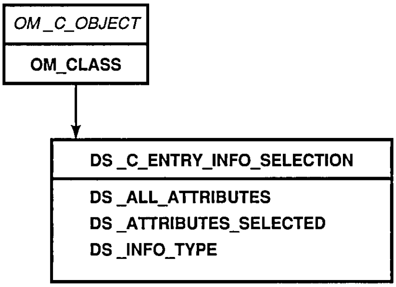
OM classes are related to each other in a tree hierarchy whose root is a special OM class called **OM_C_OBJECT**. Each of the other OM classes is the immediate subclass of precisely one other OM class. This tree structure is known as the *OM class hierarchy*. It is important because of the property of inheritance. The OM class hierarchy is defined by the XDS/XOM. DCE implements this hierarchy for GDS and adds its own set of OM classes defined in the GDS Package.

The OM attribute types that may exist in an instance of an OM class but not in an instance of the OM class above in the tree hierarchy are said to be *specific* to that OM class. OM Attributes that may appear in an object are those specific to its OM class as well as those inherited from OM classes above it in the tree. OM Classes above an instance of an OM class in the tree are *superclasses* of that OM class. OM Classes below an instance of an OM class are *subclasses* of that OM class.

For example, as shown in Figure 26-9, **DS_C_ENTRY_INFO_SELECTION** inherits its OM attributes from its superclass **OM_C_OBJECT**. The OM attributes **DS_ALL_ATTRIBUTES**, **DS_ATTRIBUTES_SELECTED**, and

DS_INFO_TYPE are attributes that are specific to the OM class **DS_C_ENTRY_INFO_SELECTION**, which has no subclasses.

Figure 26–9. The OM Class DS_C_ENTRY_INFO_SELECTION



Note: Abstract classes are shown in italics.

Another important point about OM class inheritance is that an instance of an OM class also is considered to be an instance of each of its superclasses and may appear wherever the interface requires an instance of any of those superclasses. For example, **DS_C_DS_DN** is a subclass of **DS_C_NAME**. Everywhere in an application program where **DS_C_NAME** is expected at the interface (as a parameter to **ds_read**( ), for example), it is permitted to supply **DS_C_DS_DN**.

26.1.6.2 Abstract and Concrete Classes

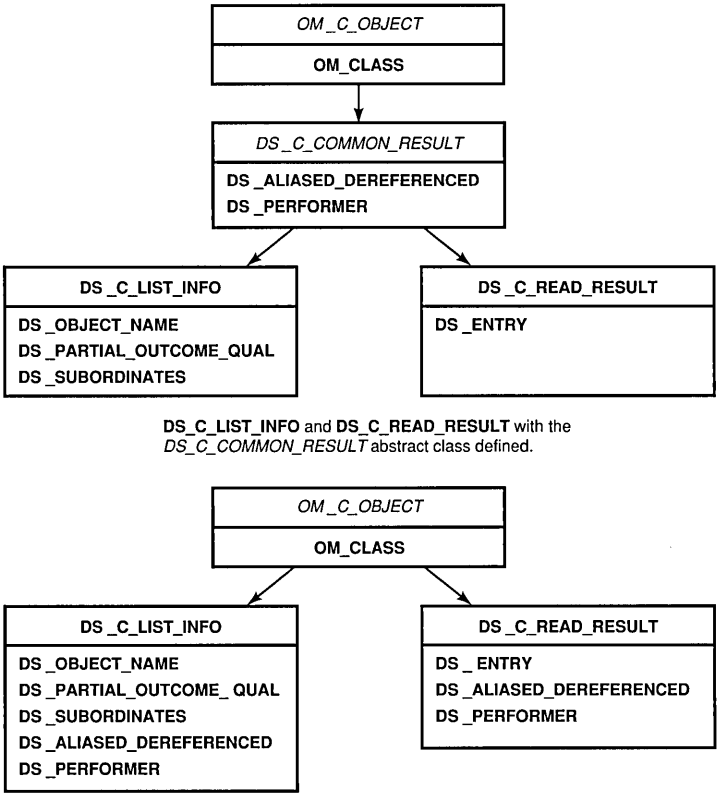
OM classes are defined as *abstract* or *concrete*.

An abstract OM class is an OM class in which instances are not permitted. An abstract OM class may be defined so that subclasses can share a common set of OM attributes between them.

In contrast to abstract OM classes, instances of OM concrete classes are permitted. However, the definition of each OM concrete class may include the restriction that a client not be allowed to create instances of that OM class. For example, consider two alternative means of defining the OM classes used in XDS: **DS_C_LIST_INFO** and **DS_C_READ_RESULT**. **DS_C_LIST_INFO** and **DS_C_READ_RESULT** are subclasses of the abstract OM class **DS_C_COMMON_RESULT**.

Figure 26-10 shows the relationship of **DS_C_LIST_INFO** and **DS_C_READ_RESULTS** when the abstract OM class **DS_C_COMMON_RESULT** is defined and when it is not defined. It demonstrates that the presence of an abstract OM class enables the programmer to develop applications that process information more efficiently.

Figure 26–10. A Comparison of Two Classes With and Without an Abstract OM Class



DS_C_LIST_INFO and **DS_C_READ_RESULT** with the *DS_C_COMMON_RESULT* abstract class defined.

DS_C_LIST_INFO and **DS_C_READ_RESULT** without the *DS_C_COMMON_RESULT* abstract class defined.

Note: Abstract classes are shown in italics.

The following list contains the hierarchy of concrete and abstract OM classes in the Directory Service Package. Abstract OM classes are shown in italics. The indentation shows the class hierarchy; for example, the abstract class *OM_C_OBJECT* is a superclass of the abstract class *DS_C_COMMON_RESULTS*, which in turn is a superclass of the concrete class **DS_C_COMPARE_RESULT**.

OM_C_OBJECT

- **DS_C_ACCESS_POINT**
- *DS_C_ADDRESS*
  - — **DS_C_PRESENTATION_ADDRESS**
- **DS_C_ATTRIBUTE**
  - — **DS_C_AVA**
  - — **DS_C_ENTRY_MOD**
  - — **DS_C_FILTER_ITEM**
- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_ATTRIBUTE_LIST**
  - — **DS_C_ENTRY_INFO**
- *DS_C_COMMON_RESULTS*
  - — **DS_C_COMPARE_RESULT**
  - — **DS_C_LIST_INFO**
  - — **DS_C_READ_RESULT**
  - — **DS_C_SEARCH_INFO**
- **DS_C_CONTEXT**
- **DS_C_CONTINUATION_REF**
  - — **DS_C_REFERRAL**
- **DS_C_ENTRY_INFO_SELECTION**
- **DS_C_ENTRY_MOD_LIST**
- *DS_C_ERROR*
  - — **DS_C_ABANDON_FAILED**

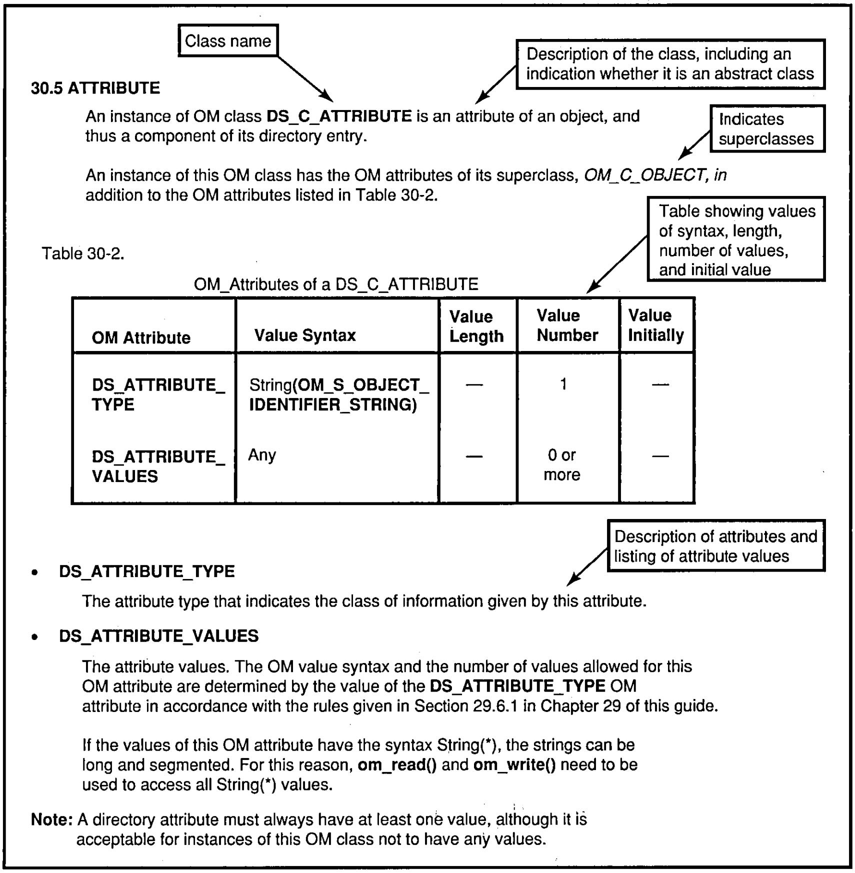
  - **DS_C_ATTRIBUTE_PROBLEM**
  - **DS_C_COMMUNICATIONS_ERROR**
  - **DS_C_LIBRARY_ERROR**
  - **DS_C_NAME_ERROR**
  - **DS_C_SECURITY_ERROR**
  - **DS_C_SERVICE_ERROR**
  - **DS_C_SYSTEM_ERROR**
  - **DS_C_UPDATE_ERROR**
- **DS_C_EXT**
- **DS_C_FILTER**
- **DS_C_LIST_INFO_ITEM**
- **DS_C_LIST_RESULT**
- *DS_C_NAME*
  - **DS_C_DS_DN**
- **DS_C_OPERATION_PROGRESS**
- **DS_C_PARTIAL_OUTCOME_QUAL**
- *DS_C_RELATIVE_NAME*
  - **DS_C_DS_RDN**
- **DS_C_SEARCH_RESULT**
- **DS_C_SESSION**

In summary, an OM class is defined with the following elements:

- OM class name (indicated by an object identifier)
- Identity of its immediate superclass
- Definitions of the OM attribute types specific to the OM class
- Indication whether the OM class is abstract or concrete
- Constraints on the OM attributes

A complete description of OM classes, OM attributes, syntaxes, and values that are defined for XDS and XOM APIs are described in Part 4C. Tables and textual descriptions, such as the one shown in Figure 26-11 for the concrete OM class **DS_C_ATTRIBUTE**, are provided for each OM class in these chapters in Part 4B.

Figure 26–11. A Complete Description of Concrete OM Class DS_C_ATTRIBUTE



Class name

Description of the class, including an indication whether it is an abstract class

Indicates superclasses

Table showing values of syntax, length, number of values, and initial value

Description of attributes and listing of attribute values

30.5 ATTRIBUTE

An instance of OM class **DS_C_ATTRIBUTE** is an attribute of an object, and thus a component of its directory entry.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT, in* addition to the OM attributes listed in Table 30-2.

Table 30-2.

OM_Attributes of a DS_C_ATTRIBUTE

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ATTRIBUTE_ TYPE	String(**OM_S_OBJECT_ IDENTIFIER_STRING)**	—	1	—
DS_ATTRIBUTE_ VALUES	Any	—	0 or more	—

- **DS_ATTRIBUTE_TYPE**

  The attribute type that indicates the class of information given by this attribute.

- **DS_ATTRIBUTE_VALUES**

  The attribute values. The OM value syntax and the number of values allowed for this OM attribute are determined by the value of the **DS_ATTRIBUTE_TYPE** OM attribute in accordance with the rules given in Section 29.6.1 in Chapter 29 of this guide.

  If the values of this OM attribute have the syntax String(*), the strings can be long and segmented. For this reason, **om_read()** and **om_write()** need to be used to access all String(*) values.

Note: A directory attribute must always have at least one value, although it is acceptable for instances of this OM class not to have any values.

The table shown in Figure 26-11 provides information under the following headings:

- OM Attribute

  The name of each of the OM attributes

- Value Syntax

  The syntaxes of each of the OM attribute's values

- Value Length

  Any constraints on the number of bits, octets, or characters in each value that is a string

- Value Number

  Any constraints on the number of values

- Value Initially

  Any value with which the OM attribute can be initialized

An OM class can be constrained to contain only one member of a set of OM attributes. In turn, OM attributes can be restricted to having no more than a fixed number of values, either 0 (zero) or 1 as an optional value, or exactly one mandatory value.

An OM attribute's value may be also constrained to a single syntax. That syntax can be further restricted to a subset of defined values.

An object passed as a parameter to an XOM and XDS function call needs to meet a minimum set of conditions:

- The type of each OM attribute must be specific to the object's OM class or one of its superclasses.
- The number of values of each OM attribute must be within OM class limits.
- The syntax of each value must be among those the OM class permits.
- The number of bits, octets, or characters in each string value must be within OM class limits.

26.2 Packages

A *package* is a collection of OM classes that are grouped together, usually by function. The packages themselves are features that are negotiated with the Directory Service using the XDS function **ds_version()**. Consider what OM classes will be required for your application programs and determine the packages that contain these OM classes.

A package is uniquely identified by an ASN.1 object identifier. DCE XDS API supports four packages of which one is mandatory and three are optional:

- The Directory Service Package (mandatory)
- The Basic Directory Contents Package (optional)
- The Global Directory Service Package (optional)
- The MHS Directory User Package (optional)

26.2.1 The Directory Service Package

The Directory Service Package is the default package and as such does not require negotiation. The optional packages have to be negotiated with the Directory Service using the **ds_version()** function.

The object identifiers for specific packages are defined in header files that are part of the XDS API and XOM API. An object identifier consists of a string of integers. The header files include **#define** preprocessor statements that assign names to these constants in order to make them more readable. These assignments alleviate the application programmer from the burden of maintaining these strings of integers. For example, the object identifiers for the Directory Service Package are defined in **xds.h**. The **xds.h** header file contains OM class and OM attribute names, OM object constants, and defines prototypes for XDS API functions, as shown in the following code fragment from **xds.h**:

```
/* DS package object identifier */
/* {iso(1) identifier-organization(3) icd-ecma(12)
 *  member-company(2)
 *  dec(1011) xopen(28) dsp(0) } */

#define OMP_O_DS_SERVICE_PKG   "\x2B\x0C\x02\x87\x1C\x00"
```

A **ds_version**( ) function call must be included within an application program to negotiate the optional features (packages) with the Directory Service. The first step is to build an array of object identifiers for the optional packages to be negotiated (the Basic Directory Contents Package and the Global Directory Service Package), as shown in the following code fragment from the **acl.h** header file:

```
DS_feature features[] = {
 { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
 { OM_STRING(OMP_O_DSX_GDS_PKG), OM_TRUE },
 { 0 }
};
```

The **OM_STRING** macro is provided for creating a data value of data type **OM_string** for octet strings and characters. XOM API macros are described in Section 26.8.2.

The array of object identifiers is stored in **features**, and passed as an input parameter to **ds_version**( ), as shown in the following code fragment from **acl.c**:

```
/* Negotiate the use of the BDCP and GDSP packages. */

if (ds_version(features) != DS_SUCCESS)
   printf("ds_version() error\n");
```

26.2.2 The Basic Directory Contents Package

The Basic Directory Contents Package contains the object identifier definition of directory classes and attribute types as defined by the X.500 standard. These definitions allow the creation of and maintainence of directory entries for a number of common objects so that the representation of all such objects is the same throughout the directory. Also included are the definitions of the OM classes and OM attributes required to support the directory attribute types.

The object identifier associated with the Basic Directory Contents Package is shown in the following code fragment from the **xdsbdcp.h** header file:

```
/* BDCP package object identifier */
/* { iso(1) identifier-organization(3) icd-ecma(12)
 *   member-company (2)
 *   dec(1011) xopen(28) bdcp(1) } */

#define OMP_DS_BASIC_DIR_CONTENTS_PKG "\x2B\x0C\x02\x87\x73\x1C\x01"
```

26.2.3 The Global Directory Service Package

The Global Directory Service Package contains the definition of a DCE extension to the XDS API. It contains the definitions of OM classes, OM attributes, and syntaxes to support extended functionality specific to DCE. Chapter 34 describes the GDSP package in detail.

The following code fragment from the **xdsgds.h** header file shows the object identifier for the GDSP package:

```
/* GDSP package object identifier */
/* { iso(1) identifier-organization(3) icd-ecma(12) member-company (2)
/*    siemens-units(1107) sni(1) directory(3) xds-api(100) gdsp(1) } */

#define OMP_O_DSX_GDS_PKG    "\x2B\x0C\x02\x88\x53\x01\x03\x64\x01"
```

Part 4C of this guide describes in detail the attributes and data types that make up the OM and directory classes defined in the XDS API packages. Chapter 28 examines in detail how these packages are used in developing the sample application programs.

26.2.4 Package Closure

An OM class may be defined to have an attribute whose OM class is defined in some other package. This avoids duplication of OM classes. This gives rise to the concept of a package closure. A package closure is the set of all OM classes that need to be supported so that all possible instances of all OM classes can be defined in the package.

26.3 Workspaces

Two application-specific APIs or two different implementations of the same service require work areas, called workspaces, to maintain private and public (service-generated) objects. The workspace is required because two implementations of the same service (or different services) can represent private objects differently. Each one has its own workspace. Using the functions provided by XOM API, such as **om_get( )** and **om_copy( )**, objects can by copied and moved from one workspace to another.

Recall that private objects are returned by a service to a workspace in private implementation-specific format. Using the OM function calls described in Section 26.7, the data can be extracted from the private object for further program processing.

Before a request to the directory can be made by an application program, a workspace must be created using the appropriate XDS function. An application creates a workspace by performing the XDS API call **ds_initialize( )**. Once the workspace is obtained, subsequent XDS API calls, such as **ds_read( )**, return a pointer to a private object in the workspace. When program processing is completed, the workspace is destroyed using the **ds_shutdown( )** XDS API function. Implicit in **ds_shutdown( )** is a call to the XOM API function **om_delete( )** to delete each private object the workspace contains.

The programs in Chapter 28 demonstrate how to initialize and shut down a workspace. The XDS functions **ds_initialize( )** and **ds_shutdown( )** are described in detail in Sections 27.1.1 and 27.1.3, respectively.

The closures of one or more packages are associated with a workspace. A package can be associated with any number of workspaces. An application program must obtain a workspace that supports an OM class before it is able to create any instances of that OM class. For example, some of these operations in an application may require involvement with GDS security, ACLs, or the DUA cache. Therefore, in addition to the basic packages provided by the Directory Service APIs, the workspace would have to support the GDSP package. The following code fragment demonstrates how an application initializes a workspace and negotiates the packages to be associated with that workspace:

```
/* Build up an array of object identifiers for the optional */
/* packages to be negotiated.                                */

DS_feature features[] = {
 { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
 { OM_STRING(OMP_O_DSX_GDS_PKG), OM_TRUE },
 { 0 }
};

CHECK_DS_CALL((OM_object) !(workspace = ds_initialize()));

CHECK_DS_CALL(ds_version(bdcp_package, workspace));
```

26.4 Storage Management

An object occupies storage. The storage occupied by a public object is allocated by the client, and is, therefore, directly accessible by the client and can be released by the client. The storage occupied by a private object is not accessible by the client and must be managed indirectly using XOM function calls.

Objects are accessed by an application program via object handles. Object handles are used as input parameters to interface functions by the client and returned as output parameters by the service. The object handle for a public object is simply a pointer to the data structure (an array of descriptors) containing the object OM attributes. The object handle for a private object is a pointer to a data structure that is in private implentation-specific format and therefore inaccessible directly by client.

The client creates a client-generated public object using normal programming language constructs; for example, static initialization. The client is responsible for managing any storage involved. The service creates service-generated public objects and allocates the necessary storage. As previously mentioned, the client must destroy service-generated public objects and release the storage by applying the XOM function **om_delete**( ) to it, as shown in the following code fragment:

```
 /*  We can now safely release all the private objects
 *  and the public objects we no longer need
 */
CHECK_OM_CALL(om_delete(session));
CHECK_OM_CALL(om_delete(result));
CHECK_OM_CALL(om_delete(entry));
CHECK_OM_CALL(om_delete(attributes));
CHECK_DS_CALL(ds_shutdown(workspace));
```

The service also creates private objects for which it allocates storage that must be managed by the application.

One of the input parameters to the **ds_read**( ) function call is **name**. The **name** parameter is a public object created by the application from a series of nested data structures (RDNs and AVAs) to represent the distinguished name containing **Peter Piper**. When the application no longer needs the public object, it issues the XDS function call **ds_shutdown**( ) to release the memory resources associated with the public object. The **ds_read**( ) call returns the pointer to a private object, **result**, deposited in the workspace by the service.

The program goes on to use the XOM function **om_get**( ) with the input parameter **result** as a pointer to extract attribute values from the returned private object. The **om_get**( ) call returns the pointer **entry** as a service-generated public object to the program so that the attribute values specified in the call can be accessed by it. Once the value is extracted, the application program can continue processing; for example, printing a message to a user with some extracted value like a phone number or postal address. The service-generated public object becomes the responsibility of the application program. The program goes on to release the resources allocated by the service by issuing a series of calls to **om_delete**( ), as shown in the following code fragment from **example.h**:

```
/*
 * extract the telephone number(s) of "name" from the result
 *
 * There are 4 stages:
 * (1) get the Entry-Information from the Read-Result.
 * (2) get the Attributes from the Entry-Information.
 * (3) get the list of phone numbers.
 * (4) scan the list and print each number.
 */

CHECK_OM_CALL(   om_get(result,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             entry_list, OM_FALSE, 0, 0, &entry,
             &total_num));

CHECK_OM_CALL(   om_get(entry->value.object.object,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             attributes_list, OM_FALSE, 0, 0, &attributes,
             &total_num));

CHECK_OM_CALL(   om_get(attributes->value.object.object,
             OM_EXCLUDE_ALL_BUT_THESE_TYPES
           + OM_EXCLUDE_SUBOBJECTS,
             telephone_list, OM_FALSE, 0, 0, &telephones,
             &total_num));

/*  We can now safely release all the private objects
 *  and the public objects we no longer need
 */
CHECK_OM_CALL(om_delete(session));
CHECK_OM_CALL(om_delete(result));
CHECK_OM_CALL(om_delete(entry));
CHECK_OM_CALL(om_delete(attributes));
```

If the client possesses a valid handle (or pointer) for an object, it has access to a private object. If the client does not possess an object handle or the handle is not a valid one, a private object is inaccessible to the client and an error is returned to the calling function. In the preceding code fragment, the handles for the objects stored in **entry**, **attributes**, and **telephones** are the pointers **&entry**, **&attributes**, and **&telephones**, respectively.

26.5 OM Syntaxes for Attribute Values

An OM attribute is made up of an integer uniquely defined within a package that indicates the OM attribute's type, an integer giving that value's syntax, and an information item called a value. The syntaxes defined by the XOM API standard are closely aligned with ASN.1 types and type constructors.

Some syntaxes are described in the standard in terms of syntax templates.

A syntax template defines a group of related syntaxes. The syntax templates that are defined are as follows:

- Enum(*)
- Object(*)
- String(*)

26.5.1 Enumerated Types

An OM attribute with syntax template Enum(*) is an enumerated type (**OM_S_ENUMERATION**) and has a set of values associated with that OM attribute. For example, one of the OM attributes of the OM class **DS_C_ENTRY_INFO_SELECTION** is **DS_INFO_TYPE**. **DS_INFO_TYPE** is listed in the OM attribute table for **DS_C_ENTRY_INFO_SELECTION** in Chapter 30 as having a value syntax of Enum(**DS_INFORMATION_TYPE**), as shown in Table 26-4. **DS_INFO_TYPE** takes one of the following values:

- **DS_TYPES_ONLY**
- **DS_TYPES_AND_VALUES**

Table 26–4. Description of an OM Attribute Using Syntax Enum(*)

OM Attributes of a DS_C_ENTRY_INFO_SELECTION				
OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_ALL_ ATTRIBUTES	**OM_S_ BOOLEAN**	—	1	**OM_TRUE**
DS_ATTRIBUTES_ SELECTED	String(**OM_S_ OBJECT_IDENTIFIER_ STRING)**	—	0 or more	—
DS_INFO_TYPE	Enum(**DS_ INFORMATION_ TYPE)**	—	1	**DS_TYPES_ AND VALUES**

The C language representation of the syntax of the OM attribute type **DS_INFO_TYPE** is **OM_S_ENUMERATION** as defined in the **xom.h** header file. The value of the OM attribute is either **DS_TYPES_ONLY** or **DS_TYPES_AND_VALUES**, as shown in the following code fragment from **example.h**:

```
/*
 * Public Object ("Descriptor List") for
 * Entry-Information-Selection
 * parameter to ds_read().
 */
 OM_descriptor selection[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
 { DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, { OM_FALSE, NULL } },
 OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR),
 { DS_INFO_TYPE,OM_S_ENUMERATION,
 { DS_TYPES_AND_VALUES,NULL } },
 OM_NULL_DESCRIPTOR
 };
```

26.5.2 Object Types

An OM attribute with syntax template Object(*) has **OM_S_OBJECT** as syntax and a subobject as a value. For example, one of the OM attributes of the OM class **DS_C_DS_DN** is **DS_RDNS**. **DS_RDNS** is listed in the OM attribute table for **DS_DS_DN** as having a value syntax of Object(**DS_C_DS_RDN**), as shown in Table 26-5.

Table 26–5. Description of an OM Attribute Using Syntax Object(*)

OM Attributes of a DS_C_DS_DN				
OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Value Initially**
DS_RDNS	Object(**DS_C_DS_RDN)**	—	0 or more	—

The C language representation of the syntax of the OM attribute type **DS_RDNS** is **OM_S_OBJECT**, as shown in following code fragment from **example.h**:

```
OM_descriptor        name[] = {
 OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
 { DS_RDNS, OM_S_OBJECT, { 0, rdn1 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn2 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn3 } },
 { DS_RDNS, OM_S_OBJECT, { 0, rdn4 } },
 OM_NULL_DESCRIPTOR
 };
```

26.5.3 Strings

An OM attribute with syntax template String(*) specifies the string syntax of its value. A string is categorized as either a *bit string*, an *octet string*, or a *character string*. The bits of a bit string, the octets of an octet string, or the octets of a character string constitute the *elements* of the string. (Refer to Chapter 35 for a list of the syntaxes that form the string group.)

The value length of a string is the number of elements in the string. Any constraints on the value length of a string are specified in the appropriate OM class definitions.

The elements of the string are numbered. The position of the first element is 0 (zero). The positions of successive elements are successive postive integers.

For example, one of the OM attributes of the OM class **DS_C_ENTRY_INFO_SELECTION** is **DS_ATTRIBUTES_SELECTED**. **DS_ATTRIBUTES_SELECTED** is listed in the OM attribute table for **DS_C_ENTRY_INFO_SELECTION** as having a value syntax of String(**OM_S_OBJECT_IDENTIFIER_STRING**), as shown in Table 26-4.

26.5.4 Other Syntaxes

The other syntaxes are defined as follows:

- **OM_S_BOOLEAN**

  A value of this syntax is a Boolean; that is, the value can be **OM_TRUE** or **OM_FALSE**.

- **OM_S_INTEGER**

  A value of this syntax is a positive or negative integer.

- **OM_S_NULL**

  The one value of this syntax is a valueless placeholder.

26.6 Service Interface Data Types

The local variables within an application program that contain the parameters and results of XDS and XOM API function calls are declared using a standard set of data types. These data types are defined by **typedef** statements in the **xom.h** header files. Some of the more commonly used data types are described in the following subsections. A complete description of service interface data types is provided in Chapter 36 and in the *OSF DCE Application Development Reference*.

26.6.1 The OM_descriptor Data Type

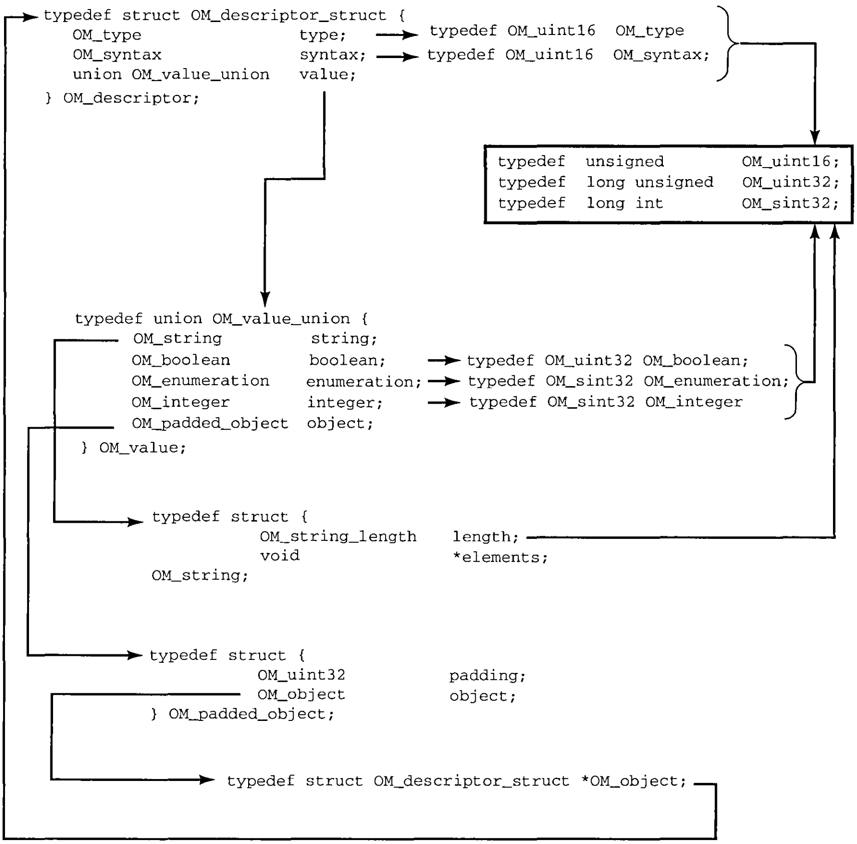
The **OM_descriptor** data type is used to describe an OM attribute type and value. A data value of this type is a descriptor, which embodies an OM attribute value. An array of descriptors can represent all the values of an object.

OM_descriptor is defined in the **xom.h** header file as follows:

```
/* Descriptor */

typedef struct OM_descriptor_struct {
        OM_type                 type;
        OM_syntax               syntax;
        union OM_value_union    value;
} OM_descriptor;
```

OM_descriptor is made up of a series of nested data structures, as shown in Figure 26-12.

Figure 26–12. Data Type OM_descriptor_struct



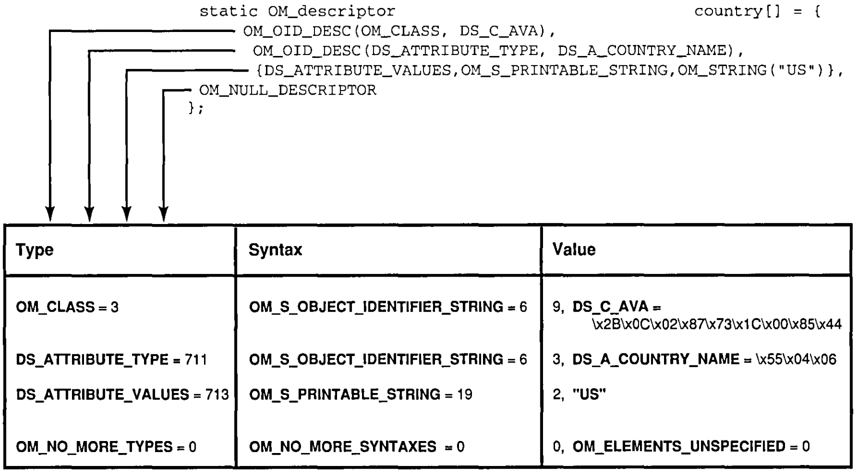
```
typedef struct OM_descriptor_struct {
   OM_type              type;        typedef OM_uint16  OM_type
   OM_syntax            syntax;      typedef OM_uint16  OM_syntax;
   union OM_value_union value;
} OM_descriptor;

                                                  typedef  unsigned       OM_uint16;
                                                  typedef  long unsigned  OM_uint32;
                                                  typedef  long int       OM_sint32;

typedef union OM_value_union {
      OM_string        string;
      OM_boolean       boolean;        typedef OM_uint32 OM_boolean;
      OM_enumeration   enumeration;    typedef OM_sint32 OM_enumeration;
      OM_integer       integer;        typedef OM_sint32 OM_integer
      OM_padded_object object;
} OM_value;

        typedef struct {
                  OM_string_length     length;
                  void                 *elements;
        OM_string;

        typedef struct {
                  OM_uint32            padding;
                  OM_object            object;
        } OM_padded_object;

             typedef struct OM_descriptor_struct *OM_object;
```

Figure 26-12 shows that **type** and **syntax** are integer constants for an OM attribute type and syntax, as shown in the following code fragment from **example.c**:

```
static OM_descriptor        country[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
 { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING,OM_STRING("US") },
 OM_NULL_DESCRIPTOR
 };
```

The code fragment initializes four descriptors, as shown in Figure 26-13. The type and syntax evaluate to integers for all four descriptors.

Figure 26–13. Initializing Descriptors



```
static OM_descriptor                              country[] = {
     OM_OID_DESC(OM_CLASS, DS_C_AVA),
      OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
      {DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING,OM_STRING("US")},
  OM_NULL_DESCRIPTOR
 };
```

Type	Syntax	Value
OM_CLASS = 3	OM_S_OBJECT_IDENTIFIER_STRING = 6	9, DS_C_AVA = \x2B\x0C\x02\x87\x73\x1C\x00\x85\x44
DS_ATTRIBUTE_TYPE = 711	OM_S_OBJECT_IDENTIFIER_STRING = 6	3, DS_A_COUNTRY_NAME = \x55\x04\x06
DS_ATTRIBUTE_VALUES = 713	OM_S_PRINTABLE_STRING = 19	2, "US"
OM_NO_MORE_TYPES = 0	OM_NO_MORE_SYNTAXES = 0	0, OM_ELEMENTS_UNSPECIFIED = 0

The **value** component is a little more complex. Figure 26-12 shows that **value** is a union of **OM_value_union**. **OM_value_union** has five members: **string**, **boolean**, **enumeration**, **integer**, and **object**. The members **boolean**, **enumeration**, and **integer** have integer values. The **string** member contains a string of type **OM_string**, which is a structure composed of a length and a pointer to a string of characters. The **object** member is a structure of type **OM_padded_object** that points to another object nested below it. Many OM attributes have other objects as values. These subobjects, in turn, may have other subobjects and so on.

For example, as shown in Figure 26-14, the OM class **DS_READ_RESULT** has one OM attribute: **DS_ENTRY**. The syntax of **DS_ENTRY** is **OM_S_OBJECT** with a value of **DS_C_ENTRY_INFO**, indicating that it points to the subobject **DS_C_ENTRY_INFO**. **DS_C_ENTRY_INFO** has the OM attribute **DS_OBJECT_NAME** with the syntax **OM_S_OBJECT**, indicating that it points to the subobject **DS_C_NAME**.

Figure 26–14. An Object and a Subordinate Object

OM Class	Attribute	Syntax and Value
DS_READ_RESULT	DS_ENTRY	Object (DS_C_ENTRY_INFO)
DS_C_ENTRY_INFO	DS_FROM_ENTRY DS_OBJECT_NAME	OM_S_BOOLEAN Object (DS_C_NAME)

The following code fragment from **example.h** shows how the data types are used to declare the variables that contain the output parameters from the XDS API function calls.

```
int main(void)
{
  DS_status          error;       /* return value from DS functions    */
  OM_return_code     return_code;/* return value from OM functions    */
  OM_workspace       workspace;   /* workspace for objects             */
  OM_private_object  session;     /* session for directory operations */
  OM_private_object  result;      /* result of read operation          */
  OM_sint            invoke_id;   /* Invoke-ID of the read operation   */
```

The code fragment shows:

- The **ds_initialize**( ) call returns a variable of type **OM_workspace** that contains a handle or pointer to a workspace.
- The **ds_bind**( ) call returns a pointer to a variable of type **OM_private_object**. The private object contains the session information required by all subsequent XDS API calls, except **ds_shutdown**( ).
- The **ds_read**( ) call returns a pointer to the result of a directory read request in a variable of type **OM_private_object**.

- The error handing macros **CHECK_DS_CALL** and **CHECK_OM_CALL**, defined in the **example.h** header file, use the data types **DS_status** and **OM_return_code**, respectively, as return values from XDS and XOM API function calls.

26.6.2 Data Types for XOM API Calls

The following code fragment from **example.h** shows how the data types are used to declare the variables that contain the input and output parameters for the XOM API function calls.

```
/*
 * variables to extract the telephone number(s)
 */
OM_type             entry_list[]      = { DS_ENTRY, 0 };
OM_type             attributes_list[] = { DS_ATTRIBUTES, 0 };
OM_type             telephone_list[]  = { DS_ATTRIBUTE_VALUES, 0 };
OM_public_object    entry;
OM_public_object    attributes;
OM_public_object    telephones;
OM_descriptor      *telephone;  /* current phone number  */
OM_value_position   total_num;  /* number of Attribute Descriptors */
```

The code fragment shows:

- The series of **om_get**( ) calls requires a list of OM attribute types that identifies the types of OM attributes to be included in the operation. The variables **entry_list**, **attribute_list**, and **telephone_list** are declared as type **OM_type**.
- The series of **om_get**( ) calls return pointers to variables of type **OM_public_object**. The **om_get**( ) call generates public objects that are accessible to the application program.
- Where the variable **total_num** is type **OM_value_position** and is used to hold the number of OM descriptors returned by **om_get**( ).

Chapter 35 contains detailed descriptions of all the data types defined by XOM API.

26.7 OM Function Calls

XOM API supports general-purpose OM functions defined by the X/Open standards body that allow an application program to manipulate objects in a workspace. Section 26.7.1 lists the OM function calls and gives a brief description of each. Section 26.7.2 illustrates the use of OM function calls using the **om_get( )** call as an example.

26.7.1 Summary of OM Function Calls

The following list of XOM API function calls contains a brief description of each function. Refer to the *OSF DCE Application Development Reference* for a detailed description of the input and output parameters, return codes, and usage of each function.

- **om_copy( )**

  Creates an independent copy of an existing private object and all of its subobjects in a specified workspace

- **om_copy_value( )**

  Replaces an existing OM attribute value or inserts a new value into a target private object with a copy of an existing OM attribute value found in a source private object

- **om_create( )**

  Creates a private object that is an instance of the specified OM class

- **om_delete( )**

  Deletes a private or service-generated public object

- **om_get( )**

  Creates a new public object that is an exact but independent copy of an existing private object; certain exclusions and/or syntax conversion may be requested for the copy

- **om_instance( )**

  Tests to determine if an object is an instance of a specified OM class (includes the case when the object is a subclass of that OM class)

- **om_put()**

  Places or replaces copies of the attribute values of the source private or public object into the target private object

- **om_read()**

  Reads a segment of a string attribute from a private object

- **om_remove()**

  Removes and discards values of an attribute of a private object

- **om_write()**

  Writes a segment of a string attribute to a private object

- **om_encode()**

  Not supported by DCE XOM API

- **om_decode()**

  Not supported by DCE XOM API

26.7.2 Using the OM Function Calls

Most application programs require the use of a series of **om_get()** function calls to create service-generated public objects from which the program can extract requested information. For this reason, this section uses the operation of **om_get()** as an example to describe how XOM API functions operate in general.

The following code fragment from **example.h** shows how a series of **om_get()** function calls extract a list of telephone numbers from a workspace. The **ds_read()** function call deposits the private object stored in **result** in the workspace and provides access to it by the pointer **&result**.

```
/*
 * extract the telephone number(s) of "name" from the result
 *
 * There are 4 stages:
 * (1) get the Entry-Information from the Read-Result.
 * (2) get the Attributes from the Entry-Information.
 * (3) get the list of phone numbers.
```

```
   * (4) scan the list and print each number.
   */

CHECK_OM_CALL(    om_get(result,
                  OM_EXCLUDE_ALL_BUT_THESE_TYPES
                + OM_EXCLUDE_SUBOBJECTS,
                  entry_list, OM_FALSE, 0, 0, &entry,
                  &total_num));

CHECK_OM_CALL(    om_get(entry->value.object.object,
                  OM_EXCLUDE_ALL_BUT_THESE_TYPES
                + OM_EXCLUDE_SUBOBJECTS,
                  attributes_list, OM_FALSE, 0, 0, &attributes,
                  &total_num));

CHECK_OM_CALL(    om_get(attributes->value.object.object,
                  OM_EXCLUDE_ALL_BUT_THESE_TYPES
                + OM_EXCLUDE_SUBOBJECTS,
                  telephone_list, OM_FALSE, 0, 0, &telephones,
                  &total_num));

/*  We can now safely release all the private objects
 *  and the public objects we no longer need
 */
CHECK_OM_CALL(om_delete(session));
CHECK_OM_CALL(om_delete(result));
CHECK_OM_CALL(om_delete(entry));
CHECK_OM_CALL(om_delete(attributes));
CHECK_DS_CALL(ds_shutdown(workspace));

for (telephone = telephones;
     telephone->type != DS_ATTRIBUTE_VALUES;
     telephone++)
    {
 if (telephone->type   != DS_ATTRIBUTE_VALUES
||  (telephone->syntax & OM_S_SYNTAX) !=
     OM_S_PRINTABLE_STRING)
      {
(void) fprintf(stderr, "malformed telephone number\n");
       exit(EXIT_FAILURE);
      }
(void) printf("Telephone number: %s\n",
        telephone->value.string.elements);
      }

CHECK_OM_CALL(om_delete(telephones));
```

The **om_get()** call makes a copy of all or a selected set of parts of a private object. The copy is a service-generated public object that is accessible to the application program. The application program extracts the list of telephone numbers from this copy.

26.7.2.1 Required Input Parameters

The **om_get()** function requires the following input parameters:

- A private object
- A set of exclusions
- A set of OM attributes to be included in the copy
- A flag to indicate whether local string processing is required
- The position of the first value to be copied (the base value)
- The position within each OM attribute that is one beyond the last attribute to be included in the copy (indicating the scope of the copy)

The **om_get()** call returns the following output parameters:

- The public object that is a copy of the private object
- The number of OM attribute descriptors returned in the public object

In the code fragment from **example.h**, the private object **result** is input to **om_get()**.

The next parameter, the *exclusions* parameter, reduces the copy to a prescribed portion of the original. The exclusions apply to the OM attributes of the object, but not to those of subobjects. The possibilities for determining the combinations of types, values, subobjects, and descriptors to be excluded depend on the creativity of the programmer. For a detailed description of all the exclusion possibilities, refer to the *OSF DCE Application Development Reference*. The values chosen for the **om_get()**

calls in **example.h** are simplified for clarity. These exclusion values are as follows:

- **OM_EXCLUDE_ALL_BUT_THESE_TYPES**
- **OM_EXCLUDE_SUBOBJECTS**

Each value indicates an exclusion, as defined by **om_get()**, and is chosen from the set of exclusions; alternatively, the single value **OM_NO_EXCLUSIONS** may be chosen, which selects the entire object. Each value, except **OM_NO_EXCLUSIONS**, is represented by a distinct bit, the presence of the value being represented as 1, and its absence as 0 (zero). Multiple exlusions are requested by adding or ORing the values that indicate the individual exclusions.

OM_EXCLUDE_ALL_THESE_TYPES indicates that the OM attributes included are only the ones defined in the list of included types supplied in the next parameter, **entry_list**. **OM_EXCLUDE_SUBOBJECTS** indicates that for each value whose syntax is **OM_S_OBJECT**, a descriptor containing an object handle for the original private subobject is returned, rather than a public copy of it. This handle makes that subobject accessible for use in subsequent function calls. This exclusion provides a means to examine an object one level at a time. The object the handle points to is used in the next **om_get()** call to get the next level.

The **entry_list** parameter is declared in **example.h** as data type **OM_type** and initialized as a two-cell array with values **DS_ENTRY** and a **NULL** terminator. **DS_ENTRY** specifies the single OM attribute type included for that **om_get()** call. This call only limits processing to the one directory entry; only one entry was defined previously in the program, the distinguished name of **Peter Piper**. The 0 (zero) marks the end of the OM attribute list.

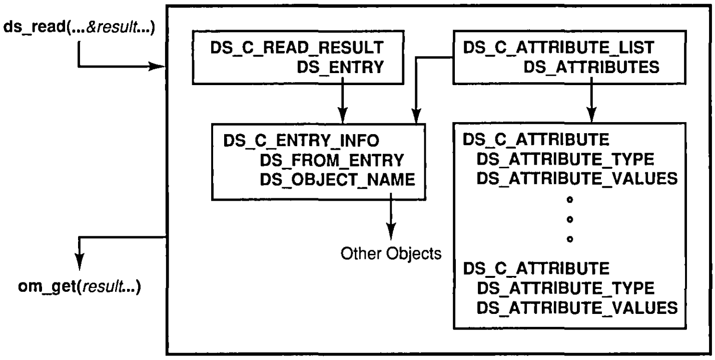
The next parameter, **OM_FALSE**, is just a place holder for a parameter that is not supported by XOM local strings. The next two parameters set the initial and limiting value to 0 (zero), meaning that no specific values are to be excluded.

The final two parameters are output parameters: **entry**, a pointer to a service-generated public object deposited by **om_get()** in the workspace, and **total_num**, an integer. Both **entry** and **total_num** are available for examination by the application program.

26.7.2.2 Extracting the Data from the Read Result

The **entry** parameter contains the result of processing by **om_get()** of the **read** parameter generated by the **ds_read()** operation. A successful call to **ds_read()** returns an instance of OM class **DS_C_READ_RESULT** in the private object **result**. **DS_C_READ_RESULT** contains the information extracted from the directory entry of the target object. Figure 26-15 shows the relationship of some of the superclasses, subclasses, and the OM attribute of **DS_C_READ_RESULT**. Consider Figure 26-15 as a partial map of the contents of **result**.

Figure 26–15. The Read Result



The **om_get()** function call creates a public object to make the information contained in **result** available to the application program. The **entry** parameter is defined as data type **OM_public_object**. As such, it is composed of several nested layers of subobjects that contain entry information, OM attributes, and OM attribute values, as shown in Figure 26-16. The series of **om_get()** calls removes these layers of objects to extract a list of telephone numbers.

Figure 26–16. Extracting Information Using om_get( )

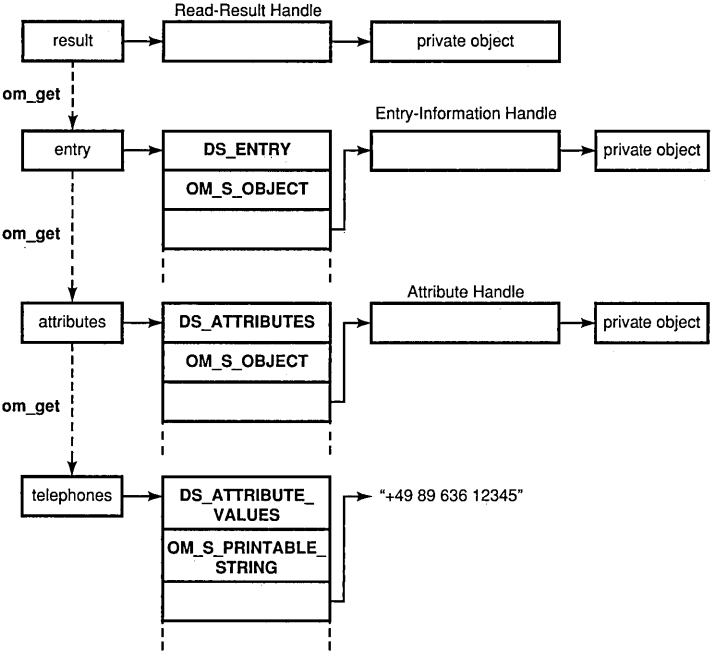


Figure 26-16 also shows that the process of exposing the subobjects continues while the syntax of the subobjects is **OM_S_OBJECT**. In effect, **example.h** is reversing the process of building up a series of public objects as input to **ds_read**( ), namely, the distinguished name of **Peter Piper** and the descriptor list for *entry_information_selection*. The following code fragment from **example.c** shows how the syntax of the variable **telephones** is tested for valid syntax, in this case, **OM_S_PRINTABLE_STRING**:

```
for (telephone = telephones;
     telephone->type != DS_ATTRIBUTE_VALUES;
     telephone++)
    {
 if (telephone->type != DS_ATTRIBUTE_VALUES ||
     (telephone->syntax & OM_S_SYNTAX) !=
     OM_S_PRINTABLE_STRING)
      {
```

```
(void) fprintf(stderr, "malformed telephone number\n");
        exit(EXIT_FAILURE);
      }

(void) printf("Telephone number: %s\n",
        telephone->value.string.elements);
      }
```

The preceding example determines whether **telephones** is in a format that can be used by the application program as string data that can be printed out, and that the syntax is correct for a list of telephone numbers. Note that the progam uses the constant **OM_S_SYNTAX** to mask off the top 5 bits. These bits are special bits that are used by XOM API. (Refer to Chapter 36 for more information on these special bits.)

26.7.2.3 Return Codes

XOM API function calls return a value of type **OM_return_code**, which indicates whether the function succeeded. If the function is successful, the value of **OM_return** is set to **OM_SUCCESS**. If the function fails, it returns one of the values listed in Chapter 36. The constants for **OM_return_code** are defined in the **xom.h** header file.

26.8 XOM API Header Files

The XOM API includes the header file **xom.h**. This header file is composed of declarations defining the C workspace interface. It supplies type definitions, symbolic constant definitions, and macro definitions.

26.8.1 XOM Type Definitions and Symbolic Constant Definitions

The **xom.h** header file includes **typedef** statements that define the data types of all OM objects used in the interface. It also provides definitions of symbolic constants used by the interface.

Refer to the *OSF DCE Application Development Reference* for a listing of the **xom.h** header file.

26.8.2 XOM API Macros

XOM API provides several macros that are useful in defining public objects in your application programs. These macros are defined in the **xom.h** header file:

- **OM_IMPORT**

  Makes object identifier symbolic constants available within a C source module

- **OM_EXPORT**

  Allocates memory and initializes object identifier symbolic constants within a C source module

- **OM_OID_DESC**

  Initializes the type, syntax, and value of an OM attribute that holds an object identifier

- **OM_NULL_DESCRIPTOR**

  Marks the end of a client-generated public object

- **OMP_LENGTH**

  Calculates the length of an object identifier

- **OM_STRING**

  Creates a data value of a string data type

26.8.2.1 The OM_EXPORT and OM_IMPORT Macros

Most application programs find it convenient to export all the names they use from the same C source module. **OM_EXPORT** allocates memory for the constants that represent an object OM class or an object identifier, as shown in the following code fragment from **example.c**:

```
/*  Define necessary Object Identifier constants
 */
OM_EXPORT(DS_A_COMMON_NAME)
OM_EXPORT(DS_A_COUNTRY_NAME)
OM_EXPORT(DS_A_ORG_NAME)
OM_EXPORT(DS_A_ORG_UNIT_NAME)
OM_EXPORT(DS_A_PHONE_NBR)
OM_EXPORT(DS_C_AVA)
OM_EXPORT(DS_C_DS_DN)
OM_EXPORT(DS_C_DS_RDN)
OM_EXPORT(DS_C_ENTRY_INFO_SELECTION)
```

In this code fragment, object identifier constants that represent OM classes that are defined in the **xds.h** and **xdsbdcp.h** header files are exported to the main program module. The object identifier constants are defined in **xds.h** with the **OMP_O** prefix followed by the variable name for the object identifier. The constant itself provides the hexadecimal value of the object identifier string.

The **OM_EXPORT** macro takes the OM class name as input and creates two new data structures: a character string and a structure of type **OM_string**. The structure of type **OM_string** contains a length and a pointer to a string that maybe used later in an application program by the **OM_OID_DESC** macro to initialize the value of an object identifier.

OM_IMPORT marks the identifiers as external for the compiler. It is used if **OM_EXPORT** is called in a different file from where its values are referenced. **OM_IMPORT** is not used in **example.c** because **OM_EXPORT** is called in the file where the object identifiers are referenced.

26.8.2.2 The OM_OID_DESC Macro

The **OM_OID_DESC** macro initializes the type, syntax, and value of an OM attribute that holds an object identifier; in other words, it initializes **OM_descriptor**. It takes as input an OM attribute type and the name of an object identifier. The object identifier should have already been exported to the program module, as shown in the previous section.

The output of the macro is an **OM_descriptor** composed of a type, syntax, and value. The type is the name of the OM class. The syntax is **OM_S_OBJECT_IDENTIFIER**. The value is a two-member structure with the length of the object identifier and a pointer to the actual object identifier string. It is defined as a pointer to **void** so that it can be used as a generic pointer; it can point to any data type.

OM_OID_DESC calls **OMP_LENGTH** to calculate the length of the object identifier string.

The following code fragment from **xom.h** shows the **OM_OID_DESC** and **OMP_LENGTH** macros:

```
/* Private macro to calculate length
 * of an object identifier
 */
#define OMP_LENGTH(oid_string) (sizeof(OMP_O_##oid_string)-1)

/* Macro to initialize the syntax and value
 * of an object identifier
 */
#define OM_OID_DESC(type, oid_name)
        { (type), OM_S_OBJECT_IDENTIFIER_STRING,
         { OMP_LENGTH(oid_name) , OMP_D_##oid_name } }
```

26.8.2.3 The OM_NULL_DESCRIPTOR Macro

The **OM_NULL_DESCRIPTOR** macro marks the end of a client-generated public object by setting the type, syntax, and value to **OM_NO_MORE_TYPES**, **OM_S_NO_MORE_SYNTAXES**, and a value of zero length and a **NULL** string, respectively.

26.8.2.4 The OM_STRING Macro

The **OM_STRING** macro creates a string data value. Data strings are of type **OM_string**, as shown from this code fragment from the **xom.h** header file:

```
/* String */

typedef struct {
        OM_string_length      length;
        void                  *elements;
} OM_string;

#define OM_STRING(string)        \
         { (OM_string_length)(sizeof(string)-1), string }
```

A string is specified in terms of its length or whether or not it terminates with a **NULL**. **OM_string_length** is the number of octets by which the string is represented, or it is the **OM_LENGTH_UNSPECIFIED** value if the string terminates with a **NULL**.

The bits of a bit string are represented as a sequence of octets. The first octet stores the number of unused bits in the last octet. The bits in the bit string, beginning with the first bit and proceeding to the trailing bit, are placed in bits 7 to 0 of the second octet. These are followed by bits 7 to 0 of the third octet, then by bits 7 to 0 of each octet in turn, followed by as many bits as are required of the final octet commencing with bit 7.

Chapter 27

XDS Programming

XDS API defines an application programming interface to directory services in the X/Open Common Applications Environment as defined in the *X/Open Portability Guide*. This interface is based on the 1988 CCITT X.500 Series of Recommendations and the ISO 9594 Standard. This joint standard is referred to from this point on simply as X.500.

This chapter describes the purpose and function of XDS API functions in a general way. Refer to the *OSF DCE Application Development Reference* for complete and detailed reference information on specific function calls.

The sections that follow describe the following types of XDS functions:

- XDS Interface Management Functions

  Interacting with the XDS interface

- Directory Connection Management Functions

  Initiating, managing, and terminating connections with the directory

- Directory Operation Functions

  Performing operations on a directory

Note: The **ds_abandon( )** and **ds_receive_result( )** functions are not supported because DCE XDS API does not support asynchronous operations. A **ds_abandon( )** call returns a **DS_C_ABANDON_FAILED** (**DS_E_TOO_LATE**) error. A **ds_receive_result( )** call returns with **DS_status** set to **DS_SUCCESS**, and the *completion_flag_return* parameter set to **DS_NO_OUTSTANDING_OPERATION**.

The following names:

- **acl.c** (**acl.h**)
- **example.c** (**example.h**)
- **teldir.c**

refer to the complete XDS example programs, which can be found in Chapter 28.

27.1 XDS Interface Management Functions

XDS API defines a set of functions that only interact with the XDS interface and have no counterpart in the directory standard definition:

- **ds_initialize( )**
- **ds_version( )**
- **ds_shutdown( )**

These interface functions perform operations that involve the initialization, management, and termination of sessions with the XDS interface service.

27.1.1 The ds_initialize( ) Function Call

Every application program must first call **ds_initialize( )** to establish a workspace where objects returned by the Directory Service are deposited. The **ds_initialize( )** function must be called before any other directory interface functions are called.

The **ds_initialize**( ) call returns a handle (or pointer) to a workspace. The application program performs operations on OM objects in this workspace. OM objects created in this workspace can be used as input parameters to the other directory interface functions. In addition, objects returned by the Directory Service are deposited in the workspace.

Within the following code fragment from **example.c**, a workspace is initialized (the declaration of the variable **workspace** and the call to **ds_initialize**( ) are found in different sections of the program):

```
int main(void)
{
  DS_status         error;       /* return value from DS functions    */
  OM_return_code    return_code;/* return value from OM functions    */
  OM_workspace      workspace;  /* workspace for objects             */
  OM_private_object session;    /* session for directory operations */
  OM_private_object result;     /* result of read operation          */
  OM_sint           invoke_id;  /* Invoke-ID of the read operation   */
  OM_value_position total_num;  /* Number of Attribute Descriptors   */

/*
 * Perform the Directory operations:
 * (1) Initialize the Directory Service and get an OM workspace.
 * (2) bind a default directory session.
 * (3) read the telephone number of "name".
 * (4) terminate the directory session.
 */

  CHECK_DS_CALL((OM_object)  !(workspace=ds_initialize()));
```

OM_workspace is a type definition in the **xom.h** header file defined as a pointer to **void**. A void pointer is a generic pointer that may point to any data type. The variable **workspace** is declared as data type **OM_workspace**. The return value is assigned to the variable **workspace** and the **CHECK_DS_CALL** macro determines if the call is successful. **CHECK_DS_CALL** is an error handling macro that is defined in **example.h**.

The **ds_initialize**( ) call returns a handle to a workspace in which OM objects can be created and manipulated. Only objects created in this workspace can be used as parameters to other directory interface functions. The **ds_initialize**( ) call returns **NULL** if it fails.

27.1.2 The ds_version( ) Function Call

The **ds_version**( ) call negotiates features of the directory interface. These features are collected into packages that define the scope of the service. Packages define such things as object identifiers for directory and OM classes and OM attributes, enumerated types, structures, and OM object constants.

XDS API defines the following packages in separate header files as part of the XDS API software product:

- Directory Service Package

  The Directory Service Package contains the OM classes and OM attributes used to interact with the Directory Service. This package is contained in the **xds.h** header file.

- Basic Directory Contents Package

  The Basic Directory Contents Package contains OM classes and OM attributes that represent values of selected attributes and selected objects defined in the X.500 standard. This package is contained in the **xdsbdcp.h** header file.

- Global Directory Service Package

  The Global Directory Service Package contains the OM classes and OM attributes that are required for GDS. This package is contained in the **xdsgds.h** header file.

- MHS Directory User Package

  The MHS (Message Handling Systems) Directory User Package contains the OM classes and OM attributes that are required for Electronic Mail API. This package is contained in the **xdsmdup.h** header file.

The application program, the client, uses **ds_version**( ) to negotiate the scope of the services the Directory Service will provide to the client. A **ds_version**( ) function call includes a list of features (or packages) that the client wants to include as part of the interface. The features are object identifiers that represent packages supported by DCE XDS API. The service returns a list of Boolean values to indicate whether or not the package was successfully negotiated.

These features are assigned to the workspace that an application program initialized (as described in Section 27.1.1). In addition, an application program must include the header files for the appropriate packages as part of the source code.

It is not necessary to negotiate the Directory Service Package. It it a mandatory requirement for XDS API and as such it is included by default. The other packages listed previously are optional and require negotiation using **ds_version( )**.

The following code fragment from **acl.h** shows how an application builds up an array of object identifiers for the optional packages to be negotiated: the Basic Directory Contents Package and the GDS Package.

```
static DS_feature features[] = {
    { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
    { OM_STRING(OMP_O_DSX_GDS_PKG), OM_TRUE },
    {0}
};
```

The **OM_STRING** macro is provided for creating a data value of data type **OM_string** for octets strings and characters. The array of object identifiers is stored in **features**, the input parameter to **ds_version( )**, as shown in the following code fragment from **acl.c**:

```
/* Negotiate the use of the BDCP and GDS packages.    */

if (ds_version(features,workspace) != DS_SUCCESS)
    printf("ds_version() error\n");
```

27.1.3 The ds_shutdown( ) Function Call

The **ds_shutdown( )** call deletes the workspace established by **ds_initialize( )** and enables the Directory Service to release resources. No other directory functions that reference that workspace may be called after this function.

The following code fragment from **acl.c** demonstrates how the application closes the directory workspace by performing a **ds_shutdown( )**.

```
/* Close the directory workspace.                         */

if (ds_shutdown (workspace) != DS_SUCCESS)
   printf ("ds_shutdown() error \n");
```

27.2 Directory Connection Management Functions

The following subsections describe the XDS functions that initiate, manage, and terminate connections with the Directory Service.

27.2.1 A Directory Session

A directory session identifies the DSA to which a directory operation is sent. It also defines the characteristics of a session, such as the distinguished name of the requestor.

An application program can request a session with specific OM attributes tailored for the program's requirements. The application passes an instance of OM class **DC_C_SESSION** with the appropriate OM attributes, or it uses the default parameters by passing the constant **DS_DEFAULT_SESSION** as a parameter to the **ds_bind**( ) function call.

27.2.2 The ds_bind( ) Function Call

The **ds_bind**( ) call establishes a session with the directory. The **ds_bind**( ) call corresponds to the **DirectoryBind** function in the Abstract Service defined in the X.500 standard.

When a **ds_bind**( ) call completes successfully, the directory returns a pointer to an OM private object of OM class **DC_C_SESSION**. This parameter is then passed as the first parameter to most interface function calls until a **ds_unbind**( ) is called to terminate the directory session.

XDS API supports multiple concurrent sessions so that an application can interact with the Directory Service using several identities, and interact directly and concurrently with different parts of the Directory Service.

The following code fragment from **example.c** shows how an application binds to the GDS server (without credentials) using the default session:

```
CHECK_DS_CALL(ds_bind(DS_DEFAULT_SESSION, workspace, &session));
```

If a user wants to specify a password as part of the user credentials and/or wants to specify the directory identifier, an instance of OM class **DSX_C_GDS_SESSION** from the GDS Package is required. **DSX_C_GDS_SESSION** identifies a particular link from an application to a DSA. Since **DSX_C_GDS_SESSION** is a subclass of the standard OM class for a session, **DS_C_SESSION**, it may be passed as a parameter to an XDS API function, such as **ds_bind()**, wherever a standard session is expected.

The following code fragment from **acl.c** shows how an application performs an authenticated bind to the GDS:

```
/*
 * Create a default session object.
 */
if ((rc = om_create(DSX_C_GDS_SESSION,OM_TRUE,workspace,&session))
        != OM_SUCCESS)
    printf("om_create() error %d\n", rc);


/*
 * Alter the default session object to include the following
 * credentials:
 *  requestor:  /C=de/O=sni/OU=ap/CN=norbert
 *  password:   "secret"
 */
if ((rc = om_put(session, OM_REPLACE_ALL, credentials, 0 ,0, 0))
        != OM_SUCCESS)
    printf("om_put() error %d\n", rc);


/*
 * Bind with credentials to the default GDS server.
 * The returned session object is stored in the private object
 * variable bound_session and is used for all further XDS
```

```
     * function calls.
     */
    if (ds_bind(session, workspace, &bound_session) != DS_SUCCESS)
        printf("ds_bind() error\n");
```

The program creates a default session object using the XOM API function **om_create**( ) and alters the default session object using **om_put**( ). The bind credentials are initialized in the following code fragment from the **example.h** header file included in the main program module:

```
/* The following descriptor list specifies
 * the bind credentials
 */

static OM_descriptor credentials[] = {
 {DS_REQUESTOR, OM_S_OBJECT, {0, dn_norbert} },
 {DSX_PASSWORD, OM_S_OCTET_STRING, OM_STRING("secret")},
 OM_NULL_DESCRIPTOR
};
```

The **credentials** parameter is provided as an input parameter to the **om_put**( ) function call to modify the existing session object in the Directory Service. A private object is returned to the workspace by **om_put**( ) that is used for all subsequent directory calls.

27.2.3 The ds_unbind( ) Function Call

The **ds_unbind**( ) call terminates a directory session and makes the session parameter unavailable for use with other interface functions. However, the unbound session can be modified by OM functions and used again as a parameter to **ds_bind**( ). When the session parameter is no longer needed, it should be deleted using OM functions such as **om_delete**( ).

The following code fragment from **example.c** shows how the application closes the connection to the GDS server using **ds_unbind**( ):

```
/* Close the connection to the GDS server.  */

if (ds_unbind(bound_session) != DS_SUCCESS)
    printf("ds_unbind() error\n");
```

The **ds_unbind( )** call corresponds to the **DirectoryUnbind** function in the Abstract Service defined in the X.500 standard.

27.2.4 Automatic Connection Management

The XDS implementation does not support automatic connection managment. A DSA connection is established when **ds_bind( )** is called and released when **ds_unbind( )** is called.

27.3 XDS Interface Class Definitions

The XDS Interface Class Definitions are described in detail in Chapter 30. The OM attribute types, syntax, and values and inheritance properties are described for each OM class.

A good way to begin to understand how the OM class hierarchy is structured and the relationship between OM classes and OM attributes to the service provided by the Directory Service Package is to look up one of the OM classes listed in Chapter 30.

27.3.1 Example: The DS_C_FILTER Class

For example, **DS_C_FILTER** inherits the OM attributes from its superclass **OM_C_OBJECT**, as do all OM classes. **OM_C_OBJECT**, as defined in Chapter 26, has one OM attribute, **OM_CLASS**, which has the value of an object identifier string that identifies the numeric representation of the object's OM class. **DS_C_FILTER**, on the other hand, has several OM attributes.

The purpose of **DS_C_FILTER** is to select or reject an object on the basis of information in its directory entry. It has the following OM attributes:

- **DS_FILTER_ITEMS**
- **DS_FILTERS**

- **DS_FILTER_TYPE**

Two of these OM attributes, **DS_FILTER_ITEMS** and **DS_FILTERS**, have values that are OM object classes themselves. The value of the OM attribute **DS_FILTER_ITEMS** is **DS_C_FILTER_ITEM**, which is an OM class. **DS_C_FILTER_ITEM** is a component of a filter and defines the nature of the filter. The value of the OM attribute **DS_FILTERS** is **DS_C_FILTER**, an OM class. Thus, **DS_FILTERS** defines a collection of filters. The OM attribute **DS_FILTER_TYPE** has a value that is an enumerated type, which takes one of the values **DS_AND**, **DS_OR**, or **DS_NOT**.

Refer to Figure 27-3 to see the relationship of **DS_C_FILTER** to its superclass **OM_C_OBJECT** and its attributes.

27.3.2 The DS_C_CONTEXT Parameter

The OM class **DS_C_CONTEXT** is the second parameter to every Directory Service request. **DS_C_CONTEXT** defines the characteristics of the Directory Service interaction that are specific to a particular Directory Service operation. These characteristics are divided into three categories of OM attributes: common parameters, service controls, and local controls.

Common parameters affect the processing of each Directory Service operation.

Service controls indicate how the Directory Service should handle requests. Included in this category are decisions about whether or not chaining is permitted, the priority of requests, the scope of referral (to DSAs within a country or within a DMD), and the maximum number of objects about which a function should return information.

Local controls include asynchronous support and automatic continuation; XDS does not currently support asynchronous operations.

27.4 Directory Class Definitions

The X.500 standards define a number of attribute types and classes. These definitions allow the creation and maintenance of directory entries for a number of common objects so that the representation of all such objects is the same throughout the directory. The Basic Directory Contents Package contains OM classes and OM attributes that model the X.500 attribute types and classes.

The X.500 object classes and attributes are defined in the following documents published by CCITT. These are the objects and the associated attributes that will be the targets of Directory Service operations in your application programs:

- *The Directory: Selected Attributes Types (Recommendation X.520)*
- *The Directory: Selected Object Classes (Recommendation X.521)*

Table 27-1 describes the OM classes, OM attributes, and their object identifiers that model the X.500 objects and attributes. (See Chapter 32 for more tables with the same type of information.)

Table 27–1. Representation of Values for Selected Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-valued	Matching Rules
DS_A_ALIASED_OBJECT_NAME	Object(**DS_C_NAME**)	None	No	E
DS_A_BUSINESS_CATEGORY	String(**OM_S_TELETEX_STRING**)	1-128	Yes	E, S
DS_A_COMMON_NAME	String(**OM_S_TELETEX_STRING**)	1-64	Yes	E, S
DS_A_COUNTRY_NAME	String(**OM_S_PRINTABLE_STRING**)[1]	2	No	E
DS_A_DESCRIPTION	String(**OM_S_TELETEX_STRING**)	1-1024	Yes	E, S

[1] As permitted by ISO 3166.

The tables in Chapter 32 contain similar categories of information as do similar tables for the attributes defined in the Directory Service Package. These information categories include the following:

- OM Value Syntax
- Value Length
- Multivalued
- Matching Rules

The OM Value Syntax column describes the structure of the values of an OM attribute. The Value Length column gives the range of lengths permitted for the string types. The Multivalued column indicates whether the attribute can have multiple values.

The CCITT standards define matching rules that are used for determining whether two values are equal, for ordering two values, or for identifying one value as a substring of another in Directory Service operations. These are indicated in the Matching Rules column.

The GDS administrator maintains the Directory Service and determines the structure of the DIT as defined by the GDS schema. The GDS standard (or default) schema is based on the recommendations in the CCITT documents mentioned previously.

Recall that the Structure Rule Table (SRT) of the GDS schema defines the structure of the DIT, the Object Class Table (OCT) defines class inheritance properties, and the Attribute Table (AT) defines the mandatory and optional attributes for each class. You will find it useful to familiarize yourself with the existing schema when developing an application program that will access the directory. This is because the public objects that your programs will create (using OM classes and OM attributes) are modeled after objects and attributes in the directory.

27.5 The Global Directory Service Package

The GDS software provides functional extensions to the standard in the following areas:

- Authentication
- Access Control
- DUA Cache

27.5.1 Authentication

An instance of OM class **DSX_C_GDS_SESSION** identifies a particular link from an application program to a DSA. This additional OM class is necessary if the user either wants to specify a password as part of user credentials, or wants to specify both a password and the directory identifier.

27.5.2 Access Control

In addition to authentication (by means of name and password), access protection is required for each object at the attribute level. A telephone number, for example, is an attribute that in general everybody is allowed to read. However, an attribute value such as a user-password normally has restricted access. In addition, even for attributes that everyone is allowed to read, it may only be acceptable for a small number of people to have authorization to change the values.

Because there can be a multitude of different attributes in the DIT, it is too expensive to define a protection mechanism for each individual attribute type. The directory attribute **DSX_A_ACL** is present for each entry in the DIT. Its syntax is Object(**DSX_C_GDS_ACL**), referencing the GDSP class **DSX_C_GDS_ACL**. These OM classes and attributes have been added to the Directory Service to specify the category of access to the individual attributes that are granted to users. There are three categories of access: Public, Standard, and Sensitive.

DSX_C_GDS_ACL has five OM attributes that define the read and modify access rights for each of these categories (read access is granted to all users; modify access implicitly grants read access):

- **DSX_MODIFY_PUBLIC**

  Specifies the user, or group of users, that can modify attributes classified as public attributes

- **DSX_READ_STANDARD**

  Specifies the user, or group of users, that can read attributes classified as standard attributes

- **DSX_MODIFY_STANDARD**

  Specifies the user, or group of users, that can modify attributes classified as standard attributes

- **DSX_READ_SENSITIVE**

  Specifies the user, or group of users, that can read attributes classified as sensitive attributes

- **DSX_MODIFY_SENSITIVE**

  Specifies the user, or group of users, that can modify attributes classified as sensitive attributes

The ACL of the default schema has no access rights when GDS is configured. Every user, including the anonymous user, has read and modify access to all attributes in the schema.

A master entry can be created only by the user who has write access to the naming attribute of the parent node. Thus, the user can create all attributes of the entry. Using the ACL class, the user can establish which objects can be accessed. If the user does not enter an ACL attribute when creating an entry, GDS automatically uses the ACL attribute of the parent node for the new entry.

A master entry can only be deleted by users who have write access to the naming attribute of the entry to be deleted.

A shadow entry created by means of shadow handling (refer to the *OSF DCE Administration Guide*) has the same ACL attribute as the corresponding master entry. This entry can therefore only be modified and deleted by the user who can also modify and delete the master entry.

27.5.3 DUA Cache

To further optimize access times, frequently requested information is automatically loaded to a section of memory in the client computer, the DUA cache, and may be overwritten again if it is not used within a certain interval of time. The cache may be periodically updated. The GDS administration program specifies the period. It can also specify that certain data is never written to the cache, or that certain data that is transferred must under no circumstances be deleted, unless it is deleted by the user. Because the DUA cache is not subject to any access control, GDS ensures that only the information that everybody is allowed to read is stored.

The GDS Package includes the OM class **DSX_C_GDS_CONTEXT** to support additional service controls for caching. **DSX_C_GDS_CONTEXT** is a subclass of **DS_C_CONTEXT**. As such, it inherits all the standard X.500 attributes associated with **DS_C_CONTEXT**, in addition to its own OM attributes related to caching. Refer to Chapter 25 for more information on how to manage the DUA cache using XDS.

27.6 Directory Operation Functions

The X.500 standard defines the operations provided by the directory in a document called the *Abstract Service Definition*. DCE implements this standard with XDS API functions calls. The XDS API functions allow an application program to interact with the Directory Service. The standard divides these interactions into three general categories: read, search, and modify.

The XDS API functions correspond to the Abstract Service functions defined in the X.500 standard, as shown in Table 27-2.

Table 27–2. Mapping of XDS API Functions to the Abstract Services

XDS Function Call	Abstract Service Equivalent
ds_read( )	**Read**
ds_compare( )	**Compare**
ds_list( )	**List**
ds_search( )	**Search**
ds_add_entry( )	**AddEntry**
ds_remove_entry( )	**RemoveEntry**
ds_modify_entry( )	**ModifyEntry**
ds_modify_rdn( )	**ModifyRDN**

27.7 Directory Read Operations

Read functions retrieve information from specific named entries in the directory where names are mapped to attributes. This is analogous to looking up some information about a name in the ‘‘White Pages’’ phone directory.

XDS API implements the following read functions:

- **ds_read( )**

  The requester supplies a distinguished name and one or more attribute types. The value(s) of requested attributes or just the attribute type(s) is returned by the DSA.

- **ds_compare( )**

  The requester gives a distinguished name and an Attribute Value Assertion (AVA). If the AVA is TRUE for the named entry, a value of TRUE is returned by the DSA.

For example, a typical read operation could request the telephone number of a particular employee. A read request would submit the distinguished name of the employee with an indication to return its telephone number: **/C=us/O=sni/OU=sales/CN=John Smith**.

27.7.1 Reading an Entry from the Directory

The following sections describe a typical read operation using the **ds_read( )** function call. They include a description of tasks directly related to the read operation. They do not include service-related tasks such as intializing the interface, allocating an OM workspace, and binding to the directory. These tasks are described in Section 27.1. The following sections also do not describe the process of extracting information from the workspace using XOM functions. Refer to Chapter 26 for a description of how to use XOM functions to access the workspace.

A typical read operation involves the following steps:

1. Define the necessary object identifier constants for the OM classes and OM attributes that will define public objects for input to **ds_read( )** using the **OM_EXPORT** macro.

2. Declare the variables that will contain the output from the XDS functions to be used in the application.

3. Build public objects (descriptor lists) for the *name* parameter to **ds_read( )**.

4. Create a descriptor list for the *selection* parameter to **ds_read()** that selects the type and scope of information in your request.
5. Perform the read operation.

These steps are demonstrated in the following code fragments from **example.c** (refer to Chapter 28 for a complete program listing). The program reads the telephone numbers of a given target entry.

27.7.2 Step 1: Export Object Identifiers for Required Directory Classes and Attributes

Most application programs find it convenient to export all the names they use from the same C source module. In the following code fragment from **example.c** the **OM_EXPORT** macro allocates memory for the constants that represent the OM object classes and directory attributes required for the read operation:

```
/*  Define necessary Object Identifier constants
 */
OM_EXPORT(DS_A_COMMON_NAME)
OM_EXPORT(DS_A_COUNTRY_NAME)
OM_EXPORT(DS_A_ORG_NAME)
OM_EXPORT(DS_A_ORG_UNIT_NAME)
OM_EXPORT(DS_A_PHONE_NBR)
OM_EXPORT(DS_C_AVA)
OM_EXPORT(DS_C_DS_DN)
OM_EXPORT(DS_C_DS_RDN)
OM_EXPORT(DS_C_ENTRY_INFO_SELECTION)
```

The **OM_EXPORT** macro performs the following steps:

1. It defines a character array called **OMP_D_** concatenated with the *class_name* input parameter.
2. It initializes this array to the value of a character string called **OMP_O_** concatenated with the *class_name* input parameter. This value has already been defined in a header file.
3. It defines an **OM_string** data structure as the *class_name* input parameter.

4. It initializes the **OM_string** data structure's first component to the length of the array initialized previously in Step 2 and initializes the second component to a pointer to the value of the array initialized in Step 2.

27.7.3 Step 2: Declare Local Variables

The local variables **session**, **result**, and **invoke_id** are defined in the following code fragment from **example.c**:

```
int main(void)
{
  DS_status          error;      /* return value from DS functions  */
  OM_return_code     return_code;/* return value from OM functions  */
  OM_workspace       workspace;  /* workspace for objects           */
  OM_private_object  session;    /* session for directory operations*/
  OM_private_object  result;     /* result of read operation        */
  OM_sint            invoke_id;  /* Invoke-ID of the read operation */
  OM_value_position  total_num;  /* Number of Attribute Descriptors */
```

These data types are defined in a **typedef** statement in the **xom.h** header file. The **session** and **result** variables are defined as data type **OM_private_object** because they are returned by **ds_bind**( ) and **ds_read**( ), respectively, to the workspace as private objects. Since asynchronous operations are not supported, the *invoke_id* functionality is redundant. The *invoke_id* parameter must be supplied to the XDS functions as described in the *OSF DCE Application Development Reference*, but its return value should be ignored.

Values in **error** and **return_code** are returned by XOM and XDS functions to indicate whether a call was successful. The **workspace** variable is defined as **OM_workspace** and is used when establishing an OM workspace. The **total_num** variable is defined as **OM_value_position** to indicate the number of attribute descriptors returned in the public object by **om_get**( ) based on the inclusion and exclusion parameters specified.

27.7.4 Step 3: Build Public Objects

A **ds_read**( ) function call may take a public object as an input parameter. A public object is generated by an application program and contains the information required to access a target directory object. This information includes the AVAs and RDNs that make up a distinguished name of an entry in the directory.

A public object is created using OM classes and OM attributes. These OM classes and OM attributes model the target object entry in the directory and provide other information required by the Directory Service to access the directory. In this case, the target object entry in the directory is the entry for **Peter Piper**.

Section 26.1.4 desribes how to create the required public objects for the **ds_read**( ) function call using macros and data structures defined in the XDS and XOM API header files.

The purpose of building the public objects for AVAs and RDNs is to provide the public objects that represent a distinguished name. The distinguished name public object is stored in the array of descriptors called *name* and provided as an input parameter to the **ds_read**( ) function call.

27.7.5 Step 4: Create an Entry-Information-Selection Parameter

The distinguished name for **Peter Piper** is an entry in the directory that the application is designed to access. The *selection* parameter of the **ds_read**( ) function call tailors its results to obtain just part of the required entry. Information on all attributes, no attributes, or a specific group of attributes can be chosen. Attribute types are always returned, but the attribute values need not be.

The value of the parameter is a public object (descriptor list) that is an instance of OM class **DS_C_ENTRY_INFO_SELECTION**, as shown in the following code fragment from **example.c**:

```
/*
 * Public Object ("Descriptor List") for
 * Entry-Information-Selection
 * parameter to ds_read().
```

```
*/
OM_descriptor selection[] = {
OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
{ DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, { OM_FALSE, NULL } },
OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR),
{ DS_INFO_TYPE,OM_S_ENUMERATION,
{ DS_TYPES_AND_VALUES,NULL } },
OM_NULL_DESCRIPTOR
};
```

DS_C_ENTRY_INFO_SELECTION is a subclass of **OM_C_OBJECT** (this information is supplied in the description of this class in Chapter 30). As such, **DS_C_ENTRY_INFO_SELECTION** inherits the OM attributes of **OM_C_OBJECT**. The only OM attribute of **OM_C_OBJECT** is **OM_CLASS**. **OM_CLASS** identifies an object's class, which in this case is **DS_C_ENTRY_INFO_SELECTION**. **DS_C_ENTRY_INFO_SELECTION** identifies information to be extracted from a directory entry and has the following OM attributes:

- **OM_C_CLASS** (inherited from **OM_C_OBJECT**)
- **DS_ALL_ATTRIBUTES**
- **DS_ATTRIBUTES_SELECTED**
- **DS_INFO_TYPE**

As part of a **ds_read()** or **ds_search()** function call, **DS_ALL_ATTRIBUTES** specifies to the Directory Service which attributes of a directory entry are relevant to the application program. It can take the values **OM_TRUE** or **OM_FALSE**. These values are defined to be of syntax **OM_S_BOOLEAN**. The value **OM_TRUE** indicates that information is requested on all attributes in the directory entry. The value **OM_FALSE**, used in the preceding sample code fragment, indicates that information is only requested on those attributes that are listed in the OM attribute **DS_ATTRIBUTES_SELECTED**.

DS_ATTRIBUTES_SELECTED lists the types of attributes in the entry from which information is to be extracted. The syntax of the value is specified as **OM_S_OBJECT_IDENTIFIER_STRING**.

OM_S_OBJECT_IDENTIFIER_STRING contains an octet string of BER-encoded integers, which are decimal representations of object identifiers of the types of attributes in the attribute list. In the preceding code fragment, the string value is the attribute name **DS_A_PHONE_NBR**

because the purpose of the read call is to read a list of telephone numbers from the directory.

DS_INFO_TYPE identifies what information is to be extracted from each attribute identified. The syntax of the value is specified as Enum(**DS_Information_Type**). **DS_INFO_TYPE** is an enumerated type that has two possible values: **DS_TYPES_ONLY** and **DS_TYPES_AND_VALUES**. **DS_TYPES_ONLY** indicates that only the attribute types of the selected attributes in the entry are returned by the Directory Service operation. **DS_TYPES_AND_VALUES** indicates that both the attribute types and the attribute values of the selected attributes in the entry are returned. The code fragment from **example.c** shown previously defines the value of **DS_INFO_TYPE** as **DS_TYPES_AND_VALUES** because the program wants to get the actual telephone numbers.

27.7.6 Step 5: Perform the Read Operation

The following code fragment from **example.c** shows the **ds_read( )** function call and the XDS calls that precede it:

```
/*
 * Perform the Directory operations:
 * (1) Initialize the Directory Service
 *     and get an OM workspace.
 * (2) bind a default directory session.
 * (3) read the telephone number of "name".
 * (4) terminate the directory session.
 */

CHECK_DS_CALL((OM_object) !(workspace = ds_initialize()));

CHECK_DS_CALL(ds_version(bdcp_package, workspace));

CHECK_DS_CALL(ds_bind(DS_DEFAULT_SESSION, workspace,
                 &session));

CHECK_DS_CALL(ds_read (session, DS_DEFAULT_CONTEXT,
                   name, selection, &result, &invoke_id));
```

CHECK_DS_CALL is an error-checking macro defined in the **example.h** header file that is included by **example.c**. The **ds_read**( ) call returns a return code of type **DS_status** to indicate whether or not the read opertion completed successfully. If the call was successful, **ds_read**( ) returns the value **DS_SUCCESS**. If the call fails, it returns an error code. (Refer to Chapter 31 for a comprehensive list of error codes.) **CHECK_DS_CALL** interprets this return value and returns successfully to the program or branches to an error-handling routine.

The *session* input parameter is a private object generated by **ds_bind**( ) prior to the **ds_read**( ) call, as shown in the preceding code fragment. **DS_DEFAULT_CONTEXT** describes the characteristics of a Directory Service interaction. Most XDS API function calls require these two input parameters because they define the operating parameters of a session with a GDS server. (Sessions are described in Section 27.2.1; contexts are described in Section 27.3.2.)

The result of a Directory Service request is returned in a private object (in this case, **result**) that is appropriate to the type of operation. The result of the operation is returned in a single OM object. The components of the result are represented by OM attributes in the operations result object:

- **DS_C_COMPARE_RESULT**

  Returned by **ds_compare**( )

- **DS_C_LIST_RESULT**

  Returned by **ds_list**( )

- **DS_C_READ_RESULT**

  Returned by **ds_read**( )

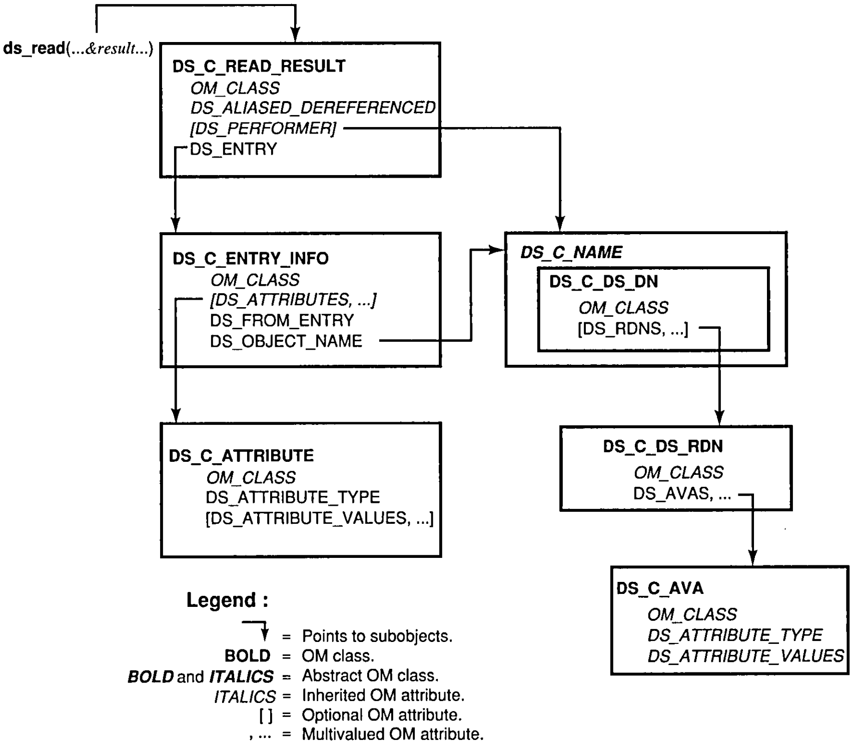
- **DS_C_SEARCH_RESULT**

  Returned by **ds_search**( )

The OM class returned by **ds_read**( ) is **DS_C_READ_RESULT**. The OM class returned by the **ds_compare**( ) call is **DS_C_COMPARE_RESULT**, and so on. (Refer to the *OSF DCE Application Development Reference* for a description of the OM classes associated with a particular function call; refer to Chapter 30 of this guide for for full descriptions of the OM attributes, syntaxes, and values associated with these OM classes.)

The superclasses, subclasses, and OM attributes for **DS_C_READ_RESULT** are shown in Figure 27-1.

Figure 27–1. Output from ds_read( ): DS_C_READ_RESULT



The **result** value is returned to the workspace in private implementation-specific format. As such, it cannot be read directly by an application program, but requires a series of **om_get()** function calls to extract the requested information from it. The following code fragment from **example.c** shows how a series of **om_get()** calls extracts the list of telephone numbers associated with the distinguished name for **Peter Piper**. Section 26.7.2 of this guide describes this extraction process in detail.

```
/*
 * extract the telephone number(s) of "name" from the result
 *
 * There are 4 stages:
 * (1) get the Entry-Information from the Read-Result.
 * (2) get the Attributes from the Entry-Information.
 * (3) get the list of phone numbers.
 * (4) scan the list and print each number.
 */

CHECK_OM_CALL(    om_get()(result,
              OM_EXCLUDE_ALL_BUT_THESE_TYPES
            + OM_EXCLUDE_SUBOBJECTS,
              entry_list, OM_FALSE, 0, 0, &entry,
              &total_num));

CHECK_OM_CALL(    om_get()(entry->value.object.object,
              OM_EXCLUDE_ALL_BUT_THESE_TYPES
            + OM_EXCLUDE_SUBOBJECTS,
              attributes_list, OM_FALSE, 0, 0, &attributes,
              &total_num));

CHECK_OM_CALL(    om_get()(attributes->value.object.object,
              OM_EXCLUDE_ALL_BUT_THESE_TYPES
            + OM_EXCLUDE_SUBOBJECTS,
              telephone_list, OM_FALSE, 0, 0, &telephones,
              &total_num));
```

27.8 Directory Search Operations

Search functions can be used to browse through the Directory Information Tree (DIT). For example, a search request could supply the distinguished name of an entry and request a list of the distinguished names of the children of that entry.

XDS API implements the following search operations:

- **ds_list( )**

  The requestor supplies a distinguished name. The Directory Service returns a list of the immediate subordinates of the named entry.

- **ds_search( )**

  The requestor supplies a search criterion known as a *filter*. The user names a subtree of the DIT, specifies some target attribute types, and formulates an expression by combining a number of attributes using logical AND, OR, or NOT operators. The Directory Service returns information from all of the entries within the specified portion of the DIT that matches the filter. Section 27.8.1 includes a description of how filters are used in **acl.c**.

27.8.1 Searching the Directory

This section describes a typical search operation using the **ds_search( )** function call. It only includes the tasks directly related to the search operation and does not include tasks related to the XDS interface or other directory operations.

A typical search operation involves the following steps:

1. Define the necessary object identifier constants for the OM classes and OM attributes that will define public objects for input to **ds_search( )** by using the **OM_EXPORT** macro.

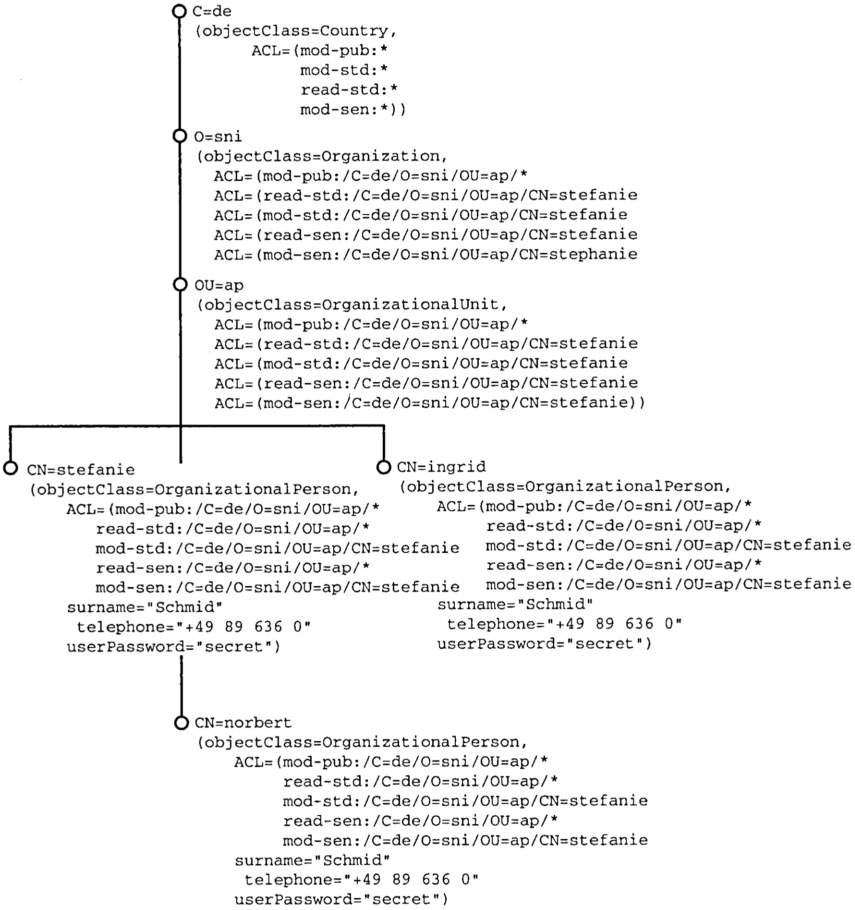
2. Declare the variables that will contain the output from the XDS functions that will be used in the application.

3. Build public objects (descriptor lists) for the *name* parameter to **ds_search( )**.

4. Specify the portion of the Directory Information Tree to be searched.
5. Create a descriptor list for the *filter* parameter to **ds_search()** that designates which entries are to be eliminated from the search.
6. Create a descriptor list for the *selection* parameter to **ds_search()** that selects the type and scope of information in your request.
7. Perform the search operation

These steps are demonstrated in the following code fragments from **acl.h**. The program includes a search operation. In order to perform the operation, the program assumes the directory contains the subtree shown in Figure 27-2.

Figure 27–2. Subtree for the acl.h Sample Program


C=de
(objectClass=Country,
ACL=(mod-pub:*
mod-std:*
read-std:*
mod-sen:*))
O=sni
(objectClass=Organization,
ACL=(mod-pub:/C=de/O=sni/OU=ap/*
ACL=(read-std:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(mod-std:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(read-sen:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(mod-sen:/C=de/O=sni/OU=ap/CN=stephanie
OU=ap
(objectClass=OrganizationalUnit,
ACL=(mod-pub:/C=de/O=sni/OU=ap/*
ACL=(read-std:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(mod-std:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(read-sen:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(mod-sen:/C=de/O=sni/OU=ap/CN=stefanie))
CN=stefanie
(objectClass=OrganizationalPerson,
ACL=(mod-pub:/C=de/O=sni/OU=ap/*
read-std:/C=de/O=sni/OU=ap/*
mod-std:/C=de/O=sni/OU=ap/CN=stefanie
read-sen:/C=de/O=sni/OU=ap/*
mod-sen:/C=de/O=sni/OU=ap/CN=stefanie
surname="Schmid"
telephone="+49 89 636 0"
userPassword="secret")
CN=ingrid
(objectClass=OrganizationalPerson,
ACL=(mod-pub:/C=de/O=sni/OU=ap/*
read-std:/C=de/O=sni/OU=ap/*
mod-std:/C=de/O=sni/OU=ap/CN=stefanie
read-sen:/C=de/O=sni/OU=ap/*
mod-sen:/C=de/O=sni/OU=ap/CN=stefanie
surname="Schmid"
telephone="+49 89 636 0"
userPassword="secret")
CN=norbert
(objectClass=OrganizationalPerson,
ACL=(mod-pub:/C=de/O=sni/OU=ap/*
read-std:/C=de/O=sni/OU=ap/*
mod-std:/C=de/O=sni/OU=ap/CN=stefanie
read-sen:/C=de/O=sni/OU=ap/*
mod-sen:/C=de/O=sni/OU=ap/CN=stefanie
surname="Schmid"
telephone="+49 89 636 0"
userPassword="secret")

27.8.2 Step 1: Export Object Identifiers

Most application programs find it convenient to export all the names they use from the same C source module. In the following code fragment from **acl.c**, the **OM_EXPORT** macro allocates memory for the constants that represent the object OM classes and OM attributes required for the search operation:

```
/* The application must export the object identifiers it  */
/* requires.                                               */

OM_EXPORT (DS_C_AVA)
OM_EXPORT (DS_C_DS_RDN)
OM_EXPORT (DS_C_DS_DN)
OM_EXPORT (DS_C_ENTRY_INFO_SELECTION)
OM_EXPORT (DS_C_ATTRIBUTE)
OM_EXPORT (DS_C_ATTRIBUTE_LIST)
OM_EXPORT (DS_C_FILTER)
OM_EXPORT (DS_C_FILTER_ITEM)
OM_EXPORT (DSX_C_GDS_SESSION)
OM_EXPORT (DSX_C_GDS_CONTEXT)
OM_EXPORT (DSX_C_GDS_ACL)
OM_EXPORT (DSX_C_GDS_ACL_ITEM)

OM_EXPORT (DS_A_COUNTRY_NAME)
OM_EXPORT (DS_A_ORG_NAME)
OM_EXPORT (DS_A_ORG_UNIT_NAME)
OM_EXPORT (DS_A_COMMON_NAME)
OM_EXPORT (DS_A_LOCALITY_NAME)
OM_EXPORT (DS_A_OBJECT_CLASS)
OM_EXPORT (DS_A_USER_PASSWORD)
OM_EXPORT (DS_A_PHONE_NBR)
OM_EXPORT (DS_A_SURNAME)
OM_EXPORT (DS_A_ACL)
OM_EXPORT (DS_TYPELESS_RDN)

OM_EXPORT (DS_O_TOP)
OM_EXPORT (DS_O_COUNTRY)
OM_EXPORT (DS_O_ORG)
OM_EXPORT (DS_O_ORG_UNIT)
OM_EXPORT (DS_O_PERSON)
OM_EXPORT (DS_O_ORG__PERSON)
```

The **OM_EXPORT** macro takes the OM class name as input and creates two new data structures: a character string and structure of type **OM_string**. The structure of type **OM_string** contains a length and a pointer that are used in Step 3 to initialize the value of the object identifier.

27.8.3 Step 2: Declare Local Variables

The local variables are defined in the following code fragment from **acl.c**:

```
OM_workspace        workspace;      /* workspace for objects          */
OM_private_object   session;        /* Session object.                */
OM_private_object   bound_session;  /* Holds the Session object which */
                                    /* is returned by ds_bind()       */
OM_public_object    context;        /* Context object.                */
OM_private_object   result;         /* Holds the search result object.*/
OM_sint             invoke_id;      /* Integer for the invoke id      */
                                    /* returned by ds_search().       */
                                    /* (this parameter must be present*/
                                    /* even though it is ignored).    */
OM_type             sinfo_list[] = { DS_SEARCH_INFO, 0 };
OM_type             entry_list[] = { DS_ENTRIES, 0 };
                                    /* Lists of types to be extracted */
OM_public_object    sinfo;          /* Search-Info object from result.*/
OM_public_object    entry;          /* Entry object from search info. */
OM_value_position   total_num;      /* Number of descriptors returned.*/
OM_return_code      rc;             /* XOM function return code.      */
register int        i;
char                user_name[MAX_DN_LEN];
                                    /* Holds requestor's name.        */
char                entry_string[MAX_DN_LEN + 7] = "[?r??] ";
                                    /* Holds entry details.           */
```

These data types are defined in a **typedef** statement in the **xom.h** header file. Since asynchronous operations are not supported, the *invoke_id* functionality is redundant. The *invoke_id* parameter must be supplied to the

XDS functions as described in the *OSF DCE Application Development Reference*, but its return value should be ignored.

27.8.4 Step 3: Build Public Objects for the name Parameter to ds_search( )

The public objects required by the search operation are defined in the **acl.h** header file. The *name* input parameter in the **ds_search( )** function call in **acl.c** is the representation of the distinguished name for the root of the DIT. The following code fragment from **acl.c** shows how the descriptor list for the distinguished name is initialized:

```
static OM_descriptor dn_root[] = {
 OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
 OM_NULL_DESCRIPTOR
};
```

27.8.5 Step 4: Specify the Portion of the DIT To Be Searched

The **ds_search( )** call requires the *subset* input parameter. The *subset* parameter specifies the portion of the DIT to be searched. It takes the value of one of the following symbolic constants, which are defined in the **xds.h** header file:

- **DS_BASE_OBJECT**, meaning to search just the given object entry
- **DS_ONE_LEVEL**, meaning to search just the immediate subordinates of the given object entry
- **DS_WHOLE_SUBTREE**, meaning to search the given object and all its subordinates

The *subset* parameter in **acl.c** takes the value **DS_WHOLE_SUBTREE**.

27.8.6 Step 5: Create a Filter

The *filter* input parameter is used to eliminate entries from the search that are not wanted. Information is only returned on entries that satisfy the filter.

DS_C_FILTER inherits the attributes from its superclass **OM_C_OBJECT**, as do all OM classes. **OM_C_OBJECT** (as defined Chapter 26) has one OM attribute, **OM_CLASS**, which has the value of an object identifier string that identifies the numeric representation of the object's OM class. **DS_C_FILTER**, on the other hand, has several OM attributes.

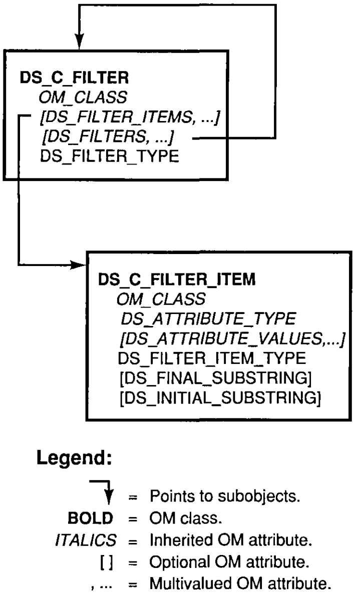
The purpose of the **DS_C_FILTER** is to select or reject an object on the basis of information in its directory entry. It has the following OM attributes:

- **DS_FILTER_ITEMS**
- **DS_FILTERS**
- **DS_FILTER_TYPE**

Two of these OM attributes, **DS_FILTER_ITEMS** and **DS_FILTERS**, have values that are OM object classes themselves. The OM attribute **DS_FILTER_ITEMS** has the value OM class **DS_C_FILTER_ITEM**. **DS_C_FILTER_ITEM** is a component of a filter and defines the nature of the filter. The OM attribute **DS_FILTERS** has the value of OM class **DS_C_FILTER** and thus defines a collection of filters. The OM attribute **DS_FILTER_TYPE** has a value that is an enumerated type, which takes one of the values **DS_AND**, **DS_OR**, or **DS_NOT**.

Figure 27-3 shows the relationship of **DS_C_FILTER** to its superclass **OM_C_OBJECT**, and its attibutes.

Figure 27–3. OM Class DS_C_FILTER



The **DS_NO_FILTER** constant can be used as the value of this parameter if all entries are searched and no entries are eliminated. This corresponds to a filter with a **DS_FILTER_TYPE** value of **DS_AND**, and no values of the **DS_FILTERS** or **DS_FILTER_ITEMS** OM attributes.

The following code fragment from **acl.c** shows the descriptor list for a filter:

```
/* The following descriptor list specifies a filter */
/* for search :                                     */
/*     (Present: objectClass)                       */

static OM_descriptor filter_item[] = {
 OM_OID_DESC(OM_CLASS, DS_C_FILTER_ITEM),
 {DS_FILTER_ITEM_TYPE, OM_S_ENUMERATION, {DS_PRESENT, 0} },
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS),
 OM_NULL_DESCRIPTOR
};

static OM_descriptor filter[] = {
 OM_OID_DESC(OM_CLASS, DS_C_FILTER),
 {DS_FILTER_ITEMS, OM_S_OBJECT, {0, filter_item} },
 {DS_FILTER_TYPE, OM_S_ENUMERATION, {DS_AND, 0} },
 OM_NULL_DESCRIPTOR
};
```

27.8.7 Step 6: Create an Entry-Information-Selection Parameter

The **ds_search**( ) call requires a *selection* input parameter to specify what information from the entry is requested. The *selection* parameter of the **ds_search**( ) call in **acl.h** requests information on all attributes, as shown in the following code fragment:

```
static OM_descriptor selection_acl[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
 {DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_FALSE},
 OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DSX_A_ACL),
 {DS_INFO_TYPE, OM_S_ENUMERATION, DS_TYPES_AND_VALUES},
 OM_NULL_DESCRIPTOR
};
```

As shown in the code fragment, **DS_ALL_ATTRIBUTES** has a syntax of **OM_S_BOOLEAN** that is set to **OM_FALSE**, indicating that only the requested attributes of the entry are to be returned. The ACL attribute's types and values are selected. **DS_INFO_TYPE** has a value of **DS_TYPES_AND_VALUES**, indicating that both the attribute types and the attribute values in the entry are returned.

27.8.8 Step 7: Perform the Search Operation

The following code fragment from **acl.c** shows the **ds_search**( ) function call:

```
/* Search the whole subtree below root.
 * The filter selects entries with an object-class attribute.
 * The selection extracts the ACL attribute from each
 * selected entry.
```

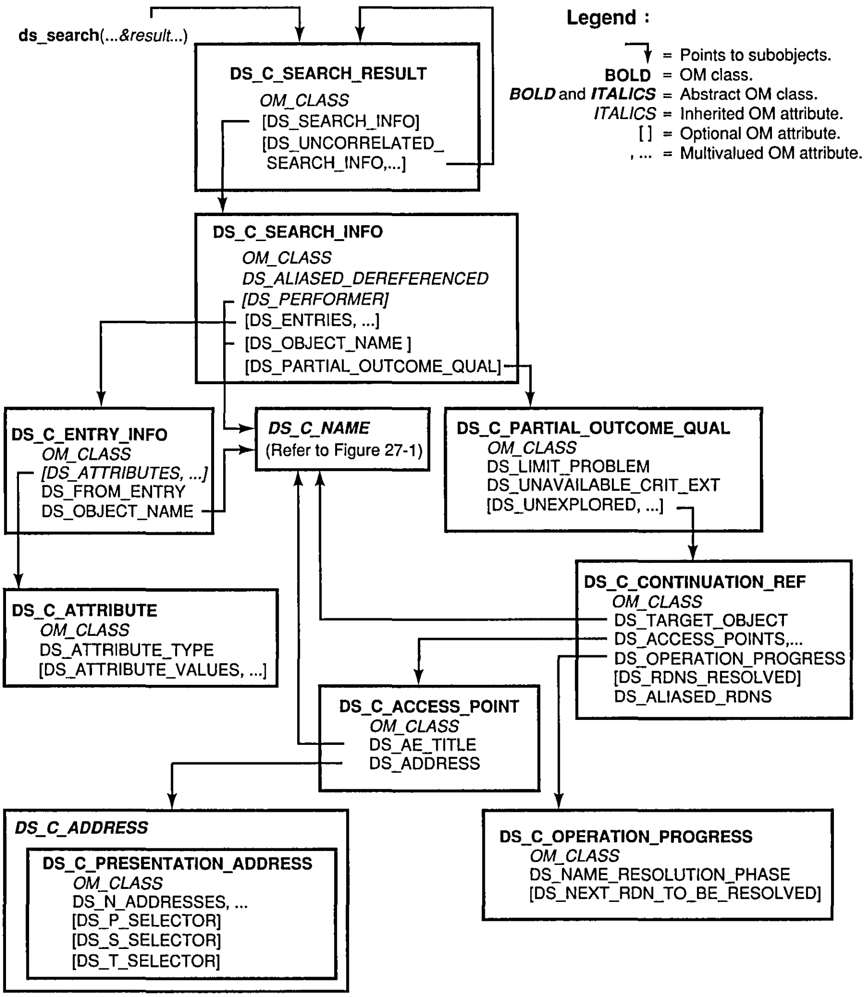
```
     * The results are returned in the private object "result".
     *
     * NOTE: Since every entry contains an object-class attribute the
     *       filter performs no function other than to demonstrate how
     *       filters may be used.
     *
     */
    if(ds_search(bound_session, context, dn_root, DS_WHOLE_SUBTREE,
      filter , OM_FALSE, selection_acl, &result, &invoke_id) != DS_SUCCESS)
            printf("ds_search() error\n");
```

The **ds_search**( ) call returns the value **DS_SUCCESS** if the call successfully completes. Otherwise, it returns an error code. (Refer to Chapter 31 for a comprehensive list of error codes.)

The result of the search operation is returned to the workspace in a private object **result**. This result is returned as a single OM object. The components of the result are represented by OM attributes in the operation's **result** object.

The OM class returned by **ds_search**( ) is **DS_C_SEARCH_RESULT**. The superclasses, subclasses, and attributes for **DS_C_SEARCH_RESULT** are shown in Figure 27-4.

Figure 27-4. OM Class DS_C_SEARCH_RESULT



The **result** object is returned to the workspace in a private implementation-specific format. As such, it cannot be read directly by an application program, but requires a series of **om_get( )** function calls to extract the requested information.

27.9 Directory Modify Operations

Modify functions alter information in the directory. For example, if an employee of an Organizational Unit transfers to a new Organizational Unit, a typical modify request would modify the **OU** name attribute in the person's directory entry to reflect the change.

XDS API implements the following modify functions:

- **ds_modify_entry( )**

  The requestor gives a distinguished name and a list of modifications to the named entry. The Directory Service carries out the specified changes if the user requesting the change has proper access rights.

- **ds_add_entry( )**

  The requestor gives a distinguished name and values for a new entry. The entry is added as a leaf node in the DIT if the user requesting the change has proper access rights.

- **ds_remove_entry( )**

  The requestor gives a distinguished name. The entry with that name is removed if the user requesting the change has proper access rights.

- **ds_modify_rdn( )**

  The requestor gives a distinguished name and a new Relative Distinguished Name (RDN) for the entry. The directory changes the entry's RDN if the user requesting the change has proper access rights.

Note that **ds_add_entry( )**, **ds_remove_entry( )**, and **ds_modify_rdn( )** only apply to leaf entries. They are not intended to provide a general facility for building and manipulating the DIT.

27.9.1 Modifying Directory Entries

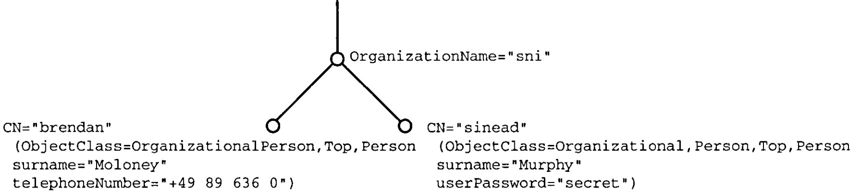
This section describes a modification and subsequent listing of the DIT using the **ds_add_entry()**, **ds_list()**, and **ds_remove_entry()** function calls. It includes a description of tasks directly related to these operations and does not include service-related tasks. It does not include a **ds_modify_entry()** function call. The modify operation is used in the context of the X.500 *Abstract Service Definition.*

A typical operation to add, remove, or list an entry involves following the same basic steps that were defined previously for the read and search operations:

1. Define the necessary object identifier constants for the OM classes and OM attributes that will define public objects for input to the function calls by using the **OM_EXPORT** macro.
2. Declare the variables that will contain the output from the XDS functions you will use in your application.
3. Build public objects (descriptor lists) for the *name* parameters to the function calls.
4. Create descriptor lists for the attributes to be added, removed, or listed.
5. Perform the operations.

These steps are demonstrated in the following code fragments. The program adds two entries to the directory, then a list operation is performed on their superior entry, and finally the two entries are removed from the directory. The directory tree shown in Figure 27-5 is used in the program.

Figure 27–5. A Sample Directory Tree


CountryName="ie"
OrganizationName="sni"
CN="brendan"
(ObjectClass=OrganizationalPerson,Top,Person
surname="Moloney"
telephoneNumber="+49 89 636 0")
CN="sinead"
(ObjectClass=Organizational,Person,Top,Person
surname="Murphy"
userPassword="secret")

27.9.2 Step 1: Export Object Identifiers for Required Directory Classes and Attributes

In the following code fragment, the **OM_EXPORT** macro allocates memory for the constants that represent the object classes and attributes required for the add, list, and remove operations:

```
/* The application has to export the object identfiers   */
/* it requires.    */

OM_EXPORT (DS_C_AVA)
OM_EXPORT (DS_C_DS_RDN)
OM_EXPORT (DS_C_DS_DN)
OM_EXPORT (DS_C_ENTRY_INFO_SELECTION)
OM_EXPORT (DS_C_ATTRIBUTE)
OM_EXPORT (DS_C_ATTRIBUTE_LIST)

OM_EXPORT (DS_A_COUNTRY_NAME)
OM_EXPORT (DS_A_ORG_NAME)
OM_EXPORT (DS_A_COMMON_NAME)
OM_EXPORT (DS_A_OBJECT_CLASS)
OM_EXPORT (DS_A_PHONE_NBR)
OM_EXPORT (DS_A_USER_PASSWORD)
OM_EXPORT (DS_A_SURNAME)

OM_EXPORT (DS_O_TOP)
OM_EXPORT (DS_O_PERSON)
OM_EXPORT (DS_O_ORG_PERSON)
```

27.9.3 Step 2: Declare Local Variables

The local variables **bound_session**, **result**, and **invoke_id** are defined in the following sample code fragment:

```
OM_private_object bound_session; /* Holds the Session object  */
                                 /* which is returned by      */
                                 /* ds_bind().                */
OM_private_object result;        /* Holds the list result     */
```

```
                                        /* object.                   */
    OM_sint             invoke_id;      /* Integer for the invoke id */
                                        /* returned by ds_search().  */
                                        /* This parameter must be    */
                                        /* present even though it is */
                                        /* ignored.                  */
```

These data types are defined in **typedef** statements in the **xom.h** header file. The **bound_session** and **result** variables are defined as data type **OM_private_object** because they are returned by **ds_bind**( ) and **ds_list**( ) operations to the workspace as private objects. Since asynchronous operations are not supported, the *invoke_id* parameter functionality is redundant. The *invoke_id* parameter must be supplied to the XDS functions as described in the *OSF DCE Application Development Reference*, but its return value should be ignored.

27.9.4 Step 3: Build Public Objects

The public objects required by the **ds_add_entry**( ), **ds_list**( ), and **ds_remove_entry**( ) operations are defined in the following code fragment:

```
/* Build up descriptor lists for the following distinguished names: */
/*     C=ie/O=sni                                                   */
/*     C=ie/O=sni/CN=brendan                                        */
/*     C=ie/O=sni/CN=sinead                                         */

static OM_descriptor    ava_ie[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("ie")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    ava_sni[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("sni")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    ava_brendan[] = {
        OM_OID_DESC(OM_CLASS, DS_C_AVA),
        OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
        {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("brendan")},
        OM_NULL_DESCRIPTOR
    };
    static OM_descriptor    ava_sinead[] = {
        OM_OID_DESC(OM_CLASS, DS_C_AVA),
        OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
        {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("sinead")},
        OM_NULL_DESCRIPTOR
    };
    static OM_descriptor    rdn_ie[] = {
        OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
        {DS_AVAS, OM_S_OBJECT, {0, ava_ie}},
        OM_NULL_DESCRIPTOR
    };
    static OM_descriptor    rdn_sni[] = {
        OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
        {DS_AVAS, OM_S_OBJECT, {0, ava_sni}},
        OM_NULL_DESCRIPTOR
    };
    static OM_descriptor    rdn_brendan[] = {
        OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
        {DS_AVAS, OM_S_OBJECT, {0, ava_brendan}},
        OM_NULL_DESCRIPTOR
    };
    static OM_descriptor    rdn_sinead[] = {
        OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
        {DS_AVAS, OM_S_OBJECT, {0, ava_sinead}},
        OM_NULL_DESCRIPTOR
    };
    static OM_descriptor dn_sni[] = {
        OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
        {DS_RDNS,OM_S_OBJECT,{0,rdn_ie}},
        {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
        OM_NULL_DESCRIPTOR
    };
    static OM_descriptor dn_brendan[] = {
        OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
        {DS_RDNS,OM_S_OBJECT,{0,rdn_ie}},
        {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
        {DS_RDNS,OM_S_OBJECT,{0,rdn_brendan}},
        OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_sinead[] = {
        OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
        {DS_RDNS,OM_S_OBJECT,{0,rdn_ie}},
        {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
        {DS_RDNS,OM_S_OBJECT,{0,rdn_sinead}},
        OM_NULL_DESCRIPTOR
};
```

27.9.5 Step 4: Create Descriptor Lists for Attributes

The following code fragments show how the attribute lists are created for the attributes to be added to the directory.

First, initialize the public object **object_class** to contain the representation of the classes in the DIT that are common to both organizational person entries, top, person, and organizational person:

```
/* Build up an array of object identifiers for the     */
/* attributes to be added to the directory.            */

static OM_descriptor    object_class[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS),
 OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_TOP),
 OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_PERSON),
 OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_ORG_PERSON),
 OM_NULL_DESCRIPTOR
};
```

Next, initialize the public objects that represent the attributes to be added: **surname** and **telephone** for the distinguished name of Brendan, and **surname2** and **password** for the distinguished name of Sinead:

```
static OM_descriptor    telephone[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_PHONE_NBR),
 {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING,
 OM_STRING("+49 89 636 0")},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    surname[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_SURNAME),
 {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("Moloney")},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    surname2[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_SURNAME),
 {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING,
 OM_STRING("Murphy")},
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    password[] = {
 OM_OID_DESC(OM_CLASS, DS_C_AVA),
 OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_USER_PASSWORD),
 {DS_ATTRIBUTE_VALUES, OM_S_OCTET_STRING, OM_STRING("secret")},
 OM_NULL_DESCRIPTOR
};
```

Finally, initialize the public objects that represent the list of attributes to be added to the directory: **attr_list1** for the distinguished name Brendan, and **attr_list2** for the distinguished name Sinead:

```
static OM_descriptor    attr_list1[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, object_class} },
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, surname} },
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, telephone} },
 OM_NULL_DESCRIPTOR
};

static OM_descriptor    attr_list2[] = {
 OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, object_class} },
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, surname2} },
 {DS_ATTRIBUTES, OM_S_OBJECT, {0, password} },
 OM_NULL_DESCRIPTOR
};
```

The **attr_list1** variable contains the public objects **surname** and **telephone**, the C representations of the attributes of the distinguished name **/C=ie/O=sni/CN=brendan** that are added to the directory. The **attr_list2** variable contains the public objects **surname2** and **password**, the C representations of the attributes of the distinguished name **/C=ie/O=sni/CN=sinead**.

27.9.6 Step 5: Perform the Operations

The following code fragments show the **ds_add_entry**( ), **ds_list**( ), and the **ds_remove_entry**( ) calls.

First, the two **ds_add_entry**( ) function calls add the attribute lists contained in **attr_list1** and **attr_list2** to the distinguished names represented by **dn_brendan** and **dn_sinead**, respectively:

```
/* Add two entries to the GDS server.                */

if (ds_add_entry(bound_session, DS_DEFAULT_CONTEXT,
   dn_brendan, attr_list1,
   &invoke_id) != DS_SUCCESS)
   printf("ds_add_entry() error\n");

if (ds_add_entry(bound_session, DS_DEFAULT_CONTEXT,
   dn_sinead, attr_list2,
   &invoke_id) != DS_SUCCESS)
   printf("ds_add_entry() error\n");
```

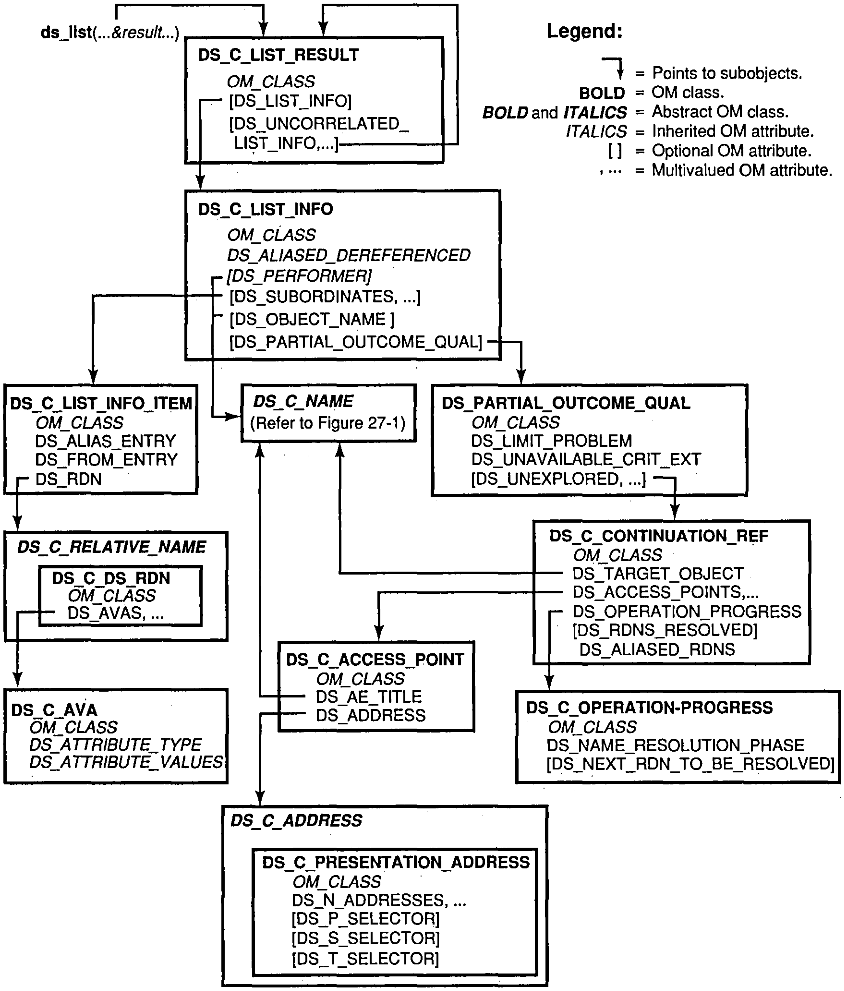
Next, list all the subordinates of the object referenced by the distinguished name **/C=ie/O=sni**:

```
if (ds_list(bound_session, DS_DEFAULT_CONTEXT, dn_sni,
   &result, &invoke_id)
```

```
    != DS_SUCCESS)
    printf("ds_list() error\n");
```

The **ds_list()** call returns the result in the private object **result** to the workspace. The components of **result** are represented by OM attributes in the OM class **DS_C_LIST_RESULT** (as shown in Figure 27-6) and can only be read by a series of **om_get()** calls.

Figure 27–6. OM Class DS_C_LIST_RESULT



Finally, remove the two entries from the directory:

```
if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT,
    dn_brendan, &invoke_id)
    != DS_SUCCESS)
    printf("ds_remove_entry() error\n");

if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT,
    dn_sinead, &invoke_id)
    != DS_SUCCESS)
    printf("ds_remove_entry() error\n");
```

Chapter 28

Sample Application Programs

This chapter contains three sample programs and the header files that are included in them (in parentheses):

- **example.c (example.h)**
- **acl.c (acl.h)**
- **teldir.c**

Most of the concepts that you will need to know to understand and use these programs are discussed in previous chapters in Part 4 of this guide. The programs are arranged so that the simplest program, **example.c**, is presented first and the most complex program, **teldir.c** is presented last. The three programs demonstrate basic XDS and XOM API principles and concepts in operation. The **teldir.c** program is considerably more complex and uses a more sophisticated approach. It allows the user to enter values dynamically; for example, a surname and phone number.

28.1 General Programming Guidelines

Writing an application program using XDS and XOM APIs involves the following general steps before you begin coding:

1. Select the interface functions that you will need for your application and determine the parameters for the function calls.
2. Check for abstract OM classes and superclasses of objects that you will manipulate for inherited OM attributes in Part 4C.
3. Find the correct symbolic constants of the appropriate packages; these can be found in the header files included with the GDS API, such as **xdsbdcp.h**.
4. Determine the error handling required.

28.2 The example.c Program

The **example.c** program uses XDS API in synchronous mode to read a telephone number or numbers of a distinguished name. The program consists of the following general steps:

1. Define the required object identifier constants.
2. Declare the variables involved with Directory Service operations (Steps 3, 4, 7, 8, and 9).
3. Build the distinguished name of **Peter Piper** as a public object for the input parameter to **ds_read( )**.
4. Build a public object for the *selection* parameter to **ds_read( )**.
5. Declare the variables to extract the telephone numbers using **om_get( )**.
6. Initialize the directory service and get an OM workspace.
7. Pull in the required packages.
8. Bind to a default directory session.
9. Perform the read operation to extract the telephone number of a distinguished name from the directory.

10. Terminate the Directory Service session.

11. Extract the telephone number(s) using a series of **om_get( )** calls.

12. Release the storage occupied by private and public objects that are no longer needed.

13. Print the telephone number string.

14. Release the storage occupied by public objects containing telephone numbers.

15. Continue processing and exit.

Note: The steps that follow are highlighted in boldface so that you can follow the sequence as you examine the **example.c** program.

Step 1 uses the **OM_EXPORT** macro to allocate memory for the object identifier constants that represent an OM class or OM attribute. These constants are the OM attribute values that are used to build the public objects that are required as input to **ds_read( )**.

Step 2 declares the variables for Directory Service operations and error handling. The **session** and **workspace** variables are required for binding a session to a server and creating a workspace into which **ds_read( )** can deposit the results of the read operation on the directory.

The **result** variable is a pointer that is returned by **ds_read( )** to the workspace. The information stored in **result** is in implementation-specific private format that is not accessible directly by the application program. Subsequent **om_get( )** calls extract the telephone number(s) requested by the program from **result** and store the information in the variable **telephones** (declared in **Step 5**).

The **error** and **return_code** variables are used by the program for error handling. The **error** variable is used for processing the return code from XDS API function calls. The **return_code** variable is used by the error handling header file **example.h** for processing return codes from **om_get( )** function calls.

Step 3 builds the public object representing the distinguished name of **Peter Piper**. The program uses statically defined public objects to demonstrate the basic principles of building public objects. However, a more sophisticated approach is presented in the last sample program in this chapter, **teldir.c**. The **teldir.c** program dynamically defines a public object from a user-supplied name in DCE string format.

In this program (**example.c**), the process starts with the definition of an array of descriptor lists as AVAs. These AVAs are public objects that are included in the definition of RDNs. The RDNs, in turn, are included in the distinguished name of **Peter Piper** stored in **name**. Using the same method of static definition, **Step 4** defines the **DS_C_ENTRY_INFO_SELECTION** public object and stores it in the variable **selection**. The **name** and **selection** variables are required as input parameters to **ds_read( )**. This process is described in detail in Chapter 27.

Step 5 declares the variables required by **om_get( )** to extract the telephone number(s) from **result**. The **entry_list**, **attributes_list**, and **telephone_list** variables are of type **OM_type** and are initialized to the values of the OM attribute types **DS_ENTRY**, **DS_ATTRIBUTES**, and **DS_ATTRIBUTE_VALUES**, respectively. **DS_ENTRY** contains the selected list of entries; **DS_ATTRIBUTES** contains the selected list of attribute types; and **DS_ATTRIBUTE_VALUES** contains the actual values of the telephone numbers.

The **entry**, **attributes**, and **telephones** variables are of type **OM_public_object** because they store the output parameters of **om_get( )**. The **om_get( )** call makes these objects available to the application program as public object data types. The program must remove layers of objects and subobjects to get at the actual string data values of the telephone numbers.

The **telephones** variable contains the actual string values of the telephone number(s). It is a descriptor in the array of descriptors that make up the public object that contains the actual string data that the program wants to extract from the directory.

Step 6 initializes the Directory Service and gets an OM workspace in which **ds_read( )** deposits the result of the read operation.

Step 7 pulls in the Basic Directory Contents Package into the program because it contains features that are required by the program not included in the default package (the Directory Service Package).

Step 8 binds the session to the default session. An application program can bind with a specifically tailored session object using an instance of OM class **DS_C_SESSION**. In most cases, however, it is sufficient to use the constant **DS_DEFAULT_SESSION**. **DS_DEFAULT_SESSION** uses the default values of **DS_C_SESSION** and the values of specific OM attributes that are set locally in the cache. These OM attributes are **DS_DSA_ADDRESS** (the address of the default DSA) and **DS_DSA_NAME** (the distinguished name of the default DSA). It is the responsibility of local administrators to make sure that these default values are set correctly in the cache.

Step 9 performs the read operation and deposits the result in the workspace in **result**. The **result** variable is one of the input parameters for the **om_get**( ) function call. The **session** variable and the **DS_DEFAULT_CONTEXT** constant are the *session* and *context* parameters required to be present in the **ds_read**( ) function call.

The **name** variable holds the public object representing the distinguished name of **Peter Piper**; the **selection** variable contains the public object the indicates which attributes and values are selected by the read operation from the entry. The *invoke_id* parameter is not used by the DCE implementation of XDS and is ignored.

Step 10 terminates the directory session.

Step 11 uses a series of **om_get**( ) calls to extract the telephone number(s). The first **om_get**( ) extracts the information about the entry from **result** and puts it in **entry**. The second **om_get**( ) extracts the attribute types from **entry** and puts them in **attributes**. The third **om_get**( ) extracts the actual values of the telephone numbers from **attributes** and puts them in **telephones**. The **telephones** variable contains the string data values of the telephone number(s).

Step 12 releases the storage occupied by the private and public objects that are no longer needed. The program has the data values in **telephone** that it needs to continue processing. A **ds_shutdown**( ) is issued that shuts down the interface established by **ds_initialize**( ).

Step 13 prints out each telephone number associated with the distinguished name **Peter Piper** in the directory, or returns an error message if the number is not in the correct format. It checks for an attribute with type **DS_ATTRIBUTE_VALUES** and a syntax of **OM_S_PRINTABLE_STRING**, the proper syntax for a telephone number. The constant **OM_S_SYNTAX** is used to mask the five high-order bits in the syntax because they are used internally by the XOM service.

Step 14 releases the storage occupied by **telephones** because it is no longer needed.

Step 15 continues processing and exits.

28.2.1 The example.c Code

The following code is a listing of the **example.c** program:

```
/*
 *      sample application that uses XDS in synchronous mode
 *
 *      This program reads the telephone number(s) of a given target name.
 */

#include <stdio.h>

#include <dce/xom.h>
#include <dce/xds.h>
#include <dce/xdsbdcp.h>

#include "example.h"           /* possible Error Handling header */


/* Step 1
 *
 * Define necessary Object Identifier constants
 */
OM_EXPORT(DS_A_COMMON_NAME)
OM_EXPORT(DS_A_COUNTRY_NAME)
OM_EXPORT(DS_A_ORG_NAME)
OM_EXPORT(DS_A_ORG_UNIT_NAME)
OM_EXPORT(DS_A_PHONE_NBR)
OM_EXPORT(DS_C_AVA)
OM_EXPORT(DS_C_DS_DN)
OM_EXPORT(DS_C_DS_RDN)
OM_EXPORT(DS_C_ENTRY_INFO_SELECTION)


/* Step 2 */

int main(void)
{
   DS_status         error;        /* return value from DS functions    */
   OM_return_code    return_code;  /* return value from OM functions    */
   OM_workspace      workspace;    /* workspace for objects             */
   OM_private_object  session;     /* session for directory operations */
   OM_private_object  result;      /* result of read operation         */
   OM_sint            invoke_id;    /* Invoke-ID of the read operation */
   OM_value_position  total_num;    /* Number of Attribute Descriptors */


    static DS_feature   bdcp_package[] = {
        { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
        { { (OM_uint32)0, (void *)0 }, OM_FALSE },
        };

/* Step 3 */
 *
 * Public Object ("Descriptor List") for Name parameter to ds_read().
 * Build the Distinguished-Name of Peter Piper.
 */

    static OM_descriptor        country[] = {
        OM_OID_DESC(OM_CLASS, DS_C_AVA),
        OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
        { DS_ATTRIBUTE_VALUES,OM_S_PRINTABLE_STRING,OM_STRING("US") },
        OM_NULL_DESCRIPTOR
        };
    static OM_descriptor        organization[] = {
        OM_OID_DESC(OM_CLASS, DS_C_AVA),
        OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
        { DS_ATTRIBUTE_VALUES,OM_S_TELETEX_STRING,
          OM_STRING("Acme Pepper Co") },
        OM_NULL_DESCRIPTOR
        };
    static OM_descriptor        organizational_unit[] = {
        OM_OID_DESC(OM_CLASS, DS_C_AVA),
        OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_UNIT_NAME),
        { DS_ATTRIBUTE_VALUES,OM_S_TELETEX_STRING,OM_STRING("Research") },
        OM_NULL_DESCRIPTOR
        };
    static OM_descriptor        common_name[] = {
        OM_OID_DESC(OM_CLASS, DS_C_AVA),
        OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
        { DS_ATTRIBUTE_VALUES,OM_S_TELETEX_STRING,OM_STRING("Peter Piper") },
        OM_NULL_DESCRIPTOR
        };

    static OM_descriptor       rdn1[] = {
        OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
        { DS_AVAS, OM_S_OBJECT, { 0, country } },
        OM_NULL_DESCRIPTOR
        };
    static OM_descriptor       rdn2[] = {
        OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
        { DS_AVAS, OM_S_OBJECT, { 0, organization } },
        OM_NULL_DESCRIPTOR
        };
    static OM_descriptor       rdn3[] = {
        OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
        { DS_AVAS, OM_S_OBJECT, { 0, organizational_unit } },
        OM_NULL_DESCRIPTOR
        };
    static OM_descriptor       rdn4[] = {
        OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
        { DS_AVAS, OM_S_OBJECT, { 0, common_name } },
        OM_NULL_DESCRIPTOR
        };

    OM_descriptor       name[] = {
        OM_OID_DESC(OM_CLASS, DS_C_DS_DN),
        { DS_RDNS, OM_S_OBJECT, { 0, rdn1 } },
        { DS_RDNS, OM_S_OBJECT, { 0, rdn2 } },
        { DS_RDNS, OM_S_OBJECT, { 0, rdn3 } },
        { DS_RDNS, OM_S_OBJECT, { 0, rdn4 } },
        OM_NULL_DESCRIPTOR
        };
```

/* **Step 4** */

```
        /*
        *
        * Public Object ("Descriptor List") for
        * Entry-Information-Selection parameter to ds_read().
        */
    OM_descriptor selection[] = {
        OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
        { DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, { OM_FALSE, NULL } },
        OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR),
        { DS_INFO_TYPE,OM_S_ENUMERATION, { DS_TYPES_AND_VALUES,NULL } },
        OM_NULL_DESCRIPTOR
        };

/*  Step 5 */

        /*
         * variables to extract the telephone number(s)
         */
        OM_type             entry_list[]       = { DS_ENTRY, 0 };
        OM_type             attributes_list[] = { DS_ATTRIBUTES, 0 };
        OM_type             telephone_list[]  = { DS_ATTRIBUTE_VALUES, 0 };
        OM_public_object  entry;
        OM_public_object  attributes;
        OM_public_object  telephones;
        OM_descriptor      *telephone;   /* current phone number  */

        /*
         * Perform the Directory Service operations:
         * (1) Initialise the Directory Service and get an OM workspace
         * (2) bind a default directory session.
         * (3) read the telephone number of "name".
         * (4) terminate the directory session.
         */

/*  Step 6 */

        CHECK_DS_CALL((OM_object) !(workspace=ds_initialize()));

/*  Step 7 */

        CHECK_DS_CALL(ds_version(bdcp_package, workspace));

/*  Step 8 */

        CHECK_DS_CALL(ds_bind(DS_DEFAULT_SESSION, workspace, &session));

/*  Step 9 */
```

```
        CHECK_DS_CALL(ds_read(session, DS_DEFAULT_CONTEXT, name,
                              selection, &result, &invoke_id));
      /*
       * NOTE: should check here for Attribute-Error (no-such-attribute)
       * in case the "name" doesn't have a telephone.
       * Then for all other cases call error_handler
       */

/* Step 10 */

        CHECK_DS_CALL(ds_unbind(session));

/* Step 11 */

        /*
         * extract the telephone number(s) of "name" from the result
         *
         * There are 4 stages:
         * (1) get the Entry-Information from the Read-Result.
         * (2) get the Attributes from the Entry-Information.
         * (3) get the list of phone numbers.
         * (4) scan the list and print each number.
         */

        CHECK_OM_CALL(   om_get(result,
                          OM_EXCLUDE_ALL_BUT_THESE_TYPES
                        + OM_EXCLUDE_SUBOBJECTS,
                          entry_list, OM_FALSE, 0, 0, &entry,
                          &total_num));

        CHECK_OM_CALL(   om_get(entry->value.object.object,
                          OM_EXCLUDE_ALL_BUT_THESE_TYPES
                        + OM_EXCLUDE_SUBOBJECTS,
                          attributes_list, OM_FALSE, 0, 0, &attributes,
                          &total_num));

        CHECK_OM_CALL(   om_get(attributes->value.object.object,
                          OM_EXCLUDE_ALL_BUT_THESE_TYPES
                        + OM_EXCLUDE_SUBOBJECTS,
                          telephone_list, OM_FALSE, 0, 0, &telephones,
                          &total_num));
```

```
/*  Step 12  */

        /*  We can now safely release all the private objects
         *  and the public objects we no longer need
         */
        CHECK_OM_CALL(om_delete(session));
        CHECK_OM_CALL(om_delete(result));
        CHECK_OM_CALL(om_delete(entry));
        CHECK_OM_CALL(om_delete(attributes));
        CHECK_DS_CALL(ds_shutdown(workspace));

/*  Step 13  */

        for (telephone = telephones;
             telephone->type != DS_ATTRIBUTE_VALUES;
             telephone++)
        {
          if (telephone->type   != DS_ATTRIBUTE_VALUES
          || (telephone->syntax & OM_S_SYNTAX) != OM_S_PRINTABLE_STRING)
          {
          (void) fprintf(stderr, "malformed telephone number\n");
                  exit(EXIT_FAILURE);
          }

                (void) printf("Telephone number: %s\n",
                              telephone->value.string.elements);
        }

/*  Step 14  */

        CHECK_OM_CALL(om_delete(telephones));

/*  Step 15  */

        /*  more application-specific processing can occur here...
         */

        /* ... and finally exit. */
        exit(EXIT_SUCCESS);
}
```

28.2.2 Error Handling

The **example.c** program includes the header file **example.h** for error handling of XDS and XOM function calls. The **example.h** program contains two error handling functions: **CHECK_DS_CALL** for handling XDS API errors, and **CHECK_OM_CALL** for handling XOM API errors. Note that **CHECK_DS_CALL** and **CHECK_OM_CALL** are created specifically for **example.c** and are not part of the XDS or XOM APIs. They are included to demonstrate a possible method for error handling.

XDS and XOM API functions return a status code. In **example.c**, **error** contains the status code for XDS API functions. If the call is successful, the function returns **DS_SUCCESS**. Otherwise, one of the error codes described in Chapter 31 is returned.

The **return_code** variable contains the status code for XOM API functions. If the call is successful, the function returns **OM_SUCCESS**. Otherwise, one of the error codes described in Chapter 36 is returned.

The contents of **example.h** are as follows:

```
/*
 * define some convenient exit codes
 */

#define EXIT_FAILURE 1
#define EXIT_SUCCESS 0


/*
 * declare an error handling function and an error checking macro for DS
 */

void handle_ds_error(DS_status error);

#define CHECK_DS_CALL(function_call)
                error = (function_call);
                if (error != DS_SUCCESS)
                        handle_ds_error(error);


/*
```

```
 * declare an error handling function and an error checking macro for OM
 */

void handle_om_error(OM_return_code return_code);

#define CHECK_OM_CALL(function_call)
                return_code = (function_call);
                if (return_code != OM_SUCCESS)
                        handle_om_error(return_code);


/*
 * the error handling code
 *
 * NOTE: any errors arising in these functions are ignored.
 */

void handle_ds_error(DS_status error)
{
  (void) fprintf(stderr, "DS error has occurred\n");

  (void) om_delete((OM_object) error);

  /* At this point, the error has been reported and storage cleaned up,
   * so the handler could return to the main program now for it to take
   * recovery action.  But we choose the simple option ...
   */

        exit(EXIT_FAILURE);
}


void handle_om_error(OM_return_code return_code)
{
  (void) fprintf(stderr, "OM error %d has occurred\n", return_code);
  /* At this point, the error has been reported and storage cleaned up,
   * so the handler could return to the main program now for it to take
   * recovery action.  But we choose the simple option ...
   */

        exit(EXIT_FAILURE);
}
```

28.3 The acl.c Program

The **acl.c** file is a program that displays the access permissions (ACLs) on each entry in the directory for a specific user. The permisions are presented in a form similar to UNIX file permissions. In addition, each entry is flagged as either a master or a shadow copy.

The distinguished name of the user requesting the access permissions is **/C=de/O=sni/OU=ap/CN=norbert**. The results of the request are presented in the following format:

[**ABCD**] *<entry's distinguished name>*

where:

A **m** master copy

s shadow copy

B **r** read access to public attributes

w write access to public attributes

- no access to public attributes

C **r** read access to standard attributes

w write access to standard attributes

- no access to standard attributes

D **r** read access to sensitive attributes

w write access to sensitive attributes

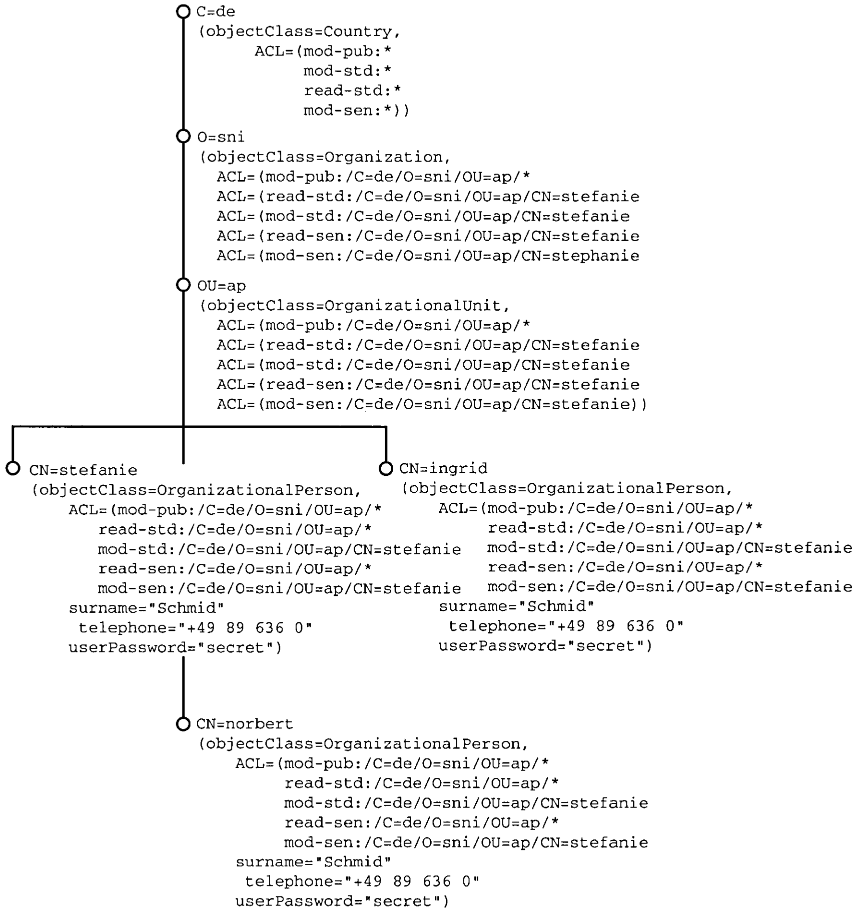
- no access to sensitive attributes

For example, the following result means that the entry **/C=de/O=sni** is a master copy and that the user making the request (**/C=de/O=sni/OU=ap/CN=norbert**) has write access to its public attributes, read access to its standard attributes, and no accesss to its sensitive attributes:

[mwr-] /C=de/O=sni

The program requires that the user perform an authenticated bind to the Directory Service. The user's credentials must already exist in the directory. For this reason, the tree of six entries shown in Figure 28-1 is added to the directory each time the program runs, and is removed again afterwards.

Figure 28–1. Entries with User Credentials Added to the Directory Tree


C=de
(objectClass=Country,
ACL=(mod-pub:*
mod-std:*
read-std:*
mod-sen:*))
O=sni
(objectClass=Organization,
ACL=(mod-pub:/C=de/O=sni/OU=ap/*
ACL=(read-std:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(mod-std:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(read-sen:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(mod-sen:/C=de/O=sni/OU=ap/CN=stephanie
OU=ap
(objectClass=OrganizationalUnit,
ACL=(mod-pub:/C=de/O=sni/OU=ap/*
ACL=(read-std:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(mod-std:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(read-sen:/C=de/O=sni/OU=ap/CN=stefanie
ACL=(mod-sen:/C=de/O=sni/OU=ap/CN=stefanie))
CN=stefanie
(objectClass=OrganizationalPerson,
ACL=(mod-pub:/C=de/O=sni/OU=ap/*
read-std:/C=de/O=sni/OU=ap/*
mod-std:/C=de/O=sni/OU=ap/CN=stefanie
read-sen:/C=de/O=sni/OU=ap/*
mod-sen:/C=de/O=sni/OU=ap/CN=stefanie
surname="Schmid"
telephone="+49 89 636 0"
userPassword="secret")
CN=ingrid
(objectClass=OrganizationalPerson,
ACL=(mod-pub:/C=de/O=sni/OU=ap/*
read-std:/C=de/O=sni/OU=ap/*
mod-std:/C=de/O=sni/OU=ap/CN=stefanie
read-sen:/C=de/O=sni/OU=ap/*
mod-sen:/C=de/O=sni/OU=ap/CN=stefanie
surname="Schmid"
telephone="+49 89 636 0"
userPassword="secret")
CN=norbert
(objectClass=OrganizationalPerson,
ACL=(mod-pub:/C=de/O=sni/OU=ap/*
read-std:/C=de/O=sni/OU=ap/*
mod-std:/C=de/O=sni/OU=ap/CN=stefanie
read-sen:/C=de/O=sni/OU=ap/*
mod-sen:/C=de/O=sni/OU=ap/CN=stefanie
surname="Schmid"
telephone="+49 89 636 0"
userPassword="secret")

The program consists of the following steps:

1. Export the required object identifiers (see the **acl.h** description in Section 28.3.2).
2. Build the descriptor lists for objects required by the program (see the **acl.h** description in Section 28.3.2).
3. Initialize a workspace.
4. Negotiate the use of the Basic Directory Contents and Global Directory Extension Packages.
5. Add a fixed tree of entries to the directory to permit an authenticated bind.
6. Create a default session object.
7. Alter the default session object to include the credentials of the requestor (**/C=de/O=sni/OU=ap/CN=norbert**).
8. Bind with credentials to the default GDS Server.
9. Create a default context object and alter it to include shadow entries.
10. Search the whole subtree below **root** and extract the ACL attribute from each selected entry.
11. Close the connection to the GDS Server.
12. Remove the user's credentials from the directory.
13. Extract the components from the search result.
14. Examine each entry and print the entry details.
15. Close the XDS workspace.

Note: The steps that follow are highlighted in boldface so that you can follow the sequence as you examine the **acl.c** program.

Step 1 through **Step 4**, **Step 6** through **Step 8**, **Step 12**, and **Step 15** are similar those performed for the previous sample application **example.c**.

Step 5 is included so that the appropriate entries will exist in the directory when the program attempts to access the access permissions.

The default session object created in **Step 9** uses **om_create**( ) to create an instance of a default session object and it uses **om_put**( ) to put in the appropriate user credentials (in this case, the descriptor list containing the password **secret**). The **credentials** parameter is a descriptor list defined in **acl.h** header file.

Step 10 used the same method as **Step 9** to alter the default context to include shadow entries. Using **om_create**( ) and **om_put**( ), the OM attribute **DS_DONT_USE_COPY** is set to a value of **OM_FALSE** to indicate that copies of entries maintained in other DSAs and copies cached locally (that is, shadow copies) can be used. The **use_copy** parameter is a descriptor list defined in **acl.h** header file.

Step 11 uses **ds_search**( ) to search the subtree below **root** to find and extract the ACL attributes from the selected entries defined in the **selection_acl** parameter. The **selection_acl** variable is a descriptor list defined in **acl.h**. The results of the search are returned to the workspace in **result**.

Step 13 and **Step 14** extract the components from **result** and examine each entry using a series of **om_get**( ) calls, as described in the previous section for **example.c**.

28.3.1 The acl.c Code

The following code is a listing of the **acl.c** program.

```
/****************************************************************
*                                                               *
*  COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG 1991    *
*                 ALL RIGHTS RESERVED                           *
*                                                               *
****************************************************************/


/*
 *
 * This sample program displays the access permissions (ACL) on each
 * entry in the directory for a specific user.  The permissions are
 * presented in a form similiar to the UNIX file permissions.
 * In addition, each entry is flagged as either a master
 * or a shadow copy.
 *
 * The distinguished name of the user performing the check is :
 *
 *      /C=de/O=sni/OU=ap/CN=norbert
 *
 * The results are presented in the following format :
 *
 *      [ABCD] <entry's distinguished name>
 *
 *      A:  'm' master copy
 *          's' shadow copy
 *
 *      B:  'r' read access to public attributes
 *          'w' write access to public attributes
 *          '-' no access to public attributes
 *
 *      C:  'r' read access to standard attributes
 *          'w' write access to standard attributes
 *          '-' no access to standard attributes
 *
 *      D:  'r' read access to sensitive attributes
 *          'w' write access to sensitive attributes
 *          '-' no access to sensitive attributes
 *
 *
 * For example, the following result means that the entry '/C=de/O=sni'
 * is a master copy and that the requesting user
 * (/C=de/O=sni/OU=ap/CN=norbert) has write access to its public
 * attributes, read access to its standard
 * attributes and no access to its sensitive attributes.
 *
 *       [mwr-] /C=de/O=sni
 *
 *
 *
 * The program requires that the specific user perform an authenticated
 * bind to the directory.  In order to achieve this the user's
 * credentials must already exist in the directory.
 * Therefore the following tree of 6 entries is added to the directory
 * each time the program runs, and removed again afterwards.
 *
```

```
*
*
*              O  C=de
*              |  (objectClass=Country,
*              |   ACL=(mod-pub: *
*              |        read-std:*
*              |        mod-std: *
*              |        read-sen:*
*              |        mod-sen: *))
*              |
*              |
*              O  O=sni
*              |  (objectClass=Organization,
*              |   ACL=(mod-pub: /C=de/O=sni/OU=ap/*
*              |        read-std:/C=de/O=sni/OU=ap/CN=stefanie
*              |        mod-std: /C=de/O=sni/OU=ap/CN=stefanie
*              |        read-sen:/C=de/O=sni/OU=ap/CN=stefanie
*              |        mod-sen: /C=de/O=sni/OU=ap/CN=stefanie))
*              |
*              O  OU=ap
*              |  (objectClass=OrganizationalUnit,
*              |   ACL=(mod-pub: /C=de/O=sni/OU=ap/*
*              |        read-std:/C=de/O=sni/OU=ap/CN=stefanie
*              |        mod-std: /C=de/O=sni/OU=ap/CN=stefanie
*              |        read-sen:/C=de/O=sni/OU=ap/CN=stefanie
*              |        mod-sen: /C=de/O=sni/OU=ap/CN=stefanie))
*              |
*      +-------+-------+
*      |       |       |
*      |       |       O  CN=ingrid
*      |       |         (objectClass=OrganizationalPerson,
*      |       |          ACL=(mod-pub: /C=de/O=sni/OU=ap/*
*      |       |               read-std:/C=de/O=sni/OU=ap/*
*      |       |               mod-std: /C=de/O=sni/OU=ap/CN=stefanie
*      |       |               read-sen:/C=de/O=sni/OU=ap/*
*      |       |               mod-sen: /C=de/O=sni/OU=ap/CN=stefanie),
*      |       |          surname="Schmid",
*      |       |          telephone="+49 89 636 0",
*      |       |          userPassword="secret")
*      |       |
*      |       O  CN=norbert
```

```
 *      |          (objectClass=OrganizationalPerson,
 *      |           ACL=(mod-pub: /C=de/O=sni/OU=ap/*
 *      |                read-std:/C=de/O=sni/OU=ap/*
 *      |                mod-std: /C=de/O=sni/OU=ap/CN=stefanie
 *      |                read-sen:/C=de/O=sni/OU=ap/*
 *      |                mod-sen: /C=de/O=sni/OU=ap/CN=stefanie),
 *      |           surname="Schmid",
 *      |           telephone="+49 89 636 0",
 *      |           userPassword="secret")
 *      |
 *      O  CN=stefanie
 *         (objectClass=OrganizationalPerson,
 *          ACL=(mod-pub: /C=de/O=sni/OU=ap/*
 *               read-std:/C=de/O=sni/OU=ap/*
 *               mod-std: /C=de/O=sni/OU=ap/CN=stefanie
 *               read-sen:/C=de/O=sni/OU=ap/*
 *               mod-sen: /C=de/O=sni/OU=ap/CN=stefanie),
 *          surname="Schmid",
 *          telephone="+49 89 636 0",
 *          userPassword="secret")
 *
 *
 */

#include <dce/xom.h>
#include <dce/xds.h>
#include <dce/xdsbdcp.h>
#include <dce/xdsgds.h>
#include <dce/xdscds.h>
#include "acl.h"       /* static initialization of data structures. */



void
main(
    int  argc,
    char *argv[]
)
{
```

```
OM_workspace       workspace;     /* workspace for objects          */
OM_private_object  session;       /* Session object.                */
OM_private_object  bound_session; /* Holds the Session object which */
                                  /* is returned by ds_bind()       */
OM_private_object   context;       /* Context object.                */
OM_private_object  result;        /* Holds the search result object.*/
OM_sint            invoke_id;     /* Integer for the invoke id      */
                                  /* returned by ds_search().       */
                                  /* (this parameter must be present*/
                                  /* even though it is ignored).    */
OM_type            sinfo_list[] = { DS_SEARCH_INFO, 0 };
OM_type            entry_list[] = { DS_ENTRIES, 0 };
                                  /* Lists of types to be extracted */
OM_public_object   sinfo;         /* Search-Info object from result.*/
OM_public_object   entry;         /* Entry object from search info. */
OM_value_position  total_num;     /* Number of descriptors returned.*/
OM_return_code     rc;            /* XOM function return code.      */
register int       i;
char               user_name[MAX_DN_LEN];
                                  /* Holds requestor’s name.        */
char               entry_string[MAX_DN_LEN + 7] = "[?r??] ";
                                  /* Holds entry details.           */

  /* Step 3 (see acl.h program code for Steps 1 and 2)
   *
   * Initialise a directory workspace for use by XOM.
   */
  if ((workspace = ds_initialize()) == (OM_workspace)0)
      printf("ds_initialize() error\n");


  /* Step 4
   *
   * Negotiate the use of the BDCP and GDS packages.
   */
  if (ds_version(features, workspace) != DS_SUCCESS)
      printf("ds_version() error\n");
```

```
/* Step 5
 *
 * Add a fixed tree of entries to the directory in order to permit
 * an authenticated bind by:  /C=de/O=sni/OU=ap/CN=norbert
 */
if (! add_tree(workspace))
    printf("add_tree() error\n");


/* Step 6
 *
 * Create a default session object.
 */
if ((rc = om_create(DSX_C_GDS_SESSION,OM_TRUE,workspace,&session))
                                                      != OM_SUCCESS)
    printf("om_create() error %d\n", rc);


/* Step 7
 *
 * Alter the default session object to include the following
 * credentials: requestor:  /C=de/O=sni/OU=ap/CN=norbert
 *  password:   "secret"
 */
if ((rc = om_put(session, OM_REPLACE_ALL, credentials, 0 ,0, 0))
                                                      != OM_SUCCESS)
    printf("om_put() error %d\n", rc);


/* Step 8
 *
 * Bind with credentials to the default GDS server.
 * The returned session object is stored in the private object
                                                                 variable
 * bound_session and is used for all further XDS function calls.
 */
if (ds_bind(session, workspace, &bound_session) != DS_SUCCESS)
    printf("ds_bind() error\n");
```

```
/* Step 9
 *
 * Create a default context object.
 */
if ((rc = om_create(DSX_C_GDS_CONTEXT,OM_TRUE,workspace,&context))
                                                  != OM_SUCCESS)
    printf("om_create() error %d\n", rc);


/*
 * Alter the default context object to include 'shadow' entries.
 */
if ((rc = om_put(context, OM_REPLACE_ALL, use_copy, 0 ,0, 0))
    != OM_SUCCESS)
    printf("om_put() error %d\n", rc);


/* Step 10
 *
 * Search the whole subtree below root.
 * The filter selects entries with an object-class attribute.
 * The selection extracts the ACL attribute from each selected entry.
 * The results are returned in the private object 'result'.
 *
 * NOTE: Since every entry contains an object-class attribute the
 *       filter performs no function other than to demonstrate how
 *       filters may be used.
 */
if (ds_search(bound_session, context, dn_root, DS_WHOLE_SUBTREE,
       filter, OM_FALSE, selection_acl, &result, &invoke_id)
       != DS_SUCCESS)
    printf("ds_search() error\n");


/* Step 11
 *
 * Close the connection to the GDS server.
 */
if (ds_unbind(bound_session) != DS_SUCCESS)
    printf("ds_unbind() error\n");
```

```
/* Step 12
 *
 * Remove the user's credentials from the directory.
 */
if (! remove_tree(workspace, session))
    printf("remove_tree() error\n");


/* Step 13
 *
 * Extract components from the search result by means of om_get().
 */
if ((rc = om_get(result,
          OM_EXCLUDE_ALL_BUT_THESE_TYPES + OM_EXCLUDE_SUBOBJECTS,
          sinfo_list, OM_FALSE, 0, 0, &sinfo, &total_num))
                                        != OM_SUCCESS)
    printf("om_get(Search-Result) error %d\n", rc);


if ((rc = om_get(sinfo->value.object.object,
          OM_EXCLUDE_ALL_BUT_THESE_TYPES + OM_EXCLUDE_SUBOBJECTS,
          entry_list, OM_FALSE, 0, 0, &entry, &total_num))
                                        != OM_SUCCESS)
     printf("om_get(Search-Info) error %d\n", rc);

/*
 * Convert the requestor's distinguished name to string format.
 */
if (! xds_name_to_string(dn_norbert, user_name))
    printf("xds_name_to_string() error\n");

printf("User:  %s\nTotal: %d\n", user_name, total_num);

/* Step 14
 *
 * Examine each entry and print the entry details.
 */
for (i = 0; i < total_num; i++) {
    if (process_entry_info((entry+i)->value.object.object,
                           entry_string, user_name))
        printf("%s\n", entry_string);
    }


    /* Step 15
     *
     * Close the directory workspace.
     */
    if (ds_shutdown(workspace) != DS_SUCCESS)
        printf("ds_shutdown() error\n");
}





/*
 * Add the tree of entries described above.
 */
int
add_tree(
    OM_workspace workspace
)
{
    OM_private_object session;     /* Holds the Session object which */
                                   /* is returned by ds_bind()       */
    OM_sint           invoke_id;   /* Integer for the invoke id      */
    int               error = 0;


/* Bind (without credentials) to the default GDS server.            */

    if (ds_bind(DS_DEFAULT_SESSION, workspace, &session) != DS_SUCCESS)
        error++;


/* Add entries to the GDS server.                                    */

    ds_add_entry(session, DS_DEFAULT_CONTEXT, dn_de, alist_C,
                 &invoke_id);

    if (ds_add_entry(session, DS_DEFAULT_CONTEXT, dn_sni, alist_O,
                                                    &invoke_id) != DS_SUCCESS)
        error++;

    if (ds_add_entry(session, DS_DEFAULT_CONTEXT, dn_ap, alist_OU,
                                                    &invoke_id) != DS_SUCCESS)
        error++;

    if (ds_add_entry(session, DS_DEFAULT_CONTEXT, dn_stefanie, alist_OP,
                                                    &invoke_id) != DS_SUCCESS)
        error++;

    if (ds_add_entry(session, DS_DEFAULT_CONTEXT, dn_norbert, alist_OP,
                                                    &invoke_id) != DS_SUCCESS)
        error++;

    if (ds_add_entry(session, DS_DEFAULT_CONTEXT, dn_ingrid, alist_OP,
                                                    &invoke_id) != DS_SUCCESS)
        error++;


/* Close the connection to the GDS server.                                   */

    if (ds_unbind(session) != DS_SUCCESS)
        error++;

    return (error?0:1);
}

/*
 * Remove the tree of entries described above.
 */
int
remove_tree(
    OM_workspace workspace,
    OM_private_object  session
)
{
    OM_private_object bound_session;  /* Holds Session object which */
                                      /* is returned by ds_bind()   */
    OM_sint           invoke_id;      /* Integer for the invoke id  */
    int               error = 0;
```

```
/* Bind (with credentials) to the default GDS server.                   */

    if (ds_bind(session, workspace, &bound_session) != DS_SUCCESS)
        error++;


/* Remove entries from the GDS server.                                  */

    if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT, dn_ingrid,
                                            &invoke_id) != DS_SUCCESS)
        error++;

    if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT, dn_stefanie,
                                            &invoke_id) != DS_SUCCESS)
        error++;

    if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT, dn_norbert,
                                            &invoke_id) != DS_SUCCESS)
        error++;

    if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT, dn_ap,
                                            &invoke_id) != DS_SUCCESS)
        error++;

    if (ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT, dn_sni,
                                            &invoke_id) != DS_SUCCESS)
        error++;

    ds_remove_entry(bound_session, DS_DEFAULT_CONTEXT, dn_de,
                        &invoke_id);


/* Close the connection to the GDS server.                              */

    if (ds_unbind(bound_session) != DS_SUCCESS)
        error++;


    return (error?0:1);
}
```

```
/*
 * Convert a distinguished name in XDS format (OM_descriptor lists) to
 * string format.
 */
int
xds_name_to_string(
    OM_public_object  name,           /* Xds distinguished name.    */
    char             *string          /* String distinguished name. */
)
{
    register OM_object  dn = name;
    register OM_object  rdn;
    register OM_object  ava;
    register char      *sp = string;
             int        error = 0;

    while ((dn->type != OM_NO_MORE_TYPES) && (! error)) {
        if ((dn->type == DS_RDNS) &&
            ((dn->syntax & OM_S_SYNTAX) == OM_S_OBJECT)) {
            rdn = dn->value.object.object;

            while ((rdn->type != OM_NO_MORE_TYPES) && (! error)) {
                if ((rdn->type == DS_AVAS) &&
                    ((rdn->syntax & OM_S_SYNTAX) == OM_S_OBJECT)) {
                    ava = rdn->value.object.object;

                    while ((ava->type != OM_NO_MORE_TYPES) &&
                             (! error)) {
                        if ((ava->type == DS_ATTRIBUTE_TYPE) &&
                            ((ava->syntax & OM_S_SYNTAX) ==
                                     OM_S_OBJECT_IDENTIFIER_STRING)) {

                            *sp++ = '/';
                            if (strncmp(ava->value.string.elements,
                                        DS_A_COUNTRY_NAME.elements,
                                        ava->value.string.length) == 0)
                                *sp++ = 'C';

                            else if (strncmp(ava->value.string.elements,
                                        DS_A_ORG_NAME.elements,
                                        ava->value.string.length) == 0)
        *sp++ = 'O';

    else if (strncmp(ava->value.string.elements,
                DS_A_ORG_UNIT_NAME.elements,
                ava->value.string.length) == 0)
        *sp++ = 'O', *sp++ = 'U';

    else if (strncmp(ava->value.string.elements,
                DS_A_COMMON_NAME.elements,
                ava->value.string.length) == 0)
        *sp++ = 'C', *sp++ = 'N';

    else if (strncmp(ava->value.string.elements,
                DS_A_LOCALITY_NAME.elements,
                ava->value.string.length) == 0)
        *sp++ = 'L';

    else if (strncmp(ava->value.string.elements,
             DSX_TYPELESS_RDN.elements,
             ava->value.string.length) != 0) {
        error++;
        continue;
    }

    if (*(sp-1) != '/'); /* no '=' if typeless*/
        *sp++ = '=';
}
if (ava->type == DS_ATTRIBUTE_VALUES) {
    switch(ava->syntax & OM_S_SYNTAX) {
        case OM_S_PRINTABLE_STRING :
        case OM_S_TELETEX_STRING :
         strncpy(sp, ava->value.string.elements,
                 ava->value.string.length);
         sp += ava->value.string.length;
         break;

        default:
         error++;
         continue;
    }
```

```
                            }
                            ava++;
                        }
                    }
                    rdn++;
                }
            }
            dn++;
        }
        *sp = ' ';

        return (error?0:1);
}




/*
 * Extract information about an entry from the Entry-Info object:
 * whether the entry is a master-copy, its ACL permissions and
 * its distinguished name.
 * Build up a string based on this information.
 */
int
process_entry_info(
    OM_private_object  entry,
    char              *entry_string,
    char              *user_name
)
{
    OM_return_code     rc;           /* Return code from XOM function. */
    OM_public_object   ei_attrs;     /* Components from Entry-Info.    */
    OM_public_object   attr;         /* Directory attribute.           */
    OM_public_object   acl;          /* ACL attribute value.           */
    OM_public_object   acl_item;     /* ACL item component.            */
    OM_value_position  total_attrs;  /* Number of attributes returned. */
    register int       i;
    register int       interp;
    register int       error = 0;
    register int       found_acl = 0;
```

```
static OM_type     ei_attr_list[] = { DS_FROM_ENTRY,
                                      DS_OBJECT_NAME,
                                      DS_ATTRIBUTES,
                                      0 };
                                    /* Attributes to be extracted.  */


/*
 * Extract three attributes from each Entry-Info object.
 */
if ((rc = om_get(entry, OM_EXCLUDE_ALL_BUT_THESE_TYPES,
                  ei_attr_list, OM_FALSE, 0, 0, &ei_attrs,
                    &total_attrs))
                                                != OM_SUCCESS) {
    error++;
    printf("om_get(Entry-Info) error %d\n", rc);
}


for (i = 0; ((i < total_attrs) && (! error)); i++, ei_attrs++) {

    /*
     * Determine if current entry is a master-copy or shadow-copy.
     */
    if ((ei_attrs->type == DS_FROM_ENTRY) &&
        ((ei_attrs->syntax & OM_S_SYNTAX) == OM_S_BOOLEAN))
        if (ei_attrs->value.boolean == OM_TRUE)
            entry_string[1] = 'm';
        else if (ei_attrs->value.boolean == OM_FALSE)
                entry_string[1] = 's';
             else
                entry_string[1] = '?';


    if ((ei_attrs->type == DS_ATTRIBUTES) &&
        ((ei_attrs->syntax & OM_S_SYNTAX) == OM_S_OBJECT)) {
        attr = ei_attrs->value.object.object;

        while ((attr->type != OM_NO_MORE_TYPES) && (! error)) {

            /*
 * Check that the attribute is an ACL attribute.
 */
if ((attr->type == DS_ATTRIBUTE_TYPE) &&
    ((attr->syntax & OM_S_SYNTAX) ==
                    OM_S_OBJECT_IDENTIFIER_STRING)) {
    if (strncmp(attr->value.string.elements,
                DSX_A_ACL.elements,
                attr->value.string.length) == 0)
        found_acl++;
}

/*
 * Examine the ACL. Check each permission for
 * the current user.
 */
if ((found_acl) &&
    (attr->type == DS_ATTRIBUTE_VALUES) &&
    ((attr->syntax & OM_S_SYNTAX) == OM_S_OBJECT)) {

    acl = attr->value.object.object;

    entry_string[2] = 'r';
    entry_string[3] = '-';
    entry_string[4] = '-';

    while (acl->type != OM_NO_MORE_TYPES) {

        if ((acl->syntax & OM_S_SYNTAX) == OM_S_OBJECT)
            acl_item = acl->value.object.object;

        switch (acl->type) {

        case OM_CLASS:
            break;

        case DSX_MODIFY_PUBLIC:
            if (permitted_access(user_name, acl_item))
                entry_string[2] = 'w';
            break;

        case DSX_READ_STANDARD:
```

```
                            if (permitted_access(user_name, acl_item))
                                entry_string[3] = 'r';
                            break;

                        case DSX_MODIFY_STANDARD:
                            if (permitted_access(user_name, acl_item))
                                entry_string[3] = 'w';
                            break;

                        case DSX_READ_SENSITIVE:
                            if (permitted_access(user_name, acl_item))
                                entry_string[4] = 'r';
                            break;

                        case DSX_MODIFY_SENSITIVE:
                            if (permitted_access(user_name, acl_item))
                                entry_string[4] = 'w';
                            break;
                        }
                        acl++;
                    }
                }
                attr++;
            }
        }


        /*
         * Convert the entry's distinguished name to a string format.
         */
        if ((ei_attrs->type == DS_OBJECT_NAME) &&
            ((ei_attrs->syntax & OM_S_SYNTAX) == OM_S_OBJECT))
            if (! xds_name_to_string(ei_attrs->value.object.object,
                                     &entry_string[7])) {
                error++;
                printf("xds_name_to_string() error\n");
            }

    }

    return (error?0:1);
```

```
}

/*
 * Check if a user is permitted access based on the ACL supplied.
 *
 */
int
permitted_access(
    char             *user_name,
    OM_public_object  acl_item
)
{
    char  acl_name[MAX_DN_LEN];
    int   interpretation;
    int   acl_present = 0;
    int   access = 0;
    int   acl_name_length;


    while (acl_item->type != OM_NO_MORE_TYPES) {

        switch (acl_item->type) {
        case OM_CLASS:
            break;

        case DSX_INTERPRETATION:
            interpretation = acl_item->value.boolean;
            break;

        case DSX_USER:
            xds_name_to_string(acl_item->value.object.object, acl_name);

            if (interpretation == DSX_SINGLE_OBJECT) {
                if (strcmp(acl_name, user_name) == 0)
                    access = 1;
            }
            else if (interpretation == DSX_ROOT_OF_SUBTREE) {
                     if ((acl_name_length = strlen(acl_name)) == 0)
                         access = 1;
                     else if (strncmp(acl_name,user_name,
                                  acl_name_length) == 0)
```

```
                              access = 1;
                    }
                    break;
                }
                acl_item++;
            }

            return (access);
        }
```

28.3.2 The acl.h Header File

The **acl.h** header file peforms the following:

1. It exports the object identifiers that **acl.c** requires.
2. It builds the descriptor lists for the following distinguished names:

   root
   C=de
   C=de/O=sni
   C=de/O=sni/OU=ap
   C=de/O=sni/OU=ap/CN=stefanie
   C=de/O=sni/OU=ap/CN=norbert
   C=de/O=sni/OU=ap/CN=ingrid

3. It builds the object identifiers for attributes to be added to the directory.
4. It builds a descriptor list for the attribute types and values that are to be selected.
5. It builds the descriptor list for bind credentials.
6. It builds the descriptor list for context.
7. It builds the descriptor list for optional packages that are to be negotiated.
8. It builds the descriptor list for search filters.

28.3.3 The acl.h Code

The following code is a listing of the **acl.h** file:

```
/***************************************************************
*                                                              *
*  COPYRIGHT (C) SIEMENS NIXDORF INFORMATIONSSYSTEME AG 1991   *
*                 ALL RIGHTS RESERVED                          *
*                                                              *
***************************************************************/

#ifndef ACL_HEADER
#define ACL_HEADER

#define MAX_DN_LEN 100
/* max length of a distinguished name in string format */


/* The application must export the object identfiers it requires.     */

OM_EXPORT (DS_C_AVA)
OM_EXPORT (DS_C_DS_RDN)
OM_EXPORT (DS_C_DS_DN)
OM_EXPORT (DS_C_ENTRY_INFO_SELECTION)
OM_EXPORT (DS_C_ATTRIBUTE)
OM_EXPORT (DS_C_ATTRIBUTE_LIST)
OM_EXPORT (DS_C_FILTER)
OM_EXPORT (DS_C_FILTER_ITEM)
OM_EXPORT (DSX_C_GDS_SESSION)
OM_EXPORT (DSX_C_GDS_CONTEXT)
OM_EXPORT (DSX_C_GDS_ACL)
OM_EXPORT (DSX_C_GDS_ACL_ITEM)

OM_EXPORT (DS_A_COUNTRY_NAME)
OM_EXPORT (DS_A_ORG_NAME)
OM_EXPORT (DS_A_ORG_UNIT_NAME)
OM_EXPORT (DS_A_COMMON_NAME)
OM_EXPORT (DS_A_LOCALITY_NAME)
OM_EXPORT (DS_A_OBJECT_CLASS)
OM_EXPORT (DS_A_USER_PASSWORD)
OM_EXPORT (DS_A_PHONE_NBR)
OM_EXPORT (DS_A_SURNAME)
OM_EXPORT (DSX_A_ACL)
OM_EXPORT (DSX_TYPELESS_RDN)

OM_EXPORT (DS_O_TOP)
OM_EXPORT (DS_O_COUNTRY)
OM_EXPORT (DS_O_ORG)
OM_EXPORT (DS_O_ORG_UNIT)
OM_EXPORT (DS_O_PERSON)
OM_EXPORT (DS_O_ORG_PERSON)


/* Build up descriptor lists for the following distinguished names: */
/*      root                                                         */
/*      /C=de                                                        */
/*      /C=de/O=sni                                                  */
/*      /C=de/O=sni/OU=ap                                            */
/*      /C=de/O=sni/OU=ap/CN=stefanie                                */
/*      /C=de/O=sni/OU=ap/CN=norbert                                 */
/*      /C=de/O=sni/OU=ap/CN=ingrid                                  */

static OM_descriptor    ava_de[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COUNTRY_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING, OM_STRING("de")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    ava_sni[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("sni")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    ava_ap[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_ORG_UNIT_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("ap")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    ava_stefanie[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("stefanie")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    ava_norbert[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("norbert")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    ava_ingrid[] = {
    OM_OID_DESC(OM_CLASS, DS_C_AVA),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_COMMON_NAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("ingrid")},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    rdn_de[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_de}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    rdn_sni[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_sni}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    rdn_ap[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_ap}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    rdn_stefanie[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_stefanie}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    rdn_norbert[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_norbert}},
    OM_NULL_DESCRIPTOR
};
```

```
static OM_descriptor    rdn_ingrid[] = {
    OM_OID_DESC(OM_CLASS, DS_C_DS_RDN),
    {DS_AVAS, OM_S_OBJECT, {0, ava_ingrid}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_root[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_de[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    {DS_RDNS,OM_S_OBJECT,{0,rdn_de}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_sni[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    {DS_RDNS,OM_S_OBJECT,{0,rdn_de}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_ap[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    {DS_RDNS,OM_S_OBJECT,{0,rdn_de}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ap}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_stefanie[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    {DS_RDNS,OM_S_OBJECT,{0,rdn_de}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ap}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_stefanie}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_norbert[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    {DS_RDNS,OM_S_OBJECT,{0,rdn_de}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ap}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_norbert}},
    OM_NULL_DESCRIPTOR
};
static OM_descriptor dn_ingrid[] = {
    OM_OID_DESC(OM_CLASS,DS_C_DS_DN),
    {DS_RDNS,OM_S_OBJECT,{0,rdn_de}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_sni}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ap}},
    {DS_RDNS,OM_S_OBJECT,{0,rdn_ingrid}},
    OM_NULL_DESCRIPTOR
};


/* Build up an array of object identifiers for the attributes to be */
/* added to the directory.                                          */

static OM_descriptor    obj_class_C[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_TOP),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_COUNTRY),
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    obj_class_O[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_TOP),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_ORG),
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    obj_class_OU[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_TOP),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_ORG_UNIT),
    OM_NULL_DESCRIPTOR
};
static OM_descriptor    obj_class_OP[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_TOP),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_PERSON),
    OM_OID_DESC(DS_ATTRIBUTE_VALUES, DS_O_ORG_PERSON),
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    att_phone_num[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_PHONE_NBR),
    {DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING,
      OM_STRING("+49 89 636 0")},
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    att_password[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_USER_PASSWORD),
    {DS_ATTRIBUTE_VALUES, OM_S_OCTET_STRING, OM_STRING("secret")},
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    att_surname[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_SURNAME),
    {DS_ATTRIBUTE_VALUES, OM_S_TELETEX_STRING, OM_STRING("Schmid")},
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    acl_item_root[] = {
    OM_OID_DESC(OM_CLASS, DSX_C_GDS_ACL_ITEM),
    {DSX_INTERPRETATION, OM_S_ENUMERATION, {DSX_ROOT_OF_SUBTREE, 0}},
    {DSX_USER, OM_S_OBJECT, {0, dn_root}},
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    acl_item_ap[] = {
    OM_OID_DESC(OM_CLASS, DSX_C_GDS_ACL_ITEM),
    {DSX_INTERPRETATION, OM_S_ENUMERATION, {DSX_ROOT_OF_SUBTREE, 0}},
    {DSX_USER, OM_S_OBJECT, {0, dn_ap}},
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    acl_item_stefanie[] = {
    OM_OID_DESC(OM_CLASS, DSX_C_GDS_ACL_ITEM),
    {DSX_INTERPRETATION, OM_S_ENUMERATION, {DSX_SINGLE_OBJECT, 0}},
    {DSX_USER, OM_S_OBJECT, {0, dn_stefanie}},
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    acl1[] = {
    OM_OID_DESC(OM_CLASS, DSX_C_GDS_ACL),
    {DSX_MODIFY_PUBLIC, OM_S_OBJECT, {0, acl_item_root}},
    {DSX_READ_STANDARD, OM_S_OBJECT, {0, acl_item_stefanie}},
    {DSX_MODIFY_STANDARD, OM_S_OBJECT, {0, acl_item_stefanie}},
    {DSX_READ_SENSITIVE, OM_S_OBJECT, {0, acl_item_stefanie}},
    {DSX_MODIFY_SENSITIVE, OM_S_OBJECT, {0, acl_item_stefanie}},
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    acl2[] = {
    OM_OID_DESC(OM_CLASS, DSX_C_GDS_ACL),
    {DSX_MODIFY_PUBLIC, OM_S_OBJECT, {0, acl_item_ap}},
    {DSX_READ_STANDARD, OM_S_OBJECT, {0, acl_item_ap}},
    {DSX_MODIFY_STANDARD, OM_S_OBJECT, {0, acl_item_stefanie}},
    {DSX_READ_SENSITIVE, OM_S_OBJECT, {0, acl_item_ap}},
    {DSX_MODIFY_SENSITIVE, OM_S_OBJECT, {0, acl_item_stefanie}},
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    att_acl1[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_ACL),
    {DS_ATTRIBUTE_VALUES, OM_S_OBJECT, {0, acl1} },
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    att_acl2[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE),
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DSX_A_ACL),
    {DS_ATTRIBUTE_VALUES, OM_S_OBJECT, {0, acl2} },
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    alist_C[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
    {DS_ATTRIBUTES, OM_S_OBJECT, {0, obj_class_C} },
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    alist_O[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
    {DS_ATTRIBUTES, OM_S_OBJECT, {0, obj_class_O} },
    {DS_ATTRIBUTES, OM_S_OBJECT, {0, att_acl1} },
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    alist_OU[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
    {DS_ATTRIBUTES, OM_S_OBJECT, {0, obj_class_OU} },
    OM_NULL_DESCRIPTOR
};

static OM_descriptor    alist_OP[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ATTRIBUTE_LIST),
    {DS_ATTRIBUTES, OM_S_OBJECT, {0, obj_class_OP} },
    {DS_ATTRIBUTES, OM_S_OBJECT, {0, att_acl2} },
    {DS_ATTRIBUTES, OM_S_OBJECT, {0, att_surname} },
    {DS_ATTRIBUTES, OM_S_OBJECT, {0, att_phone_num} },
    {DS_ATTRIBUTES, OM_S_OBJECT, {0, att_password} },
    OM_NULL_DESCRIPTOR
};


/* The following descriptor list specifies what to return from the*/
/* entry. The ACL attribute's types and values are selected.      */

static OM_descriptor selection_acl[] = {
    OM_OID_DESC(OM_CLASS, DS_C_ENTRY_INFO_SELECTION),
    {DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_FALSE},
    OM_OID_DESC(DS_ATTRIBUTES_SELECTED, DSX_A_ACL),
    {DS_INFO_TYPE, OM_S_ENUMERATION, DS_TYPES_AND_VALUES},
    OM_NULL_DESCRIPTOR
};

/* The following descriptor list specifies the bind credentials   */
```

```
static OM_descriptor credentials[] = {
    {DS_REQUESTOR, OM_S_OBJECT, {0, dn_norbert} },
    {DSX_PASSWORD, OM_S_OCTET_STRING, OM_STRING("secret")},
    OM_NULL_DESCRIPTOR
};

/* The following descriptor list specifies part of the context    */

static OM_descriptor use_copy[] = {
    {DS_DONT_USE_COPY, OM_S_BOOLEAN, {OM_FALSE, 0} },
    OM_NULL_DESCRIPTOR
};

/* Build up an array of object identifiers for the optional       */
/* packages to be negotiated.                                      */

DS_feature features[] = {
    { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
    { OM_STRING(OMP_O_DSX_GDS_PKG), OM_TRUE },
    { 0 }
};

/* The following descriptor list specifies a filter for search :  */
/*     (Present: objectClass)                                      */

static OM_descriptor filter_item[] = {
    OM_OID_DESC(OM_CLASS, DS_C_FILTER_ITEM),
    {DS_FILTER_ITEM_TYPE, OM_S_ENUMERATION, {DS_PRESENT, 0} },
    OM_OID_DESC(DS_ATTRIBUTE_TYPE, DS_A_OBJECT_CLASS),
    OM_NULL_DESCRIPTOR
};

static OM_descriptor filter[] = {
    OM_OID_DESC(OM_CLASS, DS_C_FILTER),
    {DS_FILTER_ITEMS, OM_S_OBJECT, {0, filter_item} },
    {DS_FILTER_TYPE, OM_S_ENUMERATION, {DS_AND, 0} },
    OM_NULL_DESCRIPTOR
};


#endif  /* ACL_HEADER */
```

28.4 The teldir.c Program

The sample program **teldir.c** permits a user to add, read, or delete entries in a CDS or GDS namespace in any local or remote DCE cell, assuming that permissions are granted by the ACLs. The entry consists of a person's surname and phone number. Each entry is of class Organizational Person.

The program uses predefined static XDS public objects that are never altered and partially defined static XDS public objects so that values for the surname and phone number can be entered dynamically by a user. It also uses dynamic XDS public objects that are created and filled only as needed using the **stringToXdsName** function. These techniques are a departure from the ones used in the first two sample programs where all objects are predefined.

28.4.1 Predefined Static Public Objects

The predefined static object classes and attributes are shown in the following code fragment:

```
/*
 * To hold the attributes we want to attach to the name being added.
 * One attribute is the class of the object (DS_O_ORG_PERSON), the
 * rest of the attributes are the surname (required for all objects
 * of class DS_O_ORG_PERSON) and phone number.  In addition, we need
 * an object to hold all this information to pass it into ds_add_entry().
 */
static OM_descriptor xdsObjectClass[] = {

    /* This object is an attribute--an object class. */
    OM_OID_DESC( OM_CLASS,             DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE,    DS_A_OBJECT_CLASS ),

    /* Not only must the class be listed, but also all */
    /* its superclasses.                               */
    OM_OID_DESC( DS_ATTRIBUTE_VALUES, DS_O_TOP ),
    OM_OID_DESC( DS_ATTRIBUTE_VALUES, DS_O_PERSON ),
    OM_OID_DESC( DS_ATTRIBUTE_VALUES, DS_O_ORG_PERSON ),
    /* Null terminator */
    OM_NULL_DESCRIPTOR
};
static OM_descriptor xdsAttributesToAdd[] = {

    /* This object is an attribute list. */
    OM_OID_DESC( OM_CLASS, DS_C_ATTRIBUTE_LIST ),

    /* These are "pointers" to the attributes in the list. */
    { DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsObjectClass } },
    { DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsSurname } },
    { DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsPhoneNum } },

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

/*
 * To hold the list of attributes we want to read.
 */
static OM_descriptor xdsAttributeSelection[] = {

    /* This is an entry information selection. */
    OM_OID_DESC( OM_CLASS, DS_C_ENTRY_INFO_SELECTION ),

    /* No, we don't want all attributes. */
    { DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_FALSE },

    /* These are the ones we want to read. */
    OM_OID_DESC( DS_ATTRIBUTES_SELECTED, DS_A_SURNAME ),
    OM_OID_DESC( DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR ),

    /* Give us both the types and their values. */
    { DS_INFO_TYPE, OM_S_ENUMERATION, { DS_TYPES_AND_VALUES, NULL } },

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};
```

28.4.2 Partially Defined Static Public Objects

The program partially defines static XDS objects with placeholders so that values for the surname and telephone number entered by the user can be added later, as shown in the following code fragment:

```
static OM_descriptor xdsSurname[] = {

    /* This object is an attribute--a surname. */
    OM_OID_DESC( OM_CLASS,           DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE, DS_A_SURNAME ),

    /* No default--so we need a placeholder for the actual surname. */
    OM_NULL_DESCRIPTOR,

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

static OM_descriptor xdsPhoneNum[] = {

    /* This object is an attribute--a telephone number. */
    OM_OID_DESC( OM_CLASS,           DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE, DS_A_PHONE_NBR ),

    /* By default, phone numbers are unlisted.  If the user specifies */
    /* an actual phone number, it will go into this position.         */
    { DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING,
       OM_STRING( "unlisted" ) },

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};
```

The program prompts the user for the surname of the person whose number will be changed and uses the **FILL_OMD_STRING** macro to fill in values, as shown in the following code fragment:

```
    if ( operation == 'a' ) {
        /* add operation requires additional input */
        /*
        * Get the person's real name from the user and place it in the
        * XDS object already defined at the
        * top of the program (xdsSurname).
        * We are requiring a name, so we will loop until we get one.
        */
       do {
           printf( "What is this person's surname? " );
           gets( newSurname );
       } while ( *newSurname == ' ' );
       FILL_OMD_STRING( xdsSurname, 2, DS_ATTRIBUTE_VALUES,
                        OM_S_TELETEX_STRING, newSurname )
```

28.4.3 Dynamically Defined Public Objects

The program uses the function **stringToXdsName** to convert the DCE name entered by a user into an XDS name object of OM class **DS_C_DS_DN**, which is the representation of a distinguished name. In the other two sample programs, arrays of descriptor lists are statically declared to represent the AVAs and RDNs that make up the public object that represents a distinguished name. The function **stringToXdsName** parses the DCE name and dynamically converts it to a public object.

For example, the following code fragment shows how space for a **DS_C_AVA** object is allocated and its entries are filled using the **FILL_OMD_XOM_STRING** and **FILL_OMD_NULL** macros:

```
/*
 * Allocate space for a DS_C_AVA object and fill in its entries:
 *      DS_C_AVA class identifier
 *      AVA's type
 *      AVA's value
 *      null terminator
 */
ava = (OM_descriptor *)malloc( sizeof(OM_descriptor) * 4 );
if( ava == NULL )                          /* malloc() failed */
    return OM_MEMORY_INSUFFICIENT;
FILL_OMD_XOM_STRING( ava, 0, OM_CLASS, OM_S_OBJECT_IDENTIFIER_STRING,
                     DS_C_AVA )
splitNamePiece( start, &type, &value );
```

```
FILL_OMD_XOM_STRING( ava, 1, DS_ATTRIBUTE_TYPE,
                     OM_S_OBJECT_IDENTIFIER_STRING, type )
FILL_OMD_STRING( ava, 2, DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING,
                 value )
FILL_OMD_NULL( ava, 3 )
```

The program uses the same method to build the RDNs that make up the distinguished name. The distinguished name is **NULL** terminated using the **FILL_OMD_NULL** macro and the location of the new public object is provided for the calling routine (main) in the pointer **xdsNameObj**, as shown in the following code fragment:

```
        /* Add the DS_C_RDN object to the DS_C_DS_DN object. */
        FILL_OMD_STRUCT( dsdn, index, DS_RDNS, OM_S_OBJECT, rdn )
    }

    /*
     * Null terminate the DS_C_DS_DN, tell the calling routine
     * where to find it, and return.
     */
    FILL_OMD_NULL( dsdn, index )
    *xdsNameObj = dsdn;
    return( OM_SUCCESS );

} /* end stringToXdsName() */
```

28.4.4 Main Program Procedural Steps

The program consists of the following general steps:

1. Examine the command-line argument to determine the type of operation (read, add, or delete entry) that the user wants to perform.
2. Initialize a workspace.
3. Pull in the packages with the required XDS features.
4. Prompt the user for the name entry on which the operation will be performed.

5. Convert the DCE-formatted user input string to an XDS object name.
6. Bind (without credentials) to the default server.
7. Perform the requested operation (read, add, or delete entry).
8. Perform error handling.
9. Unbind from the server.
10. Shut down the workspace, releasing resources back to the system.

Note: The steps that follow are highlighted in boldface so that you can follow the sequence as you examine the **example.c** program.

Step 1 simply involves determining which of the three options: **r** (read), **a** (add), or **d** (delete) the user has entered. **Step 2** initializes a workspace, an operation required by XDS API for every application program. **Step 3** is required because additional features not present in the Directory Service Package need to be used by the application program. An additional package, the Basic Directory Contents Package, is defined in **featureList** as a static XDS object earlier in the program.

In **Step 4**, the user is prompted for the DCE-formatted name, which is the distinguished name of the person on whose telephone number the operation is to be performed. The name must be a fully or partially qualified name that begins with either the **/...** or **/.:** prefix. An example of a fully qualified, or global, name is **/.../C=de/O=sni/OU=ap/CN=klaus**. An example of a partially qualified, or cell, name is **/.:/brad/sni/com**. Additional information is requested in **Step 5** if the user requests an add operation.

Step 5 converts the DCE-formatted name to an XDS object name (public object) using the **stringToXdsName**( ) function call. This function builds an XDS public object that represents the distinguished name entered by the user.

Step 6 binds the session to the default server without credentials; username and password are not required.

In **Step 7**, the requested operation is performed using XDS API functions calls. For an add operation, **ds_add_entry**( ) is performed; for a read operation, **ds_read**( ) is performed; and for a delete operation, **ds_remove_entry**( ) is performed. The read operation requires a series of XOM API **om_get**( ) function calls to extract the surname and phone number from the workspace. (For a detailed description of the XDS and XOM API function calls, refer to Chapters 26 and 27.)

Step 8 and **Step 9** are required for every XDS API application program in order to clean up before the program exits. The session is unbound from the server, and the public and private objects are released to the system that provided the memory allocated for them.

28.4.5 The teldir.c Code

The following is a listing of the file **teldir.c**:

```
/*
 * This sample program behaves like a simple telephone directory.
 * It permits a user to add, read or delete entries in a GDS
 * namespace or to a CDS namespace in any local or remote DCE cell
 * (assuming that permissions are granted by the ACLs).
 *
 * Each entry is of class Organizational-Person and simply contains
 * a person's surname and their phone number.
 *
 * The addition of an entry is followed by a read to verify that the
 * information was entered properly.
 *
 * All valid names should begin with one of the following symbols:
 *     /...            Fully qualified name (from global root).
 *                     e.g.  /.../C=de/O=sni/OU=ap/CN=klaus
 *
 *     /.:             Partially qualified name (from local cell root).
 *                     e.g.  /.:/brad/sni/com
 *
 * This program demonstrates the following techniques:
 * - Using completely static XDS public objects (predefined at the top
 *   of the program and never altered).  See xdsObjectClass,
 *   xdsAttributesToAdd, and xdsAttributeSelection below.
 * - Using partially static XDS public objects (predefined at the top
 *   of the program but altered later).  See xdsSurname and xdsPhoneNum
 *   below.  See also the macros whose names begin with "FILL_OMD_".
 * - Using dynamic XDS public objects (created and filled in only as
 *   needed).  See the function stringToXdsName() below.
 * - Parsing DCE-style names and converting them into XDS objects.  See
 *   the function stringToXdsName() below.
 * - Getting the value of an attribute from an object read from the
 *   namespace using ds_read().  See the function extractValue() below.
 * - Getting the numeric value of an error (type DS_status) returned by
 *   one of the XDS calls.  See the function handleDSError() below.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <xom.h>
#include <xds.h>
#include <xdsbdcp.h>
#include <xdsgds.h>
#include <xdscds.h>

OM_EXPORT( DS_A_COMMON_NAME )
OM_EXPORT( DS_A_COUNTRY_NAME )
OM_EXPORT( DS_A_LOCALITY_NAME )
OM_EXPORT( DS_A_OBJECT_CLASS )
OM_EXPORT( DS_A_ORG_UNIT_NAME )
OM_EXPORT( DS_A_ORG_NAME )
OM_EXPORT( DS_A_SURNAME )
OM_EXPORT( DS_A_PHONE_NBR )
OM_EXPORT( DS_A_TITLE )
OM_EXPORT( DS_C_ATTRIBUTE )
OM_EXPORT( DS_C_ATTRIBUTE_LIST )
OM_EXPORT( DS_C_AVA )
OM_EXPORT( DS_C_DS_DN )
OM_EXPORT( DS_C_DS_RDN )
OM_EXPORT( DS_C_ENTRY_INFO_SELECTION )
OM_EXPORT( DS_O_ORG_PERSON )
OM_EXPORT( DS_O_PERSON )
OM_EXPORT( DS_O_TOP )
OM_EXPORT( DSX_TYPELESS_RDN ) /* For "typeless" pieces of a name, as */
                              /* found in cells with bind-style names*/
                              /* and in the CDS namespace.          */

#define MAX_NAME_LEN    1024

/* These values can be found in                                     */
/* the "Directory Class Definitions" chapter.                        */
/* (One byte must be added for the null terminator.)                 */
#define MAX_PHONE_LEN   33
#define MAX_SURNAME_LEN 66

/*********************************************************************
 * Macros for help filling in static XDS objects.
 *********************************************************************/
/* Put NULL value (equivalent to OM_NULL_DESCRIPTOR) in object */
#define FILL_OMD_NULL( desc, index )
        desc[index].type = OM_NO_MORE_TYPES;
        desc[index].syntax = OM_S_NO_MORE_SYNTXES;
        desc[index].value.object.padding = 0;
        desc[index].value.object.object = OM_ELEMENTS_UNSPECIFIED;

/* Put C-style (null-terminated) string in object */
#define FILL_OMD_STRING( desc, index, typ, syntx, val )
        desc[index].type = typ;
        desc[index].syntax = syntx;
        desc[index].value.string.length = (OM_element_position)
            strlen( val );
        desc[index].value.string.elements = val;

/* Put XOM string in object */
#define FILL_OMD_XOM_STRING( desc, index, typ, syntx, val )
        desc[index].type = typ;
        desc[index].syntax = syntx;
        desc[index].value.string.length = val.length;
        desc[index].value.string.elements = val.elements;

/* Put other value in object */
#define FILL_OMD_STRUCT( desc, index, typ, syntx, val )
        desc[index].type = typ;
        desc[index].syntax = syntx;
        desc[index].value.object.padding = 0;
        desc[index].value.object.object = val;


/*********************************************************************
 * Static XDS objects.
 ********************************************************************/
```

```
/*
 * To identify which packages we need for this program.  We only need
 * the basic package because we are not doing anything fancy with
 * session parameters, etc.
 */
DS_feature featureList[] = {
        { OM_STRING(OMP_O_DS_BASIC_DIR_CONTENTS_PKG), OM_TRUE },
        { 0 }
};

/*
 * To hold the attributes we want to attach to the name being added.
 * One attribute is the class of the object (DS_O_ORG_PERSON), the
 * rest of the attributes are the surname (required for all objects
 * of class DS_O_ORG_PERSON) and phone number.  In addition, we need
 * an object to hold all this information to pass it
 * into ds_add_entry().
 */
static OM_descriptor xdsObjectClass[] = {

    /* This object is an attribute--an object class. */
    OM_OID_DESC( OM_CLASS,            DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE,   DS_A_OBJECT_CLASS ),

    /* Not only must the class be listed, but also all */
    /* its superclasses.                               */
    OM_OID_DESC( DS_ATTRIBUTE_VALUES, DS_O_TOP ),
    OM_OID_DESC( DS_ATTRIBUTE_VALUES, DS_O_PERSON ),
    OM_OID_DESC( DS_ATTRIBUTE_VALUES, DS_O_ORG_PERSON ),

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

static OM_descriptor xdsSurname[] = {

    /* This object is an attribute--a surname. */
    OM_OID_DESC( OM_CLASS,          DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE, DS_A_SURNAME ),

    /* No default--so we need a placeholder for the actual surname. */
    OM_NULL_DESCRIPTOR,

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

static OM_descriptor xdsPhoneNum[] = {

    /* This object is an attribute--a telephone number. */
    OM_OID_DESC( OM_CLASS,           DS_C_ATTRIBUTE ),
    OM_OID_DESC( DS_ATTRIBUTE_TYPE, DS_A_PHONE_NBR ),

    /* By default, phone numbers are unlisted.  If the user specifies */
    /* an actual phone number, it will go into this position.         */
    { DS_ATTRIBUTE_VALUES, OM_S_PRINTABLE_STRING,
      OM_STRING( "unlisted" ) },

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

static OM_descriptor xdsAttributesToAdd[] = {

    /* This object is an attribute list. */
    OM_OID_DESC( OM_CLASS, DS_C_ATTRIBUTE_LIST ),

    /* These are "pointers" to the attributes in the list. */
    { DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsObjectClass } },
    { DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsSurname } },
    { DS_ATTRIBUTES, OM_S_OBJECT, { 0, xdsPhoneNum } },

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};

/*
 * To hold the list of attributes we want to read.
 */
static OM_descriptor xdsAttributeSelection[] = {

    /* This is an entry information selection. */
    OM_OID_DESC( OM_CLASS, DS_C_ENTRY_INFO_SELECTION ),

    /* No, we don't want all attributes. */
    { DS_ALL_ATTRIBUTES, OM_S_BOOLEAN, OM_FALSE },

    /* These are the ones we want to read. */
    OM_OID_DESC( DS_ATTRIBUTES_SELECTED, DS_A_SURNAME ),
    OM_OID_DESC( DS_ATTRIBUTES_SELECTED, DS_A_PHONE_NBR ),

    /* Give us both the types and their values. */
    { DS_INFO_TYPE, OM_S_ENUMERATION, { DS_TYPES_AND_VALUES, NULL } },

    /* Null terminator */
    OM_NULL_DESCRIPTOR
};


/*********************************************************************
 * dce_cf_get_cell_name()
 *      Use this dummy function if CDS is not available.
 *********************************************************************/
void
dce_cf_get_cell_name(
    char **              cellname,
    unsigned long *      status
)
{
    *status = 1;

} /* end dce_cf_get_cell_name() */


/*********************************************************************
 * showUsage()
 *      Display "usage" information.
 *********************************************************************/
void
showUsage(
    char *      cmd              /* In--Name of command being called */
)
{
    fprintf( stderr, "\nusage:  %s [option]\n\n", cmd );
    fprintf( stderr, "option:  -a : add an entry\n" );
    fprintf( stderr, "         -r : read an entry\n" );
    fprintf( stderr, "         -d : delete an entry\n" );

} /* end showUsage() */


/**********************************************************************
 * numNamePieces()
 *     Returns the number of pieces in a string name.
 **********************************************************************/
int
numNamePieces(
    char *      string   /* In--String whose pieces are to be counted*/
)
{
    int         count;   /* Number of pieces found */
    char *      currSep; /* Pointer to separator between pieces */

    if( string == NULL ) /* If nothing there, no pieces */
        return( 0 );
    count = 1;           /* Otherwise, there's at least one */

    /*
     * If the first character is a /, it's not really separating
     * two pieces so we want to ignore it here.
     */
    if( *string == '/' )
        currSep = string + 1;
    else
        currSep = string;

    /* How many pieces are there? */
    while( (currSep = strchr( currSep, '/' )) != NULL ) {
        count++;
        currSep++;      /* Begin at next character */
    }

    return( count );
```

```
} /* end numNamePieces() */


/**********************************************************************
 * splitNamePiece()
 *      Divides a piece of a name (string) into its XDS attribute type
 *      and value.
 *********************************************************************/
void
splitNamePiece(
    char *      string, /* In--String to be broken down */
    OM_string * type,   /* Out--XDS type of this piece of the name */
    char **     value   /* Out--Pointer to beginning of the value */
)                       /* part of string                        */
{
    char *      equalSign;      /* Location of the = within string */

    /*
     * If the string contains an equal sign, this is probably a
     * typed name.  Check for all the attribute types allowed by
     * the default schema.
     */
    if( (equalSign = strchr( string, '=' )) != NULL ) {

        *value = equalSign + 1;

        if(( strncmp( string, "C=", 2 ) == 0 ) ||
           ( strncmp( string, "c=", 2 ) == 0 ))
            *type = DS_A_COUNTRY_NAME;

        else if(( strncmp( string, "O=", 2 ) == 0 ) ||
                ( strncmp( string, "o=", 2 ) == 0 ))
            *type = DS_A_ORG_NAME;

        else if(( strncmp( string, "OU=", 3 ) == 0 ) ||
                ( strncmp( string, "ou=", 3 ) == 0 ))
            *type = DS_A_ORG_UNIT_NAME;

        else if(( strncmp( string, "LN=", 3 ) == 0 ) ||
                ( strncmp( string, "ln=", 3 ) == 0 ))
            *type = DS_A_LOCALITY_NAME;
        else if(( strncmp( string, "CN=", 3 ) == 0 ) ||
                ( strncmp( string, "cn=", 3 ) == 0 ))
            *type = DS_A_COMMON_NAME;

        /*
         * If this did not appear to be a type allowed by the
         * default schema, consider the whole string as the
         * value (whose type is "typeless").
         */
        else {
            *type = DSX_TYPELESS_RDN;
            *value = string;
        }
    }

    /*
     * If the string does not contain an equal sign, this is a
     * typeless name.
     */
    else {
        *type = DSX_TYPELESS_RDN;
        *value = string;
    }

} /* end splitNamePiece() */


/**********************************************************************
 * extractValue()
 *      Pulls the value of a particular attribute from a private object
 *      that was received using ds_read().
 *      Returns:
 *              OM_SUCCESS              If successful.
 *              OM_NO_SUCH_OBJECT       If no values for the attribute
 *                                      were found.
 *              other                   Any value returned by one of the
 *                                      om_get() calls.
 **********************************************************************/
OM_return_code
extractValue(
    OM_private_object   object,         /* In--Object to extract from */
    OM_string *          attribute,      /* In--Attribute to extract */
    char *               value           /* Out--Value found */
)
{
    OM_public_object     attrList;
    OM_public_object     attrType;
    OM_public_object     attrValue;
    OM_public_object     entry;
    int                  i;
    OM_return_code       omStatus;
    OM_value_position    total;
    OM_value_position    totalAttributes;
    OM_type              xdsIncludedTypes[] = { 0,    /* Place holder*/
                                                0 }; /* Null terminator*/

    /*
     * Get the entry from the object returned by ds_read().
     */
    xdsIncludedTypes[0] = DS_ENTRY;
    omStatus = om_get( object,                /* Object to extract from */
                   OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
                                             /* Only want what is in    */
                                             /* xdsIncludedTypes, don't */
                                             /* include subobjects      */
                       xdsIncludedTypes,     /* What to get */
                       OM_FALSE,             /* Currently ignored */
                       OM_ALL_VALUES,        /* Start with first value */
                       OM_ALL_VALUES,        /* End with last value */
                       &entry,               /* Put the entry here */
                       &total );             /* Put number of attribute */
                                             /* descriptors here        */
    if( omStatus != OM_SUCCESS ) {
        fprintf( stderr, "om_get( entry ) returned error %d\n",
                 omStatus );
        return( omStatus );
    }
    if( total <= 0 ) {           /* Make sure something was returned */
        fprintf( stderr,
                 "Number of descriptors returned by om_get( entry )
                  was %d\n", total );
        return( OM_NO_SUCH_OBJECT );
```

```
}

/*
 * Get the attribute list from the entry.
 */
xdsIncludedTypes[0] = DS_ATTRIBUTES;
omStatus = om_get( entry->value.object.object,
                   OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
                   xdsIncludedTypes, OM_FALSE, OM_ALL_VALUES,
                   OM_ALL_VALUES, &attrList, &totalAttributes );
if( omStatus != OM_SUCCESS ) {
    fprintf( stderr, "om_get( attrList ) returned error %d\n",
             omStatus );
    return( omStatus );
}
if( total <= 0 ) {           /* Make sure something was returned */
    fprintf( stderr,
          "Number of descriptors returned by om_get( attrList )
           was %d\n", total );
    return( OM_NO_SUCH_OBJECT );
}

/*
 * Search the list for the attribute with the proper type.
 */
for( i = 0; i < totalAttributes; i++ ) {
    xdsIncludedTypes[0] = DS_ATTRIBUTE_TYPE;
    omStatus = om_get( (attrList+i)->value.object.object,
                   OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
                   xdsIncludedTypes, OM_FALSE, OM_ALL_VALUES,
                   OM_ALL_VALUES, &attrType, &total );
    if( omStatus != OM_SUCCESS ) {
        fprintf( stderr, "om_get( attrType ) [i = %d] returned
                   error %d\n", i, omStatus );
        return( omStatus );
    }
    if( total <= 0 ) {       /* Make sure something was returned */
        fprintf( stderr,
        "Number of descriptors returned by om_get( attrType )
                   [i = %d] was %d\n", i, total );
        return( OM_NO_SUCH_OBJECT );
        }
        if( strncmp( attrType->value.string.elements,
                     attribute->elements,
                     attribute->length ) == 0 )
            break;         /* If we found a match, quit looking. */
    }
    if( i == totalAttributes ) {   /* Verify that we found a match. */
        fprintf( stderr,
            "%s: extractValue() could not find requested attribute\n" );
        return( OM_NOT_PRESENT );
    }

    /*
     * Get the attribute value from the corresponding item in the
     * attribute list.
     */
    xdsIncludedTypes[0] = DS_ATTRIBUTE_VALUES;
    omStatus = om_get( (attrList+i)->value.object.object,
                   OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
                   xdsIncludedTypes, OM_FALSE, OM_ALL_VALUES,
                   OM_ALL_VALUES, &attrValue, &total );
    if( omStatus != OM_SUCCESS ) {
        fprintf( stderr, "om_get( attrValue ) returned error %d\n",
                 omStatus );
        return( omStatus );
    }
    if( total <= 0 ) {          /* Make sure something was returned */
        fprintf( stderr,
          "Number of descriptors returned by om_get( attrValue )
                 was %d\n", total );
        return( OM_NO_SUCH_OBJECT );
    }

    /*
     * Copy the value into the buffer for return to the caller.
     */
    strncpy( value, attrValue->value.string.elements,
             attrValue->value.string.length );
    value[attrValue->value.string.length] = ' ';

    /*
```

```
     * Free up the resources we don't need any more and return.
     */
    om_delete( attrValue );
    om_delete( attrType );
    om_delete( attrList );
    om_delete( entry );
    return( OM_SUCCESS );

} /* end extractValue() */


/***********************************************************************
 * stringToXdsName()
 *    Converts a string that is a DCE name to an XDS name object (class
 *    DS_C_DS_DN).  Returns one of the following:
 *           OM_SUCCESS          If successful.
 *           OM_MEMORY_INSUFFICIENT     If a malloc fails.
 *           OM_PERMANENT_ERROR If the name is not in a valid format.
 *           OM_SYSTEM_ERROR            If the local cell's name cannot
 *                                      be determined.
 *
 *      Technically, the space obtained here through malloc() needs
 *      to be returned to the system when it is no longer needed.
 *      If this was a more complex application, this function would
 *      probably malloc all the space it needs at once and require
 *      calling routines to free the space when finished with it.
 **********************************************************************/
OM_return_code
stringToXdsName(
    char *      origString,      /* In--String name to be converted */
    OM_object * xdsNameObj       /* Out--Pointer to XDS name object */
)
{
    OM_descriptor * ava;             /* DS_C_AVA object */
    char *          cellName;        /* Name of this cell */
    OM_object       dsdn;            /* DS_C_DS_DN object */
    char *          end;             /* End of name piece */
    int             index;           /* Index into DS_C_DS_DN object */
    int             numberOfPieces;  /* Number of pieces in the name */
    unsigned long   rc;              /* Return code for some functions*/
    OM_descriptor * rdn;             /* DS_C_RDN object */
```

```
char *          start;          /* Beginning of piece of name */
char *          string;     /* Copy of origString that we can use*/
OM_string       type;       /* Type of one piece of the name */
char *          value;      /* Piece of the name */

/*
 * A DS_C_AVA object only contains pointers to the strings that
 * represent the pieces of the name, not the contents of the
 * strings themselves.  So we'll make a copy of the string passed
 * in to guarantee that these pieces survive in case the programmer
 * alters or reuses the original string.
 *
 * In addition, all valid names should begin with one of the
 * following symbols:
 *     /...        Fully qualified name (from global root). For
 *                 these, we need to ignore the /...
 *     /.:         Partially qualified name (from local cell root).
 *                 For these, we must replace the /.: with the name
 *                 of the local cell name
 * If we see anything else, we'll return with an error.  (Notice
 * that /: is a valid DCE name, but refers to the file system's
 * namespace. Filenames cannot be accessed through
 * CDS, GDS, or XDS.)
 */
if( strncmp( origString, "/.../", 5 ) == 0 ) {
    string = (char *)malloc( strlen(origString+5) + 1 );
    if( string == NULL )                      /* malloc() failed */
        return OM_MEMORY_INSUFFICIENT;
    strcpy( string, origString+5 );
}
else if( strncmp( origString, "/.:/", 4 ) == 0 ) {
    dce_cf_get_cell_name( &cellName, &rc );
    if( rc != 0 )                     /* Could not get cell name */
        return OM_SYSTEM_ERROR;

    /*
     * The cell name will have /.../ on the front, so we will
     * skip over it as we add it to the string (by always
     * starting at its fifth character).
     */
    string = (char *)malloc( strlen
                (origString+4) + strlen(cellName+5) + 2 );
    if( string == NULL )                          /* malloc() failed */
        return OM_MEMORY_INSUFFICIENT;
    strcpy( string, cellName+5 );
    strcat( string, "/" );
    strcat( string, origString+4 );
}
else                                        /* Invalid name format */
    return OM_PERMANENT_ERROR;

/*
 * Count the number of pieces in the name that will have to
 * be dealt with.
 */
numberOfPieces = numNamePieces( string );

/*
 * Allocate memory for the DS_C_DS_DN object.  We will need an
 * OM_descriptor for each name piece, one for the class
 * identifier, and one for the null terminator.
 */
dsdn = (OM_object)malloc(
         (numberOfPieces + 2) * sizeof(OM_descriptor) );
if( dsdn == NULL )                                /* malloc() failed */
    return OM_MEMORY_INSUFFICIENT;

/*
 * Initialize it as a DS_C_DS_DN object by placing that class
 * identifier in the first position.
 */
FILL_OMD_XOM_STRING( dsdn, 0, OM_CLASS, OM_S_OBJECT, DS_C_DS_DN )

/*
 * For each piece of string, do the following:
 *     Break off the next piece of the string
 *     Build a DS_C_AVA object to show the type and value
 *         of this piece of the name
 *     Wrap the DS_C_AVA up in a DS_C_RDN object
 *     Add the DS_C_RDN to the DS_C_DS_DN object
 */
for( start=string, index=1 ; index <= numberOfPieces ;
  index++, start=end+1 ) {

/*
 * Find the next delimiter and replace it with a null byte
 * so the piece of the name is effectively separated from
 * the rest of the string.
 */
end = strchr( start, '/' );
if( end != NULL )
    *end = ' ';
else            /* If this is the last piece, there won't be */
                /* a '/' at the end, just a null byte.       */
    end = strchr( start, ' ' );

/*
 * Allocate space for a DS_C_AVA object and fill in its entries:
 *      DS_C_AVA class identifier
 *      AVA's type
 *      AVA's value
 *      null terminator
 */
ava = (OM_descriptor *)malloc( sizeof(OM_descriptor) * 4 );
if( ava == NULL )                       /* malloc() failed */
    return OM_MEMORY_INSUFFICIENT;
FILL_OMD_XOM_STRING( ava, 0, OM_CLASS,
                   OM_S_OBJECT_IDENTIFIER_STRING, DS_C_AVA )
splitNamePiece( start, &type, &value );
FILL_OMD_XOM_STRING( ava, 1, DS_ATTRIBUTE_TYPE,
                   OM_S_OBJECT_IDENTIFIER_STRING, type )
FILL_OMD_STRING( ava, 2, DS_ATTRIBUTE_VALUES,
                 OM_S_PRINTABLE_STRING, value )
FILL_OMD_NULL( ava, 3 )

/*
 * Allocate space for a DS_C_RDN object and fill in its entries:
 *      DS_C_RDN class identifier
 *      AVA it contains
 *      null terminator
 */
rdn = (OM_descriptor *)malloc( sizeof(OM_descriptor) * 3 );
if( rdn == NULL )                       /* malloc() failed */
```

```
            return OM_MEMORY_INSUFFICIENT;
        FILL_OMD_XOM_STRING( rdn, 0, OM_CLASS, OM_S_OBJECT, DS_C_DS_RDN )
        FILL_OMD_STRUCT( rdn, 1, DS_AVAS, OM_S_OBJECT, ava )
        FILL_OMD_NULL( rdn, 2 )

        /* Add the DS_C_RDN object to the DS_C_DS_DN object. */
        FILL_OMD_STRUCT( dsdn, index, DS_RDNS, OM_S_OBJECT, rdn )
    }

    /*
     * Null terminate the DS_C_DS_DN, tell the calling routine
     * where to find it, and return.
     */
    FILL_OMD_NULL( dsdn, index )
    *xdsNameObj = dsdn;
    return( OM_SUCCESS );

} /* end stringToXdsName() */


/**********************************************************************
 * handleDSError()
 *     Extracts the error number from a DS_status return code, prints it
 *     in an error message, then terminates the program.
 *********************************************************************/
void
handleDSError(
    char *      header,     /* In--Name of function whose return code */
                            /*     is being checked                   */
    DS_status   returnCode  /* In--Return code to be checked */
)
{
    OM_type             includeDSProblem[] = { DS_PROBLEM,
                                               0 };
    OM_return_code      omStatus;
    OM_public_object    problem;
    OM_value_position   total;

    /*
     * A DS_status return code is an object.  It will be one of the
     * subclasses of the class DS_C_ERROR.  What we want from it is
     * the value of the attribute DS_PROBLEM.
     */
    omStatus = om_get( returnCode,
                       OM_EXCLUDE_ALL_BUT_THESE_TYPES+OM_EXCLUDE_SUBOBJECTS,
                       includeDSProblem,
                       OM_FALSE,
                       OM_ALL_VALUES,
                       OM_ALL_VALUES,
                       &problem,
                       &total );

    /*
     * Make sure we successfully extracted the problem number and print
     * the error message before quitting.
     */
    if( (omStatus == OM_SUCCESS) && (total > 0) )
        printf( "%s returned error %d\n", header,
                problem->value.enumeration );
    else
        printf( "%s failed for unknown reason\n", header );

    exit( 1 );
}


/*********************************************************************
 * Main program
 */
void
main(
    int         argc,
    char *      argv[]
)
{
    DS_status               dsStatus;
    OM_sint                 invokeID;
    char                    newName[MAX_NAME_LEN];
    char                    newPhoneNum[MAX_PHONE_LEN];
    char                    newSurname[MAX_SURNAME_LEN];
    OM_return_code          omStatus;
    char                    phoneNumRead[MAX_PHONE_LEN];
```

```
int                     rc = 0;
OM_private_object       readResult;
OM_private_object       session;
char                    surnameRead[MAX_SURNAME_LEN];
OM_object               xdsName;
OM_workspace            xdsWorkspace;
int                     operation;

/* Step 1
 *
 * Examine command-line argument.
 */
 operation = getopt( argc, argv, "rad" );
 if ( (operation == '?') || (operation == EOF) ) {
     showUsage( argv[0] );
     exit( 1 );
 }

/* Step 2
 *
 * Initialize the XDS workspace.
 */
xdsWorkspace = ds_initialize( );
if( xdsWorkspace == NULL ) {
    fprintf( stderr, "ds_initialize() failed\n" );
    exit( 1 );
}

/* Step 3
 *
 * Pull in the packages that contain the XDS features we need.
 */
dsStatus = ds_version( featureList, xdsWorkspace );
if( dsStatus != DS_SUCCESS )
    handleDSError( "ds_version()", dsStatus );

/* Step 4
 *
 * Find out what name the user wants to use in the namespace and
 * convert it to and XDS object.  We do this conversion dynamically
 * (not using static structures defined at the top of the program)
 * because we don't know how long the name will be.
 */
switch( operation ) {
case 'r' :
    printf( "What name do you want to read? " );
    break;
case 'a' :
    printf( "What name do you want to add? " );
    break;
case 'd' :
    printf( "What name do you want to delete? " );
    break;
}

/* Step 5 */

gets( newName );
omStatus = stringToXdsName( newName, &xdsName );
if( omStatus != OM_SUCCESS ) {
    fprintf( stderr, "stringToXdsName() failed with OM error %d\n",
             omStatus );
    exit( 1 );
}

if ( operation == 'a' ) {
    /* add operation requires additional input */
    /*
     * Get the person's real name from the user and place it in
     * the XDS object already defined at the top of the program
     * (xdsSurname). We are requiring a name, so we will loop
     * until we get one.
     */
    do {
        printf( "What is this person's surname? " );
        gets( newSurname );
    } while ( *newSurname == ' ' );
    FILL_OMD_STRING( xdsSurname, 2, DS_ATTRIBUTE_VALUES,
                     OM_S_TELETEX_STRING, newSurname )

    /*
     * Get the person's phone number from the user and place it
```

```
      * in the XDS object already defined at the top of the
      * program (xdsPhoneNum). A phone number is not required,
      * so if none is given we will use the default already
      * stored in the structure.
      */
     printf( "What is this person's phone number? " );
     gets( newPhoneNum );
     if( *newPhoneNum != ' ' ) {
         FILL_OMD_STRING( xdsPhoneNum, 2, DS_ATTRIBUTE_VALUES,
                          OM_S_PRINTABLE_STRING, newPhoneNum )
     }
 }

 /* Step 6
  *
  * Open the session with the namespace:
  *  bind (without credentials) to the default server.
  */
 dsStatus = ds_bind( DS_DEFAULT_SESSION, xdsWorkspace, &session );
 if( dsStatus != DS_SUCCESS )
     handleDSError( "ds_bind()", dsStatus );


 /* Step 7 */

 switch( operation ) {   /* perform the requested operation */

 /*
  * Add entry to the namespace. The xdsSurname and xdsPhoneNum
  * objects are already contained within an attribute list object
  * (xdsAttributesToAdd).
  */
 case 'a' :
     dsStatus = ds_add_entry( session, DS_DEFAULT_CONTEXT, xdsName,
                              xdsAttributesToAdd, &invokeID );
     if( dsStatus != DS_SUCCESS )
         handleDSError( "ds_add_entry()", dsStatus );

     /* FALL THROUGH */

 /*
```

```
      * Read the entry of the name supplied.
      */
     case 'r' :
         dsStatus = ds_read( session, DS_DEFAULT_CONTEXT, xdsName,
                        xdsAttributeSelection, &readResult, &invokeID );
         if( dsStatus != DS_SUCCESS )
             handleDSError( "ds_read()", dsStatus );

         /*
          * Get each attribute from the object read and print them.
          */
         omStatus = extractValue( readResult, &DS_A_SURNAME,
                     surnameRead );
         if( omStatus != OM_SUCCESS ) {
             printf( "** Surname could not be read\n" );
             strcpy( surnameRead, "(unknown)" );
             rc = 1;
         }
         omStatus = extractValue( readResult, &DS_A_PHONE_NBR,
                     phoneNumRead );
         if( omStatus != OM_SUCCESS ) {
             printf( "** Phone number could not be read\n" );
             strcpy( phoneNumRead, "(unknown)" );
             rc = 1;
         }
         printf( "The phone number for %s is %s.\n", surnameRead,
                   phoneNumRead );

         break;

     /*
      * delete the entry from the namespace.
      */
     case 'd' :
         dsStatus = ds_remove_entry( session, DS_DEFAULT_CONTEXT,
                                    xdsName, &invokeID );
         if( dsStatus != DS_SUCCESS )
             handleDSError( "ds_remove_entry()", dsStatus );
         else
             printf( "The entry has been deleted.\n" );
         break;
```

```
    }


    /*
     * Clean up and exit.
     */
    /* Step 8 */
    dsStatus = ds_unbind( session );
    if( dsStatus != DS_SUCCESS )
        handleDSError( "ds_unbind()", dsStatus );

    /* Step 9 */
    dsStatus = ds_shutdown( xdsWorkspace );
    if( dsStatus != DS_SUCCESS )
        handleDSError( "ds_shutdown()", dsStatus );

    exit( rc );

} /* end main() */
```

Part 4C

XDS/XOM Supplementary Information

Part 4C provides reference material for the X/Open Object Management (XOM) programming interface.

Chapter 29

XDS Interface Description

The XDS interface comprises a number of functions, together with many OM classes of OM objects, which are used as the parameters and results of the functions. Both the functions and the OM objects are based closely on the Abstract Service that is specified in the standards (see *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511).

The interface models the directory interactions as service requests made through a number of interface functions, which take a number of input parameters. Each valid request causes an operation within the Directory Service, which eventually returns a status and any result of the operation.

All interactions between the user and the Directory Service belong to a session, which is represented by an OM object passed as the first parameter to most interface functions.

The other parameters to the functions include a context and various service-specific parameters. The context includes a number of parameters that are common to many functions, and that seldom change from operation to operation.

Each of the components of this model are described in the following sections in this chapter along with other features of the interface, such as security.

29.1 XDS Conformance to Standards

The XDS interface defines an API that application programs can use to access the functionality of the underlying Directory Service. The DCE XDS API conforms to the *X/Open CAE Specification, API to Directory Services (XDS)* (November 1991).

The DCE XDS implementation supports the following features:

- A synchronous interface. Asynchronous operations are not supported.
- All synchronous interface functions are supported. The two asynchronous-specific functions are handled as follows:
  - — **ds_abandon( )**

    This call does not issue a Directory Service abandon operation. It returns with a **DS_C_ABANDON_FAILED (DS_E_TOO_LATE)** error.
  - — **ds_receive_result( )**

    This call returns **DS_SUCCESS** with the *completion_flag_return* parameter set to **DS_NO_OUTSTANDING_OPERATION.**
- Automatic connection management is not provided. The **ds_bind**( ) and **ds_unbind**( ) functions always try, respectively, to set up and release Directory Service connections immediately.
- The **DS_FILE_DESCRIPTOR** attribute of the **DS_C_SESSION** object is not used.
- The default values for OM attributes in the **DS_C_CONTEXT** and **DS_C_SESSION** objects are described in Chapter 30.

DCE XDS supports four packages, of which one is mandatory and three are optional. Use of the optional packages is negotiated using **ds_version**( ). The packages are as follows:

- The Directory Service Package (as defined in Chapter 30), which also includes the errors (as defined in Chapter 31). This package is mandatory.
- The Basic Directory Contents Package (as defined in Chapter 32). This package is optional.

- The Global Directory Service Package (as defined in Chapter 34). This package is optional.
- The MHS Directory User Package (as defined in Chapter 33). This package is optional.

None of the OM classes defined in these four packages are encodable. Thus, DCE XDS application programmers do not require the use of the XOM functions **om_encode( )** and **om_decode( )**, which are not supported by the DCE XOM API.

29.2 The XDS Functions

As mentioned already, the standards define Abstract Services that requestors use to interact with the directory. Each of these Abstract Services maps to a single function call, and the detailed specifications are given in the XDS reference pages in the *OSF DCE Application Development Reference*. The services and the function calls to which they map are as follows:

- **DirectoryBind** (maps to **ds_bind( )**)
- **DirectoryUnbind** (maps to **ds_unbind( )**)
- **Read** (maps to **ds_read( )**)
- **Compare** (maps to **ds_compare( )**)
- **Abandon** (maps to **ds_abandon( )**)
- **List** (maps to **ds_list( )**)
- **Search** (maps to **ds_search( )**)
- **AddEntry** (maps to **ds_add_entry( )**)
- **RemoveEntry** (maps to **ds_remove_entry( )**)
- **ModifyEntry** (maps to **ds_modify_entry( )**)
- **ModifyRDN** (maps to **ds_modify_rdn( )**)

There is a function called **ds_receive_result( )**, which has no counterpart in the Abstract Service. It is used with asynchronous operations. (See the XDS **intro(3xds)** reference page in the *OSF DCE Application Development Reference* for information on how the asynchronous functions

ds_abandon() and **ds_receive_result()** are handled by the DCE XDS API.)

The **ds_initialize()**, **ds_shutdown()**, and **ds_version()** functions are used to control the XDS API and do not initiate any directory operations.

The interface functions are summarized in Table 29-1.

Table 29–1. The XDS Interface Functions

Name	Description
ds_abandon()	Abandons the result of a pending asynchronous operation. This function is not supported (see **intro(3xds)** in the *OSF DCE Application Development Reference*).
ds_add_entry()	Adds a leaf entry to the DIT.
ds_bind()	Opens a session with a DUA, which in turn connects to a DSA.
ds_compare()	Compares a purported attribute value with the attribute value stored in the DIB for a particular entry.
ds_initialize()	Initializes the XDS interface.
ds_list()	Enumerates the names of the immediate subordinates of a particular directory entry.
ds_modify_entry()	Atomically performs modification to a directory entry.
ds_modify_rdn()	Changes the RDN of a leaf entry.
ds_read()	Queries information on a particular directory entry by name.
ds_receive_result()	Retrieves the result of an asynchronously executed function. This function is not supported (see **intro(3xds)** in the *OSF DCE Application Development Reference*).

Name	Description
ds_remove_entry( )	Removes a leaf entry from the DIT.
ds_search( )	Finds entries of interest in a portion of the directory information tree.
ds_shutdown( )	Discards a workspace.
ds_unbind( )	Unbinds from a directory session.
ds_version( )	Negotiates features of the interface and service.

29.3 The XDS Negotiation Sequence

The interface has an initialization and shutdown sequence that permits the negotiation of optional features. This involves the **ds_initialize**( ), **ds_version**( ), and **ds_shutdown**( ) functions.

Every application program must first call **ds_initialize**( ), which returns a workspace. This workspace supports the standard Directory Service Package (see Chapter 30).

The workspace can be extended to support the optional Basic Directory Contents Package (see Chapter 32), the Global Directory Service Package (see Chapter 34), or the MHS Directory User Package (see Chapter 33). These packages are identified by means of OSI Object Identifiers, and these Object Identifiers are supplied to **ds_version**( ) to incorporate the extensions into the workspace.

After a workspace with the required features is negotiated in this way, the application can use the workspace as required. It can create and manipulate OM objects using the OM functions, and can start one or more directory sessions using **ds_bind**( ).

After completing its tasks, terminating all its directory sessions using **ds_unbind**( ), and releasing all its OM objects using **om_delete**( ), the application needs to ensure that resources associated with the interface are freed by calling **ds_shutdown**( ). It is possible to retain access to service-generated public objects after **ds_shutdown**( ) is called, or to start another cycle by calling **ds_initialize**( ) if so required by the application design.

29.4 The session Parameter

A session identifies the DUA and the suite of DSAs to which a particular directory operation is sent. It contains some **DirectoryBindArguments**, such as the distinguished name of the requestor. The *session* parameter is passed as the first parameter to most interface functions.

A session is described by an OM object of OM class **DS_C_SESSION**. It is created and appropriate parameter values can be set using the OM functions. A directory session then starts with **ds_bind( )** and later terminates with **ds_unbind( )**. A session with default parameters can be started by passing the constant **DS_DEFAULT_SESSION** as the **DS_C_SESSION** parameter to **ds_bind( )**.

The **ds_bind( )** function must be called before **DS_C_SESSION** can be used as a parameter to any other function in this interface. After **ds_unbind( )** is called, **ds_bind( )** must be called again if another session is to be started.

The interface supports multiple concurrent sessions so that an application implemented as a single process, such as a server in a client/server model, can interact with the directory using several identities, and a process can interact directly and concurrently with different parts of the directory.

Details of the OM Class **DS_C_SESSION** are given in Chapter 30.

29.5 The context Parameter

The context defines the characteristics of the directory interaction that are specific to a particular directory operation; nevertheless, the same characteristics are often used for many operations. Since these parameters are presumed to be relatively static for a given directory user during a particular directory interaction, these parameters are collected into an OM object of OM class **DS_C_CONTEXT**, which is supplied as the second parameter of each Directory Service request. This reduces the number of parameters passed to each function.

The context includes many administrative details, such as the **CommonArguments** defined in the Abstract Service, which affect the processing of each directory operation. These include a number of **ServiceControls**, which allow control over some aspects of the service. The **ServiceControls** include options such as **preferChaining**, **chainingProhibited**, **localScope**, **dontUseCopy**, and **dontDereferenceAliases**, together with **priority**, **timeLimit**, **sizeLimit**, and **scopeOfReferral**. Each of these is mapped onto an OM attribute in the context (see Chapter 30).

The effect of passing the *context* parameter is as if its contents were passed as a group of additional parameters for every function call. The value of each component of the context is determined when the interface function is called, and remains fixed throughout the operation.

All OM attributes in the class **DS_C_CONTEXT** have default values, some of which are administered locally. The constant **DS_DEFAULT_CONTEXT** can be passed as the value of the **DS_C_CONTEXT** parameter to the interface functions, and has the same effect as a context OM object created with default values. The context must be a private object, unless it is **DS_DEFAULT_CONTEXT**.

(See Chapter 30 for detailed specifications of the OM class **DS_C_CONTEXT**.)

29.6 The XDS Function Arguments

The Abstract Service defines specific parameters for each operation. These are mapped onto corresponding parameters to each interface function, which are also called input parameters. Although each service has different parameters, some specific parameters recur in several operations and these are briefly introduced here. (For complete details of these parameters, see Chapter 30.)

All parameters that are OM objects can generally be supplied to the interface functions as public objects (that is, descriptor lists) or as private objects. Private objects must be created in the workspace that is returned by **ds_initialize**( ). In some cases, constants can be supplied instead of OM objects.

Note: Wherever a function can accept an instance of a particular OM class as the value of a parameter, it also accepts an instance of any subclass of the OM class. For example, most functions have a *name* parameter, which accepts values of OM class *DS_C_NAME.* It is always acceptable to supply an instance of the subclass **DS_C_DS_DN** as the value of the parameter.

29.6.1 Attribute and Attribute Value Assertion

Each directory attribute is represented in the interface by an OM object of OM class **DS_C_ATTRIBUTE.** The type of the directory attribute is represented by an OM attribute, **DS_ATTRIBUTE_TYPE,** within the OM object. The values of the directory attribute are expressed as the values of the OM attribute **DS_ATTRIBUTE_VALUES.**

The representation of the attribute value depends on the attribute type and is determined as indicated in the following list. The list describes the way in which an application program must supply values to the interface; for example, in the *changes* parameter to **ds_modify_entry**( ). The interface follows the same rules when returning attribute values to the application; for example, in the **ds_read**( ) result.

- The first possibility is that the attribute type and the representation of the corresponding values can be defined in a package; for example, the selected attribute types from the standards that are defined in the Basic Directory Contents Package in Chapter 32. In this case, attribute values are represented as specified. Additional directory attribute types and their OM representations are defined by the Global Directory Service Package.
- If the attribute type is not known and the value is an ASN.1 simple type such as **IntegerType**, the representation is the corresponding type specified in Chapter 35.
- If the attribute type is not known and the value is an ASN.1 structured type, the value is represented in the Basic Encoding Rules (BER) with OM syntax String(**OM_S_ENCODING_STRING**).

Note: The distinguished encoding specified in the standards (see Clause 8.7 of *The Directory: Authentication Framework*, ISO 9594-8, CCITT X.500) must be used if the request is to be signed.

Where attribute values have OM syntax String(*), they can be long segmented strings, and the functions **om_read()** and **om_write()** need to be used to access them.

An Attribute Value Assertion (AVA) is an assertion about the value of an attribute of an entry, and can be TRUE, FALSE, or undefined. It consists of an attribute type and a single value. In general, the AVA is TRUE if one of the values of the given attribute in the entry matches the given value. An AVA is represented in the interface by an instance of OM class **DS_C_AVA**, which is a subclass of **DS_C_ATTRIBUTE** and can only have one value.

Information used by **ds_add_entry()** to construct a new directory entry is represented by an OM object of OM class **DS_C_ATTRIBUTE_LIST**, which contains a single multivalued OM attribute whose values are OM objects of OM class **DS_C_ATTRIBUTE**.

29.6.2 The Entry-Information-Selection Parameter

The *selection* parameter of the **ds_read()** and **ds_search()** operations tailors its results to obtain just part of the required entry. Information on all attributes, no attributes, or a specific group of attributes can be chosen. Attribute types are always returned, but the attribute values are not necessarily returned.

The value of the parameter is an instance of OM class **DS_C_ENTRY_INFO_SELECTION**, but one of the constants in the following list can be used in simple cases:

- To verify the existence of an entry for the purported name, use the constant **DS_SELECT_NO_ATTRIBUTES**.
- To return just the types of all attributes, use the constant **DS_SELECT_ALL_TYPES**.
- To return the types and values of all attributes, use the constant **DS_SELECT_ALL_TYPES_AND_VALUES**.

To choose a particular set of attributes, create a new instance of the OM class **DS_C_ENTRY_INFO_SELECTION** and set the appropriate OM attribute values using the OM functions.

29.6.3 The name Parameter

Most operations take a *name* parameter to specify the target of the operation. The name is represented by an instance of one of the subclasses of the OM class *DS_C_NAME*. The DCE XDS API defines the subclass **DS_C_DS_DN** to represent distinguished names and other names.

For directory interrogations, any aliases in the name are dereferenced, unless prohibited by the **DS_DONT_DEREFERENCE_ALIASES** service control. However, for modify operations, this service control is ignored if set, and aliases are never dereferenced.

RDNs are represented by an instance of one of the subclasses of the OM class *DS_C_RELATIVE_NAME*. The DCE XDS API defines the subclass **DS_C_DS_RDN** to represent RDNs.

29.7 XDS Function Call Results

All XDS functions return a **DS_status**, which is the C function result; most return data in an *invoke_id* parameter, which identifies the particular invocation, and the interrogation operations each return data in the *result* parameter. The *invoke_id* and *result* values are returned using pointers that are supplied as parameters of the C function. These three types of function results are introduced in the following subsections.

All OM objects returned by interface functions (results and errors) are private objects in the workspace returned by **ds_initialize**( ).

29.7.1 The invoke_id Parameter

All interface functions that invoke a Directory Service operation return an *invoke_id* parameter, which is an integer that identifies the particular invocation of an operation. Since asynchronous operations are not supported, the *invoke_id* return value is no longer relevant for operations. DCE application programmers must still supply this parameter as described in the XDS reference pages (see the *OSF DCE Application Development Reference*), but should ignore the value returned.

29.7.2 The result Parameter

Directory Service interrogation operations return a *result* value only if they succeed. All errors from these operations, including Directory Access Protocol (DAP) errors, are reported in **DS_status** (see Section 29.7.3), as are errors from all other operations.

The result of an interrogation is returned in a private object whose OM class is appropriate to the particular operation. The format of directory operation results is driven by the Abstract Service. To simplify processing, the result of a single operation is returned in a single OM object, which corresponds to the abstract result defined in the standards. The components of the result of an operation are represented by OM attributes in the operation's result object. All information contained in the Abstract Service result is made available to the application program. The result is inspected using the functions provided in the Object Management API, **om_get**( ).

Only the interrogation operations produce results, and each type of interrogation has a specific OM class of OM object for its result. These OM classes are as follows (see Chapter 30 for their definitions):

- **DS_C_COMPARE_RESULT**
- **DS_C_LIST_RESULT**
- **DS_C_READ_RESULT**
- **DS_C_SEARCH_RESULT**

The results of the different operations share several common components, including the **CommonResults** defined in the standards (see *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511) by inheriting OM attributes from the superclass *DS_C_COMMON_RESULTS*. An additional common component is the full DN of the target object, after all aliases are dereferenced.

The actual OM class of the result can always be a subclass of that named in order to allow flexibility for extensions. Thus, **om_instance()** always needs to be used when testing the OM class.

Any attribute values in the result are represented as discussed in Section 29.6.1.

29.7.3 The DS_status Return Value

Every interface function returns a **DS_status** value, which is either the constant **DS_SUCCESS** or an error. Errors are represented by private objects whose OM class is a subclass of *DS_C_ERROR*. Details of all errors are given in Chapter 31.

Other results of functions are not valid unless the status result has the value **DS_SUCCESS**.

29.8 Synchronous Operations

Since asynchronous use of the interface is not supported, the value of the **DS_ASYNCHRONOUS** OM attribute in **DS_C_CONTEXT** is always **OM_FALSE**, causing all operations to be synchronous.

In synchronous mode, all functions wait until the operation is complete before returning. The thread of control is blocked within the interface after calling a function, and it can use the result immediately after the function returns.

Implementations define a limit on the number of asynchronous operations that can be outstanding at any one time on any one session. The limit is given by the implementation-defined constant **DS_MAX_OUTSTANDING_OPERATIONS**. It always has the value 0 (zero) because asynchronous operations are not supported.

All errors occurring during a synchronous request are reported when the function returns. (See Chapter 31 for complete details of error handling.)

The **DS_FILE_DESCRIPTOR** OM attribute of **DS_C_SESSION** is not used by the DCE XDS API and is always set to **DS_NO_VALID_FILE_DESCRIPTOR**.

29.9 Security and XDS

The X/Open XDS specifications do not define a security interface because this can put constraints on security features of existing directory implementations.

DCE GDS provides security by means of passwords. This is achieved at the XDS API level through a new **DSX_C_GDS_SESSION** session object with an OM **DSX_PASSWORD** attribute. (See Chapter 34 for additional information.) The GDS DSA verifies this password for each directory operation.

29.10 Other Features of the XDS Interface

The following subsections describe these features of the interface:

- Automatic Connection Management
- Automatic Continuation and Referral Handling

29.10.1 Automatic Connection Management

An implementation can provide automatic management of the association or connection between the user and the Directory Service, making and releasing connections at its discretion.

The DCE XDS implementation does not support automatic connection management. A DSA connection is established when **ds_bind**( ) is called and released when **ds_unbind**( ) is called.

29.10.2 Automatic Continuation and Referral Handling

The interface provides automatic handling of continuation references and referrals in order to reduce the burden on application programs. These facilities can be inhibited to meet special needs.

A ``continuation reference'' describes how the performance of all or part of an operation can be continued at a different DSA or DSAs. A single continuation reference returned as the entire response to an operation is called a ``referral'' and is classified as an error. One or more continuation references can also be returned as part of **DS_PARTIAL_OUTCOME_QUAL** returned from a **ds_list**( ) or **ds_search**( ) operation.

A DSA returns a referral if it has administrative, operational, or technical reasons for preferring not to chain. It can return a referral if **DS_CHAINING_PROHIB** is set in the **DS_C_CONTEXT**, or instead it can report a service error (**DS_E_CHAINING_REQUIRED**) in this case.

By default, the implementation uses any continuation references it receives to try to contact the other DSA or DSAs, enabling it to make further progress in the operation, whenever practical. It only returns the result, or an error, to the application after it has made this attempt. Note that continuation references can still be returned to the application, if the relevant DSA cannot be contacted, for example.

The default behavior is the simplest for most applications, but if necessary the application can cause all continuation references to be returned to it. It does this by setting the value of the OM attribute **DS_AUTOMATIC_CONTINUATION** in the **DS_C_CONTEXT** to **OM_FALSE**.

Chapter 30

XDS Class Definitions

When referring to classes and attributes in the Directory Service, the chapters in Part 4B make a clear distinction between OM classes and directory classes, and between OM attributes and directory attributes. In both cases, the former is a construct of the closely associated Object Management interface, while the latter is a construct of the Directory Service to which XDS provides access. The terms ‘‘object class’’ and ‘‘attribute’’ indicate the directory constructs, while the phrases ‘‘OM class’’ and ‘‘OM attribute’’ indicate the Object Management constructs.

30.1 Introduction to OM Classes

This chapter defines, in alphabetical order, the OM classes that constitute the Directory Service Package. The errors defined in Chapter 31 also belong to this package. The object identifier associated with this package is **{iso(1) identified-organization(3) icd-ecma(0012) member-company(2) dec(1011) xopen(28) dsp(0)}** with the following encoding:

\x2B\xC\x2\x87\x73\x1C\x0

This object identifier is represented by the constant **DS_SERVICE_PKG**.

The Object Management notation is briefly described in the following text. See Chapters 35 through 37 for more information on Object Management.

Each OM class is described in a separate section, which identifies the OM attributes specific to that OM class. The OM classes and OM attributes for each OM class are listed in alphabetical order. The OM attributes that can be found in an instance of an OM class are those OM attributes specific to that OM class, as well as those inherited from each of its superclasses. The OM class-specific OM attributes are defined in a table. The table indicates the name of each OM attribute, the syntax of each of its values, any restrictions upon the length (in bits, octets (bytes), or characters) of each value, any restrictions upon the number of values, and the value, if any, **om_create**( ) supplies.

The constants that represent the OM classes and OM attributes in the C binding are defined in the **xds.h(4xds)** header file (see the *OSF DCE Application Development Reference*).

30.2 OM Class Hierarchy

This section shows the hierarchical organization of the OM classes defined in this chapter, and as a result, shows which OM classes inherit additional OM attributes from their superclasses. In the following list, subclassification is indicated by indentation, and the names of abstract classes are in italics. Thus, for example, the concrete class **DS_C_PRESENTATION_ADDRESS** is an immediate subclass of the abstract class *DS_C_ADDRESS*, which in turn is an immediate subclass of the abstract class *OM_C_OBJECT*. (*OM_C_OBJECT* is defined in Chapter 26 of this guide.)

OM_C_OBJECT

- **DS_C_ACCESS_POINT**
- *DS_C_ADDRESS*
  - **DS_C_PRESENTATION_ADDRESS**
- **DS_C_ATTRIBUTE**
  - **DS_C_AVA**

  - **DS_C_ENTRY_MOD**
  - **DS_C_FILTER_ITEM**
- **DS_C_ATTRIBUTE_LIST**
  - **DS_C_ENTRY_INFO**
- *DS_C_COMMON_RESULTS*
  - **DS_C_COMPARE_RESULT**
  - **DS_C_LIST_INFO**
  - **DS_C_READ_RESULT**
  - **DS_C_SEARCH_INFO**
- **DS_C_CONTEXT**
- **DS_C_CONTINUATION_REF**
- **DS_C_ENTRY_INFO_SELECTION**
- **DS_C_ENTRY_MOD_LIST**
- *DS_C_ERROR* (see Chapter 31)
- **DS_C_EXT**
- **DS_C_FILTER**
- **DS_C_LIST_INFO_ITEM**
- **DS_C_LIST_RESULT**
- *DS_C_NAME*
  - **DS_C_DS_DN**
- **DS_C_OPERATION_PROGRESS**
- **DS_C_PARTIAL_OUTCOME_QUAL**
- *DS_C_RELATIVE_NAME*
  - **DS_C_DS_RDN**
- **DS_C_SEARCH_RESULT**
- **DS_C_SESSION**

None of the classes in the preceding list are encodable using **om_encode**( ) and **om_decode**( ). The application is not permitted to create or modify instances of some OM classes because these OM classes are only returned by the interface and never supplied to it. These OM classes are as follows:

DS_C_ACCESS POINT

DS_C_COMPARE_RESULT

DS_C_CONTINUATION_REF

All subclasses of *DS_C_ERROR*

DS_C_LIST_INFO

DS_C_LIST_INFO_ITEM

DS_C_LIST_RESULT

DS_C_OPERATION_PROGRESS

DS_C_PARTIAL_OUTCOME_QUAL

DS_C_READ_RESULT

DS_C_SEARCH_INFO

DS_C_SEARCH_RESULT

30.3 DS_C_ACCESS_POINT

An instance of OM class **DS_C_ACCESS_POINT** identifies a particular point at which a DSA can be accessed.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-1.

Table 30–1. OM Attributes of DS_C_ACCESS_POINT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ADDRESS	Object(*DS_C_ADDRESS*)	—	1	—
DS_AE_TITLE	Object(*DS_C_NAME*)	—	1	—

- **DS_ADDRESS**

  This attribute indicates the address of the DSA to be used when communicating with it.

- **DS_AE_TITLE**

  This attribute indicates the name of the DSA.

30.4 DS_C_ADDRESS

The OM class *DS_C_ADDRESS* represents the address of a particular entity or service, such as a DSA.

It is an abstract class that has the OM attributes of its superclass, *OM_C_OBJECT*, and no other OM attributes.

An address is an unambiguous name, label, or number that identifies the location of the entity or service. All addresses are represented as instances of some subclass of this OM class.

The only subclass defined by the DCE XDS API is **DS_C_PRESENTATION_ADDRESS**, which is the presentation address of an OSI application entity used for OSI communications with this subclass.

30.5 DS_C_ATTRIBUTE

An instance of OM class **DS_C_ATTRIBUTE** is an attribute of an object, and is thus a component of its directory entry.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-2.

Table 30-2. OM Attributes of DS_C_ATTRIBUTE

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ATTRIBUTE_TYPE	String(**OM_S_OBJECT_IDENTIFIER_STRING**)	—	1	—
DS_ATTRIBUTE_VALUES	Any	—	0 or more	—

- **DS_ATTRIBUTE_TYPE**

  The attribute type that indicates the class of information given by this attribute.

- **DS_ATTRIBUTE_VALUES**

  The attribute values. The OM value syntax and the number of values allowed for this OM attribute are determined by the value of the **DS_ATTRIBUTE_TYPE** OM attribute in accordance with the rules given in Section 29.6.1.

  If the values of this OM attribute have the syntax String(*), the strings can be long and segmented. For this reason, **om_read**( ) and **om_write**( ) need to be used to access all String(*) values.

Note: A directory attribute must always have at least one value, although it is acceptable for instances of this OM class not to have any values.

30.6 DS_C_ATTRIBUTE_LIST

An instance of OM class **DS_C_ATTRIBUTE_LIST** is a list of directory attributes.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attribute listed in Table 30-3.

Table 30–3. OM Attribute of DS_C_ATTRIBUTE_LIST

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ATTRIBUTES	Object(**DS_C_ ATTRIBUTE**)	—	0 or more	—

- **DS_ATTRIBUTES**

  The attributes that constitute a new object's directory entry, or those selected from an existing entry.

30.7 DS_C_AVA

An instance of OM class **DS_C_AVA** (Attribute Value Assertion) is a proposition concerning the values of a directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C ATTRIBUTE**, and no other OM attributes. An additional restriction on this OM class is that there must be exactly one value of the OM attribute **DS_ATTRIBUTE_VALUES.** The **DS_ATTRIBUTE_TYPE** remains single valued. The OM value syntax of **DS_ATTRIBUTE_VALUES** must conform to the rules outlined in Section 29.6.1.

30.8 DS_C_COMMON_RESULTS

The OM class *DS_C_COMMON_RESULTS* comprises results that are returned by, and are common to, the directory interrogation operations.

It is an abstract OM class, which has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-4.

Table 30–4. OM Attributes of DS_C_COMMON_RESULTS

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALIAS_ DEREFERENCED	**OM_S_ BOOLEAN**	—	1	—
DS_PERFORMER	Object(*DS_C_ NAME*)	—	0 or 1	—

- **DS_ALIAS_DEREFERENCED**

  This attribute indicates whether the name of the target object that is passed as a function argument includes an alias that is dereferenced to determine the DN.

- **DS_PERFORMER**

  When present, this attribute gives the DN of the performer of a particular operation. It can be present when the result is signed, and it holds the name of the DSA that signed the result. The DCE Directory Service does not support the optional feature of signed results; therefore, this OM attribute is never present.

30.9 DS_C_COMPARE_RESULT

An instance of OM class **DS_C_COMPARE_RESULT** comprises the results of a successful call to **ds_compare**( ).

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_COMMON_RESULTS*, in addition to the OM attributes listed in Table 30-5.

Table 30-5. OM Attributes of DS_C_COMPARE_RESULT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_FROM_ENTRY	OM_S_ BOOLEAN	—	1	—
DS_MATCHED	OM_S_ BOOLEAN	—	1	—
DS_OBJECT_NAME	Object(*DS_C_ NAME*)	—	0 or 1	—

- **DS_FROM_ENTRY**

  This attribute indicates whether the assertion is tested against the specified object's entry, rather than a copy of the entry.

- **DS_MATCHED**

  This attribute indicates whether the assertion specified as an argument returns the value **OM_TRUE**. It takes the value **OM_TRUE** if the values are compared and matched; otherwise, it takes the value **OM_FALSE**.

- **DS_OBJECT_NAME**

  This attribute contains the distinguished name of the target object of the operation. It is present if the OM attribute **DS_ALIAS_DEREFERENCED**, inherited from the superclass *DS_C_COMMON_RESULTS*, is **OM_TRUE**.

30.10 DS_C_CONTEXT

An instance of OM class **DS_C_CONTEXT** comprises per-operation arguments that are accepted by most of the interface functions.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-6.

Table 30-6. OM Attributes of DS_C_CONTEXT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
Common Arguments				
DS_EXT	Object(**DS_C_EXT)**	—	0 or more	**NULL**
DS_OPERATION_PROGRESS	Object(**DS_C_OPERATION_PROGRESS)**	—	1	**DS_C_OPERATION_NOT_STARTED**
DS_ALIASED_RDNS	**OM_S_INTEGER**	—	0 or 1	0
Service Controls				
DS_CHAINING_PROHIB	**OM_S_BOOLEAN**	—	1	**OM_TRUE**
DS_DONT_DEREFERENCE_ALIASES	**OM_S_BOOLEAN**	—	1	**OM_FALSE**
DS_DONT_USE_COPY	**OM_S_BOOLEAN**	—	1	**OM_FALSE**
DS_LOCAL_SCOPE	**OM_S_BOOLEAN**	—	1	**OM_FALSE**
DS_PREFER_CHAINING	**OM_S_BOOLEAN**	—	1	**OM_FALSE**
DS_PRIORITY	Enum(**DS_Priority)**	—	1	**DS_MEDIUM**
DS_SCOPE_OF_REFERRAL	Enum(**DS_Scope_of_Referral)**	—	0 or 1	**DS_COUNTRY**
DS_SIZE_LIMIT	**OM_S_INTEGER**	—	0 or 1	-1

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_TIME_ LIMIT	OM_S_ INTEGER	—	0 or 1	-1
Local Controls				
DS_ ASYNCHRONOUS	OM_S_ BOOLEAN	—	1	OM_FALSE
DS_AUTOMATIC_ CONTINUATION	OM_S_ BOOLEAN	—	1	OM_TRUE

The context gathers several arguments passed to interface functions, which are presumed to be relatively static for a given directory user during a particular directory interaction. The context is passed as an argument to each function that interrogates or updates the directory. Although it is generally assumed that the context is changed infrequently, the value of each argument can be changed between every operation if required. The **DS_ASYNCHRONOUS** argument must not be changed. Each argument is represented by one of the OM attributes of the **DS_C_CONTEXT** OM class.

The context contains the common arguments defined in the standards (see *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511), except that all security information is omitted for reasons discussed in Section 29.9. These are made up of a number of service controls explained in the following text, possible extensions in the **DS_EXT** OM attribute, and operation progress and alias dereferencing information in the **DS_OPERATION_PROGRESS** OM attribute. It also contains a number of arguments that provide local control over the interface.

The OM attributes of the **DS_C_CONTEXT** OM class are as follows:

- Common Arguments

  — **DS_EXT**

  This attribute represents any future standardized extensions that need to be applied to the Directory Service operation. The DCE XDS implementation does not evaluate this optional OM attribute.

- **DS_OPERATION_PROGRESS**

  This attribute represents the state that the Directory Service assumes at the start of the operation.

  This OM attribute normally takes its default value, which is the value **DS_OPERATION_NOT_STARTED** described in the **DS_C_OPERATION_PROGRESS** OM class definition.

- **DS_ALIASED_RDNS**

  This attribute indicates to the Directory Service that the object component of the *operation* parameter is created by dereferencing of an alias on an earlier operation attempt. This value is set in the referral response of the previous operation.

- Service Controls

  - **DS_CHAINING_PROHIB**

    This attribute indicates that chaining and other methods of distributing the request around the Directory Service are prohibited.

  - **DS_DONT_DEREFERENCE_ALIASES**

    This attribute indicates that any alias used to identify the target entry of an operation is not dereferenced. This allows interrogation of alias entries (aliases are never dereferenced during updates).

  - **DS_DONT_USE_COPY**

    This attribute indicates that the request can only be satisfied by accessing directory entries, and not by using copies of entries. This includes both copies maintained in other DSAs by bilateral agreement, and copies cached locally.

  - **DS_LOCAL_SCOPE**

    This attribute indicates that the directory request will be satisfied locally. The meaning of this option is configured by an administrator. This option typically restricts the request to a single DSA or DMD.

  - **DS_PREFER_CHAINING**

    This attribute indicates that chaining is preferred to referrals when necessary. The Directory Service is not obliged to follow this preference, and can return a referral even if it is set.

— **DS_PRIORITY**

This attribute indicates the priority, relative to other directory requests, according to which the Directory Service attempts to satisfy the request. This is not a guaranteed service since there is no queuing throughout the directory. Its value must be one of the following:

— **DS_LOW**

— **DS_MEDIUM**

— **DS_HIGH**

— **DS_SCOPE_OF_REFERRAL**

This attribute indicates the part of the directory to which referrals are limited. This includes referral errors and partial outcome qualifiers. Its value must be one of the following:

— **DS_COUNTRY**, meaning DSAs within the country in which the request originates.

— **DS_DMD**, meaning DSAs within the DMD in which the request originates.

DS_SCOPE_OF_REFERRAL is an optional attribute. The lack of this attribute in a **DS_C_CONTEXT** object indicates that the scope is not limited.

— **DS_SIZE_LIMIT**

If present, this attribute indicates the maximum number of objects about which **ds_list**( ) or **ds_search**( ) needs to return information. If this limit is exceeded, information is returned about exactly this number of objects. The objects that are chosen are not specified because this can depend on the timing of interactions between DSAs, among other reasons.

— **DS_TIME_LIMIT**

If present, this attribute indicates the maximum elapsed time, in seconds, within which the service needs to be provided (not the processing time devoted to the request). If this limit is reached, a service error (**DS_E_TIME_LIMIT_EXCEEDED**) is returned, except for the **ds_list**( ) or **ds_search**( ) operations, which return an arbitrary selection of the accumulated results.

- Local Controls
  - **DS_ASYNCHRONOUS** (Optional Functionality)

    The interface currently only operates synchronously as detailed in Section 29.8. There is only one possible value:

    - **OM_FALSE,** meaning that the operation is performed sequentially (synchronously) with the application being blocked until a result or error is returned.
  - **DS_AUTOMATIC_CONTINUATION**

    This attribute indicates the requestor's requirement for continuation reference handling, including referrals and those in partial outcome qualifiers. The value is one of the following:

    - **OM_FALSE**, meaning that the interface returns all continuation references to the application program.
    - **OM_TRUE**, meaning that continuation references are automatically processed, and the subsequent results are returned to the application instead of the continuation references, whenever practical. This is a much simpler option than **OM_FALSE** unless the application has special requirements.

Note: Continuation references can still be returned to the application if, for example, the relevant DSA cannot be contacted.

Applications can assume that an object of OM class **DS_C_CONTEXT**, created with default values of all its OM attributes, works with all the interface functions. The **DS_DEFAULT_CONTEXT** constant can be used as an argument to interface functions instead of creating an OM object with default values.

30.11 DS_C_CONTINUATION_REF

An instance of OM class **DS_C_CONTINUATION_REF** comprises the information that enables a partially completed directory request to be continued; for example, following a referral.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-7.

Table 30–7. OM Attributes of DS_C_CONTINUATION_REF

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ACCESS_ POINTS	Object(**DS_C_ ACCESS_POINT)**	—	1 or more	—
DS_ALIASED_ RDNS	**OM_S_INTEGER**	—	1	—
DS_OPERATION_ PROGRESS	Object(**DS_C_ OPERATION_ PROGRESS)**	—	1	—
DS_RDNS_ RESOLVED	**OM_S_INTEGER**	—	0 or 1	—
DS_TARGET_ OBJECT	Object(*DS_C_ NAME*)	—	1	—

- **DS_ACCESS_POINTS**

  This attribute indicates the names and presentation addresses of the DSAs from where the directory request is continued.

- **DS_ALIASED_RDNS**

  This attribute indicates how many (if any) of the RDNs in the target name are produced by dereferencing an alias. Its value is 0 (zero) if no aliases are dereferenced. This value needs to be used in the **DS_C_CONTEXT** of any continued operation.

- **DS_OPERATION_PROGRESS**

  This attribute indicates the state at which the directory request must be continued. This value needs to be used in the **DS_C_CONTEXT** of any continued operation.

- **DS_RDNS_RESOLVED**

  This attribute indicates the number of RDNs in the supplied object name that are resolved (using internal references), and not just assumed to be correct (using cross-references).

- **DS_TARGET_OBJECT**

  This attribute indicates the name of the object upon which the continuation must focus.

30.12 DS_C_DS_DN

An instance of OM class **DS_C_DS_DN** represents a name of a directory object.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_NAME*, in addition to the OM attribute listed in Table 30-8.

Table 30–8. OM Attribute of DS_C_DS_DN

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_RDNS	Object(**DS_C_DS_ RDN)**	—	0 or more	—

- **DS_RDNS**

  This attribute indicates the sequence of RDNs that define the path through the DIT from its root to the object that the **DS_C_DS_DN** indicates. The **DS_C_DS_DN** of the root of the directory is the null name (no **DS_RDNS** values). The order of the values is significant; the first value is closest to the root, and the last value is the RDN of the object.

30.13 DS_C_DS_RDN

An instance of OM class **DS_C_DS_RDN** is a relative distinguished name. An RDN uniquely identifies an immediate subordinate of an object whose entry is displayed in the DIT.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_RELATIVE_NAME*, in addition to the OM attribute listed in Table 30-9.

Table 30–9. OM Attribute of DS_C_DS_RDN

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_AVAS	Object(**DS_C_AVA)**	—	1 or more	—

- **DS_AVAS**

  This attribute indicates the **DS_AVAS** that are marked by the DIB as components of the object's RDN. The assertion is TRUE of the object but not of any of its siblings, and the attribute type and value are displayed in the object's directory entry. The order of the **DS_AVAS** is not significant.

30.14 DS_C_ENTRY_INFO

An instance of OM class **DS_C_ENTRY_INFO** contains selected information from a single directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_ATTRIBUTE_LIST**, in addition to the OM attributes listed in Table 30-10.

Table 30–10. OM Attributes of DS_C_ENTRY_INFO

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_FROM_ENTRY	**OM_S_ BOOLEAN**	—	1	—
DS_OBJECT_NAME	Object(*DS_C_ NAME*)	—	1	—

The OM attribute **DS_ATTRIBUTES** is inherited from the superclass **DS_C_ATTRIBUTE_LIST**. It contains the information extracted from the directory entry of the target object. The type of each attribute requested and located is indicated in the list as are its values, if types and values are requested.

The OM class-specific OM attributes are as follows:

- **DS_FROM_ENTRY**

  This attribute indicates whether the information is extracted from the specified object's entry, rather than from a copy of the entry.

- **DS_OBJECT_NAME**

  This attribute contains the object's distinguished name.

30.15 DS_C_ENTRY_INFO_SELECTION

An instance of OM class **DS_C_ENTRY_INFO_SELECTION** identifies the information to be extracted from a directory entry.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-11.

Table 30–11. OM Attributes of DS_C_ENTRY_INFO_SELECTION

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALL_ ATTRIBUTES	**OM_S_ BOOLEAN**	—	1	**OM_TRUE**
DS_ ATTRIBUTES_ SELECTED	String(**OM_S_ OBJECT_ IDENTIFIER_ STRING**)	—	0 or more	—
DS_INFO_TYPE	Enum(**DS_ Information_ Type**)	—	1	**DS_TYPES_ AND_VALUES**

- **DS_ALL_ATTRIBUTES**

  This attribute indicates which attributes are relevant. It can take one of the following values:

  — **OM_FALSE**, meaning that information is only requested on those attributes that are listed in the OM attribute **DS_ATTRIBUTES_SELECTED.**

  — **OM_TRUE**, meaning that information is requested on all attributes in the directory entry. Any values of the OM attribute **DS_ATTRIBUTES_SELECTED** are ignored in this case.

- **DS_ATTRIBUTES_SELECTED**

  This attribute lists the types of attributes in the entry from which information will be extracted. The value of this OM attribute is used only if the value of **DS_ALL_ATTRIBUTES** is **OM_FALSE**. If an empty list is supplied, no attribute data is returned that could be used to verify the existence of an entry for a distinguished name.

- **DS_INFO_TYPE**

  This attribute identifies what information will be extracted from each attribute identified. It must take one of the following values:

  — **DS_TYPES_ONLY**, meaning that only the attribute types of the selected attributes in the entry are returned.

— **DS_TYPES_AND_VALUES**, meaning that both the attribute types and the attribute values of the selected attributes in the entry are returned.

30.16 DS_C_ENTRY_MOD

An instance of OM class **DS_C_ENTRY_MOD** describes a single modification to a specified attribute of a directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_ATTRIBUTE**, in addition to the OM attribute listed in Table 30-12.

Table 30–12. OM Attribute of DS_C_ENTRY_MOD

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_MOD_TYPE	Enum(**DS_ Modification_ Type**)	—	1	**DS_ADD_ ATTRIBUTE**

The attribute type to be modified, and the associated values, are specified in the OM attributes **DS_ATTRIBUTE_TYPE** and **DS_ATTRIBUTE_VALUES** that are inherited from the **DS_C_ATTRIBUTE** superclass.

- **DS_MOD_TYPE**

  This attribute identifies the type of modification. It must have one of the following values:

  — **DS_ADD_ATTRIBUTE**, meaning that the specified attribute is absent and will be added with the specified values.

  — **DS_ADD_VALUES**, meaning that the specified attribute is present and that one or more specified values will be added to it.

  — **DS_REMOVE_ATTRIBUTE**, meaning that the specified attribute is present and will be removed. Any values present in the OM attribute **DS_ATTRIBUTE_VALUES** are ignored.

— **DS_REMOVE_VALUES**, meaning that the specified attribute is present and that one or more specified values will be removed from it.

30.17 DS_C_ENTRY_MOD_LIST

An instance of OM class **DS_C_ENTRY_MOD_LIST** comprises a sequence of changes to be made to a directory entry.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attribute listed in Table 30-13.

Table 30–13. OM Attribute of DS_C_ENTRY_MOD_LIST

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_CHANGES	Object(**DS_C_ ENTRY_MOD)**	—	1 or more	—

- **DS_CHANGES**

  This attribute identifies the modifications to be made (in the order specified) to the directory entry of the specified object.

30.18 DS_C_EXT

An instance of OM class **DS_C_EXT** indicates that a standardized extension to the Directory Service is outlined in the standards. Such extensions will only be standardized in post-1988 versions of the standards. Therefore, this OM class is not used by the XDS API and is only included for X/Open conformance purposes.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-14.

Table 30–14. OM Attributes of DS_C_EXT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_CRIT	**OM_S_BOOLEAN**	—	1	**OM_FALSE**
DS_IDENT	**OM_S_INTEGER**	—	1	—
DS_ITEM_ PARAMETERS	Any	—	1	—

- **DS_CRIT**

  This attribute must have one of the following values:

  — **OM_FALSE**, meaning that the originator permits the operation to be performed even if the extension is not available.

  — **OM_TRUE**, meaning that the originator mandates that the extended operation be performed. If the extended operation is not performed, an error is reported.

- **DS_IDENT**

  This attribute identifies the service extension.

- **DS_ITEM_PARAMETERS**

  This OM attribute supplies the parameters of the extension. Its syntax is determined by the value of **DS_IDENT**.

30.19 DS_C_FILTER

An instance of OM class **DS_C_FILTER** is used to select or reject an object on the basis of information in its directory entry. At any point in time, an attribute filter has a value relative to every object. The value is FALSE, TRUE, or undefined. The object is selected if, and only if, the filter's value is TRUE.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-15.

Table 30–15. OM Attributes of DS_C_FILTER

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_FILTER_ITEMS	Object(**DS_C_FILTER_ITEM)**	—	0 or more	—
DS_FILTERS	Object(**DS_C_FILTER)**	—	0 or more	—
DS_FILTER_TYPE	Enum(**DS_Filter_Type**)	—	1	**DS_AND**

A *filter* is a collection of less elaborate filters and elementary **DS_FILTER_ITEMS**, together with a Boolean operation. The filter value is undefined if, and only if, all the component **DS_FILTERS** and **DS_FILTER_ITEMS** are undefined. Otherwise, the filter has a Boolean value with respect to any directory entry, which can be determined by evaluating each of the nested components and combining their values using the Boolean operation. The components whose values are undefined are ignored.

- **DS_FILTER_ITEMS**

  This attribute is a collection of assertions, each relating to just one attribute of a directory entry.

- **DS_FILTERS**

  This attribute is a collection of simpler filters.

- **DS_FILTER_TYPE**

  This attribute is the filter's type. It can have any of the following values:

  — **DS_AND**, meaning that the filter is the logical conjunction of its components. The filter is TRUE unless any of the nested filters or filter items is FALSE. If there are no nested components, the filter is TRUE.

- — **DS_OR**, meaning that the filter is the logical disjunction of its components. The filter is FALSE unless any of the nested filters or filter items is TRUE. If there are no nested components, the filter is FALSE.
- — **DS_NOT**, meaning that the result of this filter is reversed. There must be exactly one nested filter or filter item. The filter is TRUE if the enclosed filter or filter item is FALSE, and is FALSE if the enclosed filter or filter item is TRUE.

30.20 DS_C_FILTER_ITEM

An instance of OM class **DS_C_FILTER_ITEM** is a component of **DS_C_FILTER**. It is an assertion about the existence or values of a single attribute type in a directory entry.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_ATTRIBUTE**, in addition to the OM attributes listed in Table 30-16.

Table 30–16. OM Attributes of DS_C_FILTER_ITEM

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_FILTER_ ITEM_TYPE	Enum(**DS_Filter_ Item_Type)**	—	1	—
DS_FINAL_ SUBSTRING	String(*)	1 or more	0 or 1	—
DS_INITIAL_ SUBSTRING	String(*)	1 or more	0 or 1	—

Note: The following OM attributes are inherited from the superclass **DS_C_ATTRIBUTE**: **DS_ATTRIBUTE_TYPE** and **DS_ATTRIBUTE_VALUES.**

The value of the filter item is undefined in the following cases:

- The **DS_ATTRIBUTE_TYPE** is not known.
- None of the **DS_ATTRIBUTE_VALUES** conform to the attribute syntax defined for that attribute type.
- The **DS_FILTER_ITEM_TYPE** uses a matching rule that is not defined for the attribute syntax.

Access control restrictions can also cause the value to be undefined.

- **DS_FILTER_ITEM_TYPE**

  This attribute identifies the type of filter item and thus, the nature of the filter. The filter item can adopt any of the following values:

  — **DS_APPROXIMATE_MATCH**, meaning that the filter is TRUE if the directory entry contains at least one value of the specified type that is approximately equal to that specified (the meaning of ``approximately equal'' is implementation dependent); otherwise, the filter is FALSE.

  Rules for approximate matching are defined locally. For example, an approximate match may take into account spelling variations or employ phonetic comparison rules. In the absence of any such capabilities, a DSA needs to treat an approximate match as a test for equality. DCE GDS supports phonetic comparisons. There must be exactly one value of the OM attribute **DS_ATTRIBUTE_VALUES.**

  — **DS_EQUALITY**, meaning that the filter is TRUE if the entry contains at least one value of the specified type that is equal to the value specified, according to the equality matching rule in force; otherwise, the filter is FALSE. There must be exactly one value of the OM attribute **DS_ATTRIBUTE_VALUES.**

  — **DS_GREATER_OR_EQUAL**, meaning that the filter item is TRUE if, and only if, at least one value of the attribute is greater than or equal to the supplied value. There must be exactly one value of the OM attribute **DS_ATTRIBUTE_VALUES.**

  — **DS_LESS_OR_EQUAL**, meaning that the filter item is TRUE if, and only if, at least one value of the attribute is less than or equal to the supplied value. There must be exactly one value of the OM attribute **DS_ATTRIBUTE_VALUES.**

- **DS_PRESENT**, meaning that the filter is TRUE if the entry contains an attribute of the specified type; otherwise, it is FALSE.

  Any values of the OM attribute **DS_ATTRIBUTE_VALUES** are ignored.

- **DS_SUBSTRINGS**, meaning that the filter is TRUE if the entry contains at least one value of the specified attribute type that contains all of the specified substrings in the given order; otherwise, the filter is FALSE.

  Any number of substrings can be given as values of the OM attribute **DS_ATTRIBUTE_VALUES**. Similarly, no substrings can be specified. There can also be a substring in **DS_INITIAL_SUBSTRING** or **DS_FINAL_SUBSTRING**, or both. The substrings do not overlap, but they can be separated from each other or from the ends of the attribute value by zero or more string elements. However, at least one attribute of type **DS_ATTRIBUTE_VALUES**, **DS_INITIAL_SUBSTRING**, or **DS_FINAL_SUBSTRING** must exist.

- **DS_FINAL_SUBSTRING**

  If present, this attribute is the substring that will match the final part of an attribute value in the entry. This attribute can only exist if the **DS_FILTER_ITEM_TYPE** is equal to **DS_SUBSTRINGS**.

- **DS_INITIAL_SUBSTRING**

  If present, this attribute is the substring that will match the initial part of an attribute value in the entry.

30.21 DS_C_LIST_INFO

An instance of OM class **DS_C_LIST_INFO** is part of the results of **ds_list( )**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_COMMON_RESULTS*, in addition to the OM attributes listed in Table 30-17.

Table 30–17. OM Attributes of DS_C_LIST_INFO

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_OBJECT_ NAME	Object(*DS_C_ NAME*)	—	0 or 1	—
DS_PARTIAL_ OUTCOME_ QUAL	Object(**DS_C_ PARTIAL_ OUTCOME_ QUAL**)	—	0 or 1	—
DS_ SUBORDINATES	Object(**DS_C_ LIST_INFO_ ITEM**)	—	0 or more	—

- **DS_OBJECT_NAME**

  This attribute is the distinguished name of the target object of the operation. It is present if the OM attribute **DS_ALIAS_DEREFERENCED**, inherited from the superclass *DS_C_COMMON_RESULTS*, is **OM_TRUE**.

- **DS_PARTIAL_OUTCOME_QUAL**

  This OM attribute value is present if the list of subordinates is incomplete. The DSA or DSAs that provided this list did not complete the search for some reason. The partial outcome qualifier contains details of why the search is not completed, and which areas of the directory have not been searched.

- **DS_SUBORDINATES**

  This attribute contains information about zero or more subordinate objects identified by **ds_list( )**.

30.22 DS_C_LIST_INFO_ITEM

An instance of OM class **DS_C_LIST_INFO_ITEM** comprises details returned by **ds_list**( ) of a single subordinate object.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-18.

Table 30–18. OM Attributes of DS_C_LIST_INFO_ITEM

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ALIAS_ENTRY	**OM_S_ BOOLEAN**	—	1	—
DS_FROM_ENTRY	**OM_S_ BOOLEAN**	—	1	—
DS_RDN	Object(*DS_C_ RELATIVE_ NAME*)	—	1	—

- **DS_ALIAS_ENTRY**

  This attribute indicates whether the object is an alias.

- **DS_FROM_ENTRY**

  This attribute indicates whether information about the object was obtained directly from its directory entry, rather than from a copy of the entry.

- **DS_RDN**

  This attribute contains the RDN of the object. If this is the name of an alias entry, as indicated by **DS_ALIAS_ENTRY**, it is not dereferenced.

30.23 DS_C_LIST_RESULT

An instance of OM class **DS_C_LIST_RESULT** comprises the results of a successful call to **ds_list( )**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-19.

Table 30–19. OM Attributes of DS_C_LIST_RESULT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_LIST_INFO	Object(**DS_C_ LIST_INFO)**	—	0 or 1	—
DS_ UNCORRELATED_ LIST_INFO	Object(**DS_C_ LIST_RESULT)**	—	0 or more	—

Note: No instance contains values of both OM attributes.

- **DS_LIST_INFO**

  This attribute contains the full results of **ds_list( )**, or just part of them.

- **DS_UNCORRELATED_LIST_INFO**

  When the DUA requests a protection request of ‘‘signed,’’ the information returned can comprise a number of sets of results originating from, and signed by, different components of the directory. Implementations can reflect this structure by nesting **DS_LIST_RESULT** OM objects as values of this OM attribute. Alternatively, they can collapse all results into a single value of the OM attribute **DS_LIST_INFO**. The DCE Directory Service does not support the optional feature of signed results; therefore, this OM attribute is never present.

30.24 DS_C_NAME

The OM class *DS_C_NAME* represents a name of an object in the directory, or a part of such a name.

It is an abstract class, which has the attributes of its superclass, *OM_C_OBJECT*, and no other OM attributes.

A name uniquely distinguishes the object from all other objects whose entries are displayed in the DIT. However, an object can have more than one name; that is, a name need not be unique. A DN is unique; there are no other DNs that identify the same object. An RDN is part of a name, and only distinguishes the object from others that are its siblings.

Most of the interface functions take a *name* parameter, the value of which must be an instance of one of the subclasses of this OM class. Thus, this OM class is useful for amalgamating all possible representations of names.

The DCE XDS implementation defines one subclass of this OM class, and thus, a single representation for names; that is, **DS_C_DS_DN**, which provides a representation for names, including distinguished names.

30.25 DS_C_OPERATION_PROGRESS

An instance of OM class **DS_C_OPERATION_PROGRESS** specifies the progress or processing state of a directory request.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-20.

Table 30–20. OM Attributes of DS_C_OPERATION_PROGRESS

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_NAME_ RESOLUTION_ PHASE	Enum(**DS_Name_ Resolution_Phase**)	—	1	—
DS_NEXT_ RDN_ TO_BE_ RESOLVED	**OM_S_INTEGER**	—	0 or 1	—

The target name mentioned as follows is the name upon which processing of the directory request is currently focused.

- **DS_NAME_RESOLUTION_PHASE**

  This attribute indicates what phase is reached in handling the target name. It must have one of the following values:

  — **DS_COMPLETED**, meaning that the DSA holding the target object is reached.

  — **DS_NOT_STARTED**, meaning that so far a DSA is not reached with a naming context containing the initial RDNs of the name.

  — **DS_PROCEEDING**, meaning that the initial part of the name has been recognized, although the DSA holding the target object has not yet been reached.

- **DS_NEXT_RDN_TO_BE_RESOLVED**

  This attribute indicates to the DSA which of the RDNs in the target name is next to be resolved. It takes the form of an integer in the range from one to the number of RDNs in the name. This OM attribute only has a value if the value of **DS_NAME_RESOLUTION_PHASE** is **DS_PROCEEDING**.

The constant **DS_OPERATION_NOT_STARTED** can be used in the **DS_C_CONTEXT** of an operation instead of an instance of this OM class.

30.26 DS_C_PARTIAL_OUTCOME_QUAL

An instance of OM class **DS_C_PARTIAL_OUTCOME_QUAL** explains to what extent the results of a call to **ds_list**( ) or **ds_search**( ) are incomplete and why.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-21.

Table 30–21. OM Attributes of a DS_C_PARTIAL_OUTCOME_QUAL

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_LIMIT_ PROBLEM	Enum(**DS_Limit_ Problem)**	—	1	—
DS_ UNAVAILABLE_ CRIT_EXT	**OM_S_ BOOLEAN**	—	1	—
DS_ UNEXPLORED	Object(**DS_C_ CONTINUATION_ REF)**	—	0 or more	—

- **DS_LIMIT_PROBLEM**

  This attribute explains fully or partly why the results are incomplete. It can have one of the following values:

  — **DS_ADMIN_LIMIT_EXCEEDED**, meaning that an administrative limit is reached.

  — **DS_NO_LIMIT_EXCEEDED**, meaning that there is no limit problem.

  — **DS_SIZE_LIMIT_EXCEEDED**, meaning that the maximum number of objects specified as a service control is reached.

  — **DS_TIME_LIMIT_EXCEEDED**, meaning that the maximum number of seconds specified as a service control is reached.

- **DS_UNAVAILABLE_CRIT_EXT**

  If **OM_TRUE**, this attribute indicates that some part of the Directory Service cannot provide a requested critical service extension. The user requested one or more standard service extensions by including values of the OM attribute **DS_EXT** in the **DS_C_CONTEXT** supplied for the operation. Furthermore, the user indicated that some of these extensions are essential by setting the OM attribute **DS_CRIT** in the extension to **OM_TRUE**. Some of the critical extensions cannot be performed by one particular DSA or by a number of DSAs. In general, it is not possible to determine which DSA could not perform which particular extension.

- **DS_UNEXPLORED**

  This attribute identifies any regions of the directory that are left unexplored in such a way that the directory request can be continued. Only continuation references within the scope specified by the **DS_SCOPE_OF_REFERRAL** service control are included.

30.27 DS_C_PRESENTATION_ADDRESS

An instance of OM class **DS_C_PRESENTATION_ADDRESS** is a presentation address of an OSI application entity, which is used for OSI communications with this instance.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ADDRESS*, in addition to the OM attributes listed in Table 30-22.

Table 30–22. OM Attributes of DS_C_PRESENTATION_ADDRESS

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_N_ ADDRESSES	String(**OM_S_ OCTET_STRING**)	—	1 or more	—
DS_P_SELECTOR	String(**OM_S_ OCTET_STRING**)	—	0 or 1	—
DS_S_SELECTOR	String(**OM_S_ OCTET_STRING**)	—	0 or 1	—
DS_T_SELECTOR	String(**OM_S_ OCTET_STRING**)	—	0 or 1	—

- **DS_N_ADDRESSES**

  This attribute is the network addresses of the application entity.

- **DS_P_SELECTOR**

  This attribute is the presentation selector.

- **DS_S_SELECTOR**

  This attribute is the session selector.

- **DS_T_SELECTOR**

  This attribute is the transport selector.

30.28 DS_C_READ_RESULT

An instance of OM class **DS_C_READ_RESULT** comprises the result of a successful call to **ds_read()**. An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_COMMON_RESULTS*, in addition to the OM attribute listed in Table 30-23.

Table 30–23. OM Attribute of DS_C_READ_RESULT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ENTRY	Object(**DS_C_ENTRY_INFO)**	—	1	—

- **DS_ENTRY**

  This attribute contains the information extracted from the directory entry of the target object.

30.29 DS_C_RELATIVE_NAME

The OM class *DS_C_RELATIVE_NAME* represents the RDNs of objects in the directory. It is an abstract class, which has the attributes of its superclass, *OM_C_OBJECT*, and no other OM attributes.

An RDN is part of a name, and only distinguishes the object from others that are its siblings. This OM class is used to accumulate all possible representations of RDNs. An argument of interface functions that is an RDN, or an OM attribute value that is an RDN is an instance of one of the subclasses of this OM class.

The DCE XDS API defines one subclass of this OM class, and thus, a single representation for RDNs; that is, **DS_C_DS_RDN**, which provides a representation for RDNs.

30.30 DS_C_SEARCH_INFO

An instance of OM class **DS_C_SEARCH_INFO** is part of the result of **ds_search( )**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, **OM_Object** and *DS_C_COMMON_RESULTS*, in addition to the OM attributes listed in Table 30-24.

Table 30–24. OM Attributes of DS_C_SEARCH_INFO

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ENTRIES	Object(**DS_C_ENTRY_ INFO)**	—	0 or more	—
DS_OBJECT_ NAME	Object(*DS_C_NAME*)	—	0 or 1	—
DS_PARTIAL_ OUTCOME_ QUAL	Object(**DS_C_ PARTIAL_ OUTCOME_QUAL)**	—	0 or 1	—

- **DS_ENTRIES**

  This attribute contains information about zero or more objects found by **ds_search( )** that matched the given selection criteria.

- **DS_OBJECT_NAME**

  This attribute contains the distinguished name of the target object of the operation. It is present if the OM attribute **DS_ALIAS_DEREFERENCED**, inherited from the superclass *DS_C_COMMON_RESULTS*, is **OM_TRUE**.

- **DS_PARTIAL_OUTCOME_QUAL**

  This OM attribute value is only present if the list of entries is incomplete. The DSA or DSAs that provided this list did not complete the search for some reason. The partial outcome qualifier contains details of why the search was not completed, and which areas of the directory were not searched.

30.31 DS_C_SEARCH_RESULT

An instance of OM class **DS_C_SEARCH_RESULT** comprises the result of a successful call to **ds_search()**.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-25.

Table 30–25. OM Attributes of DS_C_SEARCH_RESULT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_SEARCH_INFO	Object(**DS_C_ SEARCH_INFO)**	—	0 or 1	—
DS_ UNCORRELATED_ SEARCH_INFO	Object(**DS_C_ SEARCH_ RESULT)**	—	0 or more	—

Note: No instance contains values of both OM attributes.

- **DS_SEARCH_INFO**

  This attribute contains the full result of **ds_search()**, or part of the result.

- **DS_UNCORRELATED_SEARCH_INFO**

  When the DUA requests a protection request of ''signed,'' the information returned can comprise a number of sets of results originating from and signed by different components of the Directory Service. Implementations can reflect this structure by nesting **DS_C_SEARCH_RESULT** OM objects as values of this OM attribute. Alternatively, they can collapse all results into a single value of the OM attribute **DS_SEARCH_INFO**. The DCE Directory Service does not support the optional feature of signed results; therefore, this OM attribute is never present.

30.32 DS_C_SESSION

An instance of OM class **DS_C_SESSION** identifies a particular link from the application program to a DUA.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 30-26.

Table 30–26. OM Attributes of DS_C_SESSION

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_DSA_ ADDRESS	Object(*DS_C_ ADDRESS*)	—	0 or 1	*local*[1]
DS_DSA_NAME	Object(*DS_C_ NAME*)	—	0 or 1	*local*[1]
DS_FILE_ DESCRIPTOR	**OM_S_INTEGER**	—	1	See the text
DS_ REQUESTOR	Object(*DS_C_ NAME*)	—	0 or 1	**NULL**

[1] The default values of these OM attributes are set to the address and name of the default DSA entry in the local cache. If this cache entry is not present, then these OM attributes are set to **NULL**.

The **DS_C_SESSION** gathers all the information that describes a particular directory interaction. The parameters that will control such a session are set up in an instance of this OM class, which is then passed as an argument to **ds_bind()**. This sets the OM attributes that describe the actual characteristics of this session, and then starts the session. A session started in this way must pass as the first argument to each interface function. The result of modifying an initiated session is unspecified. Finally, **ds_unbind()** is used to terminate the session, after which the parameters can be modified and a new session started using the same instance, if required. Multiple concurrent sessions can run using multiple instances of this OM class.

The OM attributes of a session are as follows:

- **DS_DSA_ADDRESS**

  This attribute indicates the address of the default DSA named by **DS_DSA_NAME**.

- **DS_DSA_NAME**

  This attribute indicates the distinguished name of the DSA that is used by default to service directory requests.

- **DS_FILE_DESCRIPTOR** (Optional Functionality)

  This OM attribute is not used by DCE XDS and is always set to **DS_NO_VALID_FILE_DESCRIPTOR**.

- **DS_REQUESTOR**

  This attribute is the distinguished name of the user of this Directory Service session.

Applications can assume that an object of OM class **DS_C_SESSION**, created with default values of all its OM attributes, works with all the interface functions. Local administrators need to ensure that this is the case. Such a session can be created by passing the constant **DS_DEFAULT_SESSION** as an argument to **ds_bind( )**.

Chapter 31

XDS Errors

This chapter defines the errors that can arise when using the Directory Service interface and describes the method for reporting them.

Errors are reported to the application program by means of **DS_status**, which is a result of every function (it is *the* function result in the C language binding for most functions). A function that completes successfully returns the value **DS_SUCCESS**, whereas one that is not successful returns an error. The error is a private object containing details of the problem that occurred. The error constant **DS_NO_WORKSPACE** can be returned by all Directory Service functions, except **ds_initialize**( ). **DS_NO_WORKSPACE** is returned if **ds_initialize**( ) is not invoked before calling any other Directory Service function.

Errors are classified into 10 OM classes. The standards (see *The Directory: Abstract Service Definition*, ISO 9594-3, CCITT X.511) classify errors into eight different groups, as follows:

- Abandoned
- Abandon Failed
- Attribute Error
- Name Error

- Referral
- Security Error
- Service Error
- Update Error

The Directory Service interface never returns an Abandoned error. The interface also defines three more kinds of errors: **DS_C_LIBRARY_ERROR**, **DS_C_COMMUNICATIONS_ERROR**, and **DS_C_SYSTEM_ERROR**. Each of these kinds of errors is represented by an OM class. These OM classes are detailed in subsequent sections of this chapter. All of them inherit the OM attribute **DS_PROBLEM** from their superclass *DS_C_ERROR*, which is described first. The OM classes defined in this chapter are part of the Directory Service Package (see Chapter 30).

The **ds_bind**( ) operation returns a Security Error or a Service Error. All other operations can also return the same errors as **ds_bind**( ). Such errors can arise in the course of following an automatic referral list.

31.1 OM Class Hierarchy

This section shows the hierarchical organization of the OM classes defined in this chapter and thus indicates how OM attributes are inherited from superclasses. In the following list, subclassification is indicated by indentation, and the names of abstract OM classes are in italics. Thus, for example, the concrete OM class **DS_C_ATTRIBUTE_PROBLEM** is an immediate subclass of the abstract OM class *DS_C_ERROR*, which in turn is an immediate subclass of the abstract OM class *OM_C_OBJECT*. The *OM_C_OBJECT* class is defined in Chapter 26 of this guide.

OM_C_OBJECT

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_CONTINUATION_REF** (see Chapter 30)
  - — **DS_C_REFERRAL**
- *DS_C_ERROR*
  - — **DS_C_ABANDON_FAILED**

- — **DS_C_ATTRIBUTE_PROBLEM**
- — **DS_C_COMMUNICATIONS_ERROR**
- — **DS_C_LIBRARY_ERROR**
- — **DS_C_NAME_ERROR**
- — **DS_C_SECURITY_ERROR**
- — **DS_C_SERVICE_ERROR**
- — **DS_C_SYSTEM_ERROR**
- — **DS_C_UPDATE_ERROR**

The application program is not permitted to create or modify any instances of any of these OM classes. None of the OM classes in the preceding list are encodable using **om_encode( )** and **om_decode( )**.

DS_C_REFERRAL is not a real error, and it is not a subclass of *DS_C_ERROR*, although it is reported in the same way as a **DS_status** result. A **DS_C_ATTRIBUTE_ERROR**, also not a subclass of *DS_C_ERROR*, is special because it can report several problems at once. Each one is reported in **DS_C_ATTRIBUTE_PROBLEM**, which is a subclass of *DS_C_ERROR*.

31.2 DS_C_ERROR

The OM class *DS_C_ERROR* comprises the parameters common to all errors.

It is an abstract OM class with the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attribute listed in Table 31-1.

Table 31–1. OM Attribute of DS_C_ERROR

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_PROBLEM	Enum(**DS_Problem**)	—	1	—

Details of errors are returned in an instance of a subclass of this OM class. Each such subclass represents a particular kind of error, and is one of the following:

- **DS_C_ABANDON_FAILED**
- **DS_C_ATTRIBUTE_PROBLEM**
- **DS_C_COMMUNICATIONS_ERROR**
- **DS_C_LIBRARY_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**
- **DS_C_SYSTEM_ERROR**
- **DS_C_UPDATE_ERROR**

A number of possible values are defined for these subclasses. DCE XDS does not return other values for error conditions described in this chapter. Information on system errors can be found in Section 31.12. Each of the following standard values is described under the relevant error OM class:

- **DS_E_ADMIN_LIMIT_EXCEEDED**
- **DS_E_AFFECTS_MULTIPLE_DSAS**
- **DS_E_ALIAS_DEREFERENCING_PROBLEM**
- **DS_E_ALIAS_PROBLEM**
- **DS_E_ATTRIBUTE_OR_VALUE_EXISTS**
- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CLASS**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_BAD_WORKSPACE**
- **DS_E_BUSY**
- **DS_E_CANNOT_ABANDON**

- **DS_E_CHAINING_REQUIRED**
- **DS_E_COMMUNICATIONS_PROBLEM**
- **DS_E_CONSTRAINT_VIOLATION**
- **DS_E_DIT_ERROR**
- **DS_E_ENTRY_EXISTS**
- **DS_E_INAPPROP_AUTHENTICATION**
- **DS_E_INAPPROP_MATCHING**
- **DS_E_INSUFFICIENT_ACCESS_RIGHTS**
- **DS_E_INVALID_ATTRIBUTE_SYNTAX**
- **DS_E_INVALID_ATTRIBUTE_VALUE**
- **DS_E_INVALID_CREDENTIALS**
- **DS_E_INVALID_REF**
- **DS_E_INVALID_SIGNATURE**
- **DS_E_LOOP_DETECTED**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_MIXED_SYNCHRONOUS**
- **DS_E_NAMING_VIOLATION**
- **DS_E_NO_INFO**
- **DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE**
- **DS_E_NO_SUCH_OBJECT**
- **DS_E_NO_SUCH_OPERATION**
- **DS_E_NOT_ALLOWED_ON_NON_LEAF**
- **DS_E_NOT_ALLOWED_ON_RDN**
- **DS_E_NOT_SUPPORTED**
- **DS_E_OBJECT_CLASS_MOD_PROHIB**
- **DS_E_OBJECT_CLASS_VIOLATION**

- **DS_E_OUT_OF_SCOPE**
- **DS_E_PROTECTION_REQUIRED**
- **DS_E_TIME_LIMIT_EXCEEDED**
- **DS_E_TOO_LATE**
- **DS_E_TOO_MANY_OPERATIONS**
- **DS_E_TOO_MANY_SESSIONS**
- **DS_E_UNABLE_TO_PROCEED**
- **DS_E_UNAVAILABLE**
- **DS_E_UNAVAILABLE_CRIT_EXT**
- **DS_E_UNDEFINED_ATTRIBUTE_TYPE**
- **DS_E_UNWILLING_TO_PERFORM**

31.3 DS_C_ABANDON_FAILED

An instance of OM class **DS_C_ABANDON_FAILED** reports a problem encountered during an attempt to abandon an operation.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the problem. Its value is one of the following:

- **DS_E_CANNOT_ABANDON**

  An attempt is made to abandon an operation for which this is prohibited, or the abandon cannot be performed.

- **DS_E_NO_SUCH_OPERATION**

  The Directory Service has no knowledge of the operation that is to be abandoned.

- **DS_E_TOO_LATE**

  The operation is already completed, either successfully or erroneously.

The Directory Abandon operation is not supported by DCE. Thus, a **ds_abandon**() XDS call always returns a **DS_E_TOO_LATE** error for the **DS_C_ABANDON_FAILED** OM class.

31.4 DS_C_ATTRIBUTE_ERROR

An instance of OM class **DS_C_ATTRIBUTE_ERROR** reports an attribute-related Directory Service error.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 31-2.

Table 31–2. OM Attributes of DS_C_ATTRIBUTE_ERROR

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_OBJECT_ NAME	Object(*DS_C_ NAME*)	—	1	—
DS_PROBLEM	Object(**DS_C_ ATTRIBUTE_ PROBLEM)**	—	1 or more	—

- **DS_OBJECT_NAME**

  This attribute contains the name of the directory entry to which the operation is applied when the failure occurs.

- **DS_PROBLEMS**

  This attribute documents the attribute-related problems encountered. Uniquely, a **DS_C_ATTRIBUTE_ERROR** can report several problems at once. All problems are related to the preceding object.

31.5 DS_C_ATTRIBUTE_PROBLEM

An instance of OM class **DS_C_ATTRIBUTE_PROBLEM** documents one attribute-related problem encountered while performing an operation as requested on a particular occasion.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, in addition to the OM attributes listed in Table 31-3.

Table 31–3. OM Attributes of DS_C_ATTRIBUTE_PROBLEM

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ATTRIBUTE_TYPE	String(**OM_S_OBJECT_IDENTIFIER_STRING**)	—	1	—
DS_ATTRIBUTE_VALUE	Any	—	0 or 1	—

- **DS_ATTRIBUTE_TYPE**

  This attribute identifies the type of attribute with which the problem is associated.

- **DS_ATTRIBUTE_VALUE**

  This attribute specifies the attribute value with which the problem is associated. Its syntax is determined by the value' of **DS_ATTRIBUTE_TYPE**. This OM attribute is present if it is necessary to avoid ambiguity.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the problem. Its value is one of the following:

- **DS_E_ATTRIBUTE_OR_VALUE_EXISTS**

  An attempt is made to add an attribute or value that is already present in the directory entry in question.

- **DS_E_CONSTRAINT_VIOLATION**

  The attribute or attribute value does not conform to the constraints imposed by the standards (see *The Directory: Models*, ISO 9594-2, CCITT X.501) or by the attribute definition; for example, the value exceeds its upper bound.

- **DS_E_INAPPROP_MATCHING**

  An attempt is made to use a matching rule that is not defined for the attribute type.

- **DS_E_INVALID_ATTRIBUTE_SYNTAX**

  A value presented as an argument does not conform to the attribute syntax of the attribute type.

- **DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE**

  The specified attribute or value is not found in the directory entry in question. This is only reported by a **ds_read**( ) or **ds_search**( ) operation if an explicit list of attributes is specified by the *selection* parameter, but none of them are present in the entry.

- **DS_E_UNDEFINED_ATTRIBUTE_TYPE**

  The attribute type, which is supplied as an argument to **ds_add_entry**( ) or **ds_modify_entry**( ), is undefined.

31.6 DS_C_COMMUNICATIONS_ERROR

An instance of OM class **DS_C_COMMUNICATIONS_ERROR** reports an error occurring in the other OSI services supporting the Directory Service.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

Communications errors include those arising in remote operation, association control, presentation, session, and transport.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the problem. Its value is **DS_E_COMMUNICATIONS_PROBLEM**.

31.7 DS_C_LIBRARY_ERROR

An instance of OM class **DS_C_LIBRARY_ERROR** reports an error detected by the interface function library.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

Each function has several possible errors that can be detected by the library itself, and that are returned directly by the subroutine. These errors occur when the library itself is incapable of performing an action, submitting a service request, or deciphering a response from the Directory Service.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the particular library error that occurred. (In the *OSF DCE Application Development Reference*, the ERRORS section of each function description lists the errors that the respective function can return.) Its value is one of the following:

- **DS_E_BAD_ARGUMENT**

  A bad argument (other than *name*) was supplied. Use of an instance of OM class **DS_C_ATTRIBUTE** with no values of the OM attribute **DS_ATTRIBUTE_VALUES** as an input argument to a Directory Service function results in this error. This is because directory attributes always have at least one value.

- **DS_E_BAD_CLASS**

  The OM class of an argument is not supported for this operation.

- **DS_E_BAD_CONTEXT**

  An invalid *context* parameter was supplied.

- **DS_E_BAD_NAME**

  An invalid *name* parameter was supplied.

- **DS_E_BAD_SESSION**

  An invalid *session* parameter was supplied.

- **DS_E_MISCELLANEOUS**

  A miscellaneous error occurred in interacting with the Directory Service. This error is returned if the interface cannot clear a transient system error by retrying the affected system call.

- **DS_E_MISSING_TYPE**

  The attribute type is not included in an AVA that is passed as part of a distinguished name argument.

- **DS_E_MIXED_SYNCHRONOUS**

  An attempt is made to start a synchronous operation when there are outstanding asynchronous operations.

- **DS_E_NOT_SUPPORTED**

  An attempt is made to use optional functionality, which is not available in this implementation.

- **DS_E_TOO_MANY_OPERATIONS**

  No more Directory Service operations can be performed until at least one asynchronous operation is completed.

- **DS_E_TOO_MANY_SESSIONS**

  No more Directory Service sessions can be started.

31.8 DS_C_NAME_ERROR

An instance of OM class **DS_C_NAME_ERROR** reports a name-related Directory Service error.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, in addition to the OM attribute listed in Table 31-4.

Table 31–4. OM Attribute of DS_C_NAME_ERROR

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_MATCHED	Object(*DS_C_NAME*)	—	1	—

- **DS_MATCHED**

  This attribute identifies the initial part (up to, but excluding, the first RDN that is unrecognized) of the name that is supplied, or of the name resulting from dereferencing an alias. It names the lowest entry (object or alias) in the DIT that is matched.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of the failure. Its value is one of the following:

- **DS_E_ALIAS_DEREFERENCING_PROBLEM**

  An alias is encountered where an alias is not permitted; for example, in a modification operation when the **DS_DONT_DEREFERENCE_ALIASES** service control is set, or when one alias points to another alias.

- **DS_E_ALIAS_PROBLEM**

  An alias is dereferenced that names an object that does not exist; that is, for which no directory entry can be found.

- **DS_E_INVALID_ATTRIBUTE_VALUE**

  The attribute value in an AVA of an RDN contained in the name does not conform to the attribute syntax prescribed for the attribute type in the AVA. This problem is called **invalidAttributeSyntax** in the standards, but that name is used only for a **DS_C_ATTRIBUTE_PROBLEM** in this interface.

- **DS_E_NO_SUCH_OBJECT**

  The specified name does not match the name of any object in the directory.

31.9 DS_C_REFERRAL

An instance of OM class **DS_C_REFERRAL** reports failure to perform an operation and redirects the requestor to one or more access points better equipped to perform the operation.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_CONTINUATION_REF**, and no additional OM attributes.

The referral is a continuation reference by means of which the operation can progress.

31.10 DS_C_SECURITY_ERROR

An instance of OM class **DS_C_SECURITY_ERROR** reports a security-related Directory Service error.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of this failure. Its value is one of the following:

- **DS_E_INAPPROP_AUTHENTICATION**

  The level of security attached to the requestor's credentials is inconsistent with the level of protection requested; for example, simple credentials are supplied whereas strong credentials are required.

- **DS_E_INSUFFICIENT_ACCESS_RIGHTS**

  The requestor does not have permission to perform the operation. A **ds_read( )** operation only returns this error when access rights preclude the reading of all requested attribute values.

- **DS_E_INVALID_CREDENTIALS**

  The requestor's credentials are invalid.

- **DS_E_INVALID_SIGNATURE**

  The signature affixed to the request is invalid.

- **DS_E_NO_INFO**

  The request produced a security error for which no other information is available.

- **DS_E_PROTECTION_REQUIRED**

  The Directory Service is unwilling to perform the operation because it is unsigned.

31.11 DS_C_SERVICE_ERROR

An instance of OM class **DS_C_SERVICE_ERROR** reports a Directory Service error related to the provision of the service.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of the failure. Its value is one of the following:

- **DS_E_ADMIN_LIMIT_EXCEEDED**

  The operation could not be performed within the administrative constraints on the directory, and no partial results are available.

- **DS_E_BUSY**

  Some part of the Directory Service is temporarily too busy to perform the operation, but will be available after a short while.

- **DS_E_CHAINING_REQUIRED**

  Chaining is required to perform the operation, but is prohibited by the **DS_CHAINING_PROHIBITED** service control.

- **DS_E_DIT_ERROR**

  An inconsistency is detected in the DIT that can be localized to a particular entry or set of entries.

- **DS_E_INVALID_REF**

  The DSA is unable to perform the request as directed; that is, via **DS_C_OPERATION_PROGRESS** in the **DS_C_CONTEXT**. This can be due to an invalid referral.

- **DS_E_LOOP_DETECTED**

  A DSA detected a loop within the directory.

- **DS_E_OUT_OF_SCOPE**

  The Directory Service cannot provide a referral or partial outcome qualifier within the required scope.

- **DS_E_TIME_LIMIT_EXCEEDED**

  The operation could not be performed within the time specified by the **DS_TIME_LIMIT** service control, and no partial results are available.

- **DS_E_UNABLE_TO_PROCEED**

  A DSA without administrative authority over a particular naming context is asked to resolve a name in that context.

- **DS_E_UNAVAILABLE**

  Some part of the directory is not currently available.

- **DS_E_UNAVAILABLE_CRIT_EXT**

  One or more critical extensions are requested, but are not available.

- **DS_E_UNWILLING_TO_PERFORM**

  Some part of the Directory Service is not willing to perform the operation because it requires excessive resources, or because doing so violates administrative policy.

31.12 DS_C_SYSTEM_ERROR

An instance of OM class **DS_C_SYSTEM_ERROR** reports an error that occurred in the underlying operating system.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes, although there can be additional implementation-defined OM attributes.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of the failure. Its value is the same as that of **errno** defined in the C language.

The standard names of system errors are defined in Volume 2 of the *X/Open Portability Guide*.

If such an error persists, a **DS_C_LIBRARY_ERROR** (**DS_E_MISCELLANEOUS**) is reported.

31.13 DS_C_UPDATE_ERROR

An instance of OM class **DS_C_UPDATE_ERROR** reports a Directory Service error peculiar to a modification operation.

An application is not permitted to create or modify instances of this OM class. An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and *DS_C_ERROR*, and no additional OM attributes.

The OM attribute **DS_PROBLEM**, which is inherited from the superclass *DS_C_ERROR*, identifies the cause of the failure. Its value is one of the following:

- **DS_E_AFFECTS_MULTIPLE_DSAS**

  The modification affects several DSAs, and such a modification is prohibited. Local agreement between DSAs can allow modifications that affect multiple DSAs; for example, adding entries whose immediate superior entry is in a different DSA. This problem is not reported in such cases.

- **DS_E_ENTRY_EXISTS**

  The name passed to **ds_add_entry**( ) already exists.

- **DS_E_NAMING_VIOLATION**

  The modification leaves the DIT structured incorrectly. This means that it adds an entry as the subordinate of an alias; or in a region of the DIT not permitted to a member of its object class; or it defines an RDN that includes a forbidden attribute type.

- **DS_E_NOT_ALLOWED_ON_NON_LEAF**

  The modification would be to an interior node of the DIT, and such a modification is prohibited.

- **DS_E_NOT_ALLOWED_ON_RDN**

  The modification alters an object's RDN.

- **DS_E_OBJECT_CLASS_MOD_PROHIB**

  The modification alters an entry's object class attribute.

- **DS_E_OBJECT_CLASS_VIOLATION**

  The modification leaves a directory entry inconsistent with its object class definition.

Chapter 32

Basic Directory Contents Package

The standards define a number of attribute types (known as the *selected attribute types*), attribute syntaxes, attribute sets, and object classes (known as the *selected object classes*). These definitions allow the creation and maintenance of directory entries for a number of common objects so that the representation of all such objects is the same throughout the directory.[1] They include such objects as Country, Person, and Organization.

This chapter outlines names for each of these items, and defines OM classes to represent those that are not represented directly by OM syntaxes. The values of attributes in the directory are not restricted to those discussed in this chapter, and new attribute types and syntaxes can be created at any time. (For further information on how the values of other syntaxes are represented in the interface, see Section 29.6.1.)

The constants and OM classes in this chapter are defined in addition to those in Chapter 30, since they are not essential to the working of the interface,

[1] These definitions are chiefly in *The Directory: Selected Attribute Types* (ISO 9594-6, CCITT X.520) and *The Directory: Selected Object Classes* (ISO 9594-7, CCITT X.521) with additional material in *The Directory: Overview of Concepts, Models, and Services* (ISO 9594-1, CCITT X.500) and *The Directory: Authentication Framework* (ISO 9594-8, CCITT X.509).

but instead allow directory entries to be utilized. The definitions belong to the Basic Directory Contents Package (BDCP), which is supported by the DCE XDS API following negotiation of its use with **ds_version( )**.

Note: The definitions for the Global Directory Service Package are provided in Chapter 34. The definitions for the MHS Directory User Package are provided in Chapter 33.

The object identifier associated with the BDC Package is **{iso(1) identified-organization(3) icd-ecma(0012) member-company(2) dec(1011) xopen(28) bdcp(1)}** with the following encoding:

\x2B\xC\x2\x87\x73\x1C\x1

This identifier is represented by the constant **DS_BASIC_DIR_CONTENTS_PKG**. The C constants associated with this package are in the **xdsbdcp.h** header file (see the *OSF DCE Application Development Reference*).

The concepts and notation used are introduced in Section 30.1. A complete explanation of the meaning of the attributes and object classes is not given since this is beyond the scope of this guide. The purpose here is simply to present the representation of these items in the interface.

The selected attribute types are presented first, followed by the selected object classes. Next, the OM class hierarchy and OM class definitions required to support the selected attribute types are presented.

32.1 Selected Attribute Types

This section presents the attribute types defined in the standards that are to be used in directory entries. Each directory entry is composed of a number of attributes, each of which comprises an attribute type together with one or more attribute values. The form of each value of an attribute is determined by the attribute syntax associated with the attribute's type.

In the interface, attributes are displayed as instances of OM class **DS_C_ATTRIBUTE** with the attribute type represented as the value of the OM attribute **DS_ATTRIBUTE_TYPE**, and the attribute value (or values) represented as the value (or values) of the OM attribute **DS_ATTRIBUTE_VALUES**. Each attribute type has an object identifier, assigned in the standards, which is the value of the OM attribute **DS_ATTRIBUTE_TYPE**. These object identifiers are represented in the interface by constants with the same name as the directory attribute, and are prefixed with **DS_A_** so that they can be easily identified.

Table 32-1 shows the names of the attribute types defined in the standards, together with the Basic Encoding Rules (BERs) for encoding of the object identifiers associated with each of them. Table 32-2 shows the names of the attribute types, together with the OM value syntax that is used in the interface to represent values of that attribute type. Table 32-2 also includes the range of lengths permitted for the string types. This indicates whether the attribute can be multivalued and which matching rules are provided for the syntax. Following the table is a brief description of each attribute.

The standards define matching rules that are used for deciding whether two values are equal (E), for ordering (O) two values, and for identifying one value as a substring (S) of another in Directory Service operations. Specific matching rules are given in this chapter for certain attributes. In addition, the following general rules apply as indicated:

- All attribute values whose syntax is as follows:

  — String(**OM_S_NUMERIC_STRING**)

  — String(**OM_S_PRINTABLE_STRING**)

  — String(**OM_S_TELETEX_STRING**)

  are considered insignificant for the following reasons:

  — Differences caused by the presence of spaces preceding the first printing character

  — Spaces following the last printing character

  — More than one consecutive space anywhere within the value

- For all attribute values whose syntax is String(**OM_S_TELETEX_STRING**), differences in the case of alphabetical characters are considered insignificant.

Note: The third and fourth columns of Table 32-1 contain the contents octets of the BER encoding of the object identifier. All these object identifiers stem from the root **{joint-iso-ccitt(2) ds(5) attributeType(4)}**.

Table 32–1. Object Identifiers for Selected Attribute Types

Package	Attribute Type	Object Identifier BER	
		Decimal	**Hexadecimal**
BDCP	**DS_A_ALIASED_OBJECT_NAME**	85, 4, 1	\x55\x04\x01
BDCP	**DS_A_BUSINESS_CATEGORY**	85, 4, 15	\x55\x04\x0F
BDCP	**DS_A_COMMON_NAME**	85, 4, 3	\x55\x04\x03
BDCP	**DS_A_COUNTRY_NAME**	85, 4, 6	\x55\x04\x06
BDCP	**DS_A_DESCRIPTION**	85, 4, 13	\x55\x04\x0D
BDCP	**DS_A_DEST_INDICATOR**	85, 4, 27	\x55\x04\x1B
BDCP	**DS_A_FACSIMILE_PHONE_NBR**	85, 4, 23	\x55\x04\x17
BDCP	**DS_A_INTERNAT_ISDN_NBR**	85, 4, 25	\x55\x04\x19
BDCP	**DS_A_KNOWLEDGE_INFO**	85, 4, 2	\x55\x04\x02
BDCP	**DS_A_LOCALITY_NAME**	85, 4, 7	\x55\x04\x07
BDCP	**DS_A_MEMBER**	85, 4, 31	\x55\x04\x1F
BDCP	**DS_A_OBJECT_CLASS**	85, 4, 0	\x55\x04\x00
BDCP	**DS_A_ORG_NAME**	85, 4, 10	\x55\x04\x0A
BDCP	**DS_A_ORG_UNIT_NAME**	85, 4, 11	\x55\x04\x0B
BDCP	**DS_A_OWNER**	85, 4, 32	\x55\x04\x20
BDCP	**DS_A_PHYS_DELIV_OFF_NAME**	85, 4, 19	\x55\x04\x13
BDCP	**DS_A_POST_OFFICE_BOX**	85, 4, 18	\x55\x04\x12
BDCP	**DS_A_POSTAL_ADDRESS**	85, 4, 16	\x55\x04\x10
BDCP	**DS_A_POSTAL_CODE**	85, 4, 17	\x55\x04\x11
BDCP	**DS_A_PREF_DELIV_METHOD**	85, 4, 28	\x55\x04\x1C
BDCP	**DS_A_PRESENTATION_ADDRESS**	85, 4, 29	\x55\x04\x1D
BDCP	**DS_A_REGISTERED_ADDRESS**	85, 4, 26	\x55\x04\x1A
BDCP	**DS_A_ROLE_OCCUPANT**	85, 4, 33	\x55\x04\x21

Package	Attribute Type	Object Identifier BER	
		Decimal	Hexadecimal
BDCP	**DS_A_SEARCH_GUIDE**	85, 4, 14	\x55\x04\x0E
BDCP	**DS_A_SEE_ALSO**	85, 4, 34	\x55\x04\x22
BDCP	**DS_A_SERIAL_NBR**	85, 4, 5	\x55\x04\x05
BDCP	**DS_A_STATE_OR_PROV_NAME**	85, 4, 8	\x55\x04\x08
BDCP	**DS_A_STREET_ADDRESS**	85, 4, 9	\x55\x04\x09
BDCP	**DS_A_SUPPORT_APPLIC_CONTEXT**	85, 4, 3	\x55\x04\x1E
BDCP	**DS_A_SURNAME**	85, 4, 4	\x55\x04\x04
BDCP	**DS_A_PHONE_NBR**	85, 4, 20	\x55\x04\x14
BDCP	**DS_A_TELETEX_TERM_IDENT**	85, 4, 22	\x55\x04\x16
BDCP	**DS_A_TELEX_NBR**	85, 4, 21	\x55\x04\x15
BDCP	**DS_A_TITLE**	85, 4, 12	\x55\x04\x0C
BDCP	**DS_A_USER_PASSWORD**	85, 4, 35	\x55\x04\x23
BDCP	**DS_A_X121_ADDRESS**	85, 4, 24	\x55\x04\x18

Table 32–2. Representation of Values for Selected Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-valued	Matching Rules
DS_A_ALIASED_OBJECT_NAME	Object(*DS_C_NAME*)	—	No	E
DS_A_BUSINESS_CATEGORY	String(**OM_S_TELETEX_STRING**)	1-128	Yes	E, S
DS_A_COMMON_NAME	String(**OM_S_TELETEX_STRING**)	1-64	Yes	E, S
DS_A_COUNTRY_NAME	String(**OM_S_PRINTABLE_STRING**)[1]	2	No	E
DS_A_DESCRIPTION	String(**OM_S_TELETEX_STRING**)	1-1024	Yes	E, S
DS_A_DEST_INDICATOR	String(**OM_S_PRINTABLE_STRING**)[2]	1-128	Yes	E, S
DS_A_FACSIMILE_PHONE_NBR	Object(**DS_C_FACSIMILE_PHONE_NBR**)	—	Yes	—
DS_A_INTERNAT_ISDN_NBR	String(**OM_S_NUMERIC_STRING**)[3]	1-16	Yes	—
DS_A_KNOWLEDGE_INFO	String(**OM_S_TELETEX_STRING**)	—	Yes	E, S
DS_A_LOCALITY_NAME	String(**OM_S_TELETEX_STRING**)	1-128	Yes	E, S
DS_A_MEMBER	Object(*DS_C_NAME*)_	—	Yes	E

Attribute Type	**OM Value Syntax**	**Value Length**	**Multi-valued**	**Matching Rules**
DS_A_OBJECT_CLASS	String(**OM_S_OBJECT_IDENTIFIER_STRING**)	—	Yes	E
DS_A_ORG_NAME	String(**OM_S_TELETEX_STRING**)	1-64	Yes	E, S
DS_A_ORG_UNIT_NAME	String(**OM_S_TELETEX_STRING**)	1-64	Yes	E, S
DS_A_OWNER	Object(*DS_C_NAME*)	—	Yes	E
DS_A_PHYS_DELIV_OFF_NAME	String(**OM_S_TELETEX_STRING**)	1-128	Yes	E, S
DS_A_POST_OFFICE_BOX	String(**OM_S_TELETEX_STRING**)	1-40	Yes	E, S
DS_A_POSTAL_ADDRESS	Object(**DS_C_POSTAL_ADDRESS**)	—	Yes	E
DS_A_POSTAL_CODE	String(**OM_S_TELETEX_STRING**)	1-40	Yes	E, S
DS_A_PREF_DELIV_METHOD	Enum(**DS_Preferred Delivery_Method**)	—	Yes	—
DS_A_PRESENTATION_ADDRESS	Object(**DS_C_PRESENTATION_ADDRESS**)	—	No	E
DS_A_REGISTERED_ADDRESS	Object(**DS_C_POSTAL_ADDRESS**)	—	Yes	—

Attribute Type	OM Value Syntax	Value Length	Multi-valued	Matching Rules
DS_A_ROLE_OCCUPANT	Object(*DS_C_NAME*)	—	Yes	E
DS_A_SEARCH_GUIDE	Object(**DS_C_SEARCH_GUIDE**)	—	Yes	—
DS_A_SEE_ALSO	Object(*DS_C_NAME*)	—	Yes	E
DS_A_SERIAL_NBR	String(**OM_S_PRINTABLE_STRING**)	1-64	Yes	E, S
DS_A_STATE_OR_PROV_NAME	String(**OM_S_TELETEX_STRING**)	1-128	Yes	E, S
DS_A_STREET_ADDRESS	String(**OM_S_TELETEX_STRING**)	1-128	Yes	E, S
DS_A_SUPPORT_APPLIC_CONTEXT	String(**OM_S_OBJECT_IDENTIFIER_STRING**)	—	Yes	E
DS_A_SURNAME	String(**OM_S_TELETEX_STRING**)	1-64	Yes	E, S
DS_A_PHONE_NBR	String(**OM_S_PRINTABLE_STRING**)[4]	1-32	Yes	E, S
DS_A_TELETEX_TERM_IDENT	Object(**DS_C_TELETEX_TERM_IDENT**)	—	Yes	—
DS_A_TELEX_NBR	Object(**DS_C_TELEX_NBR**)	—	Yes	—

Attribute Type	OM Value Syntax	Value Length	Multi-valued	Matching Rules
DS_A_TITLE	String(**OM_S_TELETEX_STRING)**	1-64	Yes	E, S
DS_A_USER_PASSWORD	String(**OM_S_OCTET_STRING)**	0-128	Yes	—
DS_A_X121_ADDRESS	String(**OM_S_NUMERIC_STRING)**[5]	1-15	Yes	E, S
[1] As permitted by ISO 3166. [2] As permitted by Recommendations F.1 and F.31. [3] As permitted by E.164. [4] As permitted by E.123 (for example, +44 582 10101). [5] As permitted by X.121.				

Throughout the descriptions that follow, the term *object* indicates the directory object whose directory entry contains the corresponding directory attributes.

- **DS_A_ALIASED_OBJECT_NAME**

  This attribute occurs only in alias entries. It assigns the Distinguished Name (DN) of the object provided with an alias using the entry in which this attribute occurs. An alias is an alternative to an object's DN. Any object can (but need not) have one or more aliases. The Directory Service is said to dereference an alias whenever it replaces the alias during name processing with the distinguished name associated with it by means of this attribute.

- **DS_A_BUSINESS_CATEGORY**

  This attribute provides descriptions of the businesses in which the object is engaged.

- **DS_A_COMMON_NAME**

  This attribute provides the names by which the object is commonly known in the context defined by its position in the DIT. The names can conform to the naming convention of the country or culture with which the object is associated. They can be ambiguous.

- **DS_A_COUNTRY_NAME**

  This attribute identifies the country in which the object is located or with which it is associated in some other important way. The matching rules require that differences in the case of alphabetical characters be considered insignificant. It has a length of two characters and its values are those listed in ISO 3166.

- **DS_A_DESCRIPTION**

  This attribute gives informative descriptions of the object.

- **DS_A_DEST_INDICATOR**

  This attribute provides the country-city pairs by means of which the object can be reached via the public telegram service. The matching rules require that differences in the case of alphabetical characters be considered insignificant.

- **DS_A_FACSIMILE_PHONE_NBR**

  This attribute provides the telephone numbers for facsimile terminals (and their parameters, if required) by means of which the object can be reached or with which it is associated in some other important way.

- **DS_A_INTERNAT_ISDN_NBR**

  This attribute provides the international ISDN numbers by means of which the object can be reached or with which it is associated in some other important way. The matching rules require that differences caused by the presence of spaces be considered insignificant.

- **DS_A_KNOWLEDGE_INFO**

  This attribute occurs only in entries that describe a DSA. It provides a human-intelligible accumulated description of the directory knowledge possessed by the DSA.

- **DS_A_LOCALITY_NAME**

  This attribute identifies geographical areas or localities. When used as part of a directory name, it specifies the localities in which the object is located or with which it is associated in some other important way.

- **DS_A_MEMBER**

  This attribute gives the names of objects that are considered members of the present object; for example, a distribution list for electronic mail.

- **DS_A_OBJECT_CLASS**

  This attribute identifies the object classes to which the object belongs, and also identifies their superclasses. All such object classes that have object identifiers assigned to them are present, except that object class **DS_O_TOP** need not (but can) be present provided that some other value is present. This attribute must be present in every entry and cannot be modified. For a further discussion, see Section 32.3.

- **DS_A_ORG_NAME**

  This attribute identifies organizations. When used as part of a directory name, it specifies an organization with which the object is affiliated. Several values can identify the same organization in different ways.

- **DS_A_ORG_UNIT_NAME**

  This attribute identifies organizational units. When used as part of a directory name, it specifies an organizational unit with which the object is affiliated. The units are understood to be parts of the organization that the **DS_A_ORG_NAME** attribute indicates. Several values can identify the same unit in different ways.

- **DS_A_OWNER**

  This attribute gives the names of objects that have responsibility for the object.

- **DS_A_PHYS_DELIV_OFF_NAME**

  This attribute gives the names of cities, towns, villages, and so on, that contain physical delivery offices through which the object can take delivery of physical mail.

- **DS_A_POST_OFFICE_BOX**

  This attribute identifies post office boxes at which the object can take delivery of physical mail. This information is also displayed as part of the **DS_A_POSTAL_ADDRESS** attribute, if it is present.

- **DS_A_POSTAL_ADDRESS**

  This attribute gives the postal addresses at which the object can take delivery of physical mail. The matching rules require that differences in the case of alphabetical characters be considered insignificant.

- **DS_A_POSTAL_CODE**

  This attribute gives the postal codes that are assigned to areas or buildings through which the object can take delivery of physical mail. This information is also displayed as part of the **DS_A_POSTAL_ADDRESS** attribute, if it is present.

- **DS_A_PREF_DELIV_METHOD**

  This attribute gives the object's preferred methods of communication, in the order of preference. The values are as follows:

  — **DS_ANY_DELIV_METHOD**, meaning that the object has no preference.

  — **DS_G3_FACSIMILE_DELIV**, meaning via the Group 3 facsimile.

  — **DS_G4_FACSIMILE_DELIV**, meaning via the Group 4 facsimile.

  — **DS_IA5_TERMINAL_DELIV**, meaning via the IA5 text.

  — **DS_MHS_DELIV**, meaning via X.400.

  — **DS_PHYS_DELIV**, meaning via the postal or other physical delivery system.

  — **DS_PHONE_DELIV**, meaning via telephone.

  — **DS_TELETEX_DELIV**, meaning via teletex.

  — **DS_TELEX_DELIV**, meaning via telex.

  — **DS_VIDEOTEX_DELIV**, meaning via videotex.

- **DS_A_PRESENTATION_ADDRESS**

  This attribute contains the OSI presentation address of the object, which is an OSI application entity. The matching rule for a presented value to match a value stored in the directory is that the P-Selector, S-Selector, and T-Selector of the two presentation addresses must be equal, and the N-Addresses of the presented value must be a subset of those of the stored value.

- **DS_A_REGISTERED_ADDRESS**

  This attribute contains mnemonics by means of which the object can be reached via the public telegram service, according to Recommendation F.1. A mnemonic identifies an object in the context of a particular city, and is registered in the country containing the city. The matching rules require that differences in the case of alphabetical characters be considered insignificant.

- **DS_A_ROLE_OCCUPANT**

  This attribute occurs only in entries that describe an organizational role. It gives the names of objects that fulfill the organizational role.

- **DS_A_SEARCH_GUIDE**

  This attribute contains the criteria that can be used to build filters for conducting searches in which the object is the base object.

- **DS_A_SEE_ALSO**

  This attribute contains the names of objects that represent other aspects of the real-world object that the present object represents.

- **DS_A_SERIAL_NBR**

  This attribute contains the serial numbers of a device.

- **DS_A_STATE_OR_PROV_NAME**

  This attribute specifies a state or province. When used as part of a directory name, it identifies states, provinces, or other geographical regions in which the object is located or with which it is associated in some other important way.

- **DS_A_STREET_ADDRESS**

  This attribute identifies a site for the local distribution and physical delivery of mail. When used as part of a directory name, it identifies the street address (for example, street name and house number) at which the object is located or with which it is associated in some other important way.

- **DS_A_SUPPORT_APPLIC_CONTEXT**

  This attribute occurs only in entries that describe an OSI application entity. It identifies OSI application contexts supported by the object.

- **DS_A_SURNAME**

  This attribute occurs only in entries that describe individuals. The surname by which the individual is commonly known, normally inherited from the individual's parent (or parents) or taken at marriage, as determined by the custom of the country or culture with which the individual is associated.

- **DS_A_PHONE_NBR**

  This attribute identifies telephones by means of which the object can be reached or with which it is associated in some other important way. The matching rules require that differences caused by the presence of spaces and dashes be considered insignificant.

- **DS_A_TELETEX_TERM_IDENT**

  This attribute contains descriptions of teletex terminals by means of which the object can be reached or with which it is associated in some other important way.

- **DS_A_TELEX_NBR**

  This attribute contains descriptions of telex terminals by means of which the object can be reached or with which it is associated in some other important way.

- **DS_A_TITLE**

  This attribute identifies positions or functions of the object within its organization.

- **DS_A_USER_PASSWORD**

  This attribute contains the passwords assigned to the object.

- **DS_A_X121_ADDRESS**

  This attribute identifies points on the public data network at which the object can be reached or with which it is associated in some other important way. The matching rules require that differences caused by the presence of spaces be considered insignificant.

32.2 Selected Object Classes

This section presents the object classes that are defined in the standards. Object classes are groups of directory entries that share certain characteristics. The object classes are arranged into a lattice, based on the object class **DS_O_TOP**. In a lattice, each element, except a leaf, has one or more immediate subordinates but also has one or more immediate superiors. This contrasts with a tree, where each element has exactly one immediate superior. Object classes closer to **DS_O_TOP** are called superclasses, and those further away are called subclasses. This relationship is not connected to any other such relationship in this guide.

Each directory entry belongs to an object class, and to all the superclasses of that object class. Each entry has an attribute named **DS_A_OBJECT_CLASS**, which was discussed in the previous section, and which identifies the object classes to which the entry belongs. The values of this attribute are object identifiers, which are represented in the interface by constants with the same name as the object class, prefixed by **DS_O_**.

Associated with each object class are zero or more mandatory and zero or more optional attributes. Each directory entry must contain all the mandatory attributes and can (but need not) contain the optional attributes associated with the object class and its superclasses.

The object classes defined in the standards are shown in Table 32-3, together with their object identifiers.

Note: The third and fourth columns of Table 32-3 contain the contents octets of the BER encoding of the object identifier. All these object identifiers stem from the root **{joint-iso-ccitt(2) ds(5) objectClass(6)}**.

Table 32–3. Object Identifiers for Selected Object Classes

Package	Attribute Type	Object Identifier BER	
		Decimal	Hexidecimal
BDCP	**DS_O_ALIAS**	85, 6, 1	\x55\x06\x01
BDCP	**DS_O_APPLIC_ENTITY**	85, 6, 12	\x55\x06\x0C
BDCP	**DS_O_APPLIC_PROCESS**	85, 6, 11	\x55\x06\x0B
BDCP	**DS_O_COUNTRY**	85, 6, 2	\x55\x06\x02
BDCP	**DS_O_DEVICE**	85, 6, 14	\x55\x06\x0E
BDCP	**DS_O_DSA**	85, 6, 13	\x55\x06\x0D
BDCP	**DS_O_GROUP_OF_NAMES**	85, 6, 9	\x55\x06\x09
BDCP	**DS_O_LOCALITY**	85, 6, 3	\x55\x06\x03
BDCP	**DS_O_ORG**	85, 6, 4	\x55\x06\x04
BDCP	**DS_O_ORG_PERSON**	85, 6, 7	\x55\x06\x07
BDCP	**DS_O_ORG_ROLE**	85, 6, 8	\x55\x06\x08
BDCP	**DS_O_ORG_UNIT**	85, 6, 5	\x55\x06\x05
BDCP	**DS_O_PERSON**	85, 6, 6	\x55\x06\x06
BDCP	**DS_O_RESIDENTIAL_ PERSON**	85, 6, 10	\x55\x06\x0A
BDCP	**DS_O_TOP**	85, 6, 0	\x55\x06\x00

32.3 OM Class Hierarchy

The remainder of this chapter defines the additional OM classes used to represent values of the selected attributes described in Section 32.1. Some of the selected attributes are represented by OM classes that are used in the interface itself, and hence are defined in Chapter 30; for example, *DS_C_NAME*. As mentioned in the introductory text to this chapter, an explanation of the purpose of these attributes is beyond the scope of this guide.

This section shows the hierarchical organization of the OM classes that are defined in the following sections, and it shows which OM classes inherit additional OM attributes from their OM superclasses. In the following list, subclassification is indicated by indentation, and the names of abstract OM classes are in italics. For example, **DS_C_POSTAL_ADDRESS** is an immediate subclass of the abstract OM class *OM_C_OBJECT*.

OM_C_OBJECT

- **DS_C_FACSIMILE_PHONE_NBR**
- **DS_C_POSTAL_ADDRESS**
- **DS_C_SEARCH_CRITERION**
- **DS_C_SEARCH_GUIDE**
- **DS_C_TELETEX_TERM_IDENT**
- **DS_C_TELEX_NBR**

None of the OM classes in the preceding list are encodable using **om_encode( )** and **om_decode( )**.

32.4 DS_C_FACSIMILE_PHONE_NBR

An instance of OM class **DS_C_FACSIMILE_PHONE_NBR** identifies and describes a facsimile terminal, if required.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 32-4.

Table 32–4. OM Attributes of DS_C_FACSIMILE_PHONE_NBR

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_PARAMETERS	Object(**MH_C_ G3_ FAX_ NBPS**)[1]	—	0 or 1	—
DS_PHONE_NBR	String(**OM_S_ PRINTABLE_ STRING**)[2]	1-32	1	—

[1] As defined in the X.400 API Specifications.

[2] As permitted by E.123 (for example, +44 582 10101).

- **DS_PARAMETERS**

  If present, this attribute identifies the facsimile terminal's nonbasic capabilities.

- **DS_PHONE_NBR**

  This attribute contains a telephone number by means of which the facsimile terminal is accessed.

32.5 DS_C_POSTAL_ADDRESS

An instance of OM class **DS_C_POSTAL_ADDRESS** is a postal address.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attribute listed in Table 32-5.

Table 32–5. OM Attribute of DS_C_POSTAL_ADDRESS

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_POSTAL_ ADDRESS	String(OM_S_ TELETEX_ STRING)	1-30	1-6	—

- **DS_POSTAL_ADDRESS**

  Each value of this OM attribute is one line of the postal address. It typically includes a name, street address, city name, state or province name, postal code, and possibly a country name.

32.6 DS_C_SEARCH_CRITERION

An instance of OM class **DS_C_SEARCH_CRITERION** is a component of a **DS_C_SEARCH_GUIDE** OM object.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 32-6.

Table 32-6. OM Attributes of DS_C_SEARCH_CRITERION

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ ATTRIBUTE_ TYPE	String(**OM_S_ OBJECT_ IDENTIFIER_ STRING**)	—	0 or 1	—
DS_CRITERIA	Object(**DS_C_ SEARCH_ CRITERION)**	—	0 or more	—
DS_FILTER_ ITEM_TYPE	Enum(**DS_Filter_ Item_Type**)	—	0 or 1	—
DS_FILTER_ TYPE	Enum(**DS_Filter_ Type)**	—	1	**DS_ITEM**

A **DS_C_SEARCH_CRITERION** suggests how to build part of a filter to be used when searching the directory. Its meaning depends on the value of its OM attribute **DS_FILTER_TYPE**. If the value is **DS_ITEM**, then the criterion suggests building an instance of OM class **DS_C_FILTER_ITEM**. If **DS_FILTER_TYPE** has any other value, it suggests building an instance of OM class **DS_C_FILTER**.

- **DS_ATTRIBUTE_TYPE**

  This attribute indicates the attribute type to be used in the suggested **DS_C_FILTER_ITEM**. This OM attribute is only present when the value of **DS_FILTER_TYPE** is **DS_ITEM**.

- **DS_CRITERIA**

  This attribute contains nested search criteria. This OM attribute is not present when the value of **DS_FILTER_TYPE** is **DS_ITEM**.

- **DS_FILTER_ITEM_TYPE**

  This attribute indicates the type of suggested filter item. Its value can be one of the following:

  — **DS_APPROXIMATE_MATCH**

  — **DS_EQUALITY**

— **DS_GREATER_OR_EQUAL**

— **DS_LESS_OR_EQUAL**

— **DS_SUBSTRINGS**

However, the filter item cannot have the value **DS_PRESENT**. This OM attribute is only present when the value of **DS_FILTER_TYPE** is **DS_ITEM**.

- **DS_FITER_TYPE**

  This attribute indicates the type of suggested filter. The value **DS_ITEM** means that the suggested component is a filter item, not a filter. The other values suggest the corresponding type of filter. Its value is one of the following:

  — **DS_AND**

  — **DS_ITEM**

  — **DS_NOT**

  — **DS_OR**

32.7 DS_C_SEARCH_GUIDE

An instance of OM class **DS_C_SEARCH_GUIDE** suggests a criterion for searching the directory for particular entries. It can be used to build a **DS_C_FILTER** parameter for **ds_search( )** operations that are based on the object in whose entry the search guide occurs.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 32-7.

Table 32–7. OM Attributes of DS_C_SEARCH_GUIDE

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_OBJECT_CLASS	String(**OM_S_ OBJECT_ IDENTIFIER_ STRING)**	—	0 or 1	—
DS_CRITERIA	Object(**DS_C_ SEARCH_ CRITERION)**	—	1	—

- **DS_OBJECT_CLASS**

  This attribute identifies the object class of the entries to which the search guide applies. If this OM attribute is absent, the search guide applies to objects of any class.

- **DS_CRITERIA**

  This attribute contains the suggested search criteria.

32.8 DS_C_TELETEX_TERM_IDENT

An instance of OM class **DS_C_TELETEX_TERM_IDENT** identifies and describes a teletex terminal.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 32-8.

Table 32–8. OM Attributes of DS_C_TELETEX_TERM_IDENT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_PARAMETERS	Object(**MH_C_ TELETEX_ NBPS**)[1]	—	0 or 1	—
DS_TELETEX_ TERM	String(**OM_S_ PRINTABLE_ STRING**)[2]	1-1024	1	—

[1] As defined in the X.400 API Specifications.

[2] As permitted by F.200.

- **DS_PARAMETERS**

  This attribute identifies the teletex terminal's nonbasic capabilities.

- **DS_TELETEX_TERM**

  This attribute identifies the teletex terminal.

32.9 DS_C_TELEX_NBR

An instance of OM class **DS_C_TELEX_NBR** identifies and describes a telex terminal.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 32-9.

Table 32–9. OM Attributes of DS_C_TELEX_NBR

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_ANSWERBACK	String(**OM_S_ PRINTABLE_ STRING)**	1-8	1	—
DS_COUNTRY_CODE	String(**OM_S PRINTABLE_ STRING)**	1-4	1	—
DS_TELEX_NBR	String(**OM_S_ PRINTABLE_ STRING)**	1-14	1	—

- **DS_ANSWERBACK**

  This attribute contains the code with which the telex terminal acknowledges calls placed to it.

- **DS_COUNTRY_CODE**

  This attribute contains the identifier of the country through which the telex terminal is accessed.

- **DS_TELEX_NBR**

  This attribute contains the number by means of which the telex terminal is addressed.

Chapter 33

MHS Directory User Package

The Message Handling Systems Directory User Package (MDUP) contains definitions to support the use of the directory in accordance with the standard 1988 X.400 User Agents and Message Transfer Agents (MTAs) for name resolution, Distribution List (DL) expansion, and capability assessment. The definitions are based upon the attribute types and syntaxes specified in *X.402, Annex A*.

The MDUP is an optional package that can be used by the XDS interface. Applications must negotiate use of this package with **ds_version( )** before using any of the MDUP features. If an application attempts to use features specific to the package without first negotiating its use, an appropriate error (for example, **OM_NO_SUCH_CLASS**) is returned by the Object Management (OM) function.

The object identifier associated with the MDUP is **{iso(1) identified-organization(3) icd-ecma(0012) member-company(2) dec(1011) xopen(28) mdup(3)}** with the following encoding:

\x2B\xC\x2\x87\x73\x1C\x3

This identifier is represented by the constant **DS_MHS_DIR_USER_PKG**. The C constants associated with this package are defined in the **xdsmdup.h**, **xmhp.h**, and **xmsga.h** header files (see the *OSF DCE Application Development Reference*).

The concepts and notation used are first mentioned in Section 30.1. They are also fully explained in Chapters 35 through 37. The attribute types are introduced first, followed by the object classes. Next, the OM class hierarchy and OM class definitions required to support the new attribute types are described.

33.1 MDUP Attribute Types

This section presents additional directory attribute types that are used with the MDUP. Each attribute type has an object identifier, which is the value of the OM attribute **DS_ATTRIBUTE_TYPE**. These object identifiers are represented in the interface by constants with the same name as the directory attribute and are prefixed by **DS_A_** so that they can be easily identified.

This section contains two tables that are used to indicate the object identifiers for MDUP attribute types (see Table 33-1), and the values for MDUP attribute types (see Table 33-2), respectively. Following these two tables is a brief description of each attribute. (See Section 32.1 for information on general matching rules).

Note: The third and fourth columns of Table 33-1 contain the contents octets of the BER encoding of the object identifier. All these object identifiers stem from the root **{joint-iso-ccitt(2) mhs-motis(6) arch(5) at(2)}**.

Table 33–1. Object Identifiers for MDUP Attribute Types

		Object Identifier BER	
Package	Attribute Type	Decimal	Hexidecimal
MDUP	**DS_A_DELIV_CONTENT_LENGTH**	86, 5, 2, 0	\x56\x05\x02\x00
MDUP	**DS_A_DELIV_CONTENT_TYPES**	86, 5, 2, 1	\x56\x05\x02\x01
MDUP	**DS_A_DELIV_EITS**	86, 5, 2, 2	\x56\x05\x02\x02
MDUP	**DS_A_DL_MEMBERS**	86, 5, 2, 3	\x56\x05\x02\x03
MDUP	**DS_A_DL_SUBMIT_PERMS**	86, 5, 2, 4	\x56\x05\x02\x04
MDUP	**DS_A_MESSAGE_STORE**	86, 5, 2, 5	\x56\x05\x02\x05
MDUP	**DS_A_OR_ADDRESSES**	86, 5, 2, 6	\x56\x05\x02\x06
MDUP	**DS_A_PREF_DELIV_METHODS**	86, 5, 2, 7	\x56\x05\x02\x07
MDUP	**DS_A_SUPP_AUTO_ACTIONS**	86, 5, 2, 8	\x56\x05\x02\x08
MDUP	**DS_A_SUPP_CONTENT_TYPES**	86, 5, 2, 9	\x56\x05\x02\x09
MDUP	**DS_A_SUPP_OPT_ATTRIBUTES**	86, 5, 2, 10	\x56\x05\x02\x0A

Table 33–2. Representation of Values for MDUP Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-valued	Matching Rules
DS_A_DELIV_ CONTENT_ LENGTH	**OM_S_ INTEGER**	—	No	—
DS_A_DELIV_ CONTENT TYPES	String(**OM_S_ OBJECT_ IDENTIFIER_ STRING)**	—	Yes	—
DS_A_DELIV_ EITS	String(**OM_S_ OBJECT_ IDENTIFIER_ STRING)**	—	Yes	—
DS_A_DL_ MEMBERS	Object(**DS_C_ OR_NAME)**	—	Yes	—
DS_A_DL_ SUBMIT PERMS	Object(**DS_C_ DL_SUBMIT_ PERMS)**	—	Yes	—
DS_A_MESSAGE_ STORE	String(**DS_C_ DS_DN)**	—	No	—
DS_A_OR_ ADDRESSES	Object(**MH_C_ OR_ADDRESS)**	—	Yes	—
DS_A_PREF_ DELIV_ METHODS	Enum(**MH_ Delivery_ Mode)**	—	No	E
DS_A_SUPP_ AUTO_ ACTIONS	String(**OM_S_ OBJECT_ IDENTIFIER_ STRING)**	—	Yes	—
DS_A_SUPP_ CONTENT_ TYPES	String(**OM_S_ OBJECT_ IDENTIFIER_ STRING)**	—	Yes	—

Attribute Type	OM Value Syntax	Value Length	Multi-valued	Matching Rules
DS_A_SUPP_OPT_ATTRIBUTES	String(OM_S_OBJECT_IDENTIFIER_STRING)	—	Yes	—

Throughout the descriptions that follow, the term *object* indicates the directory object whose directory entry contains the corresponding directory attributes.

- **DS_A_DELIV_CONTENT_LENGTH**

  This attribute identifies the maximum content length of the messages whose delivery a user will accept.

- **DS_A_DELIV_CONTENT_TYPES**

  This attribute identifies the content types of the messages whose delivery a user will accept.

- **DS_A_DELIV_EITS**

  This attribute identifies the Encoded Information Types (EITs) of the messages whose delivery a user will accept.

- **DS_A_DL_MEMBERS**

  This attribute identifies the members of a DL.

- **DS_A_DL_SUBMIT_PERMS**

  This attribute identifies the users and DLs that may submit messages to a DL.

- **DS_A_MESSAGE_STORE**

  This attribute identifies a user's Message Store (MS) by name.

- **DS_A_OR_ADDRESSES**

  This attribute specifies a user's or DL's Originator/Recipient (O/R) addresses.

- **DS_A_PREF_DELIV_METHODS**

  This attribute identifies, in the order of decreasing preference, the methods of delivery a user prefers.

- **DS_A_SUPP_AUTO_ACTIONS**

  This attribute identifies the automatic actions that an MS fully supports.

- **DS_A_SUPP_CONTENT_TYPES**

  This attribute identifies the content types of the messages whose syntax and semantics an MS fully supports.

- **DS_A_SUPP_OPT_ATTRIBUTES**

  This attribute identifies the optional attributes that an MS fully supports.

33.2 MDUP Object Classes

There are five MDUP object classes and their associated object identifiers (see Table 33-3).

Note: The third and fourth columns of Table 33-3 contain the contents octets of the BER encoding of the object identifier. MDUP object identifiers stem from the root **{joint-iso-ccitt(2) mhs-motis(6) arch(5) oc(1)}**.

Table 33–3. Object Identifiers for MDUP Object Classes

		Object Identifier BER	
Package	**Object Class**	**Decimal**	**Hexidecimal**
MDUP	**DS_O_MHS_ DISTRIBUTION_LIST**	86, 5, 1, 0	\x56\x05\x01\x00
MDUP	**DS_O_MHS_MESSAGE_ STORE**	86, 5, 1, 1	\x56\x05\x01\x01
MDUP	**DS_O_MHS_MESSAGE_ TRANS_AG**	86, 5, 1, 2	\x56\x05\x01\x02
MDUP	**DS_O_MHS_USER**	86, 5, 1, 3	\x56\x05\x01\x03
MDUP	**DS_O_MHS_USER_AG**	86, 5, 1, 4	\x56\x05\x01\x04

33.3 MDUP OM Class Hierarchy

The remainder of this chapter defines the additional OM classes used by MDUP. This section shows the hierarchical organization of the OM classes that are defined in the following sections, and shows which classes inherit additional OM attributes from their OM superclasses. In the following list, subclassification is indicated by indentation and the names of abstract OM classes are represented in italic font.

- *MH_C_OR_ADDRESS*
  - **MH_C_OR_NAME**
- **DS_C_DL_SUBMIT_PERMS**

None of the OM classes in the preceding list are encodable using **om_encode( )** and **om_decode( )**.

33.4 MH_C_OR_ADDRESS

An instance of class **MH_C_OR_ADDRESS** distinguishes one user or DL from another, and identifies its point of access to the Message Transfer System (MTS). Every user or DL is assigned one or more MTS access points and thus one or more originator/recipient (O/R) addresses.

The attributes specific to this class are listed in Table 33-4. The 1988 column indicates that the attribute applies only to the 1988 standard.

Table 33–4. Attributes Specific to MH_C_OR_ADDRESS

Attribute	Value Syntax	Value Length	Value Number	1988?
MH_T_ADMD_ NAME[1]	String(**OM_S_ PRINTABLE_ STRING)**	0-16	0 or 1	—
MH_T_COMMON_ NAME	String(**OM_S_ PRINTABLE_ STRING)** or String(**OM_S_ TELETEX_ STRING)**[2]	1-64	0-2	1988
MH_T_COUNTRY_ NAME[1]	String(**OM_S_ PRINTABLE_ STRING)**	2-3	0 or 1	—
MH_T_DOMAIN_ TYPE_1	String(**OM_S_ PRINTABLE_ STRING)** or String(**OM_S_ TELETEX_ STRING)**[2,3]	1-8	0-2[4]	—
MH_T_DOMAIN_ TYPE_2	String(**OM_S_ PRINTABLE_ STRING)** or String(**OM_S_ TELETEX_ STRING)**[2,3]	1-8	0-2[4]	—
MH_T_DOMAIN_ TYPE_3	String(**OM_S_ PRINTABLE_ STRING)** or String(**OM_S_ TELETEX_ STRING)**[2,3]	1-8	0-2[4]	—

Attribute	Value Syntax	Value Length	Value Number	1988?
MH_T_DOMAIN_TYPE_4	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2,3]	1-8	0-2[4]	—
MH_T_DOMAIN_VALUE_1	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2,3]	1-128	0-2[4]	—
MH_T_DOMAIN_VALUE_2	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2,3]	1-128	0-2[4]	—
MH_T_DOMAIN_VALUE_3	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2,3]	1-128	0-2[4]	—
MH_T_DOMAIN_VALUE_4	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2,3]	1-128	0-2[4]	—

Attribute	Value Syntax	Value Length	Value Number	1988?
MH_T_ GENERATION	String(**OM_S_ PRINTABLE_ STRING**) or String(**OM_S_ TELETEX_ STRING**)[2,3]	1-3	0-2[4]	—
MH_T_GIVEN_ NAME	String(**OM_S_ PRINTABLE_ STRING**) or String(**OM_S_ TELETEX_ STRING**)[2,3]	1-16	0-2[4]	—
MH_T_INITIALS	String(**OM_S_ PRINTABLE_ STRING**) or String(**OM_S_ TELETEX_ STRING**)[2,3]	1-5	0-2[4]	—
MH_T_ISDN_ NUMBER	String(**OM_S_ NUMERIC_ STRING**)	1-15	0 or 1	1988
MH_T_ISDN_ SUBADDRESS	String(**OM_S_ NUMERIC_ STRING**)	1-40	0 or 1[5]	1988
MH_T_NUMERIC_ USER_ IDENTIFIER	String(**OM_S_ NUMERIC_ STRING**)	1-32	0 or 1	—
MH_T_ ORGANIZATION_ NAME	String(**OM_S_ PRINTABLE_ STRING**) or String(**OM_S_ TELETEX_ STRING**)[2,3]	1-64	0-2[4, 6]	—

Attribute	Value Syntax	Value Length	Value Number	1988?
MH_T_ORGANIZATIONAL_UNIT_NAME_1	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2,3]	1-32	0-2[4]	—
MH_T_ORGANIZATIONAL_UNIT_NAME_2	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2,3]	1-32	0-2[4]	—
MH_T_ORGANIZATIONAL_UNIT_NAME_3	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2,3]	1-32	0-2[4]	—
MH_T_ORGANIZATIONAL_UNIT_NAME_4	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2,3]	1-32	0-2[4]	—
MH_T_POSTAL_ADDRESS_DETAILS	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2]	1-30	0-2	1988
MH_T_POSTAL_ADDRESS_IN_FULL	String(**OM_S_TELETEX_STRING**)	1-185	0 or 1	1988

Attribute	Value Syntax	Value Length	Value Number	1988?
MH_T_POSTAL_ ADDRESS_ IN_LINES	String(**OM_S_ PRINTABLE_ STRING)**	1-30	0-6	1988
MH_T_POSTAL_ CODE	String(**OM_S_ PRINTABLE_ STRING)**	1-16	0 or 1	1988
MH_T_POSTAL_ COUNTRY_ NAME	String(**OM_S_ PRINTABLE_ STRING)**	2-3	0 or 1	1988
MH_T_POSTAL_ DELIVERY_ POINT_NAME	String(**OM_S_ PRINTABLE_ STRING)** or String(**OM_S_ TELETEX_ STRING)**^{2}	1-30	0-2	1988
MH_T_POSTAL_ DELIV_ SYSTEM_NAME	String(**OM_S_ PRINTABLE_ STRING)**	1-16	0 or 1	1988
MH_T_POSTAL_ GENERAL_ DELIV_ADDR	String(**OM_S_ PRINTABLE_ STRING)** or String(**OM_S_ TELETEX_ STRING)**^{2}	1-30	0-2	1988
MH_T_POSTAL_ LOCALE	String(**OM_S_ PRINTABLE_ STRING)** or String(**OM_S_ TELETEX_ STRING)**^{2}	1-30	0-2	1988

Attribute	**Value Syntax**	**Value Length**	**Value Number**	**1988?**
MH_T_POSTAL_OFFICE_BOX_NUMBER	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2]	1-30	0-2	1988
MH_T_POSTAL_OFFICE_NAME	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2]	1-30	0-2	1988
MH_T_POSTAL_OFFCE_NUMBER	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2]	1-30	0-2	1988
MH_T_POSTAL_ORGANIZATION_NAME	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2]	1-30	0-2	1988
MH_T_POSTAL_PATRON_DETAILS	String(**OM_S_PRINTABLE_STRING**) or String(**OM_S_TELETEX_STRING**)[2]	1-30	0-2	1988

Attribute	Value Syntax	Value Length	Value Number	1988?
MH_T_POSTAL_ PATRON_ NAME	String(**OM_S_ PRINTABLE_ STRING**) or String(**OM_S_ TELETEX_ STRING**)[2]	1-30	0-2	1988
MH_T_POSTAL_ STREET_ ADDRESS	String(**OM_S_ PRINTABLE_ STRING**) or String(**OM_S_ TELETEX_ STRING**)[2]	1-30	0-2	1988
MH_T_ PRESENTATION_ ADDRESS	Object(**DS_C_ PRESENTATION_ ADDRESS**)	—	0 or 1	1988
MH_T_PRMD_ NAME[1]	String(**OM_S_ PRINTABLE_ STRING**)	1-16	0 or 1	—
MH_T_SURNAME	String(**OM_S_ PRINTABLE_ STRING**) or String(**OM_S_ TELETEX_ STRING**)[2,3]	1-40	0-2[4]	—
MH_T_TERMINAL_ IDENTIFIER	String(**OM_S_ PRINTABLE_ STRING**)	1-24	0 or 1	—
MH_T_TERMINAL_ TYPE	Enum(**MH_ Terminal_ Type**)	—	0 or 1	1988
MH_T_X121_ ADDRESS	String(**OM_S_ NUMERIC_ STRING**)	1-15	0 or 1	—

Footnotes to Table 33-4
[1]The value initially is the current session's attribute of the same name.
[2]If only one value is present in international communications, its syntax is String(**OM_S_PRINTABLE_STRING**). If two values are present, in either domestic or international communications, the syntax of the first is String(**OM_S_PRINTABLE_STRING**), the syntax of the second is STRIN(**OM_S_TELETEX_STRING**), and the two convey the same information such that either can be safely ignored. For example, Teletex strings allow inclusion of the accented characters commonly used in many countries. Not all input/output devices, however, permit the entry and display of such characters. Printable strings are required internationally to ensure that such device limitations do not prevent cummunications.
[3]For 1984, the syntax of the value is String(**OM_S_PRINTABLE_STRING**).
[4]For 1984, at most one value is present.
[5]This attribute is present only if the ISDN Number attribute is present.
[6]For 1988, this attribute is required if any Organization Name is present.

- **MH_T_ADMD_NAME**

  This attribute contains the name of the user's or DL's Administration Management Domain (ADMD). It identifies the ADMD relative to the country that the **MH_T_COUNTRY_NAME** attribute indicates. Its values are defined by that country.

  Note that the attribute value that comprises a single space is reserved. If permitted by the country that the **MH_T_COUNTRY_NAME** attribute indicates, a single space designates ''any;'' that is, all ADMDs within the country. This affects both the identification of users and DLs within the country and the routing of messages, probes, and reports to and among the ADMDs of that country. Regarding the former, it requires that the O/R addresses of users and DLs within the country be chosen so as to ensure their unambiguousness, even in the absence of the actual names of the users' and DLs' ADMDs. Regarding the latter, it permits both Private Management Domains (PRMD) within the country and ADMDs outside the country to route messages, probes, and reports to any of the ADMDs within the country indiscriminately. It also requires that the ADMDs within the country interconnect themselves in such a way that the messages, probes, and reports are conveyed to their destinations.

- **MH_T_COMMON_NAME**

  This attribute contains the name commonly used to refer to the user or DL. It identifies the user or DL relative to the entity indicated by another attribute; for example, **MH_T_ORGANIZATION_NAME**. Its values are defined by that entity.

- **MH_T_COUNTRY_NAME**

  This attribute contains the name of the user's or DL's country. Its defined values are the numbers that X.121 assigns to the country, or the character pairs that ISO 3166 assigns to it.

- **MH_T_DOMAIN_TYPE_1**

  This attribute contains the name of a class of information. Its values are defined by the user's or DL's ADMD and PRMD, if any, in combination.

- **MH_T_DOMAIN_TYPE_2**

  This attribute contains the name of a class of information. Its values are defined by the user's or DL's ADMD and PRMD, if any, in combination.

- **MH_T_DOMAIN_TYPE_3**

  This attribute contains the name of a class of information. Its values are defined by the user's or DL's ADMD and PRMD, if any, in combination.

- **MH_T_DOMAIN_TYPE_4**

  This attribute contains the name of a class of information. Its values are defined by the user's or DL's ADMD and PRMD, if any, in combination.

- **MH_T_DOMAIN_VALUE_1**

  This attribute is an instance of the class of information that the **MH_T_DOMAIN_TYPE_1** attribute indicates.

- **MH_T_DOMAIN_VALUE_2**

  This attribute is an instance of the class of information that the **MH_T_DOMAIN_TYPE_2** attribute indicates.

- **MH_T_DOMAIN_VALUE_3**

  This attribute is an instance of the class of information that the **MH_T_DOMAIN_TYPE_3** attribute indicates.

- **MH_T_DOMAIN_VALUE_4**

  This attribute is an instance of the class of information that the **MH_T_DOMAIN_TYPE_4** attribute indicates.

- **MH_T_GENERATION**

  This attribute contains the user's generation; for example, **Jnr.**

- **MH_T_GIVEN_NAME**

  This attribute contains the user's given name; for example, **Robert.**

- **MH_T_INITIALS**

  This attribute contains the initials of all of the user's names except the user's surname; for example, **RE.**

- **MH_T_ISDN_NUMBER**

  This attribute contains the ISDN number of the user's terminal. Its values are defined by E.163 and E.164.

- **MH_T_ISDN_SUBADDRESS**

  This attribute contains the ISDN subaddress, if any, of the user's terminal. Its values are defined by E.163 and E.164.

- **MH_T_NUMERIC_USER_IDENTIFIER**

  This attribute numerically identifies the user or DL relative to the ADMD that the **MH_T_ADMD_NAME** attribute indicates. Its values are defined by that ADMD.

- **MH_T_ORGANIZATION_NAME**

  This attribute contains the name of the user's or DL's organization. As a national matter, such names may be assigned by the country that the **MH_T_COUNTRY_NAME** attribute indicates, the ADMD that the **MH_T_ADMD_NAME** attribute indicates, the PRMD that the **MH_T_PRMD_NAME** attribute indicates, or the latter two organizations together.

- **MH_T_ORGANIZATIONAL_UNIT_NAME_1**

  This attribute contains the name of a unit (for example, a division or department) of the organization that the **MH_T_ORGANIZATION_NAME** attribute indicates. The attribute's values are defined by that organization.

- **MH_T_ORGANIZATIONAL_UNIT_NAME_2**

  This attribute contains the name of a subunit (for example, a division or department) of the unit that the **MH_T_ORGANIZATIONAL_UNIT_NAME_1** attribute indicates. The attribute's values are defined by the latter unit.

- **MH_T_ORGANIZATIONAL_UNIT_NAME_3**

  This attribute contains the name of a subunit (for example, a division or department) of the unit that the **DS_A_ORGANIZATIONAL_UNIT_NAME_2** attribute indicates. The attribute's values are defined by the latter unit.

- **MH_T_ORGANIZATIONAL_UNIT_NAME_4**

  This attribute contains the name of a subunit (for example, a division or department) of the unit that the **MH_T_ORGANIZATIONAL_UNIT_NAME_3** attribute indicates. The attribute's values are defined by the latter unit.

- **MH_T_POSTAL_ADDRESS_DETAILS**

  This attribute contains the means (for example, a room and the floor numbers in a large building) for identifying the exact point at which the user takes delivery of physical messages.

- **MH_T_POSTAL_ADDRESS_IN_FULL**

  This attribute contains the free-form and possibly multiline postal address of the user as a single Teletex string with the lines being separated as prescribed for Teletex strings.

- **MH_T_POSTAL_ADDRESS_IN_LINES**

  This attribute contains the free-form postal address of the user in a sequence of printable strings, each representing a line of text.

- **MH_T_POSTAL_CODE**

  This attribute contains the postal code for the geographical area in which the user takes delivery of physical messages. It identifies the area relative to the country that the **MH_T_POSTAL_COUNTRY_NAME** attribute indicates. Its values are defined by the postal administration of that country.

- **MH_T_POSTAL_COUNTRY_NAME**

  This attribute contains the name of the country in which the user takes delivery of physical messages. Its defined values are the numbers X.121 assigns to the country, or the character pairs ISO 3166 assigns to it.

- **MH_T_POSTAL_DELIVERY_POINT_NAME**

  This attribute identifies the locus of distribution, other than that indicated by the **MH_T_POSTAL_OFFICE_NAME** attribute (for example, a geographical area) of the user's physical messages.

- **MH_T_POSTAL_DELIV_SYSTEM_NAME**

  This attribute contains the name of the Postal Delivery System (PDS) through which the user is to receive physical messages. It identifies the PDS relative to the ADMD that the **MH_T_ADMD_NAME** attribute indicates. Its values are defined by that ADMD.

- **MH_T_POSTAL_GENERAL_DELIV_ADDRESS**

  This attribute contains the code that the user gives to the post office to collect the physical messages awaiting delivery to the user. The post office is indicated in the **MH_T_POSTAL_OFFICE_NAME** attribute. The values for the **MH_T_POSTAL_GENERAL_DELIV_ADDRESS** attribute are defined by that post office.

- **MH_T_POSTAL_LOCALE**

  This attribute identifies the point of delivery, other than that indicated by the following attributes:

  — **MH_T_POSTAL_GENERAL_DELIV_ADDR**

  — **MH_T_POSTAL_OFFICE_BOX_NUMBER**

  — **MH_T_POSTAL_STREET_ADDRESS**.

  For example, a building or a hamlet of the user's physical messages.

- **MH_T_POSTAL_OFFICE_BOX_NUMBER**

  This attribute contains the number of the post office box by means of which the user takes delivery of physical messages. The box is located at the post office that the **MH_T_POSTAL_OFFICE_NAME** attribute indicates. The attribute's values are defined by that post office.

- **MH_T_POSTAL_OFFICE_NAME**

  This attribute contains the name of the municipality (for example, city or village) where the post office is situated through which the user takes delivery of physical messages.

- **MH_T_POSTAL_OFFICE_NUMBER**

  This attribute contains the means of distinguishing among several post offices indicated by the **MH_T_POSTAL_OFFICE_NAME** attribute.

- **MH_T_POSTAL_ORGANIZATION_NAME**

  This attribute contains the name of the organization through which the user takes delivery of physical messages.

- **MH_T_POSTAL_PATRON_DETAILS**

  This attribute contains additional information (for example, the name of the organizational unit through which the user takes delivery of physical messages) necessary to identify the user for purposes of physical delivery.

- **MH_T_POSTAL_PATRON_NAME**

  This attribute contains the name under which the user takes delivery of physical messages.

- **MH_T_POSTAL_STREET_ADDRESS**

  This attribute contains the street address (for example, **43 Primrose Lane**) at which the user takes delivery of physical messages.

- **MH_T_PRESENTATION_ADDRESS**

  This attribute contains the presentation address of the user's terminal.

- **MH_T_PRMD_NAME**

  This attribute contains the name of the user's PRMD. As a national matter, such names may be assigned by the country that the **MH_T_COUNTRY_NAME** attribute indicates or the ADMD that the **MH_T_ADMD_NAME** attribute indicates.

- **MH_T_SURNAME**

  This attribute contains the user's surname; for example, **Lee**.

- **MH_T_TERMINAL_IDENTIFIER**

  This attribute contains the terminal identifier of the user's terminal; for example, a Telex answer back or a Teletex terminal identifier.

- **MH_T_TERMINAL_TYPE**

  This attribute contains the type of the user's terminal. Its value is selected from the following:

  — **MH_TT_G3_FAX**

  — **MH_TT_G4_FAX**

  — **MH_TT_IA5_TERMINAL**

  — **MH_TT_TELETEX**

  — **MH_TT_TELEX**

  — **MH_TT_VIDEOTEX**

  The meaning of each value is indicated by its name.

- **MH_T_X121_ADDRESS**

  This attribute contains the network address of the user's terminal. Its values are defined by X.121.

  Note: The strings admitted by X.121 include a telephone number preceded by the telephone escape digit (9), and a Telex number preceded by the Telex escape digit (8).

Certain attributes are grouped together for reference as follows:

- **Personal Name Attributes**

  These comprise the following:

  — **MH_T_GIVEN_NAME**

  — **MH_T_INITIALS**

  — **MH_T_SURNAME**

  — **MH_T_GENERATION**

- **Organizational Unit Name Attributes**

  These comprise the following:

  — **MH_T_ORGANIZATIONAL_UNIT_NAME_1**

  — **MH_T_ORGANIZATIONAL_UNIT_NAME_2**

  — **MH_T_ORGANIZATIONAL_UNIT_NAME_3**

  — **MH_T_ORGANIZATIONAL_UNIT_NAME_4**

- **Network Address Attributes**

  These comprise the following:

  — **MH_T_ISDN_NUMBER**

  — **MH_T_ISDN_SUBADDRESS**

  — **MH_T_PRESENTATION_ADDRESS**

  — **MH_T_X121_ADDRESS**

For any *i* in the interval [1, 4], the Domain Type *i* and Domain Value *i* attributes constitute a Domain-Defined Attribute (DDA).

Note: The widespread avoidance of DDAs produces more uniform and thus more user-friendly O/R addresses. However, it is anticipated that not all Management Domains (MDs) will be able to avoid such attributes immediately. The purpose of DDAs is to permit an MD to retain its existing native addressing conventions for a time. It is intended, however, that all MDs migrate away from the use of DDAs, and thus that DDAs are used only for an interim period.

An O/R address may take any of the forms summarized in Table 35-5. Table 35-5 indicates the attributes that may be present in an O/R address of each form. It also indicates whether it is mandatory (M) or conditional (C) that they do so. When applied to a group of attributes (the network address attributes, for example), mandatory means that at least one member of the group must be present, while conditional means that no members of the group need necessarily be present.

The presence or absence in a particular O/R address of conditional attributes is determined as follows. If a user or DL is accessed through a PRMD, the ADMD that the **MH_T_COUNTRY_NAME** and **MH_T_ADMD_NAME** attributes indicate governs whether attributes used to route messages to the PRMD are present, but it imposes no other constraints on attributes. If a user or DL is *not* accessed through a PRMD, the same ADMD governs whether all conditional attributes, except those specific to postal O/R addresses, are present. All conditional attributes specific to postal O/R addresses are present or absent so as to satisfy the postal addressing requirements of the users they identify.

Table 33–5. Forms of Originator/Recipient Address

Attribute	Mnem[1]	Num[2]	Spost[3]	Upost[4]	Term[5]
MH_T_ADMD_NAME	M	M	M	M	C
MH_T_COMMON_NAME	C	—	—	—	—
MH_T_COUNTRY_NAME	M	M	M	M	C
Domain-Defined Attributes	C	C	—	—	C
Network Address Attributes	—	—	—	—	M
MH_T_NUMERIC_USER_ IDENTIFIER	—	M	—	—	—
MH_T_ORGANIZATION_ NAME	C	—	—	—	—
Organizational Unit Name Attributes	C	—	—	—	—
Personal Name Attributes	C	—	—	—	—
MH_T_POSTAL_ ADDRESS_DETAILS	—	—	C	—	—
MH_T_POSTAL_ ADDRESS_IN_FULL	—	—	—	M	—
MH_T_POSTAL_CODE	—	—	M	M	—
MH_T_POSTAL_ COUNTRY_NAME	—	—	M	M	—
MH_T_POSTAL_ DELIVERY_POINT_ NAME	—	—	C	—	—

Attribute	Mnem[1]	Num[2]	Spost[3]	Upost[4]	Term[5]
MH_T_POSTAL_DELIV_SYSTEM_NAME	—	—	C	C	—
MH_T_POSTAL_GENERAL_DELIV_ADDR	—	—	C	—	—
MH_T_POSTAL_LOCALE	—	—	C	—	—
MH_T_POSTAL_OFFICE_BOX_NUMBER	—	—	C	—	—
MH_T_POSTAL_OFFICE_NAME	—	—	C	—	—
MH_T_POSTAL_OFFICE_NUMBER	—	—	C	—	—
MH_T_POSTAL_ORGANIZATION_NAME	—	—	C	—	—
MH_T_POSTAL_PATRON_DETAILS	—	—	C	—	—
MH_T_POSTAL_PATRON_NAME	—	—	C	—	—
MH_T_POSTAL_STREET_ADDRESS	—	—	C	—	—
MH_T_PRMD_NAME	C	C[6]	C	C	C[6]
MH_T_TERMINAL_IDENTIFIER	—	—	—	—	C
MH_T_TERMINAL_TYPE	—	—	—	—	C

Footnotes to Table 33-5
[1] Mnemonic. X.400 (1984) calls this Form 1 Variant 1. [2] Numeric. X.400 (1984) calls this Form 1 Variant 2. [3] Structured postal. For 1984 this O/R address form is undefined. [4] Unstructured postal. For 1984 this O/R address form is undefined. [5] X.400 (1984) calls this Form 1 Variant 3 and Form 2. [6] For 1984 this attribute is absent (—). For 1988 it is conditional (C).

- **Mnemonic O/R Address**

  This address mnemonically identifies a user or DL. Using the **MH_T_ADMD_NAME** and **MH_T_COUNTRY_NAME** attributes, it identifies an ADMD. Using the **MH_T_COMMON_NAME** attribute or the personal name attributes, the **MH_T_ORGANIZATION_NAME** attribute, the Organizational Unit Name attributes, the **MH_T_PRMD_NAME** attribute, or a combination of these, and optionally DDAs, it identifies a user or DL relative to the ADMD.

  The personal name attributes identify a user or DL relative to the entity indicated by another attribute; for example, **MH_T_ORGANIZATION_NAME.** The **MH_T_SURNAME** attribute will be present if any of the other three personal name attributes are present.

- **Numeric O/R Address**

  This address numerically identifies a user or DL. Using the **MH_T_ADMD_NAME** and **MH_T_COUNTRY_NAME** attributes, it identifies an ADMD. Using the **MH_T_NUMERIC_USER_IDENTIFIER** attribute and possibly the **MH_T_PRMD_NAME** attribute, it identifies the user or DL relative to the ADMD. Any DDAs provide information that is additional to that required to identify the user or DL.

- **Postal O/R Address**

  This address identifies a user by means of its postal address. Two kinds of postal O/R address are distinguished:

  — **Structured**

  Said of a postal O/R address that specifies a user's postal address by means of several attributes. The structure of the postal address is described in the following text in some detail.

  — **Unstructured**

  Said of a postal O/R address that specifies a user's postal address in a single attribute. The structure of the postal address is left largely unspecified in the following text.

  Whether structured or unstructured, a postal O/R address does the following. Using the **MH_T_ADMD_NAME** and **MH_T_COUNTRY_NAME** attributes, it identifies an ADMD. Using the **MH_T_POSTAL_CODE** and **MH_T_POSTAL_COUNTRY_NAME** attributes, it identifies the geographical region in which the user takes delivery of physical messages. Using the **MH_T_POSTAL_DELIV_SYSTEM_NAME** or **MH_T_PRMD_NAME** attribute or both, it also may identify the PDS by means of which the user is to be accessed.

  An unstructured postal O/R address also includes the **MH_T_POSTAL_ADDRESS_IN_FULL** attribute. A structured postal O/R address also includes every other postal addressing attribute that the PDS requires to identify the postal patron.

  Note: The total number of characters in the values of all attributes, except for **MH_T_ADMD_NAME**, **MH_T_COUNTRY_NAME**, and **MH_T_POSTAL_DELIV_SYSTEM_NAME**, in a postal O/R address should be small enough to permit their rendition in 6 lines of 30 characters, the size of a typical physical envelope window. The rendition algorithm, while defined by the Physical Delivery Access Unit (PDAU), is likely to include inserting delimiters (for example, spaces) between some attribute values.

- **Terminal O/R Address**

  This address identifies a user by identifying the user's terminal using the network address attributes. It also may identify the ADMD through which the terminal is accessed by using the **MH_T_ADMD_NAME** and **MH_T_COUNTRY_NAME** attributes. The **MH_T_PRMD_NAME** attribute and any DDAs, which will be present only if the **MH_T_ADMD_NAME** and **MH_T_COUNTRY_NAME** attributes are present, provide information additional to that required to identify the user.

  If the terminal is a Telematic terminal, it gives the terminal's network address and possibly, using the **MH_T_TERMINAL_TYPE** and **MH_T_TERMINAL_IDENTIFIER** attributes, its terminal type and identifier. If the terminal is a Telex terminal, it gives the terminal's Telex number.

Whenever two O/R addresses are compared for equality, the following differences are ignored:

- Whether an attribute has a value whose syntax is String(**OM_S_PRINTABLE_STRING**), a value whose syntax is String(**OM_S_TELETEX_STRING**), or both.
- Whether a letter in a value of an attribute not used in DDAs is an uppercase or lowercase letter.
- All leading, all trailing, and all but one consecutive embedded space in an attribute value.

Note: An MD may impose additional equivalence rules upon the O/R addresses it assigns to its own users and DLs. It may define, for example, rules concerning punctuation characters in attribute values, the case of letters in attribute values, or the relative order of DDAs.

As a national matter, MDs may impose additional rules regarding any attribute that may have a value whose syntax is String(**OM_S_PRINTABLE_STRING**), a value whose syntax is String(**OM_S_TELETEX_STRING**), or both. In particular, the rules for deriving from a Teletex string the equivalent printable string may be nationally prescribed.

33.5 MC_C_OR_NAME

An instance of class **MH_C_OR_NAME** comprises a directory name, an O/R address, or both. The name is considered present if, and only if, the **MH_T_DIRECTORY_NAME** attribute is present. The address comprises the attributes specific to the **MH_C_OR_ADDRESS** class and is considered present if, and only if, at least one of those attributes is present.

An O/R name's composition is context sensitive. At submission, the name, the address, or both may be present. At transfer or delivery, the address is present and the name can (but need not) be present. Whether at submission, transfer or delivery, the MTS uses the name, if it is present, only if the address is absent or invalid.

The attribute specific to this class is listed in Table 33-6.

Table 33-6. Attribute Specific to MH_C_OR_NAME

Attribute	Value Syntax	Value Length	Value Number	Value Initially	1988?
MH_T_ DIRECTORY_ NAME	Object(*DS_ C_NAME*)	—	0 or 1	—	1988

- **MH_T_DIRECTORY_NAME**

  This attribute contains the name assigned to the user or DL by the worldwide X.500 directory.

33.6 DS_C_DL_SUBMIT_PERMS

An instance of OM class **DS_C_DL_SUBMIT_PERMS** characterizes an attribute each of whose values are a submit permission. An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, and additionally the OM attributes listed in Table 33-7.

Table 33–7. OM Attributes of DS_C_DL_SUBMIT_PERMS

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DS_PERM_TYPE	Enum(**DS_ Permission_ Type)**	—	1	—
DS_INDIVIDUAL	Object(**MH_C_ OR_NAME)**	—	0 or 1	—
DS_MEMBER_OF_DL	Object(**MH_C_ OR_NAME)**	—	0 or 1	—
DS_PATTERN_MATCH	Object(**MH_C_ OR_NAME)**	—	0 or 1	—
DS_MEMBER_OF_ GROUP	Object(**DS_C_ DS_DN)**	—	0 or more	—

- **DS_PERM_TYPE**

  This attribute contains the type of the permission specified herein. Its value can be one of the following:

  — **DS_PERM_INDIVIDUAL**

  — **DS_PERM_MEMBER_OF_DL**

  — **DS_PERM_PATTERN_MATCH**

  — **DS_PERM_MEMBER_OF_GROUP**

- **DS_INDIVIDUAL**

  This attribute contains the user or unexpanded DL, any of whose O/R names is equal to the specified O/R Name.

- **DS_MEMBER_OF_DL**

  This attribute contains each member of the DL, any of whose O/R names is equal to the specified O/R name, or of each nested DL, recursively.

- **DS_PATTERN_MATCH**

  This attribute contains each user or unexpanded DL, any of whose O/R names matches the specified O/R name pattern.

- **DS_MEMBER_OF_GROUP**

  This attribute contains each member of the group-of-names whose name is specified, or of each nested group-of-names, recursively.

Note that exactly one of the four name attributes will be present at any time, according to the value of the **DS_PERM_TYPE** attribute.

Chapter 34

Global Directory Service Package

The Global Directory Service Package (GDSP) is an OSF extension to the XDS interface. Applications must negotiate use of this package with **ds_version**( ) before using any of the additional features. If an application attempts to use features specific to this package without first negotiating its use, then an appropriate error (for example, **OM_NO_SUCH_CLASS**) is returned by the Object Management function.

The object identifier associated with the GDSP is {**iso(1) identified-organisation(3) icd-ecma(0012) member-company(2) siemens-units(1107) sni(1) directory(3) xdsapi(100) gdsp(0)**} with the following encoding:

\x2B\xC\x2\x88\x53\x1\x3\x64\x0

The identifier is represented by the constant **DSX_GDS_PKG**. The C constants associated with this package are contained in the **xdsgds.h** header file (see the *OSF DCE Application Development Reference*).

The concepts and notation used are first mentioned in Section 30.1. They are also fully explained in Chapters 35 through 37. The attribute types are introduced first, followed by the object classes. Next, the OM class hierarchy and OM class definitions required to support the new attribute types are described.

34.1 GDSP Attribute Types

This section presents the additional directory attribute types that are used with GDSP. Each attribute type has an object identifier, which is the value of the OM attribute **DS_ATTRIBUTE_TYPE**. These object identifiers are represented in the interface by constants with the same name as the directory attribute, and are prefixed by **DSX_A_** so that they can be easily identified.

This section contains two tables that are used to indicate the object identifiers for GDSP attribute types (see Table 34-1), and the values for GDSP attribute types (see Table 34-2), respectively. Following these two tables is a brief description of each attribute. (See Section 32.1 for information on general matching rules.)

Table 34-1 shows the names of the GDSP attribute types, together with the BER encoding of the object identifiers associated with each of them.

Note: The third column of Table 34-1 contains the contents octets of the BER encoding of the object identifier in hexadecimal. All these object identifiers stem from the root **{iso(1) identified-organization(3) idc-ecma(0012) member-company(2) siemens-units(1107) sni(1) directory(3) attribute-type(4)}**.

Table 34–1. Object Identifiers for GDSP Attribute Types

		Object Identifier BER
Package	**Attribute Type**	**Hexadecimal**
GDSP	**DSX_A_ACL**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x01
GDSP	**DSX_A_AT**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x06
GDSP	**DSX_A_ CACHE_ATTR**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x07
GDSP	**DSX_A_CDS_ CELL**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x0D
GDSP	**DSX_A_CDS_ REPLICA**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x0E
GDSP	**DSX_A_ CLIENT**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x0A
GDSP	**DSX_A_ DEFAULT_ DSA**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x08
GDSP	**DSX_A_ DNLIST**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x0B
GDSP	**DSX_A_ LOCDSA**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x09
GDSP	**DSX_A_ MASTER_ KNOWLEDGE**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x00
GDSP	**DSX_A_OCT**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x05
GDSP	**DSX_A_ SHADOWED_ BY**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x03
GDSP	**DSX_A_ SHADOWING_ JOB**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x0C
GDSP	**DSX_A_SRT**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x04
GDSP	**DSX_A_ TIME_STAMP**	\x2B\x0C\x02\x88\x53\x01\x03\x04\x02

Table 34-2 shows the names of the attribute types, together with the OM value syntax used in the interface to represent values of that attribute type. The table also includes the range of lengths permitted for the string types, indicates whether the attribute can be multivalued, and lists which matching rules are provided for the syntax.

Table 34–2. Representation of Values for GDSP Attribute Types

Attribute Type	OM Value Syntax	Value Length	Multi-Valued	Matching Rules
DSX_A_ACL	Object(**DSX_C_GDS_ACL)**	—	No	E
DSX_A_AT	String(**OM_S_PRINTABLE_STRING)**	101	Yes	E,S
DSX_A_CACHE_ATTR	No syntax, no values	—	—	—
DSX_A_CDS_CELL	String(**OM_S_OCTET_STRING)**	36	No	E
DSX_A_CDS_REPLICA	String(**OM_S_OCTET_STRING)**	45	Yes	E
DSX_A_CLIENT	Only a cache entry	—	—	—
DSX_A_DEFAULT_DSA	Only a cache entry	—	—	—
DSX_A_DNLIST	Object(**DS_C_DS_DN)**	1K max.	Yes	E,S
DSX_A_LOCDSA	Only a cache entry	—	—	—
DSX_A_MASTER_KNOWLEDGE	Object(**DS_C_DS_DN)**	1K max.	No	E,S
DSX_A_OCT	String(**OM_S_PRINTABLE_STRING)**	310	Yes	E,S
DSX_A_SHADOWED_BY	Not used yet	—	—	—

Attribute Type	OM Value Syntax	Value Length	Multi-Valued	Matching Rules
DSX_A_ SHADOWING_ JOB	Not used yet	—	—	—
DSX_A_SRT	String(**OM_S_ PRINTABLE_ STRING)**	56	Yes	E,S
DSX_A_ TIME_STAMP	String(**OM_S_ UTC_TIME_ STRING)**	18	No	E,0

Note: With the exception of the **DSX_A_ACL** attribute, the GDSP attributes in Table 34-2 are only to be manipulated through the GDS administration interface (see the *OSF DCE Administration Guide*.)

Descriptions of the GDSP attributes follow:

- **DSX_A_ACL**

  This attribute describes the access rights for one or more Directory Service users.

- **DSX_A_AT**

  This attribute describes the attribute types permitted in GDS. For further information, see the *OSF DCE Administration Guide*.

- **DSX_A_CACHE_ATTR**

  This attribute is used internally by GDSP to separate return values that can be cached from those that cannot be cached.

- **DSX_A_CDS_CELL** and **DSX_A_CDS_REPLICA**

  These two attributes always exist together in the same object. They describe the information necessary for contacting a remote CDS cell.

- **DSX_A_CLIENT**

  This attribute is a cache entry. This naming attribute allows the DUA to retrieve its own PSAP address.

- **DSX_A_DEFAULT_DSA**

  This attribute is a cache entry. This naming attribute allows the DUA to retrieve the PSAP address of its default DSA.

- **DSX_A_DNLIST**

  This attribute is used internally by the GDS DSA.

- **DSX_A_LOCDSA**

  This attribute is a cache entry. This naming attribute allows the DSA to retrieve its own PSAP address.

- **DSX_A_MASTER_KNOWLEDGE**

  This attribute contains the Distinguished Name (DN) of the DSA that holds the master copy of this entry.

- **DSX_A_OCT**

  This attribute describes the object classes supported by the GDS DSA. (For further information, see the *OSF DCE Administration Guide*.)

- **DSX_A_SHADOWED_BY** and **DSX_A_SHADOWING_JOB**

  These two GDSP attributes are intended for future use.

- **DSX_A_SRT**

  This attribute describes the structure of the DNs permitted in GDS.

- **DSX_A_TIME_STAMP**

  This attribute is part of the **DSX_O_SCHEMA** object. It contains the creation time of the **DSX_O_SCHEMA** object.

34.2 GDSP Object Classes

The only additional GDSP object class is **DSX_O_SCHEMA** (see Table 34-3). It is stored in GDS as an object directly under root. The most important attributes of the **DSX_O_SCHEMA** object are the three recurring attributes **DSX_A_OCT**, **DSX_A_AT**, and **DSX_A_SRT**. These three objects describe the GDS DIT structure. For a more detailed explanation of the GDSP **DSX_O_SCHEMA** object, see the *OSF DCE Administration Guide*.

Note: The third column of Table 34-3 contains the contents octets of the BER encoding of the object identifier in hexadecimal. This object identifier stems from the root **{iso(1) identified-organization(3) idc-ecma(0012) member-company(2) siemens-units(1107) sni(1) directory(3) object-class(6)}**.

Table 34–3. Object Identifier for GDSP Object Classes

		Object Identifier BER
Package	**Attribute Type**	**Hexadecimal**
GDSP	**DSX_O_ SCHEMA**	\x2B\x0C\x02\x88\x53\x01\x03\x06\x00

34.3 GDSP OM Class Hierarchy

The remainder of this chapter defines the additional OM classes used by GDSP. This section shows the hierarchical organization of the OM classes that are defined in the following sections, and it shows which classes inherit additional OM attributes from their OM superclasses. In the following list, subclassification is indicated by indentation, and the names of abstract OM classes are represented in italics.

OM_C_OBJECT (defined in the OM package)

- **DS_C_SESSION** (defined in the Directory Service Package)
  - **DSX_C_GDS_SESSION**
- **DS_C_CONTEXT** (defined in the Directory Service Package)
  - **DSX_C_GDS_CONTEXT**
- **DSX_C_GDS_ACL**
- **DSX_C_GDS_ACL_ITEM**

None of the OM classes in the preceding list are encodable using **om_encode( )** and **om_decode( )**.

34.4 DSX_C_GDS_ACL

An instance of OM class **DSX_C_GDS_ACL** describes up to five categories of rights for one or more directory users.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 34-4.

Table 34–4. OM Attributes of DSX_C_GDS_ACL

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DSX_MODIFY_ PUBLIC	Object(**DSX_C_ GDS_ACL_ITEM)**	—	0-4	—
DSX_READ_ STANDARD	Object(**DSX_C_ GDS_ACL_ITEM)**	—	0-4	—
DSX_MODIFY_ STANDARD	Object(**DSX_C_ GDS_ACL_ITEM)**	—	0-4	—
DSX_READ_ SENSITIVE	Object(**DSX_C_ GDS_ACL_ITEM)**	—	0-4	—
DSX_MODIFY_ SENSITIVE	Object(**DSX_C_ GDS_ACL_ITEM)**	—	0-4	—

The OM attributes of **DSX_C_GDS_ACL** are as follows:

- **DSX_MODIFY_PUBLIC**

  This attribute specifies the user, or subtree of users, that can modify attributes classified as public attributes.

- **DSX_READ_STANDARD**

  This attribute specifies the user, or subtree of users, that can read attributes classified as standard attributes.

- **DSX_MODIFY_STANDARD**

  This attribute specifies the user, or subtree of users, that can modify attributes classified as standard attributes.

- **DSX_READ_SENSITIVE**

  This attribute specifies the user, or subtree of users, that can read attributes classified as sensitive attributes.

- **DSX_MODIFY_SENSITIVE**

  This attribute specifies the user, or subtree of users, that can modify attributes classified as sensitive attributes.

34.5 DSX_C_GDS_ACL_ITEM

An instance of OM class **DSX_C_GDS_ACL_ITEM** is a component of a **DSX_C_GDS_ACL**. It specifies the user, or subtree of users, to whom an access right applies.

An instance of this OM class has the OM attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes listed in Table 34-5.

Table 34–5. OM Attributes of DSX_C_GDS_ACL_ITEM

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DSX_ INTERPRETATION	Enum(**DSX_ Interpretation**)	—	1	—
DSX_USER	Object(**DS_C_ DS_DN**)	—	1	—

The OM attributes of a **DSX_C_GDS_ACL_ITEM** are as follows:

- **DSX_INTERPRETATION**

  This attribute specifies the scope of the access right. It can have one of the following values:

  — **DSX_SINGLE_OBJECT**, meaning that the access right is granted to the user specified in the **DSX_USER** OM attribute.

  — **DSX_ROOT_OF_SUBTREE**, meaning that the access right is granted to all users in the subtree below the name specified in the **DSX_USER** OM attribute.

- **DSX_USER**

  This attribute is the DN of the user, or subtree of users, to whom an access right applies.

34.6 DSX_C_GDS_CONTEXT

An instance of OM class **DSX_C_GDS_CONTEXT** comprises per-operation arguments that are accepted by most of the interface functions. GDSP supports additional service controls that are defined by the **DSX_C_GDS_CONTEXT** OM class.

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_CONTEXT**, in addition to the OM attributes listed in Table 34-6.

Table 34–6. OM Attributes of DSX_C_GDS_CONTEXT

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
Service Controls				
DSX_ DUAFIRST	OM_S_ BOOLEAN	—	1	OM_FALSE
DSX_DONT_ STORE	OM_S_ BOOLEAN	—	1	OM_TRUE
DSX_NORMAL_ CLASS	OM_S_ BOOLEAN	—	1	OM_FALSE
DSX_PRIV_ CLASS	OM_S_ BOOLEAN	—	1	OM_FALSE
DSX_RESIDENT_ CLASS	OM_S_ BOOLEAN	—	1	OM_FALSE
DSX_USEDSA	OM_S_ BOOLEAN	—	1	OM_TRUE
DSX_DUA_ CACHE	OM_S_ BOOLEAN	—	1	OM_FALSE
DSX_PREFER_ ADMIN_FUNCS	OM_S_ BOOLEAN	—	1	OM_FALSE

The OM attributes of the **DSX_C_GDS_CONTEXT** OM class are as follows:

- **DSX_DUAFIRST**

  This attribute defines whether the DUA cache or the DSA needs to be read first for query operations. The default value is **OM_FALSE**; that is, search the DSA first, if not found then search the DUA cache.

- **DSX_DONT_STORE**

  This attribute specifies whether the information read from the DSAs by the query functions also needs to be stored in the DUA cache. When this service control is set to **OM_TRUE** (default value), nothing is stored in the DUA cache.

  When this service control is set to **OM_FALSE**, the information read is stored in the DUA cache. The objects returned by **ds_list**( ) and **ds_compare**( ) are stored in the cache without their associated attribute information. The objects returned by **ds_read**( ) and **ds_search**( ) are stored in the cache with all their ''cacheable'' attributes; these are all public attributes, except the **ACL** attribute. This information is only cached when a list of requested attributes is supplied. If all attributes are requested, then nothing is stored in the cache.

The DUA cache categorizes the information stored into three different memory classes. The user specifies the category with the following service controls:

- **DSX_NORMAL_CLASS**

  If this attribute is set to **OM_TRUE**, the entry in the DUA cache is assigned to the class of normal objects. If the number of entries in this class exceeds a maximum value, the entry that is not addressed for the longest period of time is removed from the DUA cache.

- **DSX_PRIV_CLASS**

  If this attribute is set to **OM_TRUE**, the entry in the DUA cache is assigned to the class of privileged objects. Entries can be removed from the class in the same way as normal objects. By using this memory sparingly, the user can protect entries from deletion.

- **DSX_RESIDENT_CLASS**

  If this attribute is set to **OM_TRUE**, the entry in the DUA cache is assigned to the class of resident objects. An entry in this memory class is never removed automatically, rather it can only be removed with **ds_remove_entry**( ). The number of entries is limited; if this limit is exceeded, **ds_add_entry**( ) reports an error.

Only the service control of one memory class can be set. The **ds_add_entry**( ) function also evaluates these service control bits if the function is used on the DUA cache.

- **DSX_DUA_CACHE** and **DSX_USEDSA**

  These attributes define whether the entries in the DUA cache or in the DSA, or both, need to be used when providing the service. Depending on the values of these attributes, the following situations can arise:

  — **DSX_DUA_CACHE** and **DSX_USEDSA**, both **OM_TRUE**

  — The **ds_add_entry( )** and **ds_remove_entry( )** functions report an error.

  — The query functions evaluate the service controls **DS_DONT_USE_COPY** and **DSX_DUAFIRST**. When **DS_DONT_USE_COPY** is **OM_FALSE**, then **DSX_DUAFIRST** determines whether the DUA cache or the DSA is read first. When **DS_DONT_USE_COPY** is **OM_TRUE**, information from the DSA only is read.

  — **DSX_DUA_CACHE**, **OM_TRUE** and **DSX_USEDSA**, **OM_FALSE**

  — The **ds_add_entry( )** and **ds_remove_entry( )** functions and the query functions only go to the DUA cache.

  — **DSX_DUA_CACHE**, **OM_FALSE** and **DSX_USEDSA**, **OM_TRUE**

  — The **ds_add_entry( )** and **ds_remove_entry( )** functions and the query functions only go to the DSA.

  — **DSX_DUA_CACHE** and **DSX_USEDSA**, both **OM_FALSE**

  — The **ds_add_entry( )** and **ds_remove_entry( )** functions and the query functions report an error.

  All other functions always operate on the DSA currently connected.

- **DSX_PREFER_ADM_FUNCS**

  GDS uses the three following optional attributes:

  — **DSX_A_MASTER_KNOWLEDGE**, which contains the Distinguished Name of the DSA that holds the master copy of an entry.

  — **DSX_A_ACL**, which is used for GDS access control.

  — **DS_A_USER_PASSWORD** as an attribute of the **DS_O_DSA** object class, which is used by the GDS shadowing mechanism.

> The **DSX_A_MASTER_KNOWLEDGE** and **DSX_A_ACL** attributes are present in every GDS entry.
>
> When an application requests all attributes, it can prevent any of the three optional attributes from being returned by setting this service control to **OM_FALSE**.
>
> If GDS applications (for example, GDS administration) require these attributes, they are obtained by setting this service control to **OM_TRUE**.

Applications can assume that an object of OM class **DSX_C_GDS_CONTEXT**, created with default values of all its OM attributes, works with all the interface functions. The constant **DS_DEFAULT_CONTEXT** can be used as an argument to functions instead of creating an OM object with default values.

The default **DSX_C_GDS_CONTEXT** is defined in Table 34-7.

Table 34–7. Default DSX_C_GDS_CONTEXT

OM Attribute	Default Value
Common Arguments	
DS_EXT	NULL
DS_OPERATION_PROGRESS	DS_OPERATION_NOT_STARTED
DS_ALIASED_RDNS	0
Service Controls	
DS_CHAINING_PROHIB	OM_TRUE
DS_DONT_DEREFERENCE_ALIASES	OM_FALSE
DS_DONT_USE_COPY	OM_FALSE
DS_LOCAL_SCOPE	OM_FALSE
DS_PREFER_CHAINING	OM_FALSE
DS_PRIORITY	DS_MEDIUM
DS_SCOPE_OF_REFERRAL	DS_COUNTRY
DS_SIZE_LIMIT	-1
DS_TIME_LIMIT	-1
Local Controls	
DS_ASYNCHRONOUS	OM_FALSE
DS_AUTOMATIC_CONTINUATION	OM_TRUE
Private Extensions	
DSX_DUAFIRST	OM_FALSE
DSX_DONT_STORE	OM_TRUE
DSX_NORMAL_CLASS	OM_FALSE
DSX_PRIV_CLASS	OM_FALSE
DSX_RESIDENT_CLASS	OM_FALSE
DSX_USEDSA	OM_TRUE
DSX_DUA_CACHE	OM_FALSE
DSX_PREFER_ADM_FUNCS	OM_FALSE

34.7 DSX_C_GDS_SESSION

An instance of OM class **DSX_C_GDS_SESSION** identifies a particular link from an application program to a GDSP DUA. This additional OM class is necessary if the user either wants to specify a password as part of the user credentials, or wants to specify the GDSP directory identifier, or alternatively wants to specify both a password and the directory identifier. **DSX_C_GDS_SESSION** can be passed as an argument to **ds_bind**( ).

An instance of this OM class has the OM attributes of its superclasses, *OM_C_OBJECT* and **DS_C_SESSION**, in addition to the OM attributes listed in Table 34-8.

Table 34–8. OM Attributes of DSX_C_GDS_SESSION

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
DSX_PASSWORD	String(**OM_S_ OCTET_ STRING)**	—	0 or 1	**NULL**
DSX_DIR_ID	**OM_S_ INTEGER**	—	1	1

The OM attributes of **DSX_C_GDS_SESSION** are as follows:

- **DSX_PASSWORD**

  This attribute indicates the password for the user credentials.

- **DSX_DIR_ID**

  This attribute contains an identifier for distinguishing between several configurations of the Directory Service within a GDS installation. The valid range is from 1 to 20.

Applications can assume that an object of OM class **DSX_C_GDS_SESSION**, created with default values of all its OM attributes, works with all the interface functions. Such a session can be created by passing the constant **DS_DEFAULT_SESSION** as an argument to **ds_bind**( ), having already negotiated the GDSP package.

Table 34-9 defines **DS_DEFAULT_SESSION**.

Table 34–9. Default DSX_C_GDS_SESSION

OM Attribute	**Default Value**
DS_DSA_ADDRESS	Value obtained from the cache or **NULL**
DS_DSA_NAME	Value obtained from the cache or **NULL**
DS_FILE_DESCRIPTOR	Not used
DS_REQUESTOR	**NULL**
DSX_PASSWORD	**NULL**
DSX_DIR_ID	1

Chapter 35

Information Syntaxes

This chapter defines the syntaxes permitted for attribute values. The syntaxes are closely aligned with the types and type constructors of ASN.1. The **OM_value** data type specifies how a value of each syntax is represented in the C interface (see Section 36.2).

35.1 Syntax Templates

The names of certain syntaxes are constructed from *syntax templates*. A syntax template is a lexical construct comprising a primary identifier followed by an asterisk enclosed in parentheses:

identifier (*)

A syntax template encompasses a group of related syntaxes. Any member of the group, without distinction, is indicated by the primary identifier (*identifier*) alone.

A particular member is indicated by the template with the asterisk (*) replaced by one of a set of secondary identifiers associated with the template:

identifier$_1$ (*identifier*$_2$)

35.2 Syntaxes

A variety of syntaxes are defined. Most are functionally equivalent to ASN.1 types, as documented in Sections 35.5 through 35.8.

The following syntaxes are defined:

- **OM_S_BOOLEAN**

  A value of this syntax is a Boolean; that is, it can be **OM_TRUE** or **OM_FALSE**.

- Enum(*)

  A value of any syntax encompassed by this syntax template is one of a set of values associated with the syntax. The only significant characteristic of the values is that they are distinct.

  The group of syntaxes encompassed by this template is open-ended. Zero or more members are added to the group by each package definition. The secondary identifiers that indicate the members are also assigned there.

- **OM_S_INTEGER**

  A value of this syntax is a positive or negative integer.

- **OM_S_NULL**

  The one value of this syntax is a valueless placeholder.

- Object(*)

  A value of any syntax encompassed by this syntax template is an object, which is any instance of a class associated with the syntax.

The group of syntaxes encompassed by this template is open-ended. One member is added to the group by each class definition. The secondary identifier that indicates the member is the name of the class.

- String(*)

  A value of any syntax encompassed by this syntax template is a string (as defined in Section 35.3) whose form and meaning are associated with the syntax.

  The group of syntaxes encompassed by this template is closed. One syntax is defined for each ASN.1 string type. The secondary identifier that indicates the member is, in general, the first word of the type's name.

35.3 Strings

A *string* is an ordered sequence of zero or more bits, octets, or characters accompanied by the string's length.

The value *length* of a string is the number of bits in a *bit string*, octets in an *octet string*, or characters in a *character string*. Any constraints on the value length of a string are specified in the appropriate class definitions. The length is confined to the range 0 to 2^{32}.

Note: The length of a character string does not necessarily equal the number of characters it comprises because, for example, a single character can be represented using several octets.

The elements of a string are numbered. The position of the first element is 0 (zero). The positions of successive elements are successive positive integers.

The syntaxes that form the string group are identified in Table 35-1, which gives the secondary identifier assigned to each such syntax.

Note: The identifiers in the first, second, and third columns of Table 35-1 indicate the syntaxes of bit, octet, and character strings, respectively. The String group comprises all syntaxes identified in the table.

Table 35–1. String Syntax Identifiers

Bit String Identifier	**Octet String Identifier**	**Character String Identifier**
OM_S_BIT_STRING	**OM_S_ENCODING_STRING**[1]	**OM_S_GENERAL_STRING**[2]
	OM_S_OBJECT_IDENTIFIER_STRING[3]	**OM_S_GENERALISED_TIME_STRING**[2]
	OM_S_OCTET_STRING	**OM_S_GRAPHIC_STRING**[2]
		OM_S_IA5_STRING[2]
		OM_S_NUMERIC_STRING[2]
		OM_S_OBJECT_DESCRIPTOR_STRING[2]
		OM_S_PRINTABLE_STRING[2]
		OM_S_TELETEX_STRING[2]
		OM_S_UTC_TIME_STRING[2]
		OM_S_VIDEOTEX_STRING[2]

Bit String Identifier	Octet String Identifier	Character String Identifier
		OM_S_VISIBLE_ STRING[2]

[1] The octets are those that BER permits for the contents octets of the encoding of a value of any ASN.1 type.

[2] The characters are those permitted by ASN.1's type of the corresponding name. Values of these syntaxes are represented in their BER encoded form. The octets by means of which they are represented are those that BER permits for the contents octets of a primitive encoding of a value of that type.

[3] The octets are those that BER permits for the contents octets of the encoding of a value of ASN.1's object identifier type.

35.4 Representation of String Values

In the service interface, a string value is represented by a string data type. This is defined in Section 35.3. The length of a string is the number of octets by which it is represented at the interface. It is confined to the range 0 to 2^{32}.

The length of a character does not need to be equal to the number of characters it comprises because, for example, a single character can be represented using several octets.

It may be necessary to segment large string values when passing them across the interface. A segment is any zero or more contiguous octets of a string value. Segment boundaries are without semantic significance.

35.5 Relationship to ASN.1 Simple Types

As shown in Table 35-2, for every ASN.1 simple type except Real, there is an OM syntax that is functionally equivalent to it. The simple types are listed in the first column of the table; the corresponding syntaxes are listed in the second column.

Table 35–2. Syntax for ASN.1's Simple Types

Type	**Syntax**
Bit String	String**(OM_S_BIT_STRING)**
Boolean	**OM_S_BOOLEAN**
Integer	**OM_S_INTEGER**
Null	**OM_S_NULL**
Object Identifier	String**(OM_S_OBJECT_IDENTIFIER_STRING)**
Octet String	String**(OM_S_OCTET_STRING)**
Real	None[1]

[1] A future edition of XOM may define a syntax corresponding to this type.

35.6 Relationship to ASN.1 Useful Types

As shown in Table 35-3, for every ASN.1 useful type, there is an OM syntax that is functionally equivalent to it. The useful types are listed in the first column of the table; the corresponding syntaxes are listed in the second column.

Table 35–3. Syntaxes for ASN.1's Useful Types

Type	Syntax
External	Object(**OM_C_EXTERNAL)**
Generalized Time	String(**OM_S_GENERALISED_TIME_STRING)**
Object Descriptor	String(**OM_S_OBJECT_DESCRIPTOR_STRING)**
Universal Time	String(**OM_S_UTC_TIME_STRING)**

35.7 Relationship to ASN.1 Character String Types

As shown in Table 35-4, for every ASN.1 character string type, there is an OM syntax that is functionally equivalent to it. The ASN.1 character string types are listed in the first column of the table; the corresponding syntax is listed in the second column.

Table 35–4. Syntaxes for ASN.1's Character String Types

Type	Syntax
General String	String(**OM_S_GENERAL_STRING)**
Graphic String	String(**OM_S_GRAPHIC_STRING)**
IA5 String	String(**OM_S_IA5_STRING)**
—	String(**OM_S_LOCAL_STRING)**
Numeric String	String(**OM_S_NUMERIC_STRING)**
Printable String	String(**OM_S_PRINTABLE_STRING)**
Teletex String	String(**OM_S_TELETEX_STRING)**
Videotex String	String(**OM_S_VIDEOTEX_STRING)**
Visible String	String(**OM_S_VISIBLE_STRING)**

35.8 Relationship to ASN.1 Type Constructors

As shown in Table 35-5, there are functionally equivalent OM syntaxes for some (but not all) ASN.1 type constructors. The constructors are listed in the first column of the table; the corresponding syntaxes are listed in the second column.

Table 35–5. Syntaxes for ASN.1's Type Constructors

Type Constructor	Syntax
Any	String(**OM_S_ENCODING_STRING)**
Choice	**OM_S_OBJECT**
Enumerated	**OM_S_ENUMERATION**
Selection	None[1]
Sequence	**OM_S_OBJECT**
Sequence Of	**OM_S_OBJECT**
Set	**OM_S_OBJECT**
Set Of	**OM_S_OBJECT**
Tagged	None[2]

[1] This type constructor, a purely specification-time phenomenon, has no corresponding syntax.

[2] This type constructor is used to distinguish the alternatives of a choice or the elements of a sequence or set. This function is performed by attribute types.

The effects of the principal type constructors can be achieved, in any of a variety of ways, using objects to group attributes, or using attributes to group values. An OM application designer can (but need not) model these constructors as classes of the following kinds:

- Choice

  An attribute type can be defined for each alternative with just one being permitted in an instance of the class.

- Sequence or Set

  An attribute type can be defined for each sequence or set element. If an element is optional, then the attribute has zero or one value.

- Sequence Of or Set Of

  A single multivalued attribute can be defined.

An ASN.1 definition of an Enumerated Type component of a structured type is generally mapped to an OM attribute with an OM syntax **OM_S_ENUMERATION** in this interface. Where the ASN.1 component is optional, this is generally indicated by an additional member of the enumeration, rather than by the omission of the OM attribute. This leads to simpler programming in the application.

Chapter 36

XOM Service Interface

This chapter describes the following aspects of the XOM service interface:

- The conformance of the DCE X/Open OSI-Abstract-Data Manipulation (XOM) implementation to the X/Open specification.
- The data types whose data values are the parameters and results of the functions that the service makes available to the client.
- An overview of the functions that the service makes available to the client. For a complete description of these functions, see the corresponding reference pages in the *OSF DCE Application Development Reference*.
- The return codes that indicate the outcomes (in particular, the exceptions) that the functions can report.

See Chapter 28 for examples of using the XOM interface.

36.1 Standards Conformance

The DCE XOM implementation conforms to the following specification:

X/Open CAE Specification, OSI-Abstract-Data Manipulation (XOM) (November 1991)

The following apply to the DCE XOM implementation:

- A single workspace for XDS objects is supported.
- The OM package is supported.
- The **om_encode**( ) and **om_decode**( ) functions are not supported. The transfer of objects between workspaces is not envisaged within the DCE environment. The OM classes used by the DCE XDS/XOM API are not encodable.
- Translation to local character sets is not provided.

36.2 XOM Data Types

The data types of the XOM service interface are defined in this section and listed in Table 36-1. These data types are repeated in the XOM reference pages (see **xom.h(4xom)** in the *OSF DCE Application Development Reference*).

Table 36–1. XOM Service Interface Data Types

Data Type	Description
OM_boolean	Type definition for a Boolean data value.
OM_descriptor	Type definition for describing an attribute type and value.
OM_enumeration	Type definition for an Enumerated data value.
OM_exclusions	Type definition for the *exclusions* parameter for **om_get( )**.
OM_integer	Type definition for an Integer data value.
OM_modification	Type definition for the *modification* parameter for **om_put( )**.
OM_object	Type definition for a handle to either a private or a public object.
OM_object_identifier	Type definition for an Object Identifier data value.
OM_private_object	Type definition for a handle to an object in an implementation-defined, or private, representation.
OM_public_object	Type definition for a defined representation of an object that can be directly interrogated by a programmer.
OM_return_code	Type definition for a value returned from all OM functions, indicating either that the function succeeded or why it failed.
OM_string	Type definition for a data value of one of the String syntaxes.
OM_syntax	Type definition for identifying a syntax type.
OM_type	Type definition for identifying an OM attribute type.
OM_type_list	Type definition for enumerating a sequence of OM attribute types.
OM_value	Type definition for representing any data value.

Data Type	Description
OM_value_position	Type definition for designating a particular location within a String data value.
OM_workspace	Type definition for identifying an application-specific API that implements OM, such as directory or message handling.

Some data types are defined in terms of the following *intermediate data types*, whose precise definitions in C are defined by the system:

- **OM_sint**

  The positive and negative integers that can be represented in 16 bits

- **OM_sint16**

  The positive and negative integers that can be represented in 16 bits

- **OM_sint32**

  The positive and negative integers that can be represented in 32 bits

- **OM_uint**

  The nonnegative integers that can be represented in 16 bits

- **OM_uint16**

  The nonnegative integers that can be represented in 16 bits

- **OM_uint32**

  The nonnegative integers that can be represented in 32 bits

Note: The **OM_sint** and **OM_uint** data types are defined by the range of integers they must accommodate. As typically declared in the C interface, they are defined by the range of integers permitted by the host machine's word size. The latter range, however, always encompasses the former.

The type definitions for these data types are as follows:

```
typedef int             OM_sint;
typedef short           OM_sint16;
typedef long int        OM_sint32;
typedef unsigned        OM_uint;
typedef unsigned short  OM_uint16;
typedef long unsigned   OM_uint32;
```

36.2.1 OM_boolean

The C declaration for an **OM_boolean** data value is as follows:

```
typedef OM_uint32 OM_boolean;
```

A data value of this data type is a Boolean; that is, either FALSE or TRUE.

FALSE (**OM_FALSE**) is indicated by 0 (zero). TRUE is indicated by any other integer, although the symbolic constant **OM_TRUE** refers to the integer 1 specifically.

36.2.2 OM_descriptor

The **OM_descriptor** data type is used to describe an attribute type and value. Its C declaration is as follows:

```
typedef struct OM_descriptor_struct
{
   OM_type        type;
   OM_syntax      syntax;
   union  OM_value_union value;
}  OM_descriptor;
```

Note: Other components are encoded in high bits of the syntax member.

See the **OM_value** data type in Section 36.2.16 or the **xom.h(4xom)** reference page in the *OSF DCE Application Development Reference* for a description of the **OM_value_union** structure.

A data value of this type is a descriptor, which embodies an attribute value. An array of descriptors can represent all the values of all the attributes of an object, and is the representation called **OM_public_object**.

A descriptor has the following components:

- *type*

  An **OM_type** data type. It identifies the data type of the attribute value.

- *syntax*

  An **OM_syntax** data type. It identifies the syntax of the attribute value. Components 3 to 7 (that is, the components *long-string* through *private* that follow) are encoded in the high-order bits of this structure member. Therefore, the syntax always needs to be masked with the constant **OM_S_SYNTAX.** For example:

```
my_syntax = my_public_object[3].syntax &
            OM_S_SYNTAX;

my_public_object[4].syntax =
my_syntax + (my_public_object[4].syntax &
~OM_S_SYNTAX);
```

- *long-string*

  An **OM_boolean** data type. It is **OM_TRUE** only if the descriptor is a service-generated descriptor and the length of the value is greater than an implementation-defined limit.

  This component occupies bit 15 (0x8000) of the syntax and is represented by the constant **OM_S_LONG_STRING.**

- *no-value*

  An **OM_boolean** data type. It is **OM_TRUE** only if the descriptor is a service-generated descriptor and the value is not present because **OM_EXCLUDE_VALUES** or **OM_EXCLUDE_MULTIPLES** is set in **om_get( ).**

  This component occupies bit 14 (0x4000) of the syntax and is represented by the constant **OM_S_NO_VALUE.**

- *local-string*

  An **OM_boolean** data type, significant only if the syntax is one of the string syntaxes. It is **OM_TRUE** only if the string is represented in an implementation-defined local character set. The local character set may be more amenable for use as keyboard input or display output than the nonlocal character set, and can include specific treatment of line termination sequences. Certain interface functions can convert information in string syntaxes to or from the local representation, which may result in a loss of information.

This component occupies bit 13 (0x2000) of the syntax and is represented by the constant **OM_S_LOCAL_STRING.** The DCE XOM implementation does not support translation of strings to a local character set.

- *service-generated*

  An **OM_boolean** data type. It is **OM_TRUE** only if the descriptor is a service-generated descriptor and the first descriptor of a public object, or the defined part of a private object (see the XOM reference pages in the *OSF DCE Application Development Reference*).

  This component occupies bit 12 (0x1000) of the syntax and is represented by the constant **OM_S_SERVICE_GENERATED.**

- *private*

  An **OM_boolean** data type. It is **OM_TRUE** only if the descriptor in the service-generated public object contains a reference to the handle of a private subobject, or in the defined part of a private object.

  Note: This applies only when the descriptor is a service-generated descriptor. The client need not set this bit in a client-generated descriptor that contains a reference to a private object.

  In the C interface, this component occupies bit 11 (0x0800) of the syntax and is represented by the constant **OM_S_PRIVATE.**

- *value*

  An **OM_value** data type. It identifies the attribute value.

36.2.3 OM_enumeration

The **OM_enumeration** data type is used to indicate an Enumerated data value. Its C declaration is as follows:

```
typedef OM_sint32 OM_enumeration;
```

A data value of this data type is an attribute value whose syntax is **OM_S_ENUMERATION.**

36.2.4 OM_exclusions

The **OM_exclusions** data type is used for the *exclusions* parameter of **om_get( )**. Its C declaration is as follows:

```
typedef OM_uint OM_exclusions;
```

A data value of this data type is an unordered set of one or more values, all of which are distinct. Each value indicates an exclusion, as defined by **om_get( )**, and is chosen from the following set:

- **OM_EXCLUDE_ALL_BUT_THESE_TYPES**
- **OM_EXCLUDE_MULTIPLES**
- **OM_EXCLUDE_ALL_BUT_THESE_VALUES**
- **OM_EXCLUDE_VALUES**
- **OM_EXCLUDE_SUBOBJECTS**
- **OM_EXCLUDE_DESCRIPTORS**

Alternatively, the single value **OM_NO_EXCLUSIONS** can be chosen; this selects the entire object.

Each value except **OM_NO_EXCLUSIONS** is represented by a distinct bit. The presence of the value is represented as 1; its absence is represented as 0 (zero). Thus, multiple exclusions are requested by ORing the values that indicate the individual exclusions.

36.2.5 OM_integer

The **OM_integer** data type is used to indicate an integer data value. Its C declaration is as follows:

```
typedef OM_sint32 OM_integer;
```

A data value of this data type is an attribute value whose syntax is **OM_S_INTEGER**.

36.2.6 OM_modification

The **OM_modification** data type is used for the *modification* parameter of **om_put( )**. Its C declaration is as follows:

```
typedef OM_uint OM_modification;
```

A data value of this data type indicates a kind of modification, as defined by **om_put( )**. It is chosen from the following set:

- **OM_INSERT_AT_BEGINNING**
- **OM_INSERT_AT_CERTAIN_POINT**
- **OM_INSERT_AT_END**
- **OM_REPLACE_ALL**
- **OM_REPLACE_CERTAIN_VALUES**

36.2.7 OM_object

The **OM_object** data type is used as a handle to either a private or a public object. Its C declaration is as follows:

```
typedef struct OM_descriptor_struct *OM_object;
```

A data value of this data type represents an object, which can be either public or private. It is an ordered sequence of one or more instances of the **OM_descriptor** data type. See the **OM_private_object** and **OM_public_object** data types for restrictions on that sequence (Sections 36.2.9 and 36.2.10, respectively).

36.2.8 OM_object_identifier

The **OM_object_identifier** data type is used as an ASN.1 object identifier. Its C declaration is as follows:

```
typedef OM_string OM_object_identifier;
```

A data value of this data type contains an octet string that comprises the contents octets of the BER encoding of an ASN.1 object identifier.

36.2.8.1 C Declaration of Object Identifiers

Every application program that uses a class or another object identifier must explicitly import it into every compilation unit (C source module) that uses it. Each such class or object identifier name must be explicitly exported from just one compilation module. Most application programs find it convenient to export all the names they use from the same compilation unit. Exporting and importing is performed using the following two macros:

- The importing macro makes the class or other object identifier constants available within a compilation unit.
  - **OM_IMPORT**(*class_name*)
  - **OM_IMPORT**(*OID_name*)
- The exporting macro allocates memory for the constants that represent the class or another object identifier.
  - **OM_EXPORT**(*class_name*)
  - **OM_EXPORT**(*OID_name*)

Object identifiers are defined in the appropriate header files, with the definition identifier having the prefix **OMP_O_** followed by the variable name for the object identifier. The constant itself provides the hexadecimal value of the object identifier string.

36.2.8.2 Use of Object Identifiers in C

The following macro initializes a descriptor:

OM_OID_DESC(*type, OID_name*)

It sets the *type* component to that given, sets the *syntax* component to **OM_S_OBJECT_IDENTIFIER_STRING**, and sets the *value* component to the specified object identifier.

The following macro initializes a descriptor to mark the end of a client-allocated public object:

OM_NULL_DESCRIPTOR

For each class there is a global variable of type **OM_STRING** with the same name; for example, the External class has a variable called **OM_C_EXTERNAL**. This is also the case for other object identifiers; for example, the object identifier for BER rules has a variable called **OM_BER**. This global variable can be supplied as a parameter to functions when required.

This variable is valid only when it is exported by an **OM_EXPORT** macro and imported by an **OM_IMPORT** macro in the compilation units that use it. This variable cannot form part of a descriptor, but the value of its length and elements components can be used. The following code fragment provides examples of the use of the macros and constants.

```
/* Examples of the use of the macros and constants */

#include <xom.h>

OM_IMPORT(OM_C_ENCODING)
OM_IMPORT(OM_CANONICAL_BER)

/*  The following sequence must appear in just one compilation
 *  unit in place of the above:
 *
 *  #include <xom.h>
 *
 *  OM_EXPORT(OM_C_ENCODING)
 *  OM_EXPORT(OM_CANONICAL_BER)
 */

main()
{
/* Use #1 - Define a public object of class Encoding
 *          (Note: xxxx is a Message Handling class which can be
 *           encoded)
 */
OM_descriptor my_public_object[] = {
        OM_OID_DESC(OM_CLASS, OM_C_ENCODING),
        OM_OID_DESC(OM_OBJECT_CLASS, MA_C_xxxx),
        { OM_OBJECT_ENCODING, OM_S_ENCODING_STRING, some_BER_value },
        OM_OID_DESC(OM_RULES, OM_CANONICAL_BER),
        OM_NULL_DESCRIPTOR
        };

/* Use #2 - Pass class Encoding as a parameter to om_instance()
 */
return_code = om_instance(my_object, OM_C_ENCODING,
&boolean_result);
}
```

36.2.9 OM_private_object

The **OM_private_object** data type is used as a handle to an object in an implementation-defined or private representation. Its C declaration is as follows:

```
typedef OM_object OM_private_object;
```

A data value of this data type is the designator or handle to a private object. It comprises a single descriptor whose *type* component is **OM_PRIVATE_OBJECT** and whose *syntax* and *value* components are unspecified.

Note: The descriptor's *syntax* and *value* components are essential to the service's proper operation with respect to the private object.

36.2.10 OM_public_object

The **OM_public_object** data type is used to define an object that can be directly accessed by a programmer. Its C declaration is as follows:

```
typedef OM_object OM_public_object;
```

A data value of this data type is a public object. It comprises one or more (ususally more) descriptors, all but the last of which represent values of attributes of the object.

The descriptors for the values of a particular attribute with two or more values are adjacent to one another in the sequence. Their order is that of the values they represent. The order of the resulting groups of descriptors is unspecified.

Since the Class attribute specific to the Object class is represented among the descriptors, it must be represented before any other attributes. Regardless of whether or not the Class attribute is present, the syntax field of the first descriptor must have the **OM_S_SERVICE_GENERATED** bit set or cleared appropriately.

The last descriptor signals the end of the sequence of descriptors. The last descriptor's *type* component is **OM_NO_MORE_TYPES** and its *syntax* component is **OM_S_NO_MORE_SYNTAXES**. The last descriptor's *value* component is unspecified.

36.2.11 OM_return_code

The **OM_return_code** data type is used for a value that is returned from all OM functions, indicating either that the function succeeded or why it failed. Its C declaration is as follows:

```
typedef OM_uint OM_return_code;
```

A data value of this data type is the integer in the range 0 to 2^{16} that indicates an outcome of an interface function. It is chosen from the set specified in Section 36.4.

Integers in the narrower range 0 to 2^{15} are used to indicate the return codes they define.

36.2.12 OM_string

The **OM_string** data type is used for a data value of String syntax. Its C declaration is as follows:

```
typedef OM_uint32 OM_string_length;
typedef struct {
      OM_string_length length;
      void *elements;
} OM_string;

#define OM_STRING(string)\
      { (OM_string_length)(sizeof(string)-1), (string) }
```

A data value of this data type is a string; that is, an instance of a String syntax. A string is specified either in terms of its length or whether or not it terminates with **NULL**. A string has the following components:

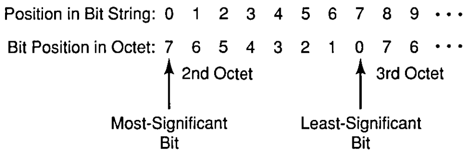
- *length* (**OM_string_length**)

  The number of octets by means of which the string is represented, or the **OM_LENGTH_UNSPECIFIED** value if the string terminates with **NULL**.

- *elements*

  The string's elements. The bits of a bit string are represented as a sequence of octets (see Figure 36-1). The first octet stores the number of unused bits in the last octet. The bits in the bit string, commencing with the first bit and proceeding to the trailing bit, are placed in bits 7 to 0 of the second octet. These are followed by bits 7 to 0 of the third octet, then by bits 7 to 0 of each octet in turn, followed by as many bits as are required of the final octet, commencing with bit 7.

Figure 36–1. OM_String Elements


Position in Bit String: 0 1 2 3 4 5 6 7 8 9 • • •
Bit Position in Octet: 7 6 5 4 3 2 1 0 7 6 • • •
2nd Octet
3rd Octet
Most-Significant Bit
Least-Significant Bit

The service supplies a string value with a specified length. The client can supply a string value to the service in either form, either with a specified length or terminated with **NULL**.

The characters of a character string are represented as any sequence of octets permitted as the primitive contents octets of the BER encoding of an ASN.1 type value. The ASN.1 type defines the type of character string. A 0 (zero) value character follows the characters of the character string, but is not encompassed by the *length* component. Thus, depending upon the type of character string, the 0 (zero) value character can delimit the characters of the character string.

The **OM_STRING** macro is provided for creating a data value of this data type, given only the value of its *elements* component. The macro, however, applies to octet strings and character strings, but not to bit strings.

36.2.13 OM_syntax

The **OM_syntax** data type is used to identify a syntax type. Its C declaration is as follows:

```
typedef OM_uint16 OM_syntax;
```

A data value of this data type is an integer in the range 0 to 2^9 that indicates an individual syntax or a set of syntaxes taken together.

The data value is chosen from among the following:

- **OM_S_BIT_STRING**
- **OM_S_BOOLEAN**

- **OM_S_ENCODING_STRING**
- **OM_S_ENUMERATION**
- **OM_S_GENERAL_STRING**
- **OM_S_GENERALISED_TIME_STRING**
- **OM_S_GRAPHIC_STRING**
- **OM_S_IA5_STRING**
- **OM_S_INTEGER**
- **OM_S_NULL**
- **OM_S_NUMERIC_STRING**
- **OM_S_OBJECT**
- **OM_S_OBJECT_DESCRIPTOR_STRING**
- **OM_S_OBJECT_IDENTIFIER_STRING**
- **OM_S_OCTET_STRING**
- **OM_S_PRINTABLE_STRING**
- **OM_S_TELETEX_STRING**
- **OM_S_VIDEOTEX_STRING**
- **OM_S_VISIBLE_STRING**
- **OM_S_UTC_TIME_STRING**

Integers in the narrower range 0 to 2^9 are used to indicate the syntaxes they define. The integers in the range 2^9 to 2^{10} are reserved for vendor extensions. Wherever possible, the integers used are the same as the corresponding ASN.1 universal class number.

36.2.14 OM_type

The **OM_type** data type is used to identify an OM attribute type. Its C declaration is as follows:

```
typedef OM_uint16 OM_type;
```

A data value of this data type is an integer in the range 0 to 2^{16} that indicates a type in the context of a package. However, the following values in Table 36-2 are assigned meanings by the respective data types.

Table 36–2. Assigning Meanings to Values

Value	Data Type
OM_NO_MORE_TYPES	OM_type_list
OM_PRIVATE_OBJECT	OM_private_object

Integers in the narrower range 0 to 2^{15} are used to indicate the types they define.

36.2.15 OM_type_list

The **OM_type_list** data type is used to enumerate a sequence of OM attribute types. Its C declaration is as follows:

```
typedef OM_type *OM_type_list;
```

A data value of this data type is an ordered sequence of zero or more type numbers, each of which is an instance of the **OM_type** data type.

An additional data value, **OM_NO_MORE_TYPES**, follows and thus delimits the sequence. The C representation of the sequence is an array.

36.2.16 OM_value

The **OM_value** data type is used to represent any data value. Its C declaration is as follows:

```
typedef struct {
      OM_uint32 padding;
      OM_object object;
} OM_padded_object;
```

```
typedef union OM_value_union {
      OM_string          string;
      OM_boolean         boolean;
      OM_enumeration     enumeration;
      OM_integer         integer;
      OM_padded_object   object;
} OM_value;
```

Note: The first type definition (in particular, its **padding** component) aligns the **object** component with the *elements* component of the **string** component in the second type definition. This facilitates initialization in C.

The identifier **OM_value_union** is defined for reasons of compilation order. It is used in the definition of the **OM_descriptor** data type.

A data value of this data type is an attribute value. It has no components if the value's syntax is **OM_S_NO_MORE_SYNTAXES** or **OM_S_NO_VALUE**. Otherwise, it has one of the following components:

- **string**

  The value if its syntax is a string syntax

- **boolean**

  The value if its syntax is **OM_S_BOOLEAN**

- **enumeration**

  The value if its syntax is **OM_S_ENUMERATION**

- **integer**

  The value if its syntax is **OM_S_INTEGER**

- **object**

  The value if its syntax is **OM_S_OBJECT**

Note: A data value of this data type is only displayed as a component of a descriptor. Thus, it is always accompanied by indicators of the value's syntax. The latter indicator reveals which component is present.

36.2.17 OM_value_length

The **OM_value_length** data type is used to indicate the number of bits, octets, or characters in a string. Its C declaration is as follows:

```
typedef OM_uint32 OM_value_length;
```

A data value of this data type is an integer in the range 0 to 2^{32} that represents the number of bits in a bit string, octets in an octet string, or characters in a character string.

Note: This data type is not used in the definition of the interface. It is provided for use by client programmers for defining attribute constraints.

36.2.18 OM_value_position

The **OM_value_position** data type is used to indicate an attribute value's position within an attribute. Its C declaration is as follows:

```
typedef OM_uint32 OM_value_position;
```

A data value of this data type is an integer in the range 0 to 2^{32}-1 that indicates the position of a value within an attribute. However, the value **OM_ALL_VALUES** has the meaning assigned to it by **om_get**( ).

36.2.19 OM_workspace

The **OM_workspace** data type is used to identify an application-specific API that implements OM; for example, directory or message handling. Its C declaration is as follows:

```
typedef void *OM_workspace;
```

A data value of this data type is the designator or handle for a workspace.

36.3 XOM Functions

This section provides an overview of the XOM service interface functions as listed in Table 36-3. For a full description of these functions, see the corresponding reference pages in the **(3xom)** section of the *OSF DCE Application Development Reference*.

Table 36–3. XOM Service Interface Functions

Function	Description
om_copy( )	Copies a private object.
om_copy_value( )	Copies a string between private objects.
om_create( )	Creates a private object.
om_decode( )	This function is not supported by the DCE XOM interface; it returns an **OM_FUNCTION_DECLINED** error.
om_delete( )	Deletes a private or service-generated object.
om_encode( )	This function is not supported by the DCE XOM interface; it returns an **OM_FUNCTION_DECLINED** error.
om_get( )	Gets copies of attribute values from a private object.
om_instance( )	Tests an object's class.
om_put( )	Puts attribute values into a private object.
om_read( )	Reads a segment of a string in a private object.
om_remove( )	Removes attribute values from a private object.
om_write( )	Writes a segment of a string into a private object.

The purpose and range of capabilities of the service interface functions can be summarized as follows:

- **om_copy( )**

  This function creates an independent copy of an existing private object and all its subobjects. The copy is placed in the workspace of the original object, or in another workspace specified by the DCE client.

- **om_copy_value( )**

  This function replaces an existing attribute value or inserts a new value in one private object with a copy of an existing attribute value found in another. Both values must be strings.

- **om_create()**

  This function creates a new private object that is an instance of a particular class. The object can be initialized with the attribute values specified as initial in the class definition.

  The service does not permit the client to explicitly create instances of all classes, but rather only those indicated by a package's definition as having this property.

- **om_delete()**

  This function deletes a service-generated public object, or makes a private object inaccessible.

- **om_get()**

  This function creates a new public object that is an exact but independent copy of an existing private object. The client can request certain exclusions, each of which reduces the copy to a part of the original. The client can also request that values be converted from one syntax to another before they are returned.

  The copy can exclude: attributes of types other than those specified, values at positions other than those specified within an attribute, values of multivalued attributes, copies of (not handles for) subobjects, or all attribute values. Excluding all attribute values reveals only an attribute's presence.

- **om_instance()**

  This function determines whether an object is an instance of a particular class. The client can determine an object's class simply by inspection. This function is useful since it reveals that an object is an instance of a particular class, even if the object is an instance of a subclass of that class.

- **om_put( )**

  This function places or replaces in one private object copies of the attribute values of another public or private object.

  The source values can be inserted before any existing destination values, before the value at a specified position in the destination attribute, or after any existing destination values. Alternatively, the source values can be substituted for any existing destination values or for the values at specified positions in the destination attribute.

- **om_read( )**

  This function reads a segment of a value of an attribute of a private object. The value must be a string. The value can first be converted from one syntax to another. This function enables the client to read an arbitrarily long value without requiring that the service place a copy of the entire value in memory.

- **om_remove( )**

  This function removes and discards particular values of an attribute of a private object. The attribute itself is removed if no values remain.

- **om_write( )**

  This function writes a segment of an attribute value to a private object. The value must be a string. The segment can first be converted from one syntax to another. The written segment becomes the value's last segment since any elements beyond it are discarded. The function enables the client to write an arbitrarily long value without having to place a copy of the entire value in memory.

36.4 XOM Return Codes

This section defines the return codes of the service interface, and thus the exceptions that can prevent the successful completion of an interface function. Table 36-4 identifies the abbreviated column headings that are used in Table 36-5; see Table 36-4 for the complete function names of the abbreviated column heads used in Table 36-5.

Table 36-5 lists the XOM return codes and the functions to which they apply. (The information in this table also appears in the ERRORS sections of the function descriptions in the **(3xom)** reference pages in the *OSF DCE Application Development Reference*.) The first column of Table 36-5 lists the return codes. The other columns identify the return codes that apply to each function by means of an x.

Table 36–4. OM Functions and their Corresponding Abbreviations

Function	Abbreviation
om_copy( )	Cop
om_copy_value( )	CoV
om_create( )	Cre
om_decode( )	Dec
om_delete( )	Del
om_encode( )	Enc
om_get( )	Get
om_instance( )	Ins
om_put( )	Put
om_read( )	Rea
om_remove( )	Rem
om_write( )	Wri

Table 36–5. XOM Service Interface Return Codes

Return Code	Cop	CoV	Cre	Dec	Del	Enc	Get	Ins	Put	Rea	Rem	Wri
OM_SUCCESS	x	x	x	x	x	x	x	x	x	x	x	x
OM_ ENCODING_ INVALID	—	—	—	x	—	—	—	—	—	—	—	—
OM_ FUNCTION_ DECLINED	—	x	x	—	—	x	—	—	x	—	x	x
OM_ FUNCTION_ INTERRUPTED	x	x	x	x	x	x	x	x	x	x	x	x
OM_ MEMORY_ INSUFFICIENT	x	x	x	x	x	x	x	x	x	x	x	x
OM_ NETWORK_ ERROR	x	x	x	x	x	x	x	x	x	x	x	x
OM_NO_ SUCH_ CLASS	x	—	x	x	—	—	—	x	x	—	—	—
OM_NO_ SUCH_ EXCLUSION	—	—	—	—	—	—	x	—	—	—	—	—
OM_NO_ SUCH_ MODIFICATION	—	—	—	—	—	—	—	—	x	—	—	—

Return Code	Cop	CoV	Cre	Dec	Del	Enc	Get	Ins	Put	Rea	Rem	Wri
OM_NO_SUCH_OBJECT	x	x	—	x	x	x	x	x	x	x	x	x
OM_NO_SUCH_RULES	—	—	—	x	—	x	—	—	—	—	—	—
OM_NO_SUCH_SYNTAX	—	—	—	—	x	—	—	x	x	—	—	x
OM_NO_SUCH_TYPE	—	x	—	—	x	—	x	—	x	x	x	x
OM_NO_SUCH_WORKSPACE	x	—	x	—	—	—	—	—	—	—	—	—
OM_NOT_AN_ENCODING	—	—	—	x	—	—	—	—	—	—	—	—
OM_NOT_CONCRETE	—	—	x	—	—	—	—	—	x	—	—	—
OM_NOT_PRESENT	—	x	—	—	—	—	—	—	x	x	—	x
OM_NOT_PRIVATE	x	x	—	x	—	x	x	—	x	x	x	x
OM_NOT_THE_SERVICES	—	—	—	—	x	—	—	x	—	—	—	—

Return Code	Cop	CoV	Cre	Dec	Del	Enc	Get	Ins	Put	Rea	Rem	Wri
OM_PERMANENT_ERROR	x	x	x	x	x	x	x	x	x	x	x	x
OM_POINTER_INVALID	x	x	x	x	x	x	x	x	x	x	x	x
OM_SYSTEM_ERROR	x	x	x	x	x	x	x	x	x	x	x	x
OM_TEMPORARY_ERROR	x	x	x	x	x	x	x	x	x	x	x	x
OM_TOO_MANY_VALUES	x	—	—	x	—	—	—	—	x	—	—	—
OM_VALUES_NOT_ADJACENT	—	—	—	—	—	—	—	—	x	—	—	—
OM_WRONG_VALUE_LENGTH	—	x	—	x	—	—	—	—	x	—	—	x
OM_WRONG_VALUE_MAKEUP	—	—	—	x	—	—	—	—	x	—	—	x
OM_WRONG_VALUE_NUMBER	—	—	—	x	—	—	—	—	x	—	—	—

Return Code	Cop	CoV	Cre	Dec	Del	Enc	Get	Ins	Put	Rea	Rem	Wri
OM_WRONG_ VALUE_ POSITION	—	—	—	—	—	—	—	—	x	—	—	x
OM_WRONG_ VALUE_ SYNTAX	—	x	—	x	—	—	x	—	x	x	—	x
OM_WRONG_ VALUE_ TYPE	—	x	—	x	—	—	x	—	x	—	—	—

The return code values are as follows:

0 **OM_SUCCESS**
The function completed successfully.

1 **OM_ENCODING_INVALID**
The octets that constitute the value of an encoding's Object Encoding attribute are invalid.

2 **OM_FUNCTION_DECLINED**
The function does not apply to the object to which it is addressed.

3 **OM_FUNCTION_INTERRUPTED**
The function is aborted by an external force; for example, a keystroke designated for this purpose at a user interface.

4 **OM_MEMORY_INSUFFICIENT**
The service cannot allocate the main memory it needs to complete the function.

5 **OM_NETWORK_ERROR**
The service could not successfully employ the network upon which its implementation depends.

6 **OM_NO_SUCH_CLASS**
A purported class identifier is not defined.

7 **OM_NO_SUCH_EXCLUSION**
A purported exclusion identifier is not defined.

8 **OM_NO_SUCH_MODIFICATION**
A purported modification identifier is not defined.

9 **OM_NO_SUCH_OBJECT**
A purported object is nonexistent, or the purported handle is invalid.

10 **OM_NO_SUCH_RULES**
A purported rules identifier is not defined.

11 **OM_NO_SUCH_SYNTAX**
A purported syntax identifier is not defined.

12 **OM_NO_SUCH_TYPE**
A purported type identifier is not defined.

13 **OM_NO_SUCH_WORKSPACE**
A purported workspace is nonexistent.

14 **OM_NOT_AN_ENCODING**
An object is not an instance of the Encoding class.

15 **OM_NOT_CONCRETE**
A class is abstract, not concrete.

16 **OM_NOT_PRESENT**
An attribute value is absent, not present.

17 **OM_NOT_PRIVATE**
An object is public, not private.

18 **OM_NOT_THE_SERVICES**
An object is a client-generated object, rather than a service-generated or private object.

19 **OM_PERMANENT_ERROR**
The service encountered a permanent difficulty other than those indicated by other return codes.

20 **OM_POINTER_INVALID**
In the C interface, an invalid pointer is supplied as a function parameter, or as the receptacle for a function result.

21 **OM_SYSTEM_ERROR**
The service could not successfully employ the operating system upon which its implementation depends.

22 **OM_TEMPORARY_ERROR**
The service encountered a temporary difficulty other than those indicated by other return codes.

23 **OM_TOO_MANY_VALUES**
An implementation limit prevents a further attribute value from being added to an object. This limit is undefined.

24 **OM_VALUES_NOT_ADJACENT**
The descriptors for the values of a particular attribute are not adjacent.

25 **OM_WRONG_VALUE_LENGTH**
An attribute has, or would have, a value that violates the value length constraints in force.

26 **OM_WRONG_VALUE_MAKEUP**
An attribute has, or would have, a value that violates a constraint on the value's syntax.

27 **OM_WRONG_VALUE_NUMBER**
An attribute has, or would have, a value that violates the value number constraints in force.

28 **OM_WRONG_VALUE_POSITION**
The use defined for value position in the parameter or parameters of a function is invalid.

29 **OM_WRONG_VALUE_SYNTAX**
An attribute has, or would have, a value whose syntax is not permitted.

30 **OM_WRONG_VALUE_TYPE**
An object has, or would have, an attribute whose type is not permitted.

Chapter 37

Object Management Package

This chapter defines the Object Management Package (OMP). The object identifier (referred to as **om**) assigned to the package, as defined by this guide, is the object identifier specified in ASN.1 as **{joint-iso-ccitt(2) mhs-motis(6) group(6) white(1) api(2) om(4)}**.

37.1 Class Hierarchy

This section shows the hierarchical organization of the OM classes. Subclassification is indicated by indentation, and the names of abstract classes are in italics. Thus, for example, **OM_C_ENCODING** is an immediate subclass of *OM_C_OBJECT*, an abstract class. The names of classes to which **om_encode**( ) applies are in boldface. (DCE XOM does not support the encoding of any OM classes.) The **om_create**( ) function applies to all concrete classes.

- *OM_C_OBJECT*
  - **OM_C_ENCODING**
  - **OM_C_EXTERNAL**

37.2 Class Definitions

The following subsections define the OM classes.

37.2.1 OM_C_ENCODING

An instance of class **OM_C_ENCODING** is an object represented in a form suitable for transmission between workspaces, for transport via a network, or for storage in a file. Encoding can also be a suitable way of indicating to an intermediate service provider (for example, a directory, or message transfer system) an object that it does not recognize.

This class has the attributes of its superclass, *OM_C_OBJECT*, in addition to the specific attributes listed in Table 37-1.

Table 37–1. Attributes Specific to OM_C_ENCODING

Attribute	Value Syntax	Value Length	Value Number	Value Initially
OM_OBJECT_CLASS	String(**OM_S_OBJECT_IDENTIFIER_STRING**)	—	1	—
OM_OBJECT_ENCODING	String[1]	—	1	—
OM_RULES	String(**OM_S_OBJECT_IDENTIFIER_STRING**)	—	1	**ber**

[1]If the Rules attribute is **ber** or **canonical-ber**, the syntax of the present attribute must be String(**OM_S_ENCODING_STRING**).

- **OM_OBJECT_CLASS**

  This attribute identifies the class of the object that the Object Encoding attribute encodes. The class must be concrete.

- **OM_OBJECT_ENCODING**

  This attribute is the encoding itself.

- **OM_RULES**

  This attribute identifies the set of rules that are followed to produce the Object Encoding attribute. Among the defined values of this attribute are those represented as follows:

  — **OM_BER**

  This value is specified in ASN.1 as **{joint-iso-ccitt(2) asnl(1) basic-encoding(1)}**. This value indicates the BER.[1]

  — **OM_CANONICAL_BER**

  This value is specified in ASN.1 as **{joint-iso-ccitt(2) mhs-motis(6) group(6) white(1) api(2) om(4) canonical-ber(4)}**. This value indicates the canonical BER.[2]

Note: In general, an instance of this class cannot appear as a value whose syntax is Object (*C*) if *C* is not **OM_C_ENCODING**, even if the class of the object encoded is *C*.

37.2.2 OM_C_EXTERNAL

An instance of class **OM_C_EXTERNAL** is a data value and one or more information items that describe the data value and identify its data type. This class corresponds to ASN.1's External type, and thus the class and the attributes specific to it are described indirectly in the specification of ASN.1.[3]

[1] (See Clause 25.2 of Recommendation X.209, ''Specification of Basic Encoding Rules for Abstract Syntax Notation 1 (ASN.1),'' *CCITT Blue Book*, Fascicle VIII.4, International Telecommunications Union, 1988. Also published by ISO as *ISO 8825*.)

[2] (See Clause 8.7 of Recommendation X.509, ''The Directory: Authentication Framework,'' *CCITT Blue Book,* International Telecommunications Union, 1988. Also published by ISO as *ISO 9594-8*.)

[3] (See Clause 34 of Recommendation X.208, ''Specification of Abstract Syntax Notation 1 (ASN.1),'' *CCITT Blue Book*, Fascicle VIII.4, International Telecommunications Union, 1988. Also published by ISO as *ISO 8824*.)

This class has the attributes of its superclass, *OM_C_OBJECT*, in addition to the OM attributes specific to this class that are listed in Table 37-2.

Table 37–2. Attributes Specific to OM_C_EXTERNAL

Attribute	Value Syntax	Value Length	Value Number	Value Initially
OM_ ARBITRARY_ ENCODING	String(**OM_S_ BIT_STRING)**	—	0 or 1[1]	—
OM_ASN1_ ENCODING	String(**OM_S_ ENCODING_ STRING)**	—	0 or 1[1]	—
OM_DATA_ VALUE_ DESCRIPTOR	String(**OM_S_ OBJECT_ DESCRIPTOR_ STRING)**	—	0 or 1	—
OM_DIRECT_ REFERENCE	String(**OM_S_ OBJECT_ IDENTIFIER_ STRING)**	—	0 or 1	—
OM_INDIRECT_ REFERENCE	**OM_S_ INTEGER**	—	0 or 1	—
OM_OCTET_ ALIGNED_ ENCODING	String(**OM_S_ OCTET_ STRING)**	—	0 or 1[1]	—

[1] Only one of these three attributes is present.

- **OM_ARBITRARY_ENCODING**

  This attribute is a representation of the data value as a bit string.

- **OM_ASN1_ENCODING**

  The data value. This attribute can be present only if the data type is an ASN.1 type.

If this attribute value's syntax is an Object syntax, the data value's representation is that produced by **om_encode( )** when its *Object* parameter is the attribute value and its *Rules* parameter is **ber**. Thus, the Object's class must be one to which **om_encode( )** applies.

- **OM_DATA_VALUE_DESCRIPTOR**

  This attribute contains a description of the data value.

- **OM_DIRECT_REFERENCE**

  This attribute contains a direct reference to the data type.

- **OM_INDIRECT_REFERENCE**

  This attribute contains an indirect reference to the data type.

- **OM_OCTET_ALIGNED_ENCODING**

  This attribute contains a representation of the data value as an octet string.

37.2.3 OM_C_OBJECT

The class **OM_C_OBJECT** represents information objects of any variety. This abstract class is distinguished by the fact that it has no superclass and that all other classes are its subclasses.

The attribute specific to this class is listed in Table 37-3.

Table 37–3. Attribute Specific to OM_C_OBJECT

Attribute	Value Syntax	Value Length	Value Number	Value Initially
OM_CLASS	String(**OM_S_ OBJECT_ IDENTIFIER_ STRING**)	—	1	—

- **OM_CLASS**

  This attribute identifies the object's class.

Part 5

DCE Distributed Time Service

Chapter 38

Introduction to the Distributed Time Service API

This chapter describes the DCE Distributed Time Service (DTS) programming routines. You can use these routines to obtain timestamps that are based on Coordinated Universal Time (UTC). You can also use the DTS routines to translate among different timestamp formats and perform calculations on timestamps. Applications can use the timestamps that DTS supplies to determine event sequencing, duration, and scheduling. Applications can call the DTS routines from server or clerk systems.

The DCE Distributed Time Service routines are written in the C programming language. You should be familiar with the basic DTS concepts before you attempt to use the Applications Programming Interface (API). The DTS chapters of the *OSF DCE Administration Guide* provides conceptual information about DTS.

The DTS API routines offer the following basic functions:

- Retrieving timestamp information
- Converting between binary timestamps that use different time structures
- Converting between binary timestamps and ASCII representations
- Converting between UTC time and local time
- Manipulating binary timestamps

- Comparing two binary time values
- Calculating binary time values
- Obtaining time zone information.

The sections that follow describe how DTS represents time, discuss the DTS time structures, discuss the DTS API header files, and briefly describe the DTS API routines.

38.1 DTS Time Representation

Coordinated Universal Time (UTC) is the international time standard that has largely replaced Greenwich Mean Time (GMT). The standard is administered by the International Time Bureau (BIH), and is widely used. DTS uses opaque binary timestamps that represent UTC for all of its internal processes. You cannot read or disassemble a DTS binary timestamp; the DTS API allows applications to convert or manipulate timestamps, but they cannot be displayed. DTS also translates the binary timestamps into ASCII text strings, which can be displayed.

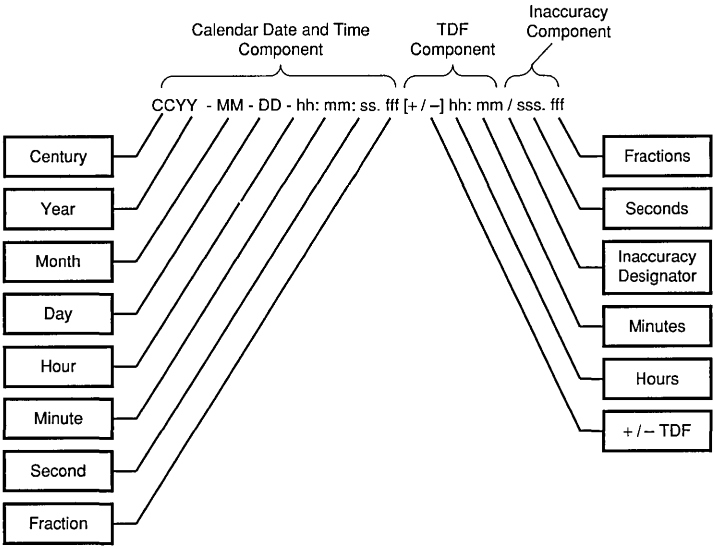
38.1.1 Absolute Time Representation

An absolute time is a point on a time scale. For DTS, absolute times reference the UTC time scale; absolute time measurements are derived from system clocks or external time-providers. When DTS reads a system clock time, it records the time in an opaque binary timestamp that also includes the inaccuracy and other information. When you display an absolute time, DTS converts the time to ASCII text as shown in the following display:

```
1990-11-21-13:30:25.785-04:00I000.082
```

DTS displays all times in a format that complies with the International Standards Organization (ISO) 8601 (1988) standard. Note that the inaccuracy portion of the time is not defined in the ISO standard; times that do not include an inaccuracy are accepted. Figure 38-1 explains the ISO format that generated the previous display.

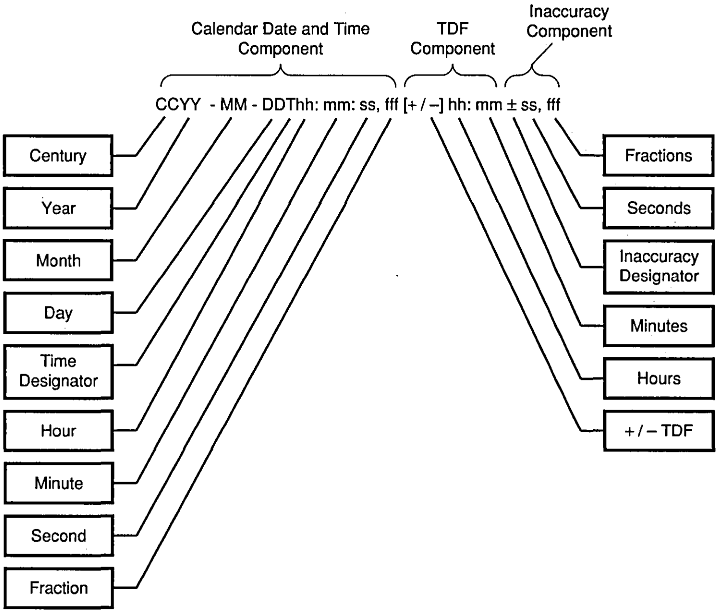
Figure 38–1. ISO Format for Time Displays



In the previous figure, the relative time preceded by the + (plus) or - (minus) character indicates the hours and minutes that the calendar date and time are offset from UTC. The presence of this Time Differential Factor (TDF) in the string also indicates that the calendar date and time are the local time of the system, not UTC. Local time is UTC plus the TDF. The Inaccuracy (I) designator indicates the beginning of the inaccuracy component associated with the time.

Although DTS displays all times in the previous format, variations to the ISO format shown in Figure 38-2 are also accepted as input for the ASCII conversion routines.

Figure 38–2. Variations to the ISO Time Format



In the previous figure, the Time (**T**) designator separates the calendar date from the time, a , (comma) separates seconds from fractional seconds, and the + (plus) or - (minus) character indicates the beginning of the inaccuracy component.

The following examples show some valid time formats.

The following represents July 4, 1776 17:01 GMT and an unspecified inaccuracy (default):

```
1776-7-4-17:01:00
```

The following represents a local time of 12:01 (17:01 GMT) on July 4, 1776 with a TDF of -5 hours and an inaccuracy of 100 seconds:

```
1776-7-4-12:01:00-05:00I100
```

Both of the following represent 12:00 GMT in the current day, month, and year with an unspecified inaccuracy:

```
12:00 and T12
```

The following represents July 14, 1792 00:00 GMT with an unspecified inaccuracy:

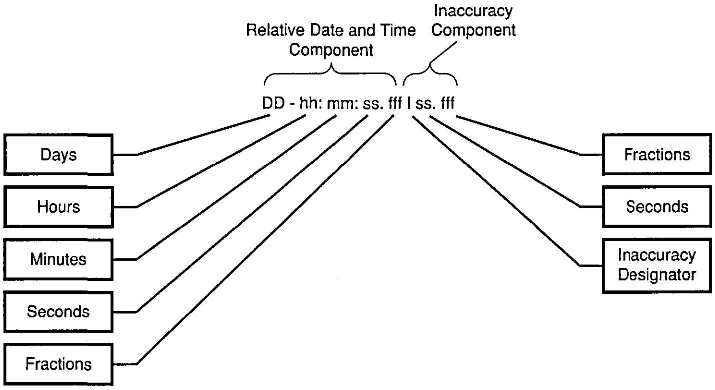
```
1792-7-14
```

38.1.2 Relative Time Representation

A relative time is a discrete time interval that is usually added to or subtracted from another time. A TDF associated with an absolute time is one example of a relative time. A relative time is normally used as input for commands or system routines.

Figure 38-3 shows the full syntax for a relative time.

Figure 38–3. Full Syntax for a Relative Time



The following example shows a relative time of 21 days, 8 hours, and 30 minutes, 25 seconds with an inaccuracy of 0.300 seconds:

```
21-08:30:25.000I00.300
```

The following example shows a negative relative time of 20.2 seconds with an unspecified inaccuracy (default):

```
-20.2
```

The following example shows a relative time of 10 minutes, 15.1 seconds with an inaccuracy of 4 seconds:

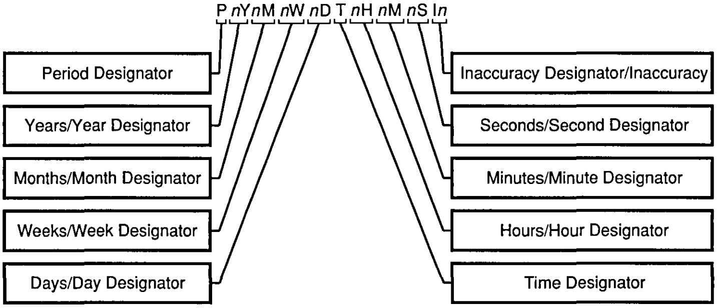
```
10:15.1I4
```

Notice that a relative time does not use the calendar date fields, since these fields concern absolute time. A positive relative time is unsigned; a negative relative time is preceded by a - (minus) sign. A relative time is often subtracted from or added to another relative or absolute time. Relative times that DTS uses internally are opaque binary timestamps. The DTS API offers several routines that can be used to calculate new times using relative binary timestamps.

Representing Periods of Time

A given duration of a period of time can be represented by a data element of variable length that uses the syntax shown in Figure 38-4.

Figure 38–4. Syntax for Representing a Duration



The Data Element Parts

The data element contains the following parts:

- The designator **P** precedes the part that includes the calendar components, including the following:
  - — The number of years followed by the designator **Y**
  - — The number of months followed by the designator **M**
  - — The number of weeks followed by the designator **W**
  - — The number of days followed by the designator **D**
- The **T** designator precedes the part that includes the time components, including the following:
  - — The number of hours followed by the designator **H**
  - — The number of minutes followed by the designator **M**
  - — The number of seconds followed by the designator **S**
- The designator **I** precedes the number of seconds of inaccuracy.

The following example represents a period of 1 year, 6 months, 15 days, 11 hours, 30 minutes, and 30 seconds and an unspecified inaccuracy:

```
P1Y6M15DT11H30M30S
```

The following example represents a period of 3 weeks and an inaccuracy of 4 seconds:

```
P3WI4
```

38.2 Time Structures

DTS can convert among several types of binary time structures that are based on different base dates and time unit measurements. DTS uses UTC-based time structures, and can convert other types of time structures to its own presentation of UTC-based time. The DTS API routines are used to perform these conversions for applications on your system.

Table 38-1 lists the absolute time structures that the DTS API uses to modify binary times for applications.

Table 38–1. Absolute Time Structures

Structure	Time Units	Base Date	Approximate Range
utc	100-nanosecond	15 October 1582	A.D. 1 to A.D. 30,000
tm	second	1 January 1900	A.D. 1 to A.D. 30,000
timespec	nanosecond	1 January 1970	A.D. 1970 to A.D. 2106

Table 38-2 lists the relative time structures that the DTS API uses to modify binary times for applications.

Table 38–2. Relative Time Structures

Structure	Time Units	Approximate Range
utc	100-nanosecond	+/- 30,000 years
tm	second	+/- 30,000 years
reltimespec	nanosecond	+/- 68 years

The remainder of this section explains the DTS time structures in detail.

38.2.1 The utc Structure

Coordinated Universal Time (UTC) is useful for measuring time across local time zones and for avoiding the seasonal changes (summer time or daylight savings time) that can affect the local time. DTS uses 128-bit binary numbers to represent time values internally; throughout this guide, these binary numbers representing time values are referred to as binary timestamps. The DTS **utc** structure determines the ordering of the bits in a binary timestamp; all binary timestamps that are based on the **utc** structure contain the following information:

- The count of 100-nanosecond units since 00:00:00.00, 15 October 1582 (the date of the Gregorian reform to the Christian calendar)
- The count of 100-nanosecond units of inaccuracy applied to the preceding item
- The Time Differential Factor (TDF), expressed as the signed quantity
- The DTS version number

The binary timestamps that are derived from the DTS **utc** structure have an opaque format. This format is a cryptic character sequence that DTS uses and stores internally. The opaque binary timestamp is designed for use in programs, protocols, and databases.

Note: Applications use the opaque binary timestamps when storing time values or when passing them to DTS.

The API provides the necessary routines for converting between opaque binary timestamps and character strings that can be displayed and read by users.

38.2.2 The tm Structure

The **tm** structure is based on the time in years, months, days, hours, minutes, and seconds since 00:00:00 GMT (Greenwich Mean Time), 1 January 1900. The **tm** structure is defined in the **time.h** header file.

The **tm** structure declaration follows:

```
struct tm {
     int tm_sec;     /* Seconds (0 - 59)                  */
     int tm_min;     /* Minutes (0 - 59)                  */
     int tm_hour;    /* Hours (0 - 23)                    */
     int tm_mday;    /* Day of Month (1 - 31)             */
     int tm_mon;     /* Month of Year (0 - 11)            */
     int tm_year;    /* Year - 1900                       */
     int tm_wday;    /* Day of Week (Sunday = 0)          */
     int tm_yday;    /* Day of Year (0 - 364)             */
```

```
             int tm_isdst;  /* Nonzero if Daylight Savings Time   */
                            /*  is in effect                      */
         };
```

Not all of the **tm** structure fields are used for each routine that converts between **tm** structures and **utc** structures. (See the parameter descriptions that accompany the routines in the *OSF DCE Application Development Reference* manual for additional information about which fields are used for specific routines.)

38.2.3 The timespec Structure

The **timespec** structure is normally used in combination with or in place of the **tm** structure to provide finer resolution for binary times. The **timespec** structure is similar to the **tm** structure, but the **timespec** structure specifies the number of seconds and nanoseconds since the base time of 00:00:00 GMT, 1 January 1970. You can find the structure in the **dce/utc.h** header file.

The **timespec** structure declaration follows:

```
struct timespec {
   unsigned long tv_sec; /*  Seconds since 00:00:00 GMT,  */
                         /*   1 January 1970              */
   long tv_nsec;         /*  Additional nanoseconds since */
                         /*   tv_sec                      */
             }    timespec_t;
```

38.2.4 The reltimespec Structure

The **reltimespec** structure represents relative time. This structure is similar to the **timespec** structure, except that the first field is *signed* in the **reltimespec** structure. (The field is *unsigned* in the **timespec** structure.) You can find the **reltimespec** structure in the **dce/utc.h** header file.

The **reltimespec** structure declaration follows:

```
struct reltimespec {
        long tv_sec;    /*  Seconds of relative time     */
        long tv_nsec;   /*  Additional nanoseconds of    */
                        /*   relative time               */
            }   reltimespec_t;
```

38.3 DTS API Header Files

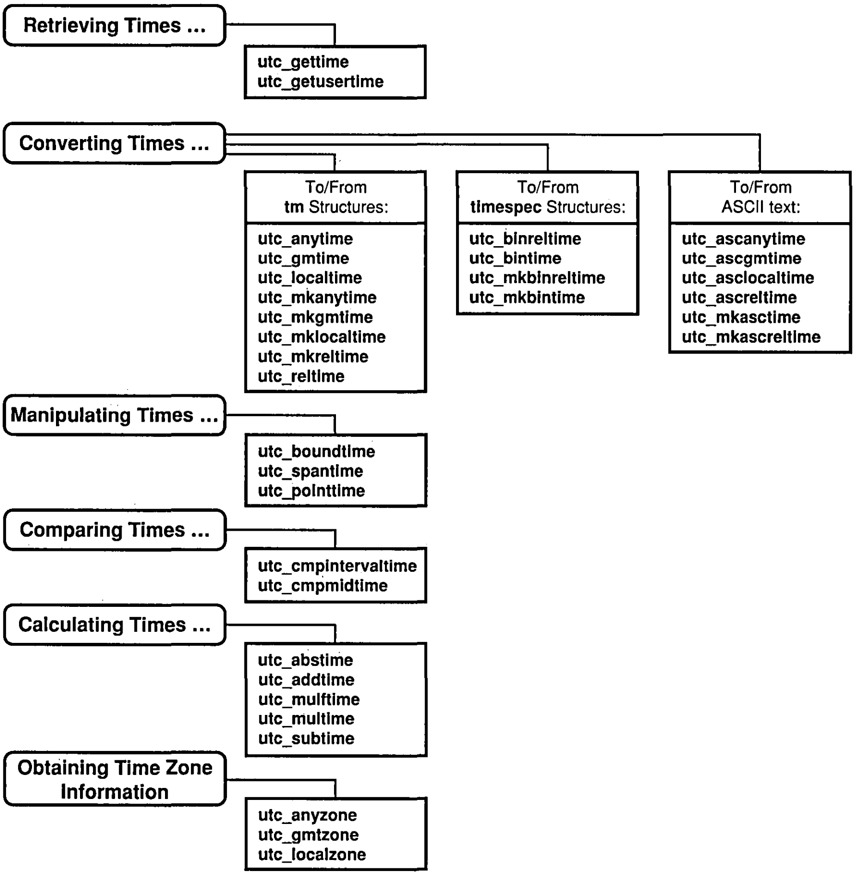
The **time.h** and **dce/utc.h** header files contain the data structures, type definitions, and define statements that are referenced by the DTS API routines. The **time.h** header file is a standard UNIX file. The **dce/utc.h** header file includes **time.h** and contains the **timespec**, **reltimespec**, and **utc** structures.

These header files are located in **/usr/include/dce**.

38.4 DTS API Routine Functions

Figure 38-5 categorizes the DTS portable interface routines by function.

Figure 38–5. DTS API Routines Shown by Functional Grouping



An alphabetical listing of the DTS portable interface routines and a brief description of each one follows:

- **utc_abstime**

  Computes the absolute value of a binary relative timestamp

- **utc_addtime**

  Computes the sum of two binary timestamps; the timestamps can be two relative times or a relative time and an absolute time

- **utc_anytime**

  Converts a binary timestamp into a **tm** structure by using the TDF information contained in the timestamp to determine the TDF returned with the **tm** structure

- **utc_anyzone**

  Gets the time zone label and offset from GMT by using the TDF contained in the input **utc**

- **utc_ascanytime**

  Converts a binary timestamp into an ASCII string that represents an arbitrary time zone

- **utc_ascgmtime**

  Converts a binary timestamp into an ASCII string that expresses a GMT time

- **utc_asclocaltime**

  Converts a binary timestamp to an ASCII string that represents a local time

- **utc_ascreltime**

  Converts a binary timestamp that expresses a relative time to its ASCII representation

- **utc_binreltime**

  Converts a relative binary timestamp into two **timespec** structures that express relative time and inaccuracy

- **utc_bintime**

  Converts a binary timestamp into a **timespec** structure

- **utc_boundtime**

  Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event

- **utc_cmpintervaltime**

  Compares two binary timestamps or two relative binary timestamps

- **utc_cmpmidtime**

  Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies

- **utc_gettime**

  Returns the current system time and inaccuracy as an opaque binary timestamp

- **utc_getusertime**

  Returns the time and process-specific TDF, rather than the system-specific TDF

- **utc_gmtime**

  Converts a binary timestamp into a **tm** structure that expresses GMT or the equivalent UTC

- **utc_gmtzone**

  Gets the time zone label, given **utc**

- **utc_localtime**

  Converts a binary timestamp into a **tm** structure that expresses local time

- **utc_localzone**

  Gets the time zone label and offset from GMT, given **utc**

- **utc_mkanytime**

  Converts a **tm** structure and TDF (expressing the time in an arbitrary time zone) into a binary timestamp

- **utc_mkascreltime**

  Converts a **NULL**-terminated character string, which represents a relative timestamp, to a binary timestamp

- **utc_mkasctime**

  Converts a **NULL**-terminated character string, which represents an absolute timestamp, to a binary timestamp

- **utc_mkbinreltime**

  Converts a **timespec** structure expressing a relative time to a binary timestamp

- **utc_mkbintime**

  Converts a **timespec** structure into a binary timestamp

- **utc_mkgmtime**

  Converts a **tm** structure that expresses GMT or UTC to a binary timestamp

- **utc_mklocaltime**

  Converts a **tm** structure that expresses local time to a binary timestamp

- **utc_mkreltime**

  Converts a **tm** structure that expresses relative time to a binary timestamp

- **utc_mulftime**

  Multiplies a relative binary timestamp by a floating-point value

- **utc_multime**

  Multiplies a relative binary timestamp by an integer factor

- **utc_pointtime**

  Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time

- **utc_reltime**

  Converts a binary timestamp that expresses a relative time into a **tm** structure

- **utc_spantime**

  Given two (possibly unordered) binary timestamps, returns a single UTC time interval whose inaccuracy spans the two input timestamps

- **utc_subtime**

  Computes the difference between two binary timestamps that express either an absolute time and a relative time, two relative times, or two absolute times

Chapter 39

Time-Provider Interface

This chapter describes the Time-Provider Interface (TPI) for DCE Distributed Time Service software. The chapter provides a brief overview of the TPI, explains how to use external time-providers with DTS, and describes the data structures and message protocols that make up the TPI.

Coordinated Universal Time (UTC) is widely used and is disseminated throughout the world by various standards organizations. Several manufacturers supply devices that can acquire UTC time values via radio, satellite, or telephone. These devices can then provide standardized time values to computer systems. Normally, one device is connected to a computer system; the device runs a process that interprets signals and translates them to time values, which can either be displayed or be provided to the server process running on the connected system.

To synchronize its system clock with UTC using an external time-provider device, a DTS server needs a software interface to the device to periodically obtain UTC. In effect, this interface serves as an intermediary between the DTS server and external time-provider processes. The DTS server requires the interface to obtain UTC time values and to determine the associated inaccuracy of each value. The interface between the DTS server process and the time-provider process is called the Time-Provider Interface (TPI).

The remainder of this chapter describes the TPI and its attendant processes in detail. The following section describes the control flow between the DTS server process, the TPI, and the Time-Provider process.

39.1 General TPI Control Flow

When you use a time-provider with a system running DTS, the external time-provider is implemented as an independent process that communicates with a DTS server process through Remote Procedure Calls (RPCs). A remote procedure call is a synchronous request and response between a main calling program and a procedure executing in another process. RPC applications are based on the client/server model. In this context, the following processes act as the client and server components in the RPC-based application:

- The DTS daemon is the client.
- The Time-Provider process (TP process) is the server.

Both the RPC-client (DTS daemon) and the server (TP process) must be running on the same system.

Applications running on RPC communicate through an interface that is well known to both the client and the server. The RPC interface consists of a set of procedures, data types, and constants that describe how a client can invoke a routine running on the server. The server offers the interface to the clients through the Interface Definition Language (IDL) file.

The IDL file defines the syntax for an operation, including the following:

- The name of the operation
- The data type of the value that the operation returns (if any)
- The order and data types of the operation's parameters (if any)

The TP process offers two procedures that DTS calls to obtain time values. These procedures are **ContactProvider** and **ServerRequestProviderTime.**

At each system synchronization, DTS makes the initial remote procedure call (**ContactProvider**) to a TP process that is assumed to be running on the same node.

If the TP process is active, the RPC call returns the following arguments:

- A successful communication status message
- A control message that DTS uses for further processing

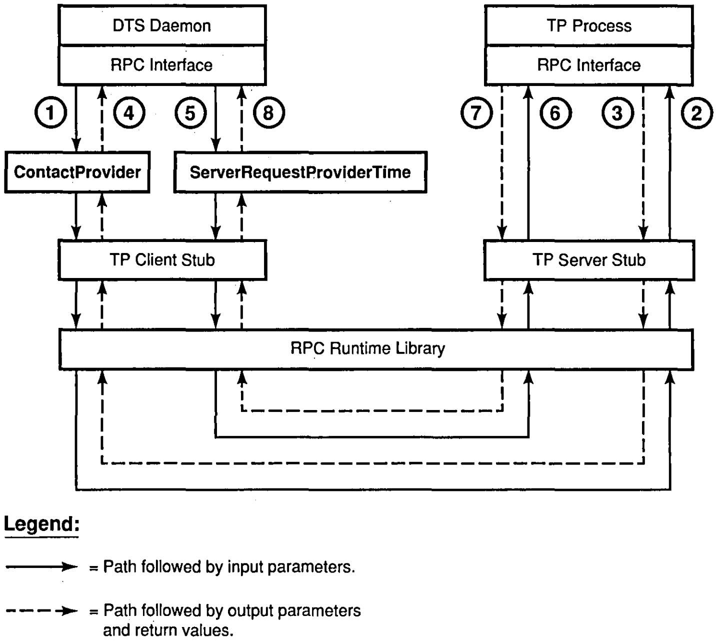
If the TP process is not active, the RPC call either returns a communication status failure or a time-out occurs. DTS then synchronizes with other servers instead of with the external time-provider.

If the initial call (**ContactProvider**) is successful, DTS makes a second call (**ServerRequestProviderTime**) to retrieve the timestamps from the external time-provider. The control message sent by the TP process in the first RPC call specifies the length of time DTS waits for the RPC call to complete. The TP process returns the following parameters in the procedure call:

- A communication status message.
- A time structure that contains timestamps collected from the external time-provider. (DTS then uses these timestamps to complete its synchronization.)

Figure 39-1 illustrates the RPC calling sequence between DTS and the TP process. Note that solid black lines represent the path followed by input parameters; dashed lines represent the path followed by output parameters and return values.

Figure 39–1. DTS/Time-Provider RPC Calling Sequence



The following steps describe the process shown in the previous figure:

1. At synchronization time, DTS calls the **ContactProvider** remote procedure. Input parameters are passed to the TP client stub, dispatched to the RPC runtime library, and then passed to the TP server stub.

2. The TP process receives the call and executes the **ContactProvider** procedure.

3. The procedure terminates and returns the results through the TP server stub, the RPC runtime library, and the TP client stub.

4. The procedure terminates in the DTS call, where the returned parameters are examined.

5. DTS then calls the **ServerRequestProviderTime** remote procedure. Input parameters are passed to the TP client stub, dispatched to the RPC runtime library, and then passed to the TP server stub.

6. The TP process receives the call and executes the **ServerRequestProviderTime** procedure.
7. The procedure terminates and returns the results through the TP server stub, the RPC runtime library, and the TP client stub.
8. The DTS remote procedure call terminates and the timestamps are returned as an output parameter. DTS then synchronizes using the timestamps returned by the external time-provider.

The following section describes the remote procedures that are exported by the TP process during the previous sequence.

39.1.1 ContactProvider Procedure

ContactProvider is the first routine called by DTS. The routine is called to verify that the TP process is running and to obtain a control message that DTS uses for subsequent communications with the TP process and for synchronization after it receives the timestamps. The parameters passed in the **ContactProvider** procedure call consist of the following elements:

- Binding Handle

  An input parameter that establishes the relationship between DTS and the TP process. A binding handle enables the client (DTS) to recognize and find a server (the TP process) that offers the same interface.

- Control Message

  An output parameter that contains information used by DTS for subsequent processing. The control message consists of the following elements:

  TPstatus One of the following values:

  — **K_TPI_SUCCESS**

  — **K_TPI_FAILURE**

  nextPoll A time value that tells DTS when to contact the TP process again. For example, once a day through dial-up, radio, or satellite.

  timeout A value that tells DTS how long to wait for a response from the TP process.

noClockSet A value that specifies whether or not DTS is allowed to alter the system clock. If *noClockSet* is specified as 0x01 (TRUE), DTS does not adjust or set the clock during the current synchronization. This option is useful for systems whose system clock is known to be accurate (such as systems equipped with special hardware) or systems that are managed by some other time service (such as Network Time Protocol (NTP)), but which still wish to function as a DTS server.

- Communication Status

  An output parameter that contains a status code returned by the DCE RPC runtime library. The status **rpc_s_ok** is returned if the TP process is successfully contacted.

39.1.2 ServerRequestProviderTime Procedure

After the TP process is successfully contacted, DTS makes the **ServerRequestProviderTime** procedure call to obtain the timestamps from the external time-provider. The parameters passed in the **ServerRequestProviderTime** procedure call consist of the following elements:

- Binding Handle

  An input parameter that establishes the relationship between DTS and the TP process. A binding handle enables the client (DTS) to recognize and find a server (the TP process) that offers the same interface.

- Time Response Message

  An output parameter that contains a TP process status value (**K_TPI_SUCCESS** or **K_TPI_FAILURE**), a count of the timestamps that are returned, and the timestamps obtained from the external time-provider. The timestamp count is an integer in the range **K_MIN_TIMESTAMPS** to **K_MAX_TIMESTAMPS**. Each timestamp consists of three **utc** time values:

  — The system clock time immediately before the TP process polls the external time source. (The TP process normally obtains the time from the **utc_gettime**( ) DTS API routine.)

- — The time value returned to the TP process by the external time source.
- — The system clock time immediately after the external time source is read. (The TP process obtains the time from the **utc_gettime( )** DTS API routine.)

- Communication Status

  An output parameter that contains a status code returned by the DCE RPC runtime library. The status **rpc_s_ok** is returned if the TP process is successfully contacted.

39.2 Time-Provider Process IDL File

A remote procedure call can only work if an interface definition that clearly defines operation signatures exists. Operation signatures define the syntax for an operation, including its name and parameters (input and output) that are passed as part of the procedure call. The TP process interface exports the two operation signatures that have been previously explained. The interface is provided in the file **examples/dts/dtsprovider.idl**. When building the TP process application, this file must be compiled using the Interface Definition Language (IDL) compiler, which creates three files:

- **dtsprovider.h** (header file)
- **dtsprovider_sstub.c** (server stub file)
- **dtsprovider_cstub.c** (client stub file)

The Time-Provider program must be compiled along with the **dtsprovider_sstub.c** code and then linked together. The TP program must also include the stub-generated file **dtsprovider.h**. The following sample code shows the structure of this interface.

```
/*
 *              Time Service Provider Interface
 *
 * This interface is defined through the Network Interface
 * Definition Language (NIDL).
 */
[uuid (bfca1238-628a-11c9-a073-08002b0dea7a),
    version(1)
]

interface time_provider
{

 import "dce/nbase.idl";
 import "dce/utctypes.idl";

/* Minimum and Maximum number of times to read time source at each
 * synchronization
 */
const long K_MIN_TIMESTAMPS   = 1;
const long K_MAX_TIMESTAMPS   = 6;

/* Message status field return values
 */
const long K_TPI_FAILURE        = 0;
const long K_TPI_SUCCESS        = 1;

/* This structure contains one reading of the TP wrapped in the
 * timestamps of the local clock.
 */
typedef struct TimeResponseType
{
   utc_t beforeTime;     /* local clk just before getting UTC */
   utc_t TPtime;         /* source UTC; inacc also supplied   */
   utc_t afterTime;      /* local clk just after getting UTC  */
} TimeResponseType;

/* Time-provider control message.  This structure is returned in
 * response to a time service request.  The status field returns TP
 * success or failure.  The nextPoll gives the client the time at
 * which to poll the TP next. The timeout value tells the client how
 * long to wait for a time response from the TP.  The noClockSet will
 * tell the client whether or not it is allowed to alter the system
 * clock after a synchronization with the TP.
 */
typedef struct TPctlMsg
{
    unsigned long       status;
    unsigned long       nextPoll;
    unsigned long       timeout;
    unsigned long       noClockSet;
} TPctlMsg;

/* TP timestamp message.  The actual time-provider synchronization
 * data.  The status is the result of the operation (success or
 * failure).  The timeStampCount parameter returns the number of
 * timestamps being returned in this message.  The timeStampList is
 * the set of timestamps being returned from the TP.
 */
typedef struct TPtimeMsg
{
    unsigned long       status;
    unsigned long       timeStampCount;
    TimeResponseType    timeStampList[K_MAX_TIMESTAMPS];

} TPtimeMsg;

/* The Time-Provider Interface structures are described here.
 * There are two types of response messages from the TP:
 * control message and data message.
 *
 *         <<<< TPI CONTROL MESSAGE >>>>
 *
 * 31                                          0
 * +--------------------------------------------+
 * |            Time-Provider Status            |
 * +--------------------------------------------+
 * |            Next Poll Delta                 |
 * +--------------------------------------------+
 * |            Message Time Out                |
 * +--------------------------------------------+
 * |            NoSet Flag                      |
 * +--------------------------------------------+
 *
 *      <<<< a single timestamp >>>>
 *
 * 128                                           0
 * +---------------------------------------------+
 * |            Before Time                      |
 * +---------------------------------------------+
 * |            TP Time                          |
 * +---------------------------------------------+
 * |            After Time                       |
 * +---------------------------------------------+
 *
 *      <<<< TPI DATA MESSAGE >>>>
 *
 * 31                                            0
 * +---------------------------------------------+
 * |            Time-Provider Status             |
 * +---------------------------------------------+
 * |            Timestamp Count                  |
 * +---------------------------------------------+
 * |                                             |
 * |            <timestamp one>                  |
 * |                                             |
 * +---------------------------------------------+
 * |                  .                          |
 * |                  .                          |
 * |                  .                          |
 * |                  .                          |
 * |                  .                          |
 * +---------------------------------------------+
 * |                                             |
 * |      <timestamp K_MAX_TIMESTAMPS>           |
 * |                                             |
 * +---------------------------------------------+
 */

/* The RPC-based Time-Provider Program (TPP) interfaces are defined
 * here. These calls are invoked by a Time Service daemon running as a
 * server (in this case it makes an RPC client call to the TPP server).
 */

/* CONTACT_PROVIDER
 * Send initial contact message to the TPP.  The TPP server
 * responds with a control message.
 */
void ContactProvider
        (
        [in]    handle_t        bind_h,
        [out]   TPctlMsg        *ctrlRespMsg,
        [out]   error_status_t  *comStatus
        );

/* SERVER_REQUEST_PROVIDER_TIME
 * The client sends a request to the TPP for times.
 * The TPP server responds with an array of timestamps
 * obtained by querying the Time-Provider hardware that it polls.
 */
void ServerRequestProviderTime
        (
        [in]    handle_t        bind_h,
        [out]   TPtimeMsg       *timesRspMsg,
        [out]   error_status_t  *comStatus
        );
}
```

39.3 Initializing the Time-Provider Process

Initializing the RPC-based TP process prepares it to receive remote procedure calls from a DTS daemon requesting the timestamps. The following steps are involved:

1. Include the header file (**dtsprovider.h**) that is created by compiling **/usr/include/dce/dtsprovider.idl**, which contains the interface definition.

2. Register the interface with the DCE RPC runtime.

3. Select one or more protocol sequences that are compatible with both the interface and the runtime library. It is recommended that the TP process application selects all protocol sequences available on the system. Available protocol sequences are obtained by calling an RPC API routine, described in the example that follows. This ensures that transport independence is maintained in RPC applications.

4. Register the TP process with the endpoint mapper (**rpcd**) database running on the system. This makes the TP process available to the DTS daemon.

5. Obtain the name of the machine's principal and then register an authentication service to use with authenticated remote procedure calls coming from the DTS daemon. Note that DTS and the Time-Provider program are presumed to be running in an authenticated environment.

6. Listen for remote procedure calls.

The following example illustrates these steps, including the sequence of calls needed.

```
/* Register the TP server interface with the RPC runtime.
 * The interface specification time_provider_v1_0_ifspec
 * is obtained from the generated header file dtsprovider.h
 * The entry point vector is normally defined at the top of
 * the TP source program similar to this:
 *
 *    globaldef time_provider_v1_0_epv_t time_provider_epv =
 *    {
 *       ContactProvider,
 *       ServerRequestProviderTime
 *    };
 */
rpc_server_register_if (time_provider_v1_0_s_ifspec,
                        NULL,
                        (rpc_mgr_epv_t) &time_provider_epv,
                        &RPCstatus);

/*
 * This call tells the DCE RPC runtime to listen for remote
 * procedure calls using all supported protocol sequences.
 * To listen for a specific protocol sequence, use the
 * rpc_server_use_protreq call.
 */
rpc_server_use_all_protseqs (max_calls,
                             &RPCstatus);
```

```
/* This routine is called to obtain a vector of binding handles
 * that were established with registration of protocol sequences.
 */
rpc_server_inq_bindings (&bind_vector,
                         &RPCstatus);

/* This routine adds the address information of the binding
 * handle for the TP server to the endpoint mapper database.
 */
rpc_ep_register (time_provider_v1_0_s_ifspec,
                 bind_vector,
                 NULL,
                 "Time Provider",
                 &RPCstatus);

/* Obtain the name of the machine's principal and register an
 * authentication service to use for authenticated remote procedure
 * calls coming from the time service daemon.
 */
dce_cf_prin_name_from_host (NULL,
                            &machinePrincipalName,
                            &status);

rpc_server_register_auth_info (machinePrincipalName,
                               rpc_c_authn_dce_private,
                               NULL,
                               NULL,
                               &RPCstatus);

/* This routine is called to listen for remote procedure calls
 * send by the DTS client.  The possible RPC calls coming from
 * the DTS client are ContactProvider and ServerRequestProviderTime.
 */
rpc_server_listen (max_calls,
                   &RPCstatus);
```

39.4 Time-Provider Algorithm

The time-provider algorithm assumes that the two remote procedure calls will come in the following order: **ContactProvider** followed by **ServerRequestProviderTime**. The algorithm to create a generic time-provider follows:

1. Initialize the TP process, as described previously. Listen for RPC calls.
2. If the **ContactProvider** procedure is invoked, perform the following steps:
   a. Initialize the control message to the appropriate values (status value to **K_TPI_SUCCESS**; *nextPoll*, *timeout*, and *noClockSet* to valid integer values).

   b. Set the communication status output parameter to **rpc_s_ok**.

   c. Return from the procedure call. (The DCE RPC runtime returns the values to DTS.)
3. If the **ServerRequestProviderTime** procedure is run, perform the following steps:
   a. Initialize the timestamp count to the appropriate number.

   b. Use the **utc_gettime( )** DTS API routine to read the system time.

   c. Poll the external time source and read a UTC value. Use the **utc_gmtime( )** routine to convert the UTC time value to a binary timestamp.

   d. Use the **utc_gettime( )** routine to read the system time.

   e. Repeat steps b, c, and d the number of times specified by the values of **K_MIN_TIMESTAMPS** and **K_MAX_TIMESTAMPS**.

   f. If steps b, c, or d return erroneous data, initialize the TP process status field (*TPstatus*) of the data message to **K_TPI_FAILURE**; otherwise, initialize the data message timestamps.

g. Set the communication status output parameter to **rpc_s_ok**.

h. Return from the procedure call. (The DCE RPC runtime sends the values back to DTS.)

4. The TP process continues listening for RPC calls.

39.5 DTS Synchronization Algorithm

DTS performs the following steps to synchronize with an external time-provider:

1. At start-up time, create the binding handle for the Time-Provider interface. The binding handle is obtained from the list of available protocol sequences on the system.

2. At synchronization time, make the remote procedure call **ContactProvider**, assuming that a TP process is running on the system. If the procedure call fails, examine the RPC communication status, checking the availability of the server. If the server is unavailable, synchronize with peer servers; otherwise, continue.

3. Wait for the procedure call to return the control message in the output parameter. If the procedure call does not return within the specified LAN time-out interval, synchronize with peer servers. Otherwise, go to step 4.

4. If the procedure call returned successfully (communication status is **rpc_s_ok**), read the data in the control message.

5. Make the remote procedure call **ServerRequestProviderTime** to obtain the timestamps from the external time-provider. If the procedure does not return within the elapsed time specified by the control message (*timeout*), then synchronize with peer servers. Schedule the next synchronization based upon the applicable DTS management parameters, ignoring *nextPoll.*

6. If the procedure returns successfully, verify that the TP process status is **K_TPI_SUCCESS**. Otherwise, synchronize with peer servers and schedule the next synchronization.

7. Extract the timestamps from the data message and synchronize using the timestamps.

8. Schedule the next synchronization time by adding the value of *nextPoll* seconds to the current time. At the next synchronization, go to step 2.

Note: Application developers do not have to perform these steps; DTS performs these steps internally during synchronization with an external time-provider.

39.6 Running the Time-Provider Process

Both the TP process and the DTS daemon must run on the same system. The TP process must be started up under the login context of the machine's principal, which has root privileges. The DTS daemon and the TP process are started independently. However, before starting the TP process, ensure that the endpoint mapper daemon (**rpcd**) is running on the system. If it is not running, start it. The TP process can always exit without affecting the DTS daemon. DTS dynamically reestablishes communications with the TP process when it creates binding handles.

39.7 Sources of Additional Information

Refer to the following for additional information:

- See **/examples/dts** for examples of time-provider programs that you can use with several different types of external time-provider devices.
- See the *OSF DCE Administration Guide* for commercial sources of external time providers.
- See the *OSF DCE Application Development Reference* for detailed information about the RPC API and DTS API routines.

Chapter 40

DTS API Routines Programming Example

This chapter contains a C programming example showing a practical application of the DTS API programming routines. The program performs the following actions:

- Prompts the user to enter two sets of time coordinates corresponding to the timestamps of two ''events.''
- Stores those coordinates in a **tm** structure.
- Converts the **tm** structure to a **utc** structure.
- Prints out the **utc** structure in ISO text format.
- Determines which event occurred first.
- Determines if Event 1 may have caused Event 2 by comparing the intervals.

```
#include time.h         /* time data structures                      */
#include dce/utc.h      /* utc structure definitions                 */

void ReadTime();
void PrintTime();

/* This program requests user input about events, then prints out
```

```
 * information about those events.
 */
main()
{
    struct utc event1,event2;
    enum utc_cmptype relation;

    /* Read in the two events.
     */
    ReadTime(&event1);
    ReadTime(&event2);

    /* Print out the two events.
     */
    printf("The first event is : ");
    PrintTime(&event1);
    printf("\nThe second event is : ");
    PrintTime(&event2);
    printf("\n");

    /* Determine which event occurred first.
     */
    if (utc_cmpmidtime(&relation,&event1,&event2))
        exit(1);

    switch( relation )
    {
        case utc_lessThan:
        printf("comparing midpoints: Event1 < Event2\n");
        break;
        case utc_greaterThan:
        printf("comparing midpoints: Event1 > Event2\n");
        break;
        case utc_equalTo:
        printf("comparing midpoints: Event1 == Event2\n");
        break;
        default:
        exit(1);
        break;
    }
    /* Could Event 1 have caused Event 2?  Compare the intervals.
     */
    if (utc_cmpintervaltime(&relation,&event1,&event2))
        exit(1);

    switch( relation )
    {
        case utc_lessThan:
        printf("comparing intervals: Event1 < Event2\n");
        break;
        case utc_greaterThan:
        printf("comparing intervals: Event1 > Event2\n");
        break;
        case utc_equalTo:
        printf("comparing intervals: Event1 == Event2\n");
        break;
        case utc_indeterminate:
        printf("comparing intervals: Event1 ? Event2\n");
        default:
        exit(1);
        break;
    }
}

/* Print out a utc structure in ISO text format.
 */
void PrintTime(utcTime)
struct utc *utcTime;
{
    char   string[50];

    /* Break up the time string.
     */
    if (utc_ascgmtime(string,      /* Out: Converted time    */
                      50,          /* In:  String length     */
                      utcTime))    /* In:  Time to convert   */
        exit(1);
    printf("%s\n",string);
}

/* Prompt the user to enter time coordinates.  Store the coordinates
```

```
 * in a tm structure and then convert the tm structure to a utc structure.
 */
void ReadTime(utcTime)
struct utc *utcTime;
{
struct tm tmTime,tmInacc;
    (void)memset((void *)&tmTime,  0, sizeof(tmTime));
    (void)memset((void *)&tmInacc, 0, sizeof(tmInacc));
    (void)printf("Year? ");
    (void)scanf("%d",&tmTime.tm_year);
    tmTime.tm_year -= 1900;
    (void)printf("Month? ");
    (void)scanf("%d",&tmTime.tm_mon);
    tmTime.tm_mon -= 1;
    (void)printf("Day? ");
    (void)scanf("%d",&tmTime.tm_mday);
    (void)printf("Hour? ");
    (void)scanf("%d",&tmTime.tm_hour);
    (void)printf("Minute? ");
    (void)scanf("%d",&tmTime.tm_min);
    (void)printf("Inacc Secs? ");
    (void)scanf("%d",&tmInacc.tm_sec);

    if (utc_mkanytime(utcTime,
                      &tmTime,
                      (long)0,
                      &tmInacc,
                      (long)0,
                      (long)0))
        exit(1);
}
```

Part 6

DCE Security Service

Chapter 41

Overview of Security

This chapter provides a brief overview of the DCE Security Service from an application programmer's perspective. Refer to the *OSF DCE Application Development Reference* for detailed information on the Application Program Interfaces (APIs) discussed in this Part of the guide.

41.1 Purpose and Organization of This Part of the Guide

The purpose of the discussions in this Part of the guide is to explain the major features of DCE Security, so that you can decide what, if anything, you need to do in order to ensure that your DCE application is sufficiently secure. A lot of security is built into DCE, so in many cases you will need to do nothing, or very little, to secure your DCE application. Furthermore, you do not need to understand all of the details of DCE Security in order to use it effectively.

Following the overview of DCE Security in this chapter are two chapters that contain conceptual discussions of authentication and authorization. The remaining chapters in this Part of the guide discuss the five Security APIs: Registry, Login, Key Management, Access Control List (ACL), and ID Map.

41.2 About Authenticated RPC

Perhaps the most important Security API is the Authenticated Remote Procedure Call facility. Authenticated RPC enables distributed applications to participate in authenticated network communications. Applications using the Authenticated RPC routines may select the authentication protocol and the authorization protocol to be used, and set various protocol-independent protection levels for communicating with remote principals (users, servers and computers).

The usage of Authenticated RPC is explained in the RPC chapters of Part 3 of this guide. This Part, however, contains conceptual information that is useful for understanding the authentication and authorization protocols that Authenticated RPC routines use; for this information, we recommend that you read Chapters 42 and 43, as well as this one.

41.3 UNIX System Security and DCE Security

UNIX system security mostly presumes that a computer's backplane can be trusted because computing operations are assumed to be local, and because the computer itself can be physically secured. In a distributed environment, the logical equivalent of the single system's backplane is the network itself. Network computing means distributed, rather than localized, computing operations, and in the case of an open network (which DCE assumes), little of the network is physically secure. Thus, the nature of distributed systems poses special security risks, in addition to those posed by nondistributed systems. Unlike UNIX system security, DCE Security is designed specifically to address those risks.

These considerations notwithstanding, network security is ultimately dependent on the security features that are local to the individual computers in the network, and what is more important, the manner in which those features are used and administered. Since any compromise to the local security of a computer in the distributed environment may introduce opportunities for compromising network security, DCE Security does not diminish the importance of local security. In fact, the relative importance of local system security is greater in the distributed environment because the consequences of a local security breach may not be local. Finally, while

DCE Security does nothing to enhance local security, neither does it introduce any new avenues for compromising local security.

In the discussions in this guide, we assume you are familiar with the authentication and authorization features that UNIX systems provide: **/etc/passwd** and **/etc/group** file processing; routines that return or change file attributes; routines that return or change real or effective user IDs (UIDs) and group IDs (GIDs); and data encryption and decryption.

41.4 What Authentication and Authorization Mean

There are two questions that DCE Security can answer for a principal about another principal with which it might want to communicate:

- Is this principal really who it says it is?
- Does it have the right to do what it wants to do?

Depending on the answers to these questions, a security-sensitive principal takes different courses of action with respect to a principal with which it is communicating.

To authenticate a principal means to verify that the principal is representing its true identity. To authorize a principal means to grant permission for the principal to perform an operation. While distinct, the concepts of authentication and authorization are also intertwined. For one thing, a principal's authorization is explicitly linked to its identity. For another, there is the possibility that authorization data concerning an authenticated principal can be falsified, which raises the additional question, ''Should the authorization data concerning this principal be believed?'' To this question also, DCE Security can provide an answer to a principal for which this issue is a concern.

We refer to the specific mechanisms by which authentication and authorization are performed as authentication and authorization protocols, and DCE Security supports at least one of each. However, RPC documentation refers to authentication and authorization protocols as services. We use the term ''protocol'' instead of ''service'' in this context in order to prevent confusion between the protocol-independent DCE authentication and authorization services, and the various authentication and authorization protocols that those services may support.

41.5 Authentication, Authorization, and Data Protection in Brief

When one principal talks to another in a distributed computing environment, there is a risk that communications between the two will provide a means for compromising the security of one or the other. For example, a client may attack a server, or a server may set a trap for clients. An attack is most likely to succeed if the malevolent principal can convince its victim that it is something other than what it really is (an attacker), and/or that it possesses authorization that it does not really have. A counterfeit identity and/or authorization data grants an attacker access that it presumably would not otherwise have, and so provides an opportunity for the attacker to do damage. Therefore, a security-sensitive principal needs assurances that the identity and authorization of any principal with which it communicates are authentic.

Another security risk is that posed by a third party that attempts to modify or steal data passed between a client and a server. In these cases, it is not the identity or authorization of the eavesdropping party that is particularly relevant, but the security of transmitted data. DCE Security enables a principal to detect whether some party has attempted to modify data sent to the principal from another legitimate principal, and to be reasonably certain that an eavesdropper can make no use of stolen data.

Figure 41-1 is an extremely condensed and highly stylized representation of the essentials of DCE Security in terms of the DCE Shared-Secret authentication protocol and the DCE Authorization authorization protocol. Unless we note otherwise, assume that discussions in this Part of this guide refer to these two protocols, used in conjunction with one another.

Following is a description of the events depicted in the illustration:

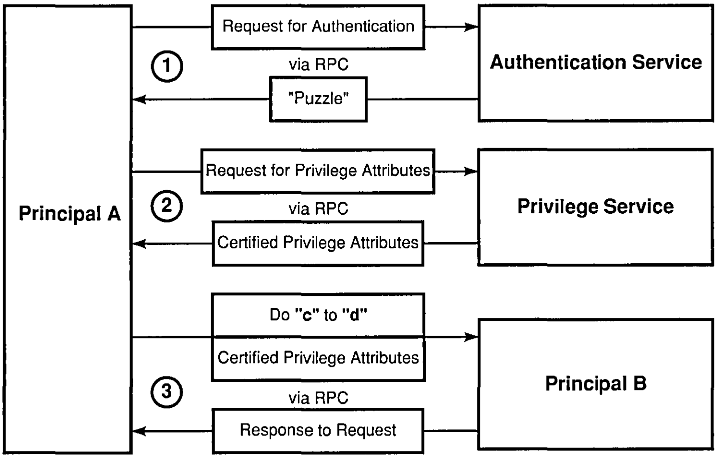
1. Principal A, which might actually be an attacker masquerading as Principal A, seeks authentication of its identity from the Authentication Service. The Authentication Service responds with a kind of puzzle; that is, it responds with information about how to contact the Privilege Service that is encrypted under a secret key known only to Principal A and the Authentication Service.

2. Other than the Authentication Service, only the real Principal A possesses the key that solves the puzzle (unless its key has been compromised). So, if the principal that requested authentication is

really A, it learns how to ask the Privilege Service to authenticate its privilege attributes (privilege attributes are data used in making authorization decisions; they consist of the principal's name and group memberships). If Principal A fails to get authenticated privilege attributes (sometimes referred to as ''credentials''), it may simply assert its privilege attributes to Principal B.

3. Principal A now makes a request to Principal B to perform some operation that requires the **c** permission to object **d**, and presents its certified privilege attributes. Principal B may grant or deny **c** access to **d** after examining the Access Control List (ACL) that protects object **d** (an ACL associates the privilege attributes of principals with permissions to an object). If **c** is one of the permissions listed in the ACL entry that specifies the permission set that may be granted to Principal A, then Principal A is allowed to perform the operation; if the **c** permission is not listed in that entry, A is denied access.

Figure 41–1. Shared-Secret Authentication and DCE Authorization in Brief



Had Principal A failed to solve the puzzle that the Authentication Service provided, it would have been unauthenticated, and as a consequence, unable to acquire certified privilege attributes from the Privilege Service. In that case, Principal A might have simply asserted its privilege attributes to B; that is, claimed them for itself, without the benefit of having the Privilege Service certify this data as being genuine. Had Principal A then presented asserted privilege attributes to Principal B, then B might have denied the requested permission or granted it, depending on whether B grants permissions to unauthenticated principals, and whether **c** is among the permissions that B grants to such principals.

If Principals A and B are especially sensitive to security concerns, they may request that RPC data be checked for integrity to establish whether it has been modified in transit, and possibly also encrypted to ensure that the data is unintelligible to any party other than Principals A and B.

41.6 Summary of DCE Security Services and Facilities

The DCE Security Service comprises three services and four facilities. The three services are

- The Registry Service, which maintains a database of principals, groups, organizations, accounts, and administrative policies.
- The Authentication Service, which verifies the identity of a principal and issues tickets that the principal uses to access remote services (a ticket is data about a principal that is presented to the principal providing the service).
- The Privilege Service, which certifies a principal's privilege attributes (that is, its name and group memberships, which are represented as UUIDs).

The three security services are implemented as a single daemon, the Security Server.

The four facilities are

- The Login facility, which enables a principal to establish its network identity.
- The Access Control List (ACL) facility, which enables a principal's access to an object to be determined by a comparison of the principal's privilege attributes to the object's permissions.
- The Key Management facility, which enables noninteractive principals (most frequently, servers) to manage their secret keys.

- The ID Map facility, which maps cell-relative principal names to global principal names, and global principal names to cell-relative principal names. This facility is used in connection with the transmission of information about principals that are members of different DCE cells.

For UNIX system compatibility with DCE, the Security Service also provides implementations of UNIX system C library interfaces to the **/etc/passwd** and **/etc/group** files.

41.6.1 Interfaces to the Security Server

Following are the user interfaces to the Security Server itself (see the *OSF DCE Administration Guide* and the *OSF DCE Administration Reference*):

- **secd**

  Starts the Security daemon

- **sec_create_db**

  Creates the Security databases

- **sec_admin**

  Administers instances of the Security daemon, which is a replicated server

- **sec_salvage_db**

  Salvages a corrupted Security database

- **sec_clientd**

  Enables clients of the Security Server to communicate with it

All other interfaces to the Security Server are more precisely characterized as interfaces to its three services: Registry, Authentication, and Privilege.

41.6.1.1 Registry Service Interfaces

User interfaces to the Registry Service are described in the *OSF DCE User's Guide and Reference*, the *OSF DCE Administration Guide*, and the *OSF DCE Administration Reference*. Following is a summary of them:

- **rgy_edit**

  Edits Registry database entries

- **passwd_import**

  Creates Registry database entries from UNIX system **/etc/passwd** and **/etc/group** files

- **passwd_export**

  Creates local Registry information that corresponds to network Registry database entries; used when the Security Server is unavailable

- **chpass**

  Changes a user's password in a Registry database entry

The API to the Registry Service consists of calls that are prefixed **sec_rgy...**( ). Since this is the same interface that the Registry Service user and administration tools call, few applications make use of it, unless they are to replace some or all of the functionality of the default Registry tools.

41.6.1.2 Authentication Service Interfaces

Following is a summary of the user interfaces to the Authentication Service when the default authentication protocol is in effect (the default protocol is DCE Shared-Secret, which is based on the Kerberos Version 5 network authentication system). These interfaces are described in the *OSF DCE User's Guide and Reference*:

- **kinit**

  Obtains a login session's ticket(s) to remote services (the **login** and **su** tools also perform this service)

- **klist**

  Lists a login session's tickets to remote services

- **kdestroy**

  Destroys a login session's tickets to remote services

The API to the Authentication Service is the Authenticated RPC facility, which is the Security API that a distributed application is most likely to call. Authenticated RPC is not, however, the API that the authentication tools call; these tools call Kerberos program interfaces and are undocumented.

41.6.1.3 Privilege Service Interfaces

There are neither user interfaces nor application program interfaces to the Privilege Service; the Login facility and authenticated RPC encapsulate interactions between a principal and the Privilege Service.

41.6.2 Interfaces to the Login Facility

User interfaces to the Login facility consist of the following tools (see the *OSF DCE User's Guide and Reference* for a description):

- **dce_login**

  Enables an interactive principal to log into the DCE, but does not change the principal's local identity

- **login**

  Enables an interactive principal to log in

- **su**, enables a logged-in interactive principal to assume a different principal identity

The API to the Login facility consists of calls that are prefixed **sec_login...**( ). This API enables application processes to assume their network identities. Network login and system login programs are examples of applications that call this API.

41.6.3 Interfaces to the Key Management Facility

For a distributed application, it may be important for a server to have a network identity that is distinct from the principal identity it inherits from the user who invokes it or the host on which it runs. The Key Management facility provides features that enable noninteractive principals to manage their secret keys.

The user interface to the Key Management facility consist of a few **rgy_edit** subcommands that enable an administrator to change or delete the secret key of a noninteractive principal, in the event that the administrator discovers that such a key has been compromised. These subcommands call the Key Management API, which consists of several calls with the prefix **sec_key...( )**.

41.6.4 Interfaces to the ID Map Facility

There are no user interfaces to the ID Map facility. The API to this facility consists of calls that are prefixed **sec_id...( )**. These routines map a global principal or group name into a cell name and a cell-relative principal or group name, and generate a global principal or group name from a cell name and a cell-relative principal or group name. This API also converts the internal (UUID) representation of a name to a human-readable string and back again.

41.6.5 Interfaces to the Access Control List Facility

The only user interface to the Access Control List facility is the tool **acl_edit**. This tool edits an object's ACL, the entries of which specify the permissions to the object that may be granted to principals possessing specified privilege attributes (**acl_edit** is described in the *OSF DCE User's Guide and Reference*).

The ACL API consists of routines that are prefixed **sec_acl...( )**. This is the same API that **acl_edit** calls, so an ACL editor or browser that is intended to replace **acl_edit** would call this API. A different case is that of an application server that needs to store and retrieve application-specific, access-control information for its clients. Such an application needs to implement its own ACL manager using the subset of the ACL API routines that are prefixed **sec_acl_mgr...( )**; and if its clients are remote, then the ACL manager must also export the ACL manager interface (refer to Chapter 47 for more information on ACL managers).

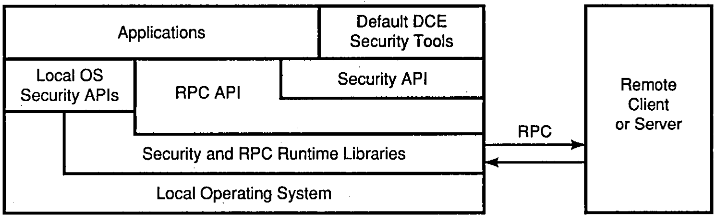
41.6.6 DCE Implementations of UNIX System Program Interfaces

DCE Security provides implementations of UNIX system C library interfaces related to security. These are **getpwent()** and the related program interfaces to the **/etc/passwd** file, and **getgrent()** and the related program interfaces to the **/etc/group** file. Applications that bind with **libdce.a** are bound with the DCE Security implementations of these interfaces; otherwise, they are bound with the local operating system implementations.

41.7 Relationships Between the Security Service and DCE Applications

Figure 41-2 is a schematic illustration of the relationships among the interfaces to the DCE Security Service, and the relationship of Security interfaces to DCE applications.

Figure 41–2. DCE Security and the DCE Application Environment



41.8 DTS, the Cell Namespace, and Security

The following subsections discuss the dependencies of DCE Security on the Distributed Time Service (DTS), and the relationship between the Security namespace and the Cell Directory Service (CDS) namespace. For information about how DCE components such as CDS use features of DCE Security, refer to the documentation on the component of interest (for example, the section of the *OSF DCE Administration Guide* on CDS).

41.8.1 DTS and Security

The Security Service depends on a relatively close synchronization of network clocks, a service provided by the Distributed Time Service. When network clocks become too skewed, unexpired tickets to services may be regarded as invalid, and/or expired tickets considered valid. Excessive skewing can inconvenience users and introduce opportunities for security breaches; in the latter case, administrative intervention is required.

41.8.2 The Cell Namespace and the Security Namespace

The Registry database maintains three Security namespaces: the principal, group, and organization (PGO) namespaces. These namespaces are distinct from the cell namespace maintained by CDS. Security names take the form:

/.../cell_name/pgo_name

whereas CDS names take the form:

/.../cell_name/mount_point/object_name

Since the Security namespace is rooted in the CDS namespace, Security names have equivalent CDS names. Thus, for example, an entry for a principal in the Registry database has this form in the Security namespace:

/.../cell_name/principal_name

and this form in the CDS namespace:

/.../cell_name/sec/principal/principal_name

The Security mount-point name (*sec* as shown in the preceding syntax) is determined when DCE is configured. Therefore, the name may differ at individual sites.

There is no ambiguity about the Security namespace to which a name refers because Security names are always supplied in contexts that identify the namespace in question. For example, logging into DCE requires a principal name to be supplied.

However, an ACL is an object that is referenced not directly, but by the name of the object it protects. Since protected objects are not always Security objects (and therefore may be registered *only* in the CDS namespace), ACL management interfaces always take CDS names rather than Security names as input, whether or not it is the ACL of a Security object (such as a Registry database entry) that is being read or modified.

Chapter 42

Authentication

This chapter explains concepts related to authentication. The Authenticated RPC facility enables you to select the authentication protocol that your application uses to perform authentication. Among the authentication protocols that may be supported by DCE Security for use by Authenticated RPC is DCE Shared-Secret Authentication, which is the default and the chief subject of this chapter. Other authentication protocols that the Security component may support include DCE Public Key Authentication, which this guide does not discuss.

For specific information about using the Authenticated RPC routines, refer to Part 3 of this guide.

42.1 Background Concepts

The following subsections present a few background concepts that are useful for understanding the discussions of authentication in this chapter:

- Principals, which are the subjects of authentication.
- The cell, which is the environment in which authentication takes place.
- The Shared-Secret Authentication protocol, which is the mechanism by which authentication is effected when applications specify this protocol via the Authenticated RPC facility.
- Protection levels, which are the various degrees to which RPC data may be protected.
- Data encryption algorithms, which are the mechanisms that the Security Server and client runtimes use to encrypt and decrypt data exchanged between principals.

42.1.1 Principals

Previously in this guide, we defined the term ''principal'' rather loosely. ''Principal'' is more precisely defined as follows: an entity that is capable of believing that it can communicate securely with another entity. In DCE Security, principals are represented as entries in the Registry database. DCE principals include the following:

- Users, who are also referred to as ''interactive principals''
- Instances of DCE servers
- Instances of application servers
- Computers in a DCE cell
- Authentication Service surrogates

The Registry database entry representing every principal contains the name of the principal and a secret key that the principal shares with the Authentication Service. (It is the secret key that enables a principal to solve the ''puzzle'' provided by the Authentication Service.) In the case of a user, the secret key is derived from the user's password. In order to establish its identity as a principal, a noninteractive principal, such as a server or computer, must store its secret key in a data file or hardware device, or rely on a system administrator to enter it.

The Security Server itself comprises three principals that correspond to the three services that it provides: Registry, Privilege, and Authentication.

Note: The Authentication Service is an exceptional principal in that it does not share its key with any other principal. Authentication Service surrogates are also exceptional in that they are not autonomous participants in authenticated communications, as other kinds of principals are. Authentication surrogates more resemble aliases for the Authentication Services of cells. (Refer to the discussion of intercell authentication in Section 42.3 of this chapter for more information on these subjects.)

In the theory of Shared-Secret Authentication (and perhaps some other authentication protocols as well), all principals are untrusted, except for the Authentication Service itself. Therefore, a security-sensitive application would authenticate all such principals with which it may communicate. However, since the Security Service implements the Registry Service, the Privilege Service, and the Authentication Service (including its surrogates) as a single server process, it is not necessary for any DCE application to authenticate these principals.

42.1.2 Cells and Realms

The cell is the basic unit of configuration and administration in DCE. In terms of Security, a cell is the set of principals that share a secret key with an instance of the Authentication Service. Therefore, each instance of a Security Server (not counting its replicas) defines a separate cell.

From the perspective of security only, a cell is also known as a ‘‘realm.’’ We mention this because the term ‘‘realm’’ is more familiar to some readers than is the term ‘‘cell.’’ A security cell is always configured to coincide with a corresponding CDS cell, and perhaps Distributed File System (DFS) cell as well. DCE documentation refers to such a collective configuration of services as a ‘‘cell.’’

42.1.3 The Shared-Secret Authentication Protocol

Authenticated RPC enables you to specify the authentication protocol to be used in authenticating principals. Authentication protocols other than DCE Shared-Secret Authentication may or may not be supported.

DCE Shared-Secret Authentication implements an extended version of the Kerberos Version 5 system as its authentication protocol. The Kerberos system was developed at the Massachusetts Institute of Technology as part of Project Athena, and provides a trustworthy, shared-secret authentication system. The walk-through of the authentication protocol in this chapter describes the protocol in general terms.

42.1.4 Protection Levels

The protection level that an application may set using Authenticated RPC determines how much of network messages exchanged by principals are encrypted. As a rule, the higher the protection level, the greater the negative impact on performance. The Authenticated RPC facility provides several levels of protection so that applications can control tradeoffs between security and performance. Following is a summary of some of the protection levels that an application using Authenticated RPC may specify:

- Connect Level: Performs authentication only when a client and server establish a relationship
- Call Level: Attaches a verifier to each client call and server response
- Packet-Integrity Level: Ensures that none of the data transferred between two principals has been modified in transit
- Packet-Privacy Level: Incorporates lesser protection levels and in addition encrypts all RPC argument values

Refer to the discussion of Authenticated RPC in Part 3 of this guide for complete information about protection levels.

42.1.5 Data Encryption Mechanisms

Authentication protocols assume the availability of a data encryption mechanism. One that is frequently used is the Data Encryption Standard (DES), although DCE supports at least one other encryption mechanism. Your version of DCE Security may use DES for data privacy, or for principal authentication and data-integrity checking; or it may use another encryption mechanism, or no encryption at all. Consult the documentation supplied by your DCE vendor for specific information.

42.2 A Walk-Through of the Shared-Secret Authentication Protocol

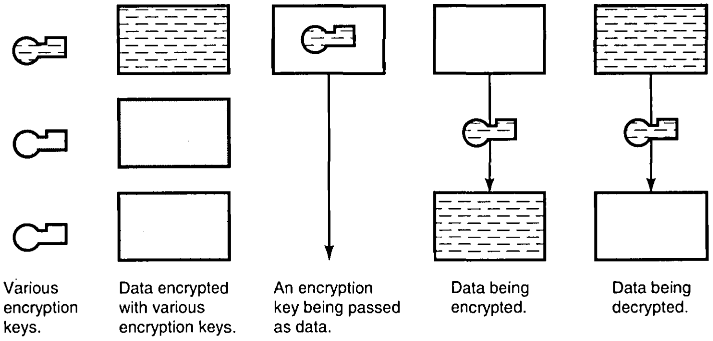
This section presents a two-part walk-through of the Shared-Secret Authentication protocol:

- The first part (''A Walk-Through of User Authentication'' in Section 42.2.1) explains what happens when a user logs in using the default DCE login tool.
- The second part (''A Walk-Through of DCE Application Authentication'' in Section 42.2.2) explains what happens when the logged-in user runs an authenticated application.

The walk-through is seen primarily from the user and the associated application-client side. Schematic representations of events related to the protocol accompany the discussions. The illustrations in this chapter do not show what literally happens when a user logs in and runs an authenticated application; they are intended only to provide a general understanding of the protocol.

In these illustrations, fill patterns represent encryption key values and encrypted data. When the key symbol appears in a box, it indicates a key is being passed as data. When the key symbol appears on a line, it indicates that encryption or decryption is taking place, depending on whether the resulting data is represented as encrypted or not (see Figure 42-1).

Figure 42–1. Representational Conventions Used in Authentication Walk-Through Illustrations


Various encryption keys.
Data encrypted with various encryption keys.
An encryption key being passed as data.
Data being encrypted.
Data being decrypted.

Note: All computer-to-computer communications are via the Remote Procedure Call mechanism, although client and server RPC runtimes are not illustrated.

Finally, note that it is unnecessary to understand the Shared-Secret protocol in order to use it. We have described it here so that application developers who may be uncertain about whether it is sufficiently secure for their needs can decide whether it is or not. If you already know that it is adequate for your needs (or are simply uninterested), go to the next chapter.

42.2.1 A Walk-Through of User Authentication

This section explains how DCE Security authenticates a user. This feature of DCE Security requires no modification of DCE nor of the applications that run on it.

Note: Refer to Figure 42-2 as you read the following steps.

1. The user logs in, entering the correct username. The login tool invokes **sec_login_setup_identity**( ), which takes the user's principal name as one of its arguments. This call causes the client Security runtime to request a Ticket-Granting Ticket (TGT) and passes the user's name (represented as a UUID) to the Authentication Service. A TGT enables a principal to be granted a ticket to a service of interest; in this case, it is the Privilege Service.

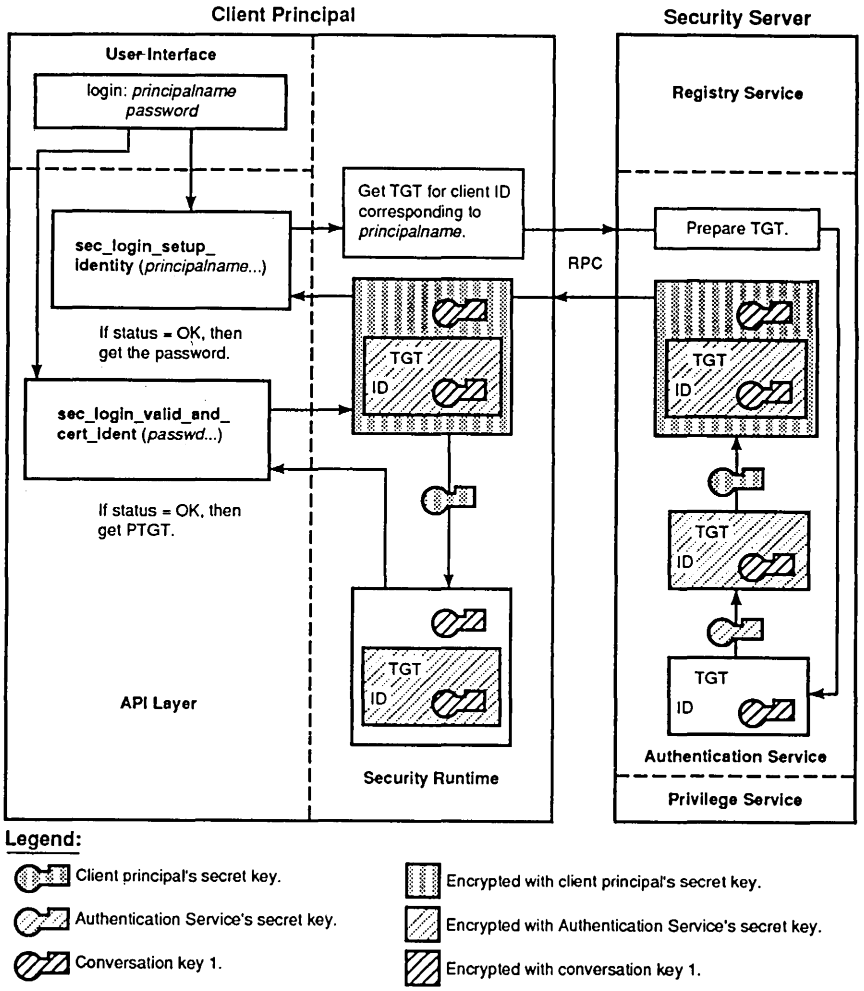
2. Upon receiving the request for a TGT, the Authentication Service obtains the user's secret key from the Registry database (where the secret keys of all principals in the cell are stored). Using its own secret key, the Authentication Service encrypts the user's identity, along with a conversation key, in a TGT. The Authentication Service seals the TGT in an ''envelope'' that is encrypted using the user's secret key. The envelope also contains the same conversation key that is encrypted in the TGT, and is returned to the client.

3. When the TGT envelope arrives, the login tool prompts the user for the password and invokes **sec_login_valid_and_cert_ident**( ). This call passes the password to the local Security runtime library. The Security runtime derives the user's secret key from the password, and uses it to decrypt the envelope. (If the user enters the wrong password, the envelope is undecryptable.) The envelope reveals the conversation key, but the Security runtime cannot decrypt the TGT, since it does not know the Authentication Service's secret key. (A validated TGT is the principal's certificate of identity.)

   Note: One of the functions of **sec_login_valid_and_cert_ident**( ) is to demonstrate that the Authentication Service knows the key of the host computer at which the principal is logging in (a server pretending to be the Security server is unlikely to know the host's key). How this is accomplished is not illustrated here, but is explained in Chapter 45.

Dividing the login sequence into two parts (**sec_login_setup_identity**( ) and **sec_login_validand_cert_ident**( )) minimizes the time that the password remains in clear text in the client's address space. From this point on, the client principal uses four different conversation keys, rather than the key derived from the user's password, to talk with other principals. This design feature obviates the need for principals (other than the Authentication Service) to know the secret keys of other principals, which is itself a security risk. Also, by using a number of short-lived conversation keys for encryption instead of a long-lived secret key, cracking encryption keys becomes a considerably more difficult task for an attacker: there are more encryption keys to discover and less time in which to discover them.

Figure 42–2. Client Acquires Ticket-Granting Ticket


Client Principal
Security Server
User Interface
login: principalname
password
Registry Service
Get TGT for client ID corresponding to principalname.
Prepare TGT.
sec_login_setup_identity (principalname...)
RPC
If status = OK, then get the password.
TGT
ID
sec_login_valid_and_cert_ident (passwd...)
If status = OK, then get PTGT.
API Layer
Security Runtime
Authentication Service
Privilege Service
Legend:
Client principal's secret key.
Authentication Service's secret key.
Conversation key 1.
Encrypted with client principal's secret key.
Encrypted with Authentication Service's secret key.
Encrypted with conversation key 1.

Note: Refer to Figure 42-3 as you read the following steps.

4. When the Security client runtime has succeeded in decrypting the envelope, the API calls a network layer interface that requests a Privilege-Ticket Granting Ticket (PTGT) from the Privilege Service. For a PTGT to be granted, however, the user must first acquire a

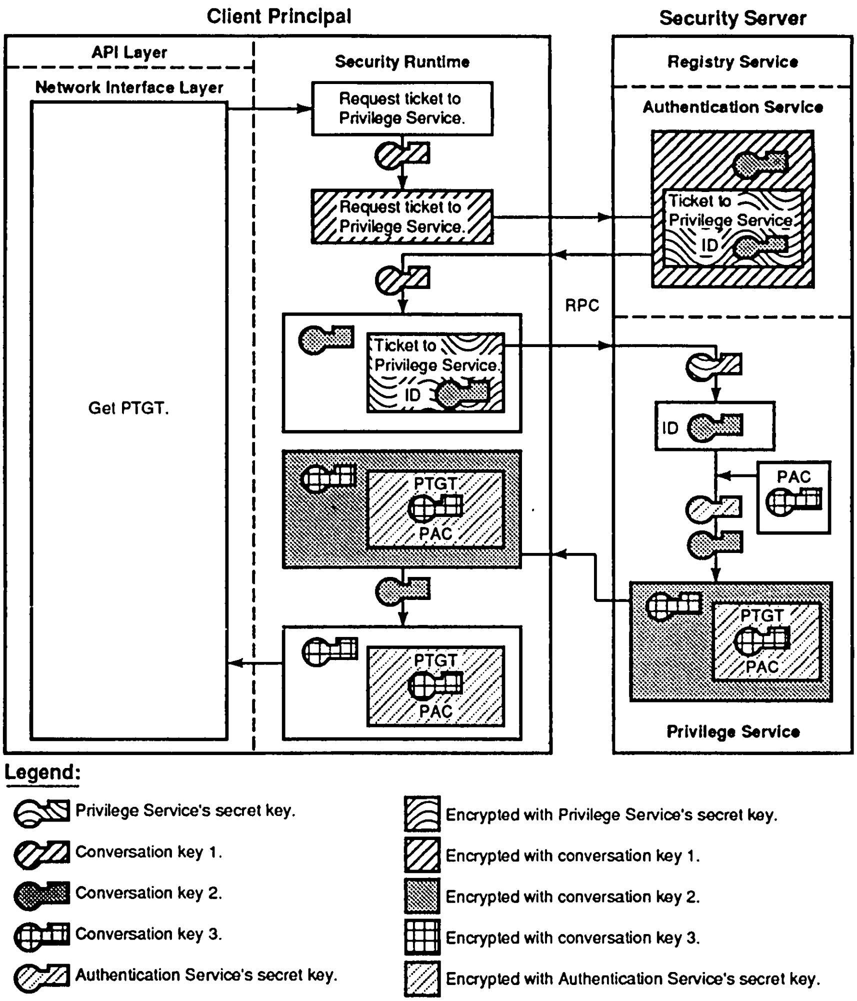
ticket to talk to the Privilege Service, which is a principal distinct from the Registry and Authentication Service. The Security runtime therefore requests such a ticket from the Authentication Service. The Security runtime encrypts this request using the conversation key it learned when it decrypted the TGT envelope.

5. Since the request for a ticket to the Privilege Service is encrypted under the conversation key associated with the TGT, the Authentication Service believes that the identity of the user is authentic; that is, no other principal could have sent a message so encrypted because no other principal knows the secret key under which the Authentication Service encrypted that conversation key. Since the user has proved to the Authentication Service knowledge of the key, the Authentication Service allows the user to talk to the Privilege Service, and so prepares a ticket to that service. This ticket contains the identity of the user (and a second conversation key) encrypted under the secret key of the Privilege Service. Like the TGT envelope, the envelope containing the ticket to the Privilege Service also contains the second conversation key, for use in conversing with the Privilege Service, and is encrypted with the first conversation key.

   Note: Beginning with Figure 42-3, the illustrations do not show the Authentication Service decrypting and reencrypting requests for tickets, since it knows all of the keys.

6. Upon receipt of the envelope containing the ticket to the Privilege Service, the Security client runtime decrypts the envelope using the first conversation key, and in the process learns the second conversation key. The client RPC runtime sends the Privilege Service ticket to the Privilege Service.

Figure 42–3. Client Acquires Privilege-Ticket-Granting Ticket


Client Principal
Security Server
API Layer
Network Interface Layer
Security Runtime
Registry Service
Authentication Service
Request ticket to Privilege Service.
Request ticket to Privilege Service.
Ticket to Privilege Service
ID
RPC
Ticket to Privilege Service.
ID
Get PTGT.
ID
PAC
PTGT
PAC
PTGT
PAC
PTGT
PAC
Privilege Service
Legend:
Privilege Service's secret key.
Conversation key 1.
Conversation key 2.
Conversation key 3.
Authentication Service's secret key.
Encrypted with Privilege Service's secret key.
Encrypted with conversation key 1.
Encrypted with conversation key 2.
Encrypted with conversation key 3.
Encrypted with Authentication Service's secret key.

7. The Privilege Service decrypts the ticket sent to it, learning both the identity of the user and the conversation key it will use to encrypt its response. The Privilege Service believes the identity is authentic because the ID information was encrypted under its own secret key, and no principal other than the Authentication Service could have encrypted the information using this secret key. Because the Privilege Service trusts the authenticity of the user's identity, it prepares a Privilege Attribute Certificate (PAC), which describes the user's privilege attributes (data that is used in making authorization decisions). The Privilege Service incorporates the user's PAC and a third conversation key into the PTGT, which is encrypted using the Authentication Service's secret key. (The Authentication Service and Privilege Service cooperate to prepare the PTGT, although the illustration only shows the Privilege Service preparing it). The PTGT envelope is encrypted using the second conversation key and also includes the third conversation key. (The Authentication Service supplies the third conversation key, although the illustration does not show this detail.)

8. The Security client runtime decrypts the PTGT envelope using the second conversation key, and learns the third conversation key. It cannot decrypt the PTGT itself, since the PTGT is encrypted under the secret key of the Authentication Service.

At this point, the Security Server has authenticated the user's identity, and as a result, the user has been able to acquire information about its privilege attributes that the Privilege Service has certified. The client now calls **sec_login_setup_identity( )** to set the login context (a handle to this user's network identity and privilege attributes) to the identity that has been established. Henceforth, processes invoked by this user assume the user's login context, and among these processes is the client-side of an application that is the subject of the rest of the walk-through.

42.2.2 A Walk-Through of DCE Application Authentication

This section explains how DCE Security authenticates an application to which the application developer has added Authenticated RPC calls. It is a continuation of the walk-through in the previous section.

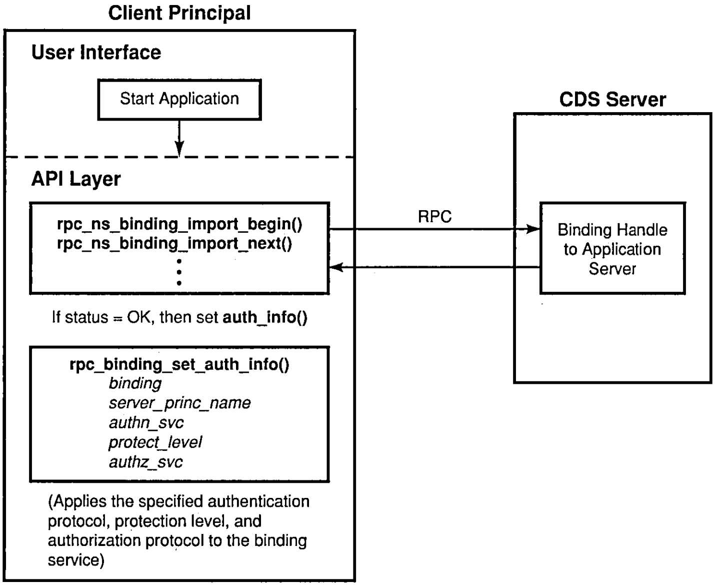
Note: Refer to Figure 42-4 as you read the following steps.

1. Having been authenticated and having acquired a PTGT, the user now invokes an application. The client side of the application calls **rpc_binding_import_begin**( ), **rpc_binding_import_next**( ), and the like. These calls specify the remote interfaces required by the client for the application.

2. The Cell Directory Service returns the client binding handles to the specified interfaces. (For this example, we have arbitrarily chosen the binding model in which the client consults the CDS for the server principal name.)

3. The client next sets authorization information for the binding handles by calling **rpc_binding_set_auth_info**( ). Among other parameters that it sets, **rpc_binding_set_auth_info**( ) sets the authentication protocol, the protection level, and authorization protocol for the binding handle corresponding to the remote interface. In this case, assume the following: the authentication protocol (*authn_svc* parameter) is DCE Shared-Secret Authentication; the protection level (*protect_level*) is Packet Privacy (all RPC argument values are encrypted); and the authorization protocol (*authz_svc*) is DCE Authorization (a PAC contains UUIDs representing the client's privilege attributes, and the server is most likely to compare this information with the ACLs protecting the objects of interest in order to determine the principal's authorization).

Figure 42–4. Client Sets Authentication and Authorization Information


Client Principal
User Interface
Start Application
API Layer
rpc_ns_binding_import_begin()
rpc_ns_binding_import_next()
If status = OK, then set auth_info()
rpc_binding_set_auth_info()
binding
server_princ_name
authn_svc
protect_level
authz_svc
(Applies the specified authentication protocol, protection level, and authorization protocol to the binding service)
RPC
CDS Server
Binding Handle to Application Server

Note: Refer to Figure 42-5 as you read the following steps.

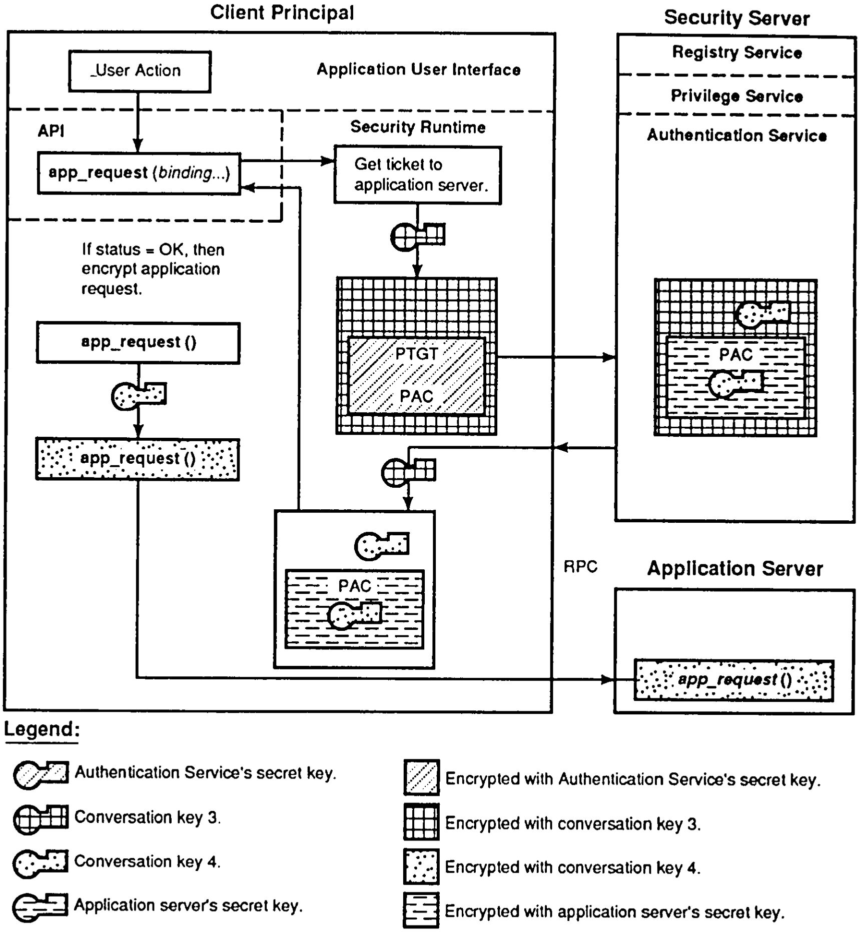
4. The client now requests some operation to be performed by the server. The client RPC runtime determines the binding handle that corresponds to the remote interface that can perform the operation, and requests a ticket to the principal that supports that interface. To acquire the ticket, the Security runtime encloses the PTGT, along with the principal name of the application server, in an envelope encrypted under the third conversation key. The client sends the envelope to the Authentication Service.

5. The Authentication Service uses the application server's secret key to reencrypt the PAC and a fourth conversation key. The ticket to the application server is in turn encrypted with the third conversation key in an envelope that also includes the fourth conversation key. The Authentication Service returns the envelope to the client's Security runtime.

6. The Security runtime decrypts the envelope using the third conversation key, in the process learning the fourth conversation key. The Security runtime then uses the fourth conversation key to encrypt the application request to the server, and the client RPC runtime sends the application request to the server.

7. The Security runtime receives the client's request, and learns from the header that the request is authenticated.

Figure 42–5. Client Principal Makes Application Request


Client Principal
Security Server
User Action
Application User Interface
Registry Service
Privilege Service
API
Security Runtime
Authentication Service
app_request (binding...)
Get ticket to application server.
If status = OK, then encrypt application request.
app_request ()
PTGT
PAC
PAC
app_request ()
PAC
RPC
Application Server
app_request ()
Legend:
Authentication Service's secret key.
Conversation key 3.
Conversation key 4.
Application server's secret key.
Encrypted with Authentication Service's secret key.
Encrypted with conversation key 3.
Encrypted with conversation key 4.
Encrypted with application server's secret key.

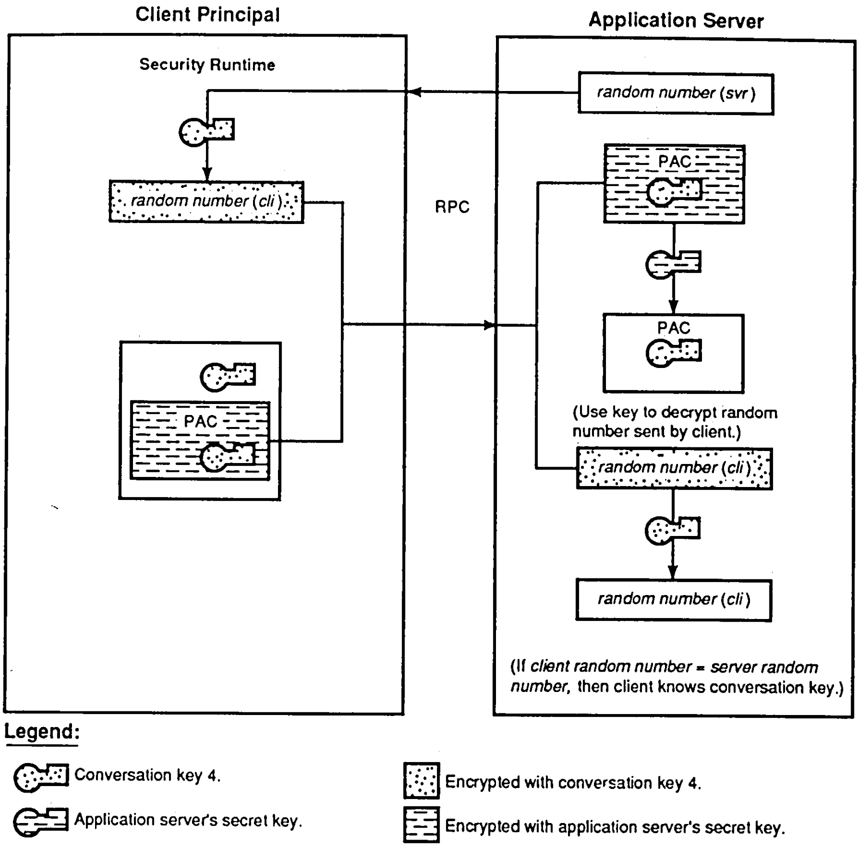
Note: Refer to Figure 42-6 as you read the following steps.

8. Before fulfilling the client's request, the Security runtime must learn the conversation key for communicating with the client, and the client's authorization. To begin the challenge to the client's identity and authorization, the runtime generates a random number and sends it (in plaintext) to the client.

9. The Security runtime encrypts the random number using the fourth conversation key, which the Authentication Service gave it for the purpose of talking to the application server. The RPC runtime sends the encrypted random number and the server ticket to the application server.

10. The Security runtime decrypts the ticket using its secret key, in the process learning the conversation key and the client's authorization. It uses the conversation key to decrypt the number sent by the client. Since the number is the same random number that the server sent previously, the runtime concludes that the client knows the conversation key, and therefore that the client's identity is authentic.

Figure 42–6. Application Server Challenges Client

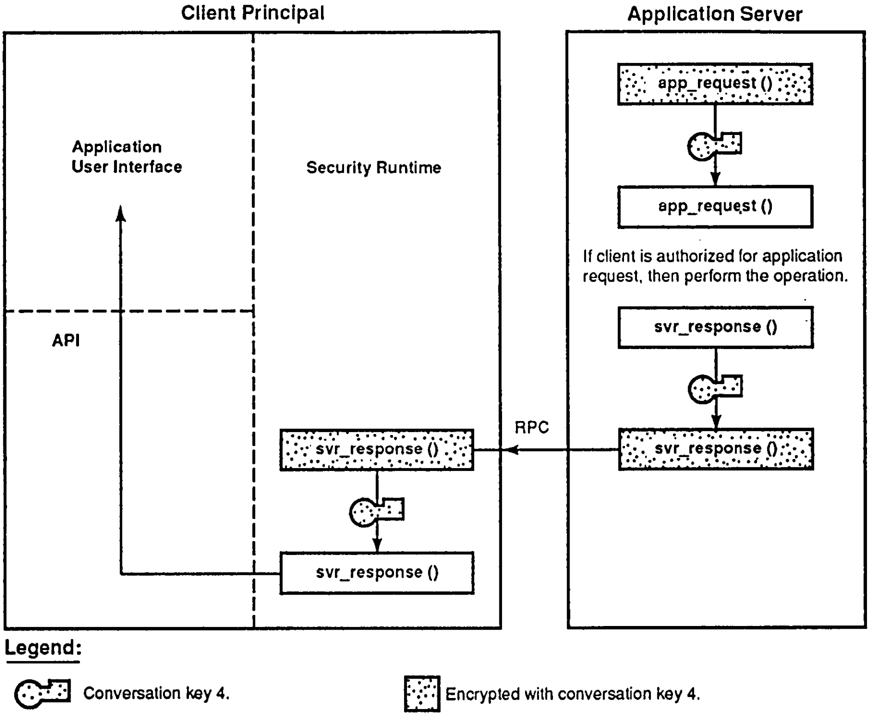

Client Principal
Application Server
Security Runtime
random number (svr)
PAC
random number (cli)
RPC
PAC
PAC
(Use key to decrypt random number sent by client.)
random number (cli)
random number (cli)
(If client random number = server random number, then client knows conversation key.)
Legend:
Conversation key 4.
Application server's secret key.
Encrypted with conversation key 4.
Encrypted with application server's secret key.

Note: Refer to Figure 42-7 as you read the following steps.

11. The Security runtime for the application server uses the fourth conversation key to decrypt the client's request, and if it determines from the information in the PAC that the client is authorized, it performs the server operation and prepares a response. The server runtime encrypts the response using the conversation key and sends it to the client.

12. The client runtime receives and decrypts the response, and returns data to the application interface through the API.

Figure 42–7. Application Server Responds to Client's Request



The application server and client can continue to use the fourth conversation key indefinitely for subsequent conversations. If the server receives an application request after discarding the conversation key, which it may do if it has not heard from client for some time, then the server challenges the client to learn the key (see Figure 42-6). If the client's ticket to the application server expires, it must acquire a new one (see Figure 42-5), and so on. If the client wishes to talk to a new service, it must acquire a ticket to that service (see Figure 42-5).

Note: The illustrations in the walk-through show the authentication protocol in the context of a datagram-based network communications protocol. In the case of a connection-oriented protocol, the client sends both the application request and the ticket to the server at connection setup, rather than separately, as illustrated in Figures 42-5 and 42-6.

42.3 Intercell Authentication

While the intercell authentication model is an extension of intracell authentication, there are certain concepts that are particular to intercell authentication. The following subsections discuss those concepts.

42.3.1 Authentication Service Surrogates

A principal trusts another principal in its cell because it trusts the Authentication Service to authenticate all principals that are members of the cell, except for the Authentication Service itself, which its member principals trust *a priori*. The Authentication Service can authenticate all principals in its cell because it shares a secret key with each of them. A principal that wants to talk to a foreign principal (that is, a principal in another cell) must acquire a ticket to that principal. Furthermore, the ticket must be encrypted in the secret key of the foreign principal, or else the foreign principal may disregard the initiator of the conversation. The local principal cannot get such a ticket from its own Authentication Service because the local Authentication Service does not know the secret keys of any foreign principals. Therefore, there must be some other means by which the two instances of the Authentication Service can securely convey information about their respective principals to one another.

Besides the fact that it is trusted *a priori*, a cell's Authentication Service is an exceptional principal in this other respect: other kinds of principals share their secret keys with the local Authentication Service, whereas the Authentication Service's key is private; that is, known to no other principal. Thus, one problem of intercell authentication is the means by which the Authentication Service in one cell may communicate securely with that in another cell without either of them having to share their private keys, which would introduce an unacceptable security risk.

Note: The Kerberos network authentication specification makes a distinction between the terms ''secret'' and ''private.'' The term ''secret'' refers to data that is known to two principals, and the term ''private'' refers to data that is known to only one principal. We make the same distinction in this guide.

The solution to this problem is an extension of the Shared-Secret Authentication model previously discussed in this chapter; that is, an entry in the Registry database of one cell specifies the same secret key as that in an entry in the other cell's Registry database. The two Registry database entries are known as mutual authentication surrogates, and the two cells that maintain mutual authentication surrogates are called ''trust peers.'' It is through their surrogates that two instances of the Authentication Service are enabled to convey information about their respective principals to one another, thus enabling a principal from one cell to acquire a ticket to a principal in another cell.

An authentication surrogate is a principal in the sense that it is represented by an entry in a Registry database, but it is not an autonomous participant in authenticated communications in the same sense that, for example, a user or a server is. Rather, it is more like an alias that is assumed by a cell's Authentication Service when it communicates with a trust peer. The establishment of a trust peer relationship between two cells is an implicit expression of mutual trust in the two Authentication Services on the part of the cell administrators who establish the relationship; administrators use the **rgy_edit** tool to establish the relationship.

42.3.2 Intercell Authentication by Trust Peers

This section explains how a client principal in one cell is authenticated by an Authentication Service in a peer cell so that the client principal may communicate with another principal that is a member of the foreign cell.

1. A client principal, having already been authenticated by its Authentication Service and having acquired its PAC, requests a service from a foreign cell. The client specifies the server principal that provides the service by its fully qualified name, which identifies the foreign cell as well as the cell-relative server principal name.

2. Recognizing by its name that the server principal is foreign, the client's Security runtime makes a request to the local Authentication service for a TGT to the Authentication service of the foreign cell of which the server principal is a member. The request for the foreign TGT (FTGT) proceeds like a ticket-granting request for any other target principal. The local Authentication Service constructs the ticket, preserving PAC data from client's existing PTGT, and encrypts it using the secret key that the two Authentication surrogates share.

3. Upon receiving the request for the FTGT, the foreign Authentication Service decrypts it using the surrogates' secret key, and returns a ticket to the foreign Privilege Service to the client's Security runtime.

4. The client's Security runtime uses the ticket to the foreign Privilege Service to obtain a Foreign Privilege-Ticket-Granting Ticket (FPTGT). The FPTGT is simply the client's original PAC encrypted with the key of the foreign Privilege Service.

5. After the client principal receives the FPTGT, it requests a ticket to the foreign server principal from the foreign Authentication Service, exactly as it would request a ticket to a local principal from its own Authentication Service. The client principal may also reuse the FPTGT to the foreign cell to acquire tickets to any other principals in that cell.

Chapter 43

Authorization

This chapter explains concepts related to authorization. The Authenticated RPC facility enables you to select the authorization protocol that your application uses. Among the authorization protocols supported by DCE Security for use by Authenticated RPC is DCE Authorization (the default), and Name-Based Authorization.

This chapter first discusses DCE Authorization, and more particularly, DCE Access Control Lists (ACLs). At the end of this chapter, we also briefly discuss the Name-Based Authorization protocol.

43.1 DCE Authorization

The DCE Authorization protocol is based in part on the UNIX system file-protection model, but is extended with ACLs. An ACL is a list of access control entries that protects an object. Each entry in the ACL specifies a set of permissions. Usually, most of the entries in the ACL specify a privilege attribute (such as membership in a group) and the set of permissions that may be granted to the principal(s) that possesses that privilege attribute. Some other entries specify a set of permissions that may mask the permission set in a privilege attribute entry.

Every ACL is managed by an ACL manager type. An ACL manager type determines a principal's authorization to perform an operation on an object by reading the object's ACL to find the appropriate entry (or entries) that matches some privilege attribute possessed by the principal. If the type of access requested by the principal is one of the permissions listed in the matching entry, and assuming no applicable mask entry denies that permission, then the ACL manager type allows the principal to perform the requested operation. If the requested permission is not listed in the matching ACL entry, or is denied by a mask, permission to perform the operation is denied. Permission to perform the operation is also denied if the ACL contains no matching privilege attribute entry.

Unlike UNIX system file permissions, DCE ACLs are not limited to the protection of file system objects; that is, files, directories, and devices. ACLs may also control access to nonfile-system objects, such as the individual entries in a database.

Note: The implementation of DCE ACLs is aligned with POSIX P1003.6 Draft 12.

In the discussions in this chapter, we use the general term ''name'' to refer to a principal, group, or cell identifier; but readers should always bear in mind that these names have two representations: as UUIDs in ACL program interfaces, and as print strings in user interfaces.

43.1.1 Object Types and ACL Types

The ACL facility distinguishes between two types of objects: container objects and simple objects. Container objects contain other objects, which may be simple and/or other container objects. Simple objects do not contain other objects. Examples of container objects may include a file-system directory or a database; examples of simple objects may include a file or a database entry.

To protect both object types, and to enable newly created objects to inherit default ACLs from their parent container objects, the ACL facility supports two basic kinds of ACLs:

- An Object ACL is associated with either a container or a simple object, and controls access to it.

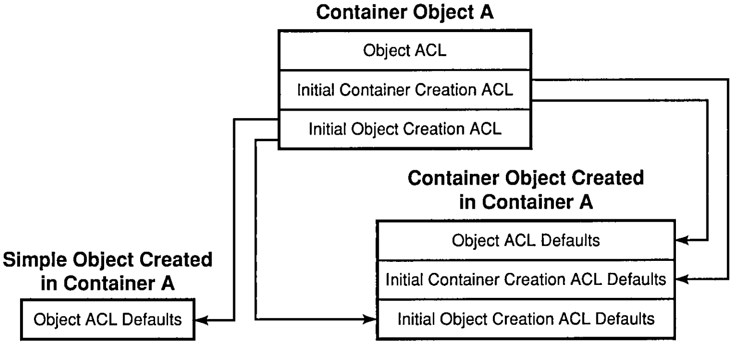
- A Creation ACL is associated with a container object only. Its function is not to control access to the container, but to supply default values for the ACLs of objects created in the container. There are two types of Creation ACLs:

  — An Initial Object Creation ACL supplies default values for a simple object's Object ACL and for a container object's Initial Object Creation ACL.

  — An Initial Container Creation ACL supplies default values for both a container object's Object ACL and its Initial Container Creation ACL.

Figure 43-1 illustrates how ACL defaults are derived from Creation ACLs.

Figure 43–1. Derivation of ACL Defaults


Container Object A
Object ACL
Initial Container Creation ACL
Initial Object Creation ACL
Container Object Created in Container A
Object ACL Defaults
Initial Container Creation ACL Defaults
Initial Object Creation ACL Defaults
Simple Object Created in Container A
Object ACL Defaults

Aside from the distinctions previously described, there are no differences between Object ACLs and Creation ACLs; therefore, the information about ACLs in the rest of this chapter does not differentiate between them.

43.1.2 ACL Manager Types

A separate ACL manager type manages the ACLs for each class of objects for which permissions are uniquely defined. The manager type defines the permissions for those objects whose ACLs it manages: the number of permissions, the meanings of the permissions, and the tokens that represent the permissions in user interfaces to ACL manipulation tools (the default DCE tool is **acl_edit**).

For example, for the purpose of access control, five classes of objects are defined in the Registry database, and five ACL manager types manage the ACLs for the Registry database objects (the five Registry manager types run in a single Security Server process). Other DCE components implement their own manager types, and applications layered on DCE may also implement manager types for the objects that the applications protect.

Refer to the *OSF DCE Administration Guide* and the *OSF DCE Administration Reference* for information about standard DCE ACL manager types and the permissions they implement. Refer to Part 1 and to Chapter 47 of this guide for information about implementing ACL manager types for distributed applications.

43.1.3 Access Control Lists

An ACL consists of

- An ACL manager type identifier, which identifies the manager type of the ACL.
- A default cell identifier, which specifies the cell of which a principal or group identified as local is assumed to be a member. A DCE global pathname is necessary to specify a principal or a group from a nondefault cell; this consists of a pair of UUIDs representing the principal or group, and the cell of which it is a member. It is necessary to use the ID Map API to convert the global print string names of foreign principals and groups to the UUID representations that DCE ACL managers recognize. (Refer to Chapter 48 for more information on this subject.)
- At least one ACL entry.

The rest of this chapter discusses ACLs primarily from a user-interface point of view, since this perspective provides an orientation to the discussion of the ACL API in this part.

43.1.4 ACL Entries

DCE Authorization defines two basic kinds of ACL entries:

1. Those that associate a specified privilege attribute with a permission set; these are privilege attribute entries.
2. Those that specify a permission set that masks a permission set specified in a privilege attribute entry; these are mask entries.

The following subsections describe the two kinds of ACL entries in detail.

43.1.4.1 Privilege Attribute Entry Types

The privilege attributes of a principal are based on identity and include the principal's name, its group membership(s), and native cell. Note that not all ACL manager types implement all privilege attribute entry types. For example, the ACL manager type of a database object probably would not support the **user_obj** and **group_obj** entry types.

Note: The term ''local cell'' means the cell specified in the ACL header; this is not necessarily the cell in which the protected object resides.

Following are descriptions of the ACL entry types that specify privilege attributes:

- **user_obj**

  The **user_obj** entry establishes the permissions for the object's ''user'' (in the UNIX system sense). An ACL may contain only one entry of this type. The identity of the principal to which this ACL entry refers is assumed to be local and is specified somewhere other than in this entry. In the case of a file, for example, the identity is attached to the file's inode.

- **user**

  The **user** entry establishes the permissions for the local principal named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the principal it specifies.

- **foreign_user**

  The **foreign_user** entry establishes the permissions for the foreign principal named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the foreign principal it specifies. This entry type is exactly like the **user** entry type, except that this entry explicitly names a cell (for the entry type **user**, the principal inherits the cell specified by the default cell identifier in the ACL header).

- **group_obj**

  The **group_obj** entry establishes the permissions for the object's ''group'' (in the UNIX system sense). An ACL may contain only one entry of this type. As is the case with the **user_obj** entry, the identity of the group is assumed to be local and is specified elsewhere than in the **group_obj** entry itself.

- **group**

  The **group** entry establishes the permissions for the local group named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the group it specifies.

- **foreign_group**

  The **foreign_group** entry establishes the permissions for the foreign group named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the foreign group it specifies. This entry type is exactly like the **group** entry type, except that this entry explicitly names a cell (for the entry type **group**, the principals inherit the default cell identifier).

- **other_obj**

  The **other_obj** entry establishes the permissions for local principals whose identities do not correspond to any entry type that explicitly names a principal or group; an ACL may contain only one entry of this type.

- **foreign_other**

  The **foreign_other** entry establishes the permissions for all principals that are members of a specified foreign cell and whose identities do not correspond to any **foreign_user** or **foreign_group** entry. An ACL may contain a number of entries of this type, but each entry must specify a different foreign cell.

- **any_other**

  The **any_other** entry establishes the permissions for principals whose privilege attributes do not match those specified in any other entry type. An ACL may contain only one entry of this type.

ACL entries for privilege attributes consist of three fields in the form:

entry_type[:key]:permissions

Following are descriptions of the fields:

- The ACL *entry_type* specifies an ACL entry type as described in the previous list.

- The *key* field specifies the privilege attribute to which the permissions listed in the entry apply. The key field for the ACL entry types **user, group, foreign_user, foreign_group,** and **foreign_other** explicitly names a principal, group, or cell. For the entry types **foreign_user**, **foreign_group**, and **foreign_other**, the key field must contain a global DCE pathname of the forms */.../cellname/principalname, /.../cellname/groupname,* or */.../cellname,* respectively. The entry types **user_obj, group_obj, other_obj,** and **any_other** do not use the key field.

- The *permissions* field lists the permissions that may be granted to the principal possessing the privilege attribute specified in the entry, unless a mask (or masks) further restricts the permissions that may be granted to the principal. As noted previously, the number and meaning of the permissions that may protect an object are defined by the object's ACL manager type. Therefore, the permissions that an ACL entry may specify must be the set, or a subset, of the permissions implemented by the manager type of the ACL in which the entry appears.

  A principal is denied access when a **user** or **foreign_user** entry that names the principal contains an empty permission set.

43.1.4.2 Mask Entry Types

Following are descriptions of the ACL entry types that specify masks:

- **mask_obj**

  The **mask_obj** entry establishes the permission set that masks all privilege attribute entry types except the **user_obj** and **other_obj** types.

- **unauthenticated**

  The **unauthenticated** entry establishes the permission set that masks the permission set in a privilege attribute entry that corresponds to a principal whose privilege attributes have not been certified by an authority such as the Privilege Service.

The two masks are similar in this respect: the permission set specified in the mask entry is intersected (logically ANDed) with the permission set in a privilege attribute entry. This masking operation yields the effective permission set (that is, the permissions that may be granted to the principal) for the principal possessing the privilege attribute. For example, if a privilege attribute entry specifies the permissions **ab**, and a mask entry that specifies the permissions **bc** masks that privilege attribute entry, then the effective permission set is **b**. Similarly, a mask entry that specifies the empty permission set means that none of the permissions in any privilege attribute entry that that mask entry masks is granted to the principal possessing the privilege attribute.

The two masks are dissimilar in one notable respect. Adding an **unauthenticated** mask entry with an empty permission set to an ACL is equivalent to omitting the **unauthenticated** mask entry from the ACL: in both cases, the set of effective permissions for principals possessing unauthenticated privilege attributes is empty. However, adding a **mask_obj** entry with an empty permission set to an ACL is different from having no **mask_obj** entry in the ACL: in the first case, the effective permission set is empty; in the second case, the effective permission set is identical to the permission set in the privilege attribute entry.

ACL entries for masks consist of two fields in the form:

entry_type:permissions

Following are descriptions of the fields:

- The *entry_type* field specifies one of the two masks entry types: **mask_obj** or **unauthenticated.**
- The *permissions* field specifies the permission set that masks the permission set in any privilege attribute entry masked by the mask entry.

43.1.4.3 The Extended ACL Entry Type

The ACL entry type **extended** is a special entry type for ensuring the compatibility of ACL data created by different software revisions. It enables old application clients to copy ACLs from one newer revision object store to another without losing data. It also enables obsolete clients to manipulate ACL data that they understand without corrupting the extended entries that they do not understand.

43.1.5 Access Checking

Standard DCE ACL manager types use a common access-check algorithm to determine the permissions they grant to a principal. Access checking is executed in up to six stages, in the following order:

1. The **user_obj** entry check
2. The check for a matching **user** or **foreign_user** entry
3. The **group_obj** entry check and the check for matching **group** or **foreign_group** entries
4. The **other_obj** entry check
5. The check for a matching **foreign_other** entry
6. The **any_other** check

If during any stage of access checking an ACL manager type finds a privilege attribute entry that matches a privilege attribute possessed by a principal, then the manager type does not execute any subsequent stages, even though the principal may possess other privilege attributes for which there are other matching entries.The following subsections describe the algorithms used at each stage of access checking.

43.1.5.1 The user_obj Entry Check

The pseudocode that follows illustrates the **user_obj** check algorithm. If the principal seeking access is the identity to which the **user_obj** entry refers, then the remaining checks are not executed.

```
IF (no USER_OBJ principal name is available)
THEN
  the requested permission is denied
ELSE IF (the principal name matches the user name associated
        with the USER_OBJ entry) AND (the cell name matches
        the cell name for that entry)
THEN
  IF (the requested permission is listed in the USER_OBJ entry)
  THEN
    IF (the principal's privilege attributes are certified)
    THEN
      the requested permission is granted
    ELSE
      IF (the requested permission is listed in the
         unauthenticated mask entry)
      THEN
        the permission is granted
      ELSE
        the permission is denied
      ENDIF
    ENDIF
  ENDIF
ELSE
  the permission is denied
ENDIF
```

43.1.5.2 The User Entries Check

The pseudocode that follows illustrates the algorithm for checking **user** or **foreign_user** entries. If the principal's identity matches one of these entries, then the remaining checks are not executed.

```
IF (the principal name matches the user name of any USER
      or FOREIGN_USER entry) AND (the principal's cell name
      matches the cell name for that entry)
THEN
  IF (the requested permission is listed in the USER or
     FOREIGN_USER entry) AND ((the requested permission
     is listed in the mask_obj entry) OR (there is no
     mask_obj entry))
  THEN
    IF (the principal's privilege attributes are certified)
    THEN
      the requested permission is granted
    ELSE
      IF (the requested permission is listed in the
         unauthenticated mask entry)
      THEN
        the permission is granted
      ELSE
        the permission is denied
      ENDIF
    ENDIF
  ELSE
    the permission is denied
  ENDIF
ENDIF
```

43.1.5.3 The Group Entries Check

The pseudocode that follows illustrates the algorithm for checking group entries. If a principal is associated with a concurrent group set, more than one search of the ACL entries for groups is executed: one for the primary group (the one specified in the principal's account information), and one for each group in the concurrent group set.

The permissions granted are the union (the logical OR operation) of the permissions yielded by each search of the group entries. For example, if two groups of which an authenticated principal is a member specify the permission sets **abc** and **cde**, then the principal is granted the permission set **abcde**.

If one or more matching group entries are found, then the remaining checks are not executed.

```
IF (a group name among the principal's privilege
      attributes matches the group ID of any GROUP_OBJ, GROUP,
      or FOREIGN_GROUP entry) AND (the principal's cell name
      matches the cell name for that entry)
THEN
  IF (the requested permission is listed in the group entry)
     AND ((the requested permission is listed in the
     mask_obj entry) OR (there is no mask_obj entry))
  THEN
    IF (the principal's privilege attributes are certified)
    THEN
      the permission is granted
    ELSE
      IF (the requested permission is listed in the
         unauthenticated mask entry)
      THEN
        the permission is granted
      ELSE
        the permission is denied
      ENDIF
    ENDIF
  ELSE
    the permission is denied
  ENDIF
ENDIF
```

43.1.5.4 The other_obj Entry Check

The pseudocode that follows illustrates the algorithm for checking the **other_obj** entry.

```
IF (the requested permission is listed in the OTHER_OBJ entry
      AND (the principal's cell name matches the cell name for
      that entry)
THEN
  IF (the principal's privilege attributes are certified)
  THEN
    the permission is granted
  ELSE
    IF (the requested permission is listed in the
       unauthenticated mask entry)
    THEN
      the permission is granted
    ELSE
      the permission is denied
    ENDIF
  ENDIF
ELSE
  the permission is denied
ENDIF
```

43.1.5.5 The foreign_other Entries Check

The pseudocode that follows illustrates the algorithm for checking the **foreign_other** entries.

```
IF (the requested permission is listed in a FOREIGN_OTHER
      entry) AND (the principal's cell name matches the cell name
      for that entry) AND ((the requested permission is listed
      in the mask_obj entry) OR (there is no mask_obj entry))
THEN
  IF (the principal's privileges are certified)
  THEN
    the permission is granted
    ELSE
      IF (the requested permission is listed in the
         unauthenticated mask entry)
      THEN
        the permission is granted
      ELSE
        the permission is denied
      ENDIF
    ENDIF
  ELSE
    the permission is denied
  ENDIF
```

43.1.5.6 The any_other Entry Check

The pseudocode that follows illustrates the **any_other** check algorithm. If no privilege attribute possessed by a principal matches any entry checked in any preceding stage of access checking, then the principal may be granted the effective permissions yielded by this check.

Note that if an ACL listing this entry also lists the **other_obj** entry, then only undistinguished foreign identities can match this entry. However, if the ACL does not list the **other_obj** entry, then all undistinguished identities, whether foreign or local, match this entry.

```
IF (the requested permission is listed in the any_other entry
      AND ((the requested permission is listed in the mask_obj
      entry) OR (there is no mask_obj entry))
THEN
  IF (the principal's privilege attributes are certified)
  THEN
    the permission is granted
  ELSE
    IF (the requested permission is listed in the
       unauthenticated mask entry)
    THEN
      the permission is granted
    ELSE
      the permission is denied
    ENDIF
  ENDIF
ELSE
  the permission is denied
ENDIF
```

43.1.6 Examples of ACL Checking

The following subsections provide some examples that illustrate ACLs and the access-check algorithms. The examples use the arbitrary convention of ordering entries as follows: masks, principals, groups, and ‘‘other’’ entries. However, the access check algorithm disregards the order in which entries appear in an ACL. Also note that the permissions in these examples do not refer to any particular permissions implemented by any ACL manager type.

43.1.6.1 Example 1

Following is an ACL that protects an object to which three principals, **janea**, **/.../cella/fritzb**, and **mariac**, seek access:

```
mask_obj:ab
user_obj:abc
user:janea:abdef
foreign_user:/.../cella/fritzb:abc
group:projectx:abcf
group:projecty:bcg
```

Note: The numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access-check algorithm referred to in Section 43.1.5.

The principal **janea** requests permission **c** to the object protected by the ACL. Assume that the principal **janea** has the privilege attributes of being a member of the groups **projectx** and **projecty** (as well as having a **user** entry that names her) and that **janea** is the principal to which the **user_obj** entry refers. Assume also that this principal's privilege attributes are certified:

1. The **user_obj** check yields the permissions **abc**.

The result of this check is that the effective permission set for **janea** is **abc**. Because a matching entry is found during the first stage of access checking, none of the remaining stages of access checking is executed, even though there are three other matching entries. The **mask_obj** entry does not mask the **user_obj** entry, so **janea**'s effective permissions are the permissions in the **user_obj** entry. Since **janea** requested a permission that is a member of the effective permission set, her request is granted.

The second principal seeking access to the protected object is **/.../cella/fritzb**. This principal requests permission **b**. Assume that **user_obj** resolves to some identity other than **/.../cella/fritzb,** and that this principal's privilege attributes are uncertified:

1. The **user_obj** check yields no permissions because **/.../cella/fritzb**'s identity does not match that of the **user_obj** (no foreign principal can ever match this entry).

2. The **foreign_user** entry for **/.../cella/fritzb** specifies the permissions **abc**. The application of the **mask_obj**, which specifies the permissions **ab** to this permission set, yields the permissions **ab**. Since the **unauthenticated** mask entry is missing from the ACL, all permissions for unauthenticated identities are masked, yielding an empty effective permission set.

The result of these checks is that **/.../cella/fritzb**'s request is denied (and would be denied, regardless of the permission requested). In this case, only the first two stages of access checking are executed.

The third principal seeking access is **mariac**, who requests permission **a**. Assume that the privilege attributes of **mariac** are certified, that **mariac** is not the principal that corresponds to the **user_obj** entry, and that **mariac** is a member of the groups **projectx** and **projecty**:

1. The **user_obj** check yields no permissions.

2. There is no matching user entry.

3. The group check finds two matching entries. The permissions associated with **projectx** (**abcf**) when masked by the **mask_obj** entry

(**ab**) yield the permissions **ab**. The permissions associated with **projecty** (**bcg**) when masked by the **mask_obj** entry yield the permission **b**. The union of the permission sets **ab** and **b** is the set **ab**.

The effective permission set for **mariac** is **ab**, and since the requested permission (**a**) is a member of that set, **mariac**'s request is granted. The remaining stages of access checking are not executed.

43.1.6.2 Example 2

Following is the ACL for an object to which two principals, **ugob** and **/.../cellb/lolad**, seek access:

```
mask_obj:bcde
unauthenticated:ab
user_obj:abcdef
user:ugob:abcdefg
group:projectz:abh
foreign_other:/.../cellb/:abc
```

Note: The numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access check algorithm referred to in Section 43.1.5.

The principal **ugob** requests permission **b.** Assume that **ugob** is not the principal to which the **user_obj** entry refers. Assume also that the privilege attributes of **ugob** include membership in the group **projectz**, in addition to the **user** entry that names him. In this case, the principal has failed to acquire certified privilege attributes:

1. The **user_obj** check yields no permissions.
2. The matching entry among the user entries specifies the permissions **abcdefg.** Applying **mask_obj** (**bcde**) yields the permission set **bcde.** Applying the **unauthenticated** mask (**ab**) to the permission set **bcde** yields the effective permission set **b.**

Since the principal **ugob** requests a permission (**b**) that is a member of the effective permissions set, this principal's request is granted.

A case that illustrates how access is determined for otherwise undifferentiated members of a specified foreign cell is that of the principal **/.../cellb/lolad**, who requests permission **e**. Assume that the privilege attributes of this principal are certified:

1. The principal is foreign, so the **user_obj** check cannot be a match.
2. There are no **foreign_user** entries.
3. There are no **foreign_group** entries.
4. The principal **lolad** is a member of **cellb**, meaning that the privilege attributes match those in the **foreign_other** entry for **cellb.** The permissions specified by the **foreign_other** entry for **cellb** (**abc**) as masked by **mask_obj** (**bcde**) yields the effective permission set **bc**.

The permission requested (**e**) is not a member of the effective permission set (**bc**), so the request is denied.

43.1.6.3 Example 3

Following is the ACL for an object to which one principal, **silviob**, seeks access.

```
unauthenticated:a
user:jeand:abcde
user:denisf:-
group:projectx:abcd
foreign_other:/.../cella:-
foreign_other:/.../cellc:abc
any_other:ab
```

Note: The **user** entry for **denisf** and the **foreign_other** entry for **cella** both specify an empty permission set with notation – (dash), meaning that identities corresponding to these entries are explicitly denied all permissions. Also, the numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access-check algorithm referred to in Section 43.1.5.

The principal **silviob** requests permission **a**. Assume that this principal's privileges include membership in the group **projecty** and that they are not certified:

1. There is no **user_obj** entry, so this check can yield no permissions.
2. There is no **user** entry for this principal, so this check yields no permissions.
3. There is no entry for the group **projecty**, so this check yields no permissions.
4. There is no **other_obj** entry, so this check can yield no permissions.
5. The principal is local, so no **foreign_other** entry can be a match; this check yields no permissions.
6. Having failed to match any entry examined in the preceding checks, the principal matches the **any_other** entry, which yields the permission set **ab**. There is no **mask_obj** entry, but there is the **unauthenticated** mask entry, which specifies the permission set **a**. Applying the **unauthenticated** mask to this privilege attribute entry yields the effective permission **a**.

The permission requested (**a**) is a member of the effective permission set (**a**), so this principal's request is granted.

43.2 Name-Based Authorization

The Kerberos authentication service, upon which the DCE Shared-Secret Authentication protocol is based, authenticates the string name representation of a principal. DCE Security converts these string representations to UUIDs, and it is these UUIDs that an ACL manager uses to make authorization decisions. However, since some existing (non-DCE) applications implement Kerberos authentication, DCE Security supports an authorization protocol based on principal string names: Name-Based Authorization.

It is assumed that applications that use Name-Based Authorization have a means to associate string names with permissions, since DCE Security offers no such facility. Because in Name-Based Authorization there is no UUID representation of privilege attribute data, and because DCE ACL managers recognize only UUIDs, if an application uses Name-Based Authorization, then a principal's privilege attributes are represented as an anonymous PAC. Such PAC data can only match the ACL entry types **other_obj**, **foreign_other**, or **any_other**, and are masked by the **unauthenticated** mask.

Also note that there is essentially no intercell security for an application that uses the Name-Based Authorization protocol: such applications never communicate with the Privilege Service, which evaluates intercell trust.

Chapter 44

The Registry Application Program Interface

This chapter describes the Registry API. Like the other Security APIs, this one provides a simpler binding mechanism than the standard RPC handle structure. It includes facilities for creating and maintaining the Registry database. Applications that run in the default DCE Registry environment (that is, those that assume the presence of the default Registry tools and servers) have no reason to call this API.

44.1 Binding to a Registry Site

Although it is often convenient to speak of the Registry database in a way that implies that it is a single physical database, the Registry database is replicated in all but the very smallest cells. Replication reduces network traffic and increases the availability of Registry data to clients. A cell's Registry database usually consists of an update site (also known as the master site), and a number of query sites (also known as read-only, or slave sites). Changes to data at the master site are propagated to its slaves by messages sent by the master. Query sites can only satisfy requests for data (for example, **sec_rgy_acct_lookup**( ), which returns account information). Requests for database changes (for example, **sec_rgy_acct_passwd**( ), which changes the password for an account) must be directed to the master site; a query site that receives such a request returns an error.

To submit requests to the Registry Server, a client must first select a site and bind to it. The client may select a site by name, ask the Directory Service to bind to the master site, or select an arbitrary site.

The Registry API enables a client to communicate with the Registry server using a specified authentication protocol, at a specified protection level, and using a specified authorization protocol. For instance, a developer may decide that the protection level for communicating with an update site should be higher (that is, more secure) than that for a query site; that is, the developer may feel that, on the one hand, the relatively infrequent changes to Registry data should be done in a highly secure manner, and that on the other hand, authentication overhead should be reduced for the more frequent requests for Registry data. The Registry API accommodates these varying needs.

The following calls bind a client to a Registry Server in preparation for Registry operations. The argument list of these calls enables an application to specify the authentication protocol, the protection level, and the authorization protocol to be used:

- **sec_rgy_site_bind( )**

  Binds to a specified site

- **sec_rgy_site_bind_update( )**

  Binds to any update site

- **sec_rgy_site_bind_query( )**

  Binds to any query site

- **sec_rgy_site_binding_get_info( )**

  Extracts the Registry site name and security information from the binding handle

The following calls are similar to the binding calls just described, except that an application cannot specify security information. By default, however, the following calls use DCE Shared-Secret Authentication, the packet-integrity level of protection, and DCE Authorization.

- **sec_rgy_site_open( )**

Binds to the specified site

- **sec_rgy_site_open_update( )**

  Binds to any update site

- **sec_rgy_site_open_query( )**

  Binds to any query site

- **sec_rgy_site_get( )**

  Gets the Registry site name from the binding handle

The following calls provide miscellaneous binding management functionality:

- **sec_rgy_site_close( )**

  Terminates binding to a Registry site

- **sec_rgy_site_is_readonly( )**

  Tests whether a bound site is an update or query site

44.2 The Registry Database

The Registry database comprises three container objects:

- **principal**

  Contains principal names; each name is associated with account information that is also specified here (for example, the name of the primary group).

- **group**

  Contains groups and the names of their member principals.

- **organization**

  Contains organizations and the names of their member principals.

These three objects are referred to as name domains, and each member of a domain is referred to as a PGO item. Principal items are contained in the principal domain, groups in the group domain, and organizations in the organization domain. A principal may have a name such as **/rd/writers/tom,** from which you might infer that **tom** is a member of the group **writers** and the organization **rd**. However, this is not the case because the name **/rd/writers/tom** only indicates that **tom** and the data corresponding to the account of this principal (if any) reside in **/rd/writers** in the principal domain. There may also be a group named **/rd/writers** in the group domain, but the principal **tom** is not a member unless he is explicitly named in the group **/rd/writers** in the group domain.

Each PGO item consists of a printstring name, a UUID, and a UNIX number (for compatibility with UNIX system security interfaces). For various administrative reasons, it is frequently convenient to be able to refer to a PGO item by more than one name. Consequently, some PGO items are aliases for other items. An alias uses the same UUID and UNIX number as the PGO item to which it refers, but contains only a pointer to that item.

The Registry also contains the **rgy** object, which describes Registry properties and policies, and organization policies.

44.2.1 Creating and Maintaining PGO Items

The PGO items in the Registry database are created and maintained with routines that are prefixed **sec_rgy_pgo...( )**. The contents of a PGO item vary with the domain. If the domain is **group** or **organization**, the contents are the membership list of principal names. If the domain is **principal**, the contents are the data corresponding to the Registry account using that name.

The **sec_rgy_pgo...( )** interface contains the following calls for maintaining the PGO trees:

- **sec_rgy_pgo_add( )**

  Adds a PGO item

- **sec_rgy_pgo_delete( )**

  Deletes a PGO item

- **sec_rgy_pgo_rename( )**

  Changes the name of a PGO item

- **sec_rgy_pgo_replace( )**

  Replaces information corresponding to the specified PGO item

The **sec_rgy_pgo...( )** interface contains the following calls for maintaining PGO membership lists:

- **sec_rgy_pgo_add_member( )**

  Adds a member to a group or organization membership list

- **sec_rgy_pgo_delete_member( )**

  Deletes a member from a group or organization membership list

- **sec_rgy_pgo_get_members( )**

  Returns a list of members of a group or organization

- **sec_rgy_pgo_is_member( )**

  Tests whether a principal is a member of a specified group or organization

The **sec_rgy_pgo...( )** interface contains the following calls for retrieving PGO item data:

- **sec_rgy_pgo_get_by_id( )**

  Returns the PGO item with the specified UUID

- **sec_rgy_pgo_get_by_name( )**

  Returns the PGO item with the specified name

- **sec_rgy_pgo_get_by_unix_num( )**

  Returns the PGO item with the specified UNIX number

- **sec_rgy_pgo_get_next( )**

  Returns the PGO item that follows the last PGO item returned

The **sec_rgy_pgo...( )** interface also contains routines that convert PGO item specifiers, as follows:

- **sec_rgy_pgo_id_to_name( )**
- **sec_rgy_pgo_id_to_unix_num( )**
- **sec_rgy_pgo_name_to_id( )**

- **sec_rgy_pgo_unix_num_to_id( )**
- **sec_rgy_pgo_name_to_unix_num( )**
- **sec_rgy_pgo_unix_num_to_name( )**

44.2.2 Creating and Maintaining Accounts

The login-name field of an account contains a principal name, a primary group name, and an organization name. The account may also contain a project list (also known as a concurrent group set) that specifies all the groups to which the principal corresponding to the account belongs, but the login-name field itself specifies only one group name.

An account can be added to the Registry database only when all of its constituent PGO items are established. For instance, to create an account with the principal name **tom,** the group name **writers**, and the organization name **rd**, all three names must exist as individual PGO items in the database; and the **writers** group and the **rd** organization must specify that **tom** is a member.

When an account is created with **sec_rgy_acct_add( )** (and if a project list is enabled for the new account), the call scans the groups in the Registry and creates a project list containing all the groups in which the principal name appears. Subsequently, the project list may be modified with the **sec_rgy_pgo_add_member( )** and **sec_rgy_pgo_delete_member( )** calls.

The following calls create and maintain accounts:

- **sec_rgy_acct_add( )**

  Adds an account to an existing principal item

- **sec_rgy_acct_delete( )**

  Deletes an account, leaving the principal item

- **sec_rgy_acct_rename( )**

  Changes an account login name, perhaps moving the account to a different principal item

The following calls return the information in an account:

- **sec_rgy_acct_get_projlist( )**

  Returns the project list for an account

- **sec_rgy_acct_lookup( )**

  Returns all the account data

The following calls modify the information in an account:

- **sec_rgy_acct_passwd( )**

  Changes an account password

- **sec_rgy_acct_replace_all( )**

  Replaces all of an account's data

- **sec_rgy_acct_admin_replace( )**

  Replaces only the administrative account data

- **sec_rgy_acct_user_replace( )**

  Replaces only the account data that is accessible to the user of the account

44.2.3 Registry Properties and Policies

The following subsections outline some Registry API parameters that affect the cell as a whole, and the routines that enable an application to retrieve and set values for them.

44.2.3.1 Registry Properties

Several Registry parameters and flags affect all accounts in the Registry. These Registry properties include the following:

- The version number of the Registry software used to create and read the Registry

- The name and UUID of the cell associated with the Registry, and whether the current Registry site is an update site or a query site
- Minimum and default lifetimes for certificates of identity issued to principals
- Bounds on the UNIX numbers used for principals, and whether the UUIDs of principals also contain embedded UNIX numbers

The routines associated with this parameter set are

- **sec_rgy_properties_get_info()**
- **sec_rgy_properties_set_info()**

44.2.3.2 The Registry Authentication Policy

Another set of parameters affecting all principals is the Registry authentication policy. This set only controls the maximum lifetime of certificates of identity, upon first issue and renewal. Accounts also have authentication policies, and the policy in effect for any principal is the most restrictive combination of the Registry policy and the policy for a principal's account. The associated routines are

- **sec_rgy_auth_plcy_get_info()**
- **sec_rgy_auth_plcy_get_effective()**
- **sec_rgy_auth_plcy_set_info()**

44.2.3.3 Organization Policies

Another parameter set controls the set of accounts of principals that are members of an organization. These parameters control the lifetime and length of passwords, as well as the set of characters from which passwords may be composed. This parameter set also specifies the default lifespan of accounts associated with the organization.

The routines associated with this parameter set are

- **sec_rgy_plcy_get_info( )**
- **sec_rgy_plcy_get_effective( )**
- **sec_rgy_plcy_set_info( )**

44.2.4 Miscellaneous Registry Routines

The Registry API includes a few miscellaneous routines, as follows:

- **sec_rgy_login_get_info( )**

  Returns login information for the specified account.

- **sec_rgy_login_get_effective( )**

  Applies local overrides (if such data is available) to Registry account information and returns information about which account information fields have been overridden.

- **sec_rgy_wait_until_consistent( )**

  Blocks until all previous database updates have been propagated to all sites. This is useful for applications that first bind and write to an update site, and then bind to an arbitrary query site and depend upon up-to-date information.

- **sec_rgy_cursor_reset( )**

  Resets the database cursor to return the first suitable entry.

Chapter 45

The Login Application Program Interface

The Login API communicates with the Security Server to establish, and possibly change, a principal's login context. A login context contains the information necessary for a principal to qualify for (although not necessarily be granted) access to network services and possibly local resources as well. Login context information normally includes the following:

- Identity information concerning the principal, including its certificate of identity (in Shared-Secret Authentication, this is the TGT); its PAC; and Registry policy information, such as the maximum lifetime of certificates of identity.

- The context state; that is, whether the Authentication Service has validated the context or not.

- The source of authentication information (it may originate from the network Authentication Service, or locally, if that network service is unavailable).

45.1 Establishing Login Contexts

The basic procedure by which a network login context is established (see Chapter 42 for more details) is as follows:

1. The client calls **sec_login_setup_identity**( ), specifying the name of the principal whose network identity is to be established. The Authentication Service creates an encrypted message containing a certificate of identity (the TGT) and returns it to the client's security runtime, which in turn passes a login context handle back to the client.

2. The client submits the context handle and the principal's password to **sec_login_valid_and_cert_ident**( ), which attempts to decrypt the TGT. If the attempt succeeds, the client's security runtime forwards the ticket to the Privilege Service, which creates a PAC for the principal and encloses it in a Privilege-Ticket Granting Ticket (PTGT), which is returned to the client's security runtime. The runtime decrypts the message containing the PTGT and returns information about the source of the authentication information to the API (if the authentication information comes from the network Security Server, then the login context is validated).

3. Finally, the client invokes **sec_login_set_context**( ), which enables child processes spawned from the calling process to inherit the validated context.

In the walk-through of user authentication in Chapter 42, we mentioned that one of the functions of **sec_login_valid_and_cert_ident**( ) is to demonstrate that a valid trust path exists between the Authentication Service and the host computer on which the principal is logging in. After setting up and validating a login context, any application that sets identity information for local processes should check to be sure that the server that provided the certificate of identity is legitimate in order to demonstrate that the trust path between the client and the Authentication Service is valid.

45.1.1 Validating the Login Context and Certifying the Security Server

Whereas a validated login context is one that is regarded as legitimate by the local security runtime, a validated and certified login context is one that is not only regarded as legitimate, but also can be demonstrated to have been (in all likelihood, that is) issued by a legitimate Security Server. Certifying that the Security Server is legitimate prevents faked identity information from being propagated to local processes. For example, a spurious server could collaborate with a dishonest user in order to obtain an identity that conferred comprehensive permissions (for example, the **root** identity). With such an identity, the dishonest user could gain access to sensitive local objects, such as key-storage files for server principals that run on the host. (Servers running on other hosts would not trust this principal, however, because it does not know their keys.) Of course, if a spurious server can return to the application a ticket encrypted with the host's secret key, it means the server has access to the host's key; but if this is the case, network security has already been seriously undermined.

When an application needs to certify the originator of a certificate of identity, it may call **sec_login_certify_identity( )**. This routine makes an authenticated remote procedure call to the local **sec_clientd** daemon in order to acquire a ticket to the host principal. If **sec_clientd** succeeds in decrypting the message containing the ticket, then the server that granted the certificate of identity must know the host principal's secret key; this evidence indicates that it is a legitimate Security Server. Since **sec_clientd** runs with the identity **root** (in order to access the host's key), the process calling **sec_login_certify_identity( )** need not.

The **sec_login_valid_and_cert_ident( )** is similar to **sec_login_certify_identity( )**, except that it combines the validation and certification procedures (and therefore, the password of the principal that is logging in must be known to the process making this call). The **sec_login_valid_and_cert_ident( )** routine calls the Security Server for a ticket to the host and attempts decryption. The process calling **sec_login_valid_and_cert_ident( )** must have access to the host's secret key, and so must run as **root**.

Note: Because system login programs should not set local identities derived from an uncertified context, all Login API routines that return data from an uncertified context issue a warning.

45.1.2 Validating the Login Context Without Certifying the Security Server

An application that does not use login contexts to set local identity information does not need to certify its login contexts. Since an illegitimate Security Server is unlikely to know the key of a remote server principal with which the application may communicate, the application will simply be refused the service requested from the remote server principal. If local operating system identity information is assumed to be neither of interest nor of concern to an application, it may call **sec_login_validate_identity( )**, which does not attempt to verify the Security Server's knowledge of the host principal's key.

The **sec_login_validate_identity**( ) routine does not acquire a PTGT, unlike the **sec_login_certify_identity**( ) and **sec_login_valid_and_cert_ident**( ) routines. Instead, the PTGT is acquired when the application first makes an authenticated remote procedure call.

45.1.3 Example of a System Login Program

Following is an example of a system login program that obtains a login context that can be trusted for both network and local operations.

Note: One of the function calls that appears in the following example, **sec_login_purge_context**( ), is described in Section 45.6.

```
if (sec_login_setup_identity(principal,sec_login_no_flags,
    &login_context,&st))
{
    ...get password...

    if (sec_login_valid_and_cert_ident(login_context, password,
        &reset_passwd, &auth_src&st))
    {
        if(auth_src==sec_login_auth_src_network)
        {
            if (GOOD_STATUS(&st)
                sec_login_set_context(login_context);
```

```
         }
      }
      if (reset_passwd)
      {
         ...reset the user's password...

         if (passwd_reset_fails)
         {
            sec_login_purge_context(login_context)

            ...application login-failure actions...
         }

         ...application-specific login-valid actions...
      }
}
```

45.2 Context Inheritance

A process inherits the login context of its parent process unless the child process is associated with a principal that has logged in and so established a separate login context. The following subsections describe two additional aspects of context inheritance:

- How the initial context is established.
- How a process may inhibit context inheritance.

45.2.1 The Initial Context

An application invokes **sec_login_setup_identity**( ) so that it can then make other authenticated RPC calls. However, **sec_login_setup_identity**( ) is itself a local interface to an authenticated remote procedure call, and authenticated RPC needs a validated login context in order to execute. For applications like system login, the daemon **sec_clientd** supplies the validated context. However, a daemon that is started before **sec_clientd** is running on the host, needs to be able to assume its host's identity. The initial context is established at boot time with **sec_login_init_first**( ), which establishes the default context inheritance for processes running on the host. The routines **sec_login_setup_first**( ) and **sec_login_validate_first**( ) then set up and validate the context in a procedure like that used for user context validation.

45.2.2 Private Contexts

A process may inhibit context inheritance by setting a flag in **sec_login_setup_identity**( ). If the flag indicates that the login context is private, then children of the calling process cannot inherit it. A child process can neither set a private context (since it is the function of **sec_login_set_context**( ) to make the context inheritable) nor export it to any other process (see Section 45.4 for more information on this subject).

45.3 Handling Expired Certificates of Identity

For a dishonest principal to make use of an intercepted certificate of identity, it must succeed in decrypting it. In order to make the task of decryption more difficult, a certificate of identity has a limited lifespan; and once it expires, the associated login context is no longer valid.

Because this security feature may inconvenience users, an application may wish to warn a user when the certificate of identity is about to expire. The **sec_login_get_expiration**( ) routine returns the expiration date of a certificate of identity. When a certificate of identity is about to expire, the application may call **sec_login_refresh_identity**( ), which may be used to refresh any login context.

Similarly, a server principal may need to determine whether a certificate of identity may expire during some long network operation, and if the certificate of identity is likely to expire, refresh it to ensure that the operation is not prevented from completion. Following is an example:

```
sec_login_get_expiration (login_context,&expire_time,&st);

if (expire_time < (current_time + operation_duration))
```

```
{
   if (!sec_login_refresh_identity(login_context,&st))
   {
      ...identity has changed and must be validated again...
   }
   else
   {
      ...login context cannot be renewed...

      exit(0);
   }
}

operation();
```

Because **sec_login_refresh_identity**( ) acquires a certificate of identity, refreshed contexts must be revalidated with **sec_login_validate_identity**( ) or **sec_login_valid_and_cert**( ) before they can be used.

The expiration date of a login context has no meaning with respect to local identity information; for the same reason, **sec_login_refresh_identity**( ) cannot refresh a login context that has been authenticated locally.

45.4 Importing and Exporting Contexts

Under some circumstances, an application may need two processes to run using the same login context. A process may acquire its login context in a form suitable for imparting to another process by calling **sec_login_export_context**( ). This call collects the login context from the local context cache and loads it into a buffer. Another process may then call **sec_login_import_context**( ) to unpack the buffer and create its own login context cache to store the imported context. Since the context has already been validated, the process that imports it may use it immediately. (The CDS Clerk is an example of a context importer.)

These operations are strictly local: the exporting and importing processes must be running on the same host. In addition, a process cannot export a private context.

45.5 Changing a Groupset

The **sec_login_newgroups**( ) routine enables a principal to assume the minimum groupset that is required to accomplish a given task. For example, a user may have privilege attributes that include membership in an administrative group associated with a comprehensive permission set, and membership in a user group associated with a more restricted permission set. Such a user may not want the permissions associated with the administrative group, except when those permissions are essential to an administrative task (so as to avoid inadvertant damage to objects that are accessible to members of the administrative group, but not to members of the user group).

To offer users the capability of removing groups from their groupsets, an application may use the Login API as shown in the following example.

Note: Two of the function calls that appear in the following cxamplc, **sec_login_get_current_context**( ) and **sec_login_inquire_net_info**( ), are described in Section 45.6.

```
sec_login_get_current_context(&login_context,&st);

sec_login_inquire_net_info(login_context,&net_info,&st);

for (i=0; i < num_groups; i++)
{
   ...query whether the user wants to discard any of the current
   group memberships. Copy new group set to the new_groups array...
}

if (!sec_login_newgroups(login_context,sec_login_no_flags,
    num_new_groups, new_groups, &restricted_context,&st))
{
   if (st == sec_login_s_groupset_invalid)

      printf("Newgroupsetinvalid\n");

   ...application-specific error handling...
}
```

Note that the **sec_login_newgroups**( ) call can only return a restricted groupset: it cannot return a groupset larger than the one associated with the login context that is passed to it. This routine also enables the calling process to flag the new login context as private to the calling process.

45.6 Miscellaneous Login API Functions

The following subsections describe a few miscellaneous Login API routines, some of which have appeared previously in examples in this chapter.

45.6.1 Getting the Current Context

The **sec_login_get_current_context**( ) routine returns a handle to the login context for the currently established principal. This routine is useful for several Login API functions that take a login context handle as input.

45.6.2 Getting Information from a Login Context

The **sec_login_inquire_net_info**( ) routine returns a data structure comprising the principal's PAC, account expiration date, password expiration date, and identity expiration date. The **sec_login_free_net_info**( ) frees the memory allocated to this data structure.

45.6.3 Getting Password and Group Information for Local Process Identities

Two calls, **sec_login_get_pwent**( ) and **sec_login_get_groups**( ), are useful for setting the local identity of a process. These routines return password or group information from the network Registry, if that service is available, or from the local files of password and group information, if the network service is unavailable.

45.6.4 Releasing and Purging a Context

When a process is finished using a login context, it may call **sec_login_release_context**( ) to free storage occupied by the context handle. When a process releases a login context, the context is still available to other processes that use it. If an application needs to destroy a login context, it may call **sec_login_purge_context**( ), which also frees storage occupied by the handle. Since a destroyed context is unavailable to all processes that use it, application developers should be careful when using **sec_login_purge_context**( ).

Chapter 46

The Key Management Application Program Interface

Every principal has an entry in the Registry database that specifies a secret key. In the case of an interactive principal (that is, a user), the secret key is derived from the principal's password. Just as users need to keep their passwords secure by memorizing them (rather than writing them down, for example), a noninteractive principal also needs to be able to store and retrieve its secret key in a secure manner. The Key Management API provides simple key management functions for noninteractive principals.

While the key management routines themselves are relatively secure, it is up to the application to ensure the security of the file or other device used to store the key. By default, server principals that run on the same computer share a local key file; however, the Key Management API also allows principals to specify an alternative local file.

When users change their passwords, they are free to forget their old passwords. When a noninteractive principal changes its secret key, however, there may be clients with valid tickets to that principal that are encoded with the old key. To save clients the trouble of having to request new tickets to a noninteractive principal when the principal's key has changed, every key is flagged with a version number, and old key versions are retained until all tickets that could have been encoded with that key have expired.

Finally, if a noninteractive principal's key has been compromised, it may be invalidated (along with all the corresponding tickets held by any clients) by simply deleting it from the local key storage.

Note: The Key Management API is for use only by applications using the DCE Shared-Secret Authentication protocol and the key-type DES (Data Encryption Standard).

Part 1 of this guide contains additional information, in the context of writing a distributed application, of the Key Management API.

46.1 Retrieving a Key

The Key Management API provides two functions for retrieving a key from the local key storage. The **sec_key_mgmt_get_key( )** function returns a specified key version for a specified principal. The meaning of specifying version 0 (zero) in this routine may vary depending on the authentication protocol in effect (if the protocol is DCE Shared-Secret, the value 0 for the version identifier means the version that was most recently added to the local storage). In any case, a principal's login is almost always successful if the principal uses the version 0 key.

When there are valid tickets that are encoded with different key versions, an application may need to retrieve more than one key version. In that case, the application may call **sec_key_mgmt_initialize_cursor( )** to set a cursor in the local storage to the first suitable entry corresponding to the named principal and key type, and then call **sec_key_mgmt_get_next_key( )** to get all versions of that key in storage. The application may then call **sec_key_mgmt_release_cursor( )**, which disposes of information associated with the cursor. Neither of the key-retrieval routines can return keys that have been explicitly deleted, or that have been ''garbage collected'' after expiring.

The two key-retrieval functions dynamically allocate the memory for the returned key(s). To enable the efficient allocation of memory, an application may call **sec_key_mgmt_free_key( ),** which frees the memory occupied by the key and returns it to the allocation pool.

46.2 Changing a Key

The **sec_key_mgmt_change_key()** function communicates with the Registry to change the principal's key to a specified string, and also places the new string in the local key storage. The *keydata* input argument for this call may be a new key that the application specifies, or a random key returned by the **sec_key_mgmt_gen_rand_key()** routine. An application may call **sec_key_mgmt_get_next_kvno()** to determine the next key version number that should be assigned to the new key, so that it may reference this key version when retrieving a key.

In some circumstances, a principal may need to change its key in the local key storage, but not immediately update the Registry database. For example, a database application may maintain replicas of a master database that are managed by servers running on different computers. If these servers all provide exactly the same service, it makes sense for them to share the same key (meaning that they share the same principal identity). This way, a user with a ticket to the principal can be directed to whichever server is least busy.

When the Registry database obtains a new key for a principal, the Authentication Service can immediately begin issuing tickets to the principal that are encoded under the new key. However, suppose the master for a single-principal replicated service were to call **sec_key_mgmt_change_key()**, and a client presented a ticket encoded with the latest key to a replica that had not yet learned that key. In this case, the replica would refuse service, even though the ticket was valid. Therefore, if an application employs replicated servers that are also instances of a single principal identity, the application should:

1. Generate a new key by calling **sec_key_mgmt_gen_rand_key()**. This routine simply returns a key to the calling process, without updating the Registry or local storage.

2. Disseminate the new key to all replicas.

3. Cause the replicas to call **sec_key_mgmt_set_key()**. This call updates the local storage to the new key, but does not update the Registry database entry for the principal (the key version specified in this routine must not be 0 (zero).) The replicas should notify the master when they have completed setting their local stores to the new key.

4. Cause the master to call **sec_key_mgmt_change_key**( ) (here again, the key version must not be 0) after all replicas have set the new key locally, thereby updating both the master's local storage and the Registry database entry.

Of course, if the master and each replica has its own principal identity, each server may call **sec_key_mgmt_change_key**( ) without coordinating this activity with any others.

46.3 Automatic Key Management

It is sometimes convenient for a principal to be able to change its key on a schedule determined by the password expiration policy for that principal, rather than to rely on a network administrator to decide when this should be done. In this case, the application may call **sec_key_mgmt_manage_key**( ). This function invokes **sec_key_mgmt_gen_rand_key**( ) shortly before the current key is due to expire, updates both the local key storage and the Registry database entry with the new key, and then calls **sec_key_mgmt_garbage_collect**( ) to discard any obsolete keys. This function runs indefinitely; it will never return during normal operation and so should be invoked from a thread dedicated to key management. It is not intended for use by server principals that share the same key.

46.4 Deleting Expired Keys

In order to prevent service interruptions, the Key Management API does not immediately discard keys that have been replaced; instead, it maintains the keys, with a version number and key-type identifier, in the local key storage. However, after a key has been out of use for longer than the maximum life of a ticket to the principal, it is no longer possible that any client of that principal has a valid ticket encoded with that key. At this time, the key storage may have its ''garbage'' collected.

The **sec_key_mgmt_garbage_collect()** routine collects garbage in the local key storage by deleting all keys older than the maximum ticket lifetime for the cell. The *garbage_collect_time* argument, which is returned by **sec_key_mgmt_change_key()**, specifies when key-storage garbage is to be collected.

46.5 Deleting a Compromised Key

When a principal's key has been compromised, it should be deleted as soon as the damage has been discovered in order to prevent another party from masquerading as that principal. Two routines delete a principal's key:

- The **sec_key_mgmt_delete_key()** routine removes all key types having the specified key version identifier from the local key storage, thus invalidating all extant tickets encoded with that key.
- The **sec_key_mgmt_delete_key_type()** routine removes only a specified version of a specified key type.

If the compromised key is the current one, the application should first change the key with **sec_key_mgmt_change_key()**. It is not an error for a process to delete the current key as long as it is done after the login context has been established, but it may inconvenience legitimate clients of a service. The inconvenience may be justified, however, if the application data is sensitive.

Since an application may have no means to discover that its key has been compromised, the **rgy_edit** tool provides interfaces that call **sec_key_mgmt_delete_key()**, **sec_key_mgmt_change_key()**, and **sec_key_mgmt_gen_rand_key()** so that a network administrator, who is more likely to detect that a key has been compromised, may handle a security breach of this kind. As an alternative, the application may provide user interfaces to these routines.

Chapter 47

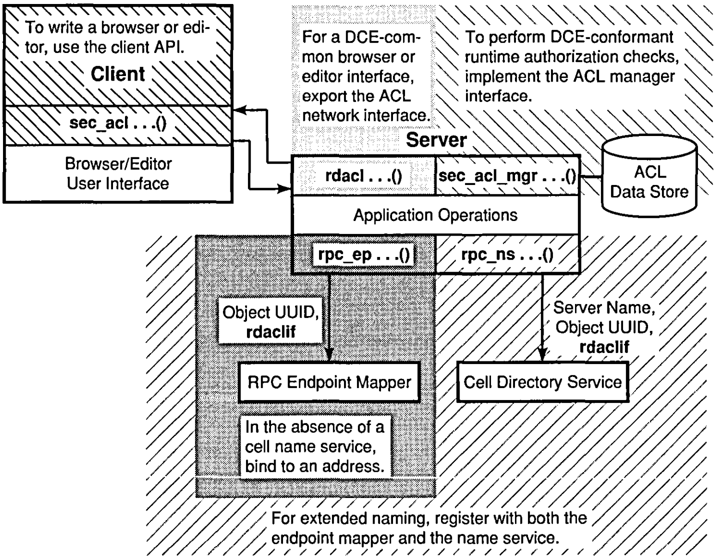
The Access Control List Application Program Interfaces

As a rule, DCE Security program interfaces are local client-side APIs only. The ACL facility includes this kind of interface, and some others as well, as follows:

- The DCE ACL interface, **sec_acl...()**, which enables clients to browse or edit DCE ACLs.
- The DCE ACL manager interface, **sec_acl_mgr...()**, which enables servers to perform DCE-conformant authorization checks at runtime.
- The DCE ACL network interface, **rdacl...()**, which enables servers that manage access control (such as **sec_acl_mgr**-based ACL managers) to communicate with **sec_acl**-based clients.

Figure 47-1 provides a schematic view of the relationships and usage of these interfaces, as well as some relevant RPC interfaces. This chapter first discusses the client API, and then the two server program interfaces.

Figure 47–1. ACL Program Interfaces


To write a browser or editor, use the client API.
Client
sec_acl...()
Browser/Editor User Interface
For a DCE-common browser or editor interface, export the ACL network interface.
To perform DCE-conformant runtime authorization checks, implement the ACL manager interface.
Server
rdacl...()
sec_acl_mgr...()
ACL Data Store
Application Operations
rpc_ep...()
rpc_ns...()
Object UUID, rdaclif
RPC Endpoint Mapper
In the absence of a cell name service, bind to an address.
Server Name, Object UUID, rdaclif
Cell Directory Service
For extended naming, register with both the endpoint mapper and the name service.

47.1 The Client-Side API

The client-side API is a local interface consisting of a set of routines that are prefixed **sec_acl**. This is the interface from which the default DCE ACL editor (**acl_edit**) is built. An application that needs to replace **acl_edit** with a DCE ACL editor or browser of its own calls this interface and binds to **libdce.a** The following subsections provide specific information on the functionality that this API supports.

47.1.1 Binding to an ACL

Any operation performed on an ACL uses an ACL handle to identify the target of the operation. The handle is bound to the object protected by the ACL, not to the ACL itself. Since an object may be protected by more than one ACL manager type, the ACL itself can only be uniquely identified by the ACL handle in combination with the manager type that manages it. ACL editing calls must also specify the ACL type to be read or otherwise manipulated (the object, default container, or default object ACL types).

An application may call **sec_acl_bind**( ) to get an ACL handle. The handle itself is opaque to the calling program, which needs none of the information encoded in it to use the ACL interface. A program can obtain a list ACL manager types protecting an object with the **sec_acl_get_manager_types**( ) call, which returns a list of UUIDs corresponding to these manager types; and pass this data, along with the ACL type identifier, to another client-side routine. (In the absence of a Cell Directory Service, an application may call **sec_acl_bind_to_addr**( ); this call binds to a network address rather than a cell namespace entry.)

Once an application is finished using an ACL handle, it may call **sec_acl_release_handle**( ) to dispose of it.

47.1.2 ACL Editors and Browsers

After obtaining a handle to the object in question (and using **sec_acl_get_manager_types**( ) to determine the ACL manager types protecting the object), editors and browsers use the **sec_acl_lookup**( ) function to return a copy of an object's ACL. Once an object's ACL is loaded in memory, the editor can call **sec_acl_get_printstring**( ) to receive instructions about how to display the permissions of the ACL in a human-readable form. This call returns a symbol or word for each permission, as well as common combinations of permissions. In addition, the printstring structure includes a short explanation of each permission.

An ACL cannot be modified in part. To change an ACL, an editor must read the entire ACL (the **sec_acl_t** structure), modify it, and replace it entirely by calling **sec_acl_replace**( ).

An ACL can occupy a substantial amount of memory. The memory management routine, **sec_acl_release( )**, frees the memory occupied by an ACL, and returns it to the pool. This is implemented strictly as a local operation.

47.1.3 Testing Access

Access testing by clients is not definitive because the state of an ACL can change between the access test and the request to the server to perform an application operation. More typically, a client simply requests an operation; then, upon receiving the request, the server performs the access test, and depending on the result, either executes the client's request or returns an error to the client. However, if an application server acts as a client of another server that manages ACLs for the application objects, the application server needs the results of access tests from the ACL manager server in order to process requests from application clients.

After calling **sec_acl_bind( )** to acquire an ACL handle to the target object, such an application server would call **sec_acl_test_access( )** with the returned handle, the UUID of the ACL manager, and the permission(s) requested in order to perform the requested operation. The access-test function returns TRUE if the object's ACL allows the client to perform the operation; otherwise, it returns FALSE. An alternative to **sec_acl_test_access( )**, **sec_acl_get_access( )** is useful for implementing operations like the conventional UNIX system access function.

Some applications need to check an ACL on behalf of a principal other than the one represented by the calling process. For example, a replicated database server would presumably need to check the privilege attributes of its clients against the database ACL entries. In this case, the server would use the **sec_acl_test_access_on_behalf( )** function, which is identical to the **sec_acl_test_access( )** function, except that it also requires the PAC of the principal for which the server principal is acting as an agent. This function checks both the privilege attributes of the principal represented by the calling process and those encoded in the PAC. It returns TRUE if the most restrictive combination of the two permission sets grants the requested permission(s).

47.1.4 Errors

Although the ACL API saves errors received from the DCE RPC runtime (or other APIs) in ACL handle data, it returns an error describing the ACL operation that failed as a result of the RPC error. However, if an error occurs and the client needs to know the cause of the ACL operation failure, it may call **sec_acl_get_error_info( )**. This routine returns the error code last stored in the handle.

47.2 The Server-Side API

The server-side API consists of a set of routines that are prefixed **sec_acl_mgr**. This is the interface from which the default DCE ACL managers are built. It would be used by any application that accesses persistent storage of access control information in order to make runtime authorization decisions that are DCE-conformant (in terms of the representation of identities, the access-check algorithm, and so on). This is a local interface that is supplied in source code form (interfaces to these routines are described in the *OSF DCE Application Development Reference*). This interface can be tailored as necessary for integration into application code.

A second program interface is the ACL network interface (or ''wire'' protocol). This interface consists of a set of routines that are prefixed **rdacl**. This is a remote interface that enables any server program that manages access control information (preferably, though not necessarily, DCE ACL managers) to communicate with **sec_acl**-based clients.

47.2.1 The ACL Manager Interface

Following is a summary of ACL manager routines:

- **sec_acl_mgr_configure( )**

  Creates an ACL database and returns a handle to it

- **sec_acl_mgr_is_authorized( )**

  Takes a principal's PAC and the requested permission set and returns TRUE if the permission set is granted

- **sec_acl_mgr_get_access( )**

  Returns a principal's permissions to an object (useful for implementing operations like the conventional UNIX system access function)

- **sec_acl_mgr_replace( )**

  Replaces the specified ACL

- **sec_acl_mgr_lookup( )**

  Returns a copy of the object's ACL

- **sec_acl_mgr_get_manager_types( )**

  Returns a list of manager types protecting the object

- **sec_acl_mgr_get_printstring( )**

  Returns human-readable representations of permissions

47.2.1.1 A Sample ACL Manager

Following is sample application server code that tests a client's access to an object that the application protects:

```
application_op(handle_th, ...)
{
   ...
   rpc_binding_inq_auth_client(h,&PAC, &server_name,
                                    &protect_level, &authn_svc,
                                    &authz_svc, &st);

   rpc_inq_object(h, &object, &st);

   if (authentication_levels_are_appropriate(server_name,
                                                  protect_level,
                                                  authn_svc,
                                                  authz_svc)
      && sec_acl_mgr_is_authorized(sec_acl_mgr_handle_t,
                                   sec_acl_permset_t, &PAC,
                                   (sec_acl_key_t)&object,
                                   sec_acl_type_object, NULL,
                                   NULL, &st))
   {
      ...Application code to perform operation
   }
   else
   {
      ...Perform appropriate application logging etc.
   }
}
```

For more information about writing ACL managers, refer to Part 1 of this guide.

47.2.1.2 Extended Naming of Protected Objects

The DCE ACL model supports extended naming, which enables ACL managers to protect separately objects that are not registered in the cell namespace. For example, suppose an application manages different kinds of printers. The application may register only printer types, such as **laser** and **line**, with the Cell Directory Service. Among the laser printers is a high-resolution printer that is available only to members of the group **writers**, and low-resolution laser printers that anyone may use. When the Cell Directory Service receives a name such as **/laser/high-resolution**, it passes the residual part of the name (**high-resolution**) to the appropriate ACL manager, which resolves the residual and makes a determination as to whether the principal requesting to print on the high-resolution laser printer may do so.

To take advantage of extended naming, an ACL manager must register the server name, object UUID, and **rdaclif.idl** interface with the Cell Directory Service (refer to the CDS chapters in Part 4A of this guide for more information). In addition, the ACL manager must register the object UUID and **rdaclif.idl** interface with the RPC endpoint mapper (refer to the RPC chapters in Part 3 of this guide).

47.2.2 The ACL Network Interface

The ACL network interface, **rdacl...( )**, provides a DCE-common interface to ACL managers. It is the interface exported by the default DCE ACL managers to the default DCE ACL client (that is, the **acl_edit** tool), and any other **sec_acl-**based client.

The client API, **sec_acl...( )**, is a local interface that calls a client-side implementation of the ACL network interface. However, application developers are responsible for implementing the server side of this interface. The implementation needs to conform the reference pages in *OSF DCE Application Development Reference* that describe the **rdacl...( )** routines; following is a summary of them:

- **rdacl_lookup( )**

  Retrieves a copy of the object's ACL.

- **rdacl_replace( )**

  Replaces the specified ACL.

- **rdacl_get_access( )**

  Returns a principal's permissions to an object (useful for implementing operations like the conventional UNIX system access function).

- **rdacl_test_access( )**

  Determines whether the calling principal has the requested permission(s).

- **rdacl_test_access_on_behalf( )**

  Determines whether the principal represented by the calling principal has the requested permission(s). This function returns TRUE if both the principal and the calling principal acting as its agent have the requested permission(s).

- **rdacl_get_manager_types( )**

  Returns a list of manager types protecting the object.

- **rdacl_get_printstring( )**

  Obtains human-readable representations of permissions.

- **rdacl_get_referral( )**

  Returns a referral to an ACL update site. This function enables a client that attempts to modify an ACL at a read-only site to recover from the error and rebind to an update site.

Chapter 48

The ID Map Application Program Interface

In the multicell environment, the global print string representation of a principal identity can be ambiguous, even though every principal and its native cell have unique names in the form of UUIDs to which the print string representations normally resolve. For example, all ACLs maintain UUIDs as the definitive representations of principal and cell names. The **acl_edit** tool, on the other hand, takes as input (and also outputs) this same information as print strings. This string-to-UUID mapping is accomplished easily enough when an ACL entry refers to a local identity; that is, a member of the local cell. However, when a user adds an ACL entry for a foreign principal identity such as **/. . . /world/dce/rd/writers/tom**, it is not evident to the ACL Manager which part of the name identifies the cell, and which identifies the principal within the cell. The name **/. . . /world/dce** may refer to a cell containing the principal **/rd/writers/tom**, or the cell name may be **/. . . /world/dce/rd** and the principal name, **/writers/tom**.

To parse the fully qualified principal name that the user types into its cell name and local principal-name components, and for these components to be mapped to UUIDs, ACL Managers that support entries for foreign identities use the ID Map API. For the same reasons, many other kinds of servers in a DCE multicell environment need a facility to parse global names and translate UUIDs into print string names.

The ID Map API provides a simple interface to translate a fully qualified name (that is, the global representation of a name) into its components and back again. This API consists of the following calls:

- The **sec_id_parse_name()** call takes as input a context handle and a fully qualified principal name, and returns the principal's print string name and UUID, and the print string name and UUID of the principal's native cell.
- The **sec_id_gen_name()** call translates a principal UUID and the UUID of its native cell UUID into a cell-relative principal name, a cell name, and a fully qualified principal name.
- The **sec_id_parse_group()** call is like **sec_id_parse_name()**, except that it operates on group names.
- The **sec_id_gen_group()** call is like **sec_id_gen_name()**, except that it operates on group names.

Part 7

DCE Distributed File Service

Chapter 49

DCE Distributed File Service Overview

This chapter describes the architecture of the DCE Distributed File Service (DFS), a high-performance distributed file system that provides transparent local and remote file access, and also provides background information for writing DFS applications. DFS is designed to maximize reliability and interoperability with other file systems. When accessing remote data, DFS uses DCE Remote Procedure Calls (RPCs) to communicate with participating systems, to exchange access requests, authentication information, and file and directory data, and to synchronize information.

DFS also includes a high-performance, log-based local file system, the DCE Local File System (DCE LFS). The DCE LFS offers capabilities not generally available in conventional UNIX File Systems (UFSs), such as support for logical groups of files (filesets) within a disk partition, and support for DCE Access Control Lists (ACLs). The DCE LFS maintains a log (on disk) of all actions that affect file and directory metadata, such as file creation and modification dates, file sizes, and directory entries. Maintaining a log of file system operations provides a fast, robust mechanism for recovering from computer system problems that do not actually involve damage to the physical storage media.

This chapter and the ones that follow it assume that you are familiar with the *Introduction to OSF DCE.*

49.1 Writing DFS Applications

The capabilities of the DFS are used in writing applications that depend heavily on fileset manipulation, file server process management, and client management. Such applications include backup systems, mail and bulletin board programs, and graphical fileset management tools for system administrators. In addition, customers may wish to replace stock applications with customized versions; this is made possible by the programming interface provided.

49.1.1 Related DCE Components

Because DFS is built on top of other DCE components, a complete understanding of the other components is necessary for an understanding of DFS. The information in this section is intended only as an overview of the other components; it is assumed that you have read about and understand the following DCE components:

- DCE Remote Procedure Calls
- DCE Security Service, especially how to use and interpret Access Control Lists (ACLs)
- DCE Directory Service, especially details about the namespace
- DCE Distributed Time Service, especially client and server machine synchronization
- DCE Threads

For more information about these components, consult the appropriate parts of this guide and the *OSF DCE Application Development Reference*.

49.1.1.1 DCE Remote Procedure Calls

DCE Remote Procedure Calls are central to the DFS client/server model. All communications between server machines and client machines happen through RPCs. DFS functions generally take handles as arguments, which direct the functions to use the correct RPC interfaces. An application programmer will need to know how to obtain these handles, and should have a thorough understanding of the use of RPCs in general.

49.1.1.2 DCE Security Service

The DCE Security Service is composed of several different components that are used in conjunction with DFS: the Authentication Service, the Privilege Service, the Access Control List component, and the Registry Service.

The DCE Authentication Service component performs several security functions that interact with DFS. It ensures that only certified users can log in and use the system, and it ensures that only authorized machines can access the system.

The DCE Privilege Service component ensures that those people using the system have the correct access rights to perform the operations they request.

The DCE Access Control List component provides an interface that allows you to set different levels of protection on file system objects such as directories and files. The different ACLs interact with the UNIX file system protection mode bits. You can grant permissions to individuals, or you can define groups of users and grant permissions to the groups. For specific information on ACLs and file system objects, see the *OSF DCE Administration Guide* and the **afs_syscall( )** description in this guide and in the *OSF DCE Application Development Reference*.

The DCE Registry Service maintains a Registry Database. This database contains information similar to that stored in UNIX password files, such as user, group, and account information. An account defines who can log in to the system and includes information about passwords and home directories.

The DCE Security Service operates through principals. A principal is a representation of a user or process that can read and/or alter user or system data. Principals have keys associated with them, which are, roughly, passwords (see Part 6 of this guide); a process must know a principal's key to exercise the principal's access rights. Typically, there is a one-to-one mapping between people and principals. DFS server processes on a single machine all share a single principal, while some other DCE components assign separate principals to each server process.

49.1.1.3 DCE Directory Service

The DCE Directory Service provides a consistent way to identify and locate users and resources, including files and directories, anywhere in a networked computing environment. The Directory Service has three main components: the Cell Directory Service, the Global Directory Service, and a Global Directory Agent (a gateway between the local and the global naming environments).

The Cell Directory Service (CDS) manages names within a cell. The Global Directory Service (GDS) supports the global naming environment between cells and outside of cells. The Global Directory Agent (GDA) makes cell interoperability possible by allowing CDS to participate in the global naming environment.

DFS uses the Directory Service to locate fileset, backup, and other servers.

49.1.1.4 DCE Distributed Time Service

The DCE Distributed Time Service (DTS) provides precise clock synchronization for system clocks in a network. It is used to keep DFS client and server machines synchronized.

DTS is important for communications between client machines using the Cache Manager and server machines running the File Exporter and other server processes. Clients must remain in contact with server machines whose tokens they hold to ensure that they have the most recent copies of cached data. Clients and servers must refer to a common time standard for communications to remain constant and data to remain current.

DTS is also important for replicated Fileset Location Databases (FLDBs), which must be coordinated on different server machines. Like clients and servers, machines housing replicated databases must remain in constant contact to ensure that each server has the current copy of the database.

Failure to maintain synchronization can result in unnecessary disruption of database access.

49.1.1.5 DCE Threads

DCE Threads provides a parallel processing-like environment. DFS uses DCE Threads, and DFS programmers need to understand DCE Threads to use any of the DFS application programming interfaces, except the **pioctl( )** and **afs_syscall( )** interfaces. Applications are required to coexist with DCE Threads. It is possible that nonserver applications can be written without DCE Threads if the applications do not make remote procedure calls.

49.1.2 The DFS Application Programming Interface

The DFS Application Programming Interface (API) consists of the following parts:

- Cache Manager (**pioctl( ), ioctl( ), afs_syscall( )**): Provides calls to manipulate system information, client cache information, ACLs, and related items
- General Fileset Functions (**VC_...( )**): Manipulates filesets and the Fileset Location Database at a high level
- Fileset Location Server (**VL...( )**): Provides functions to manipulate Fileset Location Database entries
- Fileset Server (**FTSERVER...( )**): Provides functions to manipulate filesets on servers
- BOS Server (**BOSSVR...( )**): Provides functions to manipulate processes on File Server machines

Some parts of the DFS API can be called by any user, but those functions that involve altering fileset, FLDB, process, or Cache Manager information are restricted to users with the appropriate privileges, usually a DFS administrator or root on the local machine. More information about these authorization issues is given in later sections that decribe these functions. In general, if the application is going to perform these operations, it must be installed with the appropriate privileges.

The rest of this chapter describes the DFS architecture in detail. While an interface is not provided for each component, some level of understanding of the various pieces and how they fit together greatly assists in understanding DFS. The chapters following this one describe the Cache Manager, the functions for manipulating filesets, and the Basic OverSeer Server.

49.2 Overview of the DCE Distributed File Service Architecture

Each node, or system, in a DFS installation is either a server machine, running a DFS server process or maintaining local file systems on disk and making them available to other nodes (exporting them), a client machine, running applications that access files that are exported by File Server machines, or both. A client node can act as a server if it exports a local file system. An exported local file system can either be a DCE LFS or a conventional UNIX File System (UFS).

To expedite file system response, DFS clients maintain a local cache of recently requested file and directory information. To synchronize locally cached file and directory information with requests made by other systems for that same information, DFS uses a token-based synchronization system. When a client requests file or directory information from a File Server machine, the File Exporter returns that information and grants an access token that describes the remote system's capabilities with respect to that file or directory. Subsequent requests made by other clients for conflicting operations on that same information are not granted until the first token can be revoked. On systems with a local hard disk, cache information is stored on disk. On diskless client machines, cache information is stored in system memory.

DFS uses an enhanced Virtual File System interface (called VFS+) to ensure consistency between requests for remote files (from File Server machines) and requests for local files present in exported physical file systems on the client machine. To guarantee consistency, all exported DFS file systems types share the same access mechanism. The VFS+ interface guarantees that local requests for files located on the local file system generate the same types of access tokens as requests for these files that originate from a remote system.

The three main components of DFS are the Virtual File System, the Cache Manager, and the File Exporter. These components have the following responsibilities:

- The enhanced Virtual File System (VFS+) component supports both local and remote file access. Part of the server VFS+ interface is the concept of DCE LFS filesets, which are logical groups of files present in single disk partitions.

- The Cache Manager runs on client machines, and maintains a copy of data that has been obtained from a server in the ''recent'' past. If the file or information requested is already present in this cache, no remote call is necessary. If the file or information requested by the application is not already present in the local cache, the Cache Manager makes calls to the File Exporter on the appropriate File Server machine to retrieve that information, and stores it. How long information is kept in the local cache depends on the size of the cache and the amount of data retrieved.

- The File Exporter makes any exported VFS that is physically located on the File Server machine available to client machines. This process also synchronizes requests for access to files and directories to prevent conflicting access to files and directories by clients. (These clients may include the machine on which the files and directories reside, if a client is also acting as a server.) The synchronization mechanism is called the Token Manager. Access rights are granted through tokens that summarize the current read, write, and access capabilities of a client with respect to each cached file or directory.

These primary components are supported by three other modules: the Fileset Location Server, the Fileset Server, and the Replication Server. These components have the following responsibilities:

- The Fileset Location Server, which maintains the Fileset Location Database (FLDB), records the location of all available filesets and the File Server machines associated with those filesets. Fileset location lookups are necessary when translating new references to different logical file systems when, for example, file system ''mount points'' (see the glossary in the *Introduction to the OSF DCE*) are encountered.

- The Fileset Server performs operations that involve entire filesets. These include making entire filesets available to DFS requests, producing incremental dumps of filesets, and moving filesets from one File Server machine to another to aid in load and resource balancing.

- The Replication Server makes it possible to dynamically create read-only replicas of an active fileset.

49.3 Component Overview

DFS provides a transparent distributed file system environment to both users and applications. However, developing system-level DFS applications requires an understanding of the purpose of each of the DFS modules and their interactions. The following subsections discuss each DFS module in detail, examining the functions provided by each module, its role in the DFS architecture, and the interactions with the various services required and provided by DFS. Not all of these components are visible via the API.

49.3.1 The DCE Local File System

DCE LFS is a fast-restarting UNIX file system that integrates the capabilities needed for a large-scale distributed system with a sophisticated recovery mechanism, providing performance equal to or better than most existing physical file systems. DCE LFS offers several capabilities not generally available in UNIX file systems, such as support for logical groups of files (filesets) within a disk partition, support for ACLs, and the ability to recover quickly from system failures.

DCE LFS is based on UNIX disk partitions and is integrated into the kernel. DCE LFS is designed to take advantage of a multithreaded environment and asynchronous I/O. DCE LFS can be accessed both as a local file system when individual filesets are mounted and as a remote file system exported from File Server machines. To provide uniform local and remote access, DCE LFS implements a compatible extension of the standard VFS functions.

The next subsections describe the DCE LFS capabilities that represent advances over older physical file systems. These include the fileset and aggregate concepts, the log-based file system, and support for ACLs.

These subsections also introduce some implementation issues, such as how DCE LFS and conventional UNIX File Systems are integrated, and the anode abstraction, which is comparable to the inode structure in UNIX File Systems.

49.3.1.1 Filesets and Aggregates

In many UNIX environments, a file system is associated with a partition, which is a disk or logical portion of a disk that can be mounted, addressed, and administered as a single unit. DCE LFS provides an additional logical level of organization by introducing filesets, which are mountable subtrees within a standard disk or partition. Filesets can be administered and referenced individually. To avoid confusion with conventional UNIX terminology, DFS uses the term ‘‘aggregate’’ (see the glossary in the *Introduction to OSF DCE*) to refer to a unit of disk storage (equivalent to a UNIX partition) because it can be logically composed of multiple filesets. The use of the term aggregate also provides a reminder that partitions composed of filesets enable DFS-specific administrative operations such as moving and cloning, which are not possible (or meaningful) on UNIX file systems.

The distinction between mountable logical filesets and physical partitions allows DCE LFS filesets to be moved among partitions on a single server or from one server to another. This provides the system administrator with a mechanism for balancing the system load across File Server machines by redistributing frequently accessed filesets among the available file servers. DCE LFS supports dynamic fileset motion, which does not require taking down any servers or prohibiting the use of any standard facilities when moving filesets. During such a move, the fileset itself is temporarily unavailable. This absence is fairly transparent to application programs, which are temporarily blocked from accessing the files and applications in that fileset.

In addition to the standard interfaces required of any VFS, DCE LFS provides higher-level fileset and aggregate interfaces. Although each VFS is a mountable fileset, these fileset interfaces are separate from the VFS interface. This ensures that fileset operations such as moving, cloning, and replicating can be performed on filesets that are not mounted.

49.3.1.2 The Log-Based File System

Conceptually, the data available in filesets and aggregates can be divided into two general types: user data and metadata. User data is the data in a fileset, such as applications and data files, that is created and referenced by users of the system.

Metadata is the data in any part of a file system that is used to describe and organize the files and directories in that fileset or aggregate. Metadata is logged in DCE LFS, but user data is not.

Tracking the actions of a program or system is generally referred to as keeping a log of that program or system. DCE LFS is a log-based file system because all changes to metadata in a given aggregate are recorded in the log for that aggregate. To provide a method for organizing the changes recorded in these log records, associated changes to metadata are grouped together into ''atomic transactions.'' The term ''transaction'' indicates that all of the individual changes in that group are related to a larger logical operation. The term ''atomic'' means that no single change that is a part of a transaction takes effect unless all the changes associated with that transaction are performed. The log entry for each change records the old and new values for all changed metadata and the identity of the transaction of which the change is part.

A transaction has ''committed'' once all of the log records associated with that transaction are written to the log on disk. A separate log entry notes when a transaction commits. If a system failure occurs, the file system recovery procedure replays the log, completing transactions that have been recorded as committed, and undoing any effects of transactions that did not commit. Log-based file systems are frequently referred to as ''recoverable'' file systems because, even with system problems, any file system changes that have been logged are easily recovered by simply replaying the operations recorded in the log on disk.

49.3.1.2.1 DFS Logging Versus Conventional Logging

The log maintained by DCE LFS records the most recent changes to file system metadata in a given aggregate. In conventional UNIX file systems, system failures that interrupt certain operations can leave the file system in an inconsistent state, making it potentially unsafe to use and requiring the use of the **fsck** command after reboot to check for and repair any damage. Maintaining a record of the changes made to metadata enables the DCE LFS to resume normal operation as quickly as possible after a system failure; it eliminates the need to call **fsck**, unless the medium that stores the file system is physically damaged. After a system crash, DCE LFS applies recovery techniques that either undo operations that have begun but not finished, or complete operations that have finished but have not yet been written to the file system. The time spent in recovery is proportional to the size of the active portion of the log instead of the size of the file system.

In conventional UNIX file systems, file system operations that can introduce file system inconsistencies must be written to disk as soon as possible after they have been performed. Although these writes are generally performed asynchronously, they generate so much disk traffic that they can affect the overall performance of the system. In addition, subsequent updates to a disk block already queued for one of these asynchronous writes generally wait until the first write operation completes, also impairing system performance.

In comparison, a log-based file system does not need to force modified metadata to the disk in order to guarantee consistency in the event of a crash. The file system is always consistent, even if slightly out-of-date, after the log is replayed. The file system may periodically commit all pending transactions, but fidelity to the spirit of the UNIX file system requires this to occur only every 30 seconds. For this reason, the DCE LFS log usually maintains log records for changes to metadata within only the last 30 to 45 seconds of file system activity.

All pending transactions are automatically written to the log whenever a **sync()** or **fsync()** system call is executed. These ‘‘batch’’ commits have very low overhead because of the sequential organization of the log, and because disks are especially efficient at performing these types of writes. This means that the DCE LFS actually generates considerably fewer disk updates than a UNIX file system, especially when performing operations that primarily change file system metadata, such as file creation, deletion, and truncation.

49.3.1.2.2 How Logging Works

Each aggregate has its own log, which is simply another portion of a physical disk. The size of the log for each aggregate is fixed when that aggregate is initialized. When the amount of data held in the log approaches the limits of the space allocated to the log for that aggregate, the processing of new file system requests is halted to allow pending transactions to commit. File system changes are then flushed to disk, ensuring that the log space previously required for records of those operations can be freed.

DCE LFS only logs file system metadata, such as ACLs, directory entries, UNIX protection modes, and file and directory status information. It does not keep a log of the changes to the user data in a partition. Changes to metadata can be logged quickly in small log records. The overhead introduced by writing these small log records does not affect the performance of the file system. On the other hand, maintaining a log of changes to user data would require larger, longer transactions that could easily require gigabytes of storage for log records.

To optimize the efficiency of the log and to minimize its size, some restrictions are placed on the scope and type of operations that can occur during a single transaction. DCE LFS restricts transactions to a maximum of one VFS interface call. This is especially important when a large number of log records are pending and their size nears the physical limits of the log. In this case, new file system operations are temporarily blocked to allow pending operations to complete, reducing the number of records that must be retained in the log. If transactions were allowed to span VFS interface calls, all pending transactions could end up waiting for additional high-level VFS operations to complete. In this case, none of these operations could complete successfully, no log space could be freed, and all subsequent system operations would be blocked. Bounding the scope of a transaction eliminates this possibility, guaranteeing smaller, more easily completed transactions.

Similarly, long file system operations are broken into sequences of short-lived transactions, each of which leaves the file system in a consistent state. For example, a single operation that truncates a file may be logged as a number of smaller transactions, each of which truncates only a few blocks at a time. This guarantees that transactions are short-lived, minimizing the size of the log without requiring complex algorithms for log truncation, and identifying the parts of the log that it is ‘‘safe’’ to reuse.

The logging system is also integrated into the management of the disk cache. File system functions must not directly modify data held in the disk cache because all file system functions must use the logging primitives in order to be recoverable. To guarantee this, most higher-level functions simply release cached data when done, leaving the logging system with the responsibility for writing that data back to the file system or File Server machine. The log mechanism keeps a record of the most recent log entry for each unit of locally cached data. The cached data cannot be written back to a file system until all relevant records are logged.

49.3.1.3 Access Control Lists

DCE ACLs are an enhancement to UNIX file protection and access control mode bits. ACLs are not supported by conventional UNIX file systems, although some vendors modify their file systems to support specific ACL implementations. ACLs expand the control that the owner of a file has over granting or denying file and directory access to specific users or groups of users. ACLs allow a finer granularity of access control for specific files and directories than do UNIX protection bits.

The DCE LFS supports both file and directory ACLs. A description of the access control mechanisms provided by ACLs, and the use of ACLs when developing applications, are presented in the ACL section of Chapter 50 of this guide and in the security sections of the *OSF DCE Administration Guide* and Part 6 of this guide.

49.3.1.4 Anodes

For an applications programmer, one of the most important abstractions provided by a file is its open-endedness. Programs can write to files without having to directly allocate the storage associated with the file. In conventional UNIX file systems, this open-endedness is implemented at the level of the inode. Other file-oriented abstractions, such as authorization (mode bits and ACLs), status information (access and update times), position in the directory hierarchy, and reference (link) counts are also associated with UNIX inodes.

Bundling all of these abstractions into a primary file system structure can be inconvenient to the kernel programmer. When these high-level abstractions are irrelevant, the additional overhead of having them can discourage taking advantage of a file's open-endedness. This encourages either abandoning open-endedness in favor of fixed limits, or reimplementing a primary form of open-ended resource allocation.

The DFS's anode abstraction provides a convenient descriptor for referencing an open-ended address space of storage, and nothing more. All DFS objects that require disk storage (for example, files, ACLs, filesets, aggregates, transaction logs, and low-level disk allocation bitmaps) are implemented as anodes. Files are implemented as anodes with additional information including a set of status bytes, a pointer to an ACL, and a position in the directory hierarchy.

Because both user and metadata are referenced through anodes, disk utilities can access the disk uniformly, regardless of the level of file system access that is required. This simplifies the construction of utilities, such as the logging system and disk recovery mechanisms, which might otherwise have to distinguish between anode and nonanode disk areas.

49.3.1.5 DCE LFS and UNIX File System Differences

In DCE, UNIX file systems can be accessed from remote machines if they are exported to the DFS filespace. The only practical way to export a UFS to the DCE environment is to export it as a unit at the lowest level for which a parallel DFS organization exists. This means that a UFS partition is exported as a single fileset. Once exported, this partition is referred to as an aggregate that contains at most one fileset. The Virtual File System interface (VFS+) provides a consistent mechanism for accessing the UFS files either via DFS or by local processes running on the machine exporting the UFS.

Not all fileset operations are supported for these non-LFS filesets because DCE LFS functionality is a superset of the functionality provided by standard UFSs. DCE LFS aggregates can accommodate multiple DCE LFS filesets; UNIX disk partitions can store only a single fileset. Because of this, DCE LFS filesets are usually smaller than non-LFS filesets and easier to manage. Lone fileset operations, such as dump and restore, can be supported for non-LFS filesets. The cloning, move, and replication fileset operations are not supported for non-LFS filesets because these require the capability of accommodating multiple filesets in an aggregate.

49.3.2 The Virtual File System Interface (VFS+)

The VFS+ interface used in DFS is an enhancement of the Virtual File System interface found in most UNIX kernels, or the Generic File System (GFS) found in Ultrix. The VFS+ interface extends the VFS interface in several ways, adapting it to provide the operational requirements of a distributed computing environment.

The VFS+ interface adds the following enhancements to standard VFS interfaces:

- Generalized Credentials: The VFS+ interface implements an open credential mechanism that allows the use of a variety of authentication mechanisms. The new credential can therefore carry both conventional UNIX authentication information and any other form of authentication information required, such as authentication tickets.
- Synchronization: A synchronization package is incorporated into the VFS+ interface operations, implementing a mechanism for 2-way communications between servers and their clients. When an object in any type of VFS is changed, the Token Manager notifies all interested parties.
- Fileset/Aggregate Interface: The VFS+ interface supports operations on DFS filesets and aggregates, including creating new filesets within an aggregate, putting those filesets online, taking existing filesets offline, iterating through the files in a fileset, and enumerating existing filesets.

The enhancements implemented by the VFS+ interface ensure compatibility among local and remote processes simultaneously accessing files. For example, a VFS+ server can make guarantees about when an exported file changes while, at the same time, clients attempt to modify these files. All of these types of access are synchronized through the VFS+ interface.

The VFS+ interface does not require that changes be made to external servers or software in order to access standard VFS or vnode-level functions. To eliminate potential problems with non-DCE software and servers, the standard VFS vnode-level functions are redefined to invoke the synchronization package that keeps track of the guarantees made by the various servers. These updated function definitions are implemented by writing ‘‘wrapper’’ functions for existing virtual file system operations. These wrapper functions perform the appropriate synchronization calls before and after the original VFS call.

When using the VFS+ interface, calling a VFS function initially executes an internal locking routine that provides a consistent interface to all VFS types. The call declares the operation to be performed and also identifies the file IDs involved in that operation. As part of this function, other virtual file systems may be called upon to relinquish conflicting access guarantees (such as tokens) that they may already have issued for the same files. Once the required access guarantees are obtained, the original virtual file system operation is performed. When that operation is complete, the locks are released. Releasing these locks does not directly return the access guarantees required by the VFS operation, but simply notifies the local synchronization package that the access guarantees for this operation may now be reclaimed whenever necessary.

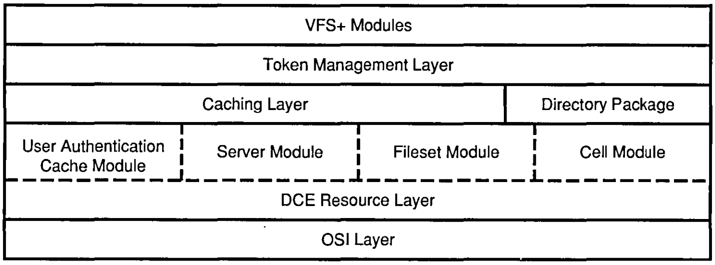
49.3.3 The Cache Manager

The DFS Cache Manager is a kernel-resident part of DFS that is responsible for the local caching of file and directory data on machines used as file system clients. Caching means that once any portion of a file is requested and retrieved by a client, a copy of that data is kept on the client machine. If the same part of the file is requested again, the locally cached copy is immediately available; it is not necessary to re-retrieve the data. The tokens associated with the data the first time it was retrieved ensure that the file has not changed on the server since it was first cached. If the file has changed, a new copy is retrieved from the File Server machine. Caching provides a substantial performance improvement over using remote operations to fetch file and directory information each time it is required.

The client’s Cache Manager presents a VFS+ interface to the UNIX kernel. Logically, the Cache Manager sits between the File Exporter, from which it receives file services, and the kernel, to which it provides files. Neither the kernel nor the processes that use the kernel to perform file operations need to know the physical location of the files referenced. The VFS+ interface ensures that all files are obtained through the same mechanism, regardless of whether their source was a conventional UNIX file system or a DFS file system. If the target fileset/file for a particular operation is local, avoiding an RPC to the local File Exporter results in increased system performance. This is handled by consulting the local fileset registry to see if this fileset belongs to the local disk before executing any vnode-layer procedure. If the fileset is local, the local operation can be executed directly. If the fileset is not local, the standard Cache Manager VFS+ call is executed.

The Cache Manager is divided into several layers, as shown in Figure 49-1 .The lowest layer is the Operating System Independent (OSI) layer, which is responsible for isolating all low-level operating system calls such as basic I/O and network, physical file system, and virtual memory calls. This layer helps isolate higher-level Cache Manager functions from vendor-specific or system-specific issues.

Figure 49–1. The Organization of the DFS Cache Manager



The next layer is the DCE Resource Layer, which is responsible for maintaining RPC connections, authentication, fileset location information, and other information about the state of the DCE environment. The resource layer is divided into several modules:

- A User Authentication Cache Module maintains a per-user list of tickets and additional per-user credential information.

- A Server Module maintains a set of per-server structures for tracking the status of recently contacted servers.

- A Fileset Module maintains a list of accessed filesets, their mounted positions in the global file system tree, and their physical locations on one or more File Server machines.

- A Cell Module maintains a list of the administrative DCE cells that have been accessed by the particular Cache Manager.

Above the resource layer is the Caching Layer itself. This layer maintains the set of cached file information, separated into status and data components. The basic operations performed at the caching layer are fetching and storing the data and status information associated with files. A client uses the directory package to keep its cached directories synchronized with the server's originals. When an RPC call to a file server causes the contents of a remote directory to be updated, the Cache Manager's directory package can update the cached directory. This avoids having to refetch directory information each time a directory is updated.

The highest levels of the DFS Cache Manager are the VFS modules, which implement the functions exported to the VFS+ interface, and therefore offer the functions required by the kernel in order to treat DCE DFS as a true external file system. This layer also contains the VFS functions that provide the OS-specific code needed to support different VFS/kernel interfaces. This isolates the Cache Manager from OS-related differences in virtual file systems in the same way the Cache Manager's OSI layer isolates the Cache Manager from OS differences in lower kernel layers.

49.3.4 The File Exporter

The File Exporter is the kernel-resident part of a DFS File Server machine that evaluates client requests and sends file and directory information in response to those requests. The File Exporter actually consists of several components:

- Token Manager
- Host Module
- VFS+ Interface
- Fileset Registry
- Server Procedure

The Token Manager, called by the File Exporter, maintains the set of file and directory tokens that have been granted to existing clients of that File

Server machine. Among these tokens is a binary compatibility relation telling which other tokens can be simultaneously granted to other clients. The Token Manager is discussed in more detail in the next section.

The File Exporter's Host Module associates a pair of structures with every client request. One structure describes the state of the client Cache Manager that made the call and contains such information as the delivery status of any token revocation messages issued to the client. The other structure contains authentication information about the user who made the file system request on the client machine. This includes the user's authentication system identity and group memberships.

The File Exporter's VFS+ Interface interacts with the Cache Manager, File Exporter, and DCE LFS to access the data stored at the server.

The File Exporter's Fileset Registry is a table that enumerates the filesets residing locally on the server. It is the ''glue'' that sits between the File Exporter and the Cache Manager and UFS. Given a fileset name or ID, the fileset registry must be able to identify and locate the mount-level directory of that fileset. The fileset registry fills two important functions in the processing of an incoming DFS request. First, it tells the File Exporter whether a fileset is local, which determines whether the request can be filled by that File Exporter. Second, it maps the fileset specified in the request into a VFS identifier, so that the appropriate file system anode can be located for a given request.

Finally, the File Exporter's Server Procedure implements the RPC interface in terms of calls to the previous components.

Processing a request for file data or status information starts with the host module, which keeps track of the information about each client that is needed by the file server. To expedite client requests, the host module maintains a cache of authenticated users on that File Server machine and the RPC connections they have previously used to contact the server.

The information maintained by the host module for each user is associated with a credential structure before the VFS file system layer is called. This allows the File Exporter to invoke various types of virtual file systems while still allowing each file system to use its own authentication information.

Once a request is associated with a host module client structure, the File Exporter checks with the fileset registry for the fileset referenced in the request, and passes the file IDs associated with the request to the Token Manager. This gives all other DFS File Exporters that may be holding tokens for the fileset an opportunity to revoke the appropriate tokens or other forms of promises they may have made to external clients. At this time, the Token Manager locks the fileset IDs for the duration of the server call. Other File Exporters can issue callbacks or revoke tokens at this stage of handling the call.

After the relevant file IDs are locked, the File Exporter converts the incoming file IDs into vnode structures in a local fileset by calling the appropriate VFS layer function. At this time, the actual VFS operation is invoked to perform the desired function.

49.3.4.1 The File Exporter's Token Manager

In order to guarantee that file or directory information in use by a client machine is not simultaneously being modified, each DCE File Server kernel includes a Token Manager to guarantee file access synchronization. The Token Manager keeps track of the clients that are referencing files located on that File Server machine. A Token Manager is local to each File Server machine, and maintains tokens for, and monitors access to, only files and directories on that particular File Server machine. For example, the File Exporter for a given File Server machine may grant a read token for a specific file, allowing a client to read the contents of that file until otherwise notified. The File Exporter records that the client has received a guarantee, and does not allow any other client to write data to the file without first revoking that guarantee by notifying the client that its cached data must no longer be used.

When a File Exporter obtains any tokens on behalf of a client, it registers a procedure to be called in the event that the token later has to be revoked. Calls registered for later communications with other processes are referred to as ``callbacks.'' When an incompatible token must be revoked, the Token Manager executes that callback to the client to revoke the token.

The Token Manager is invoked by all calls through the VFS+ interface, and ensures that any access incompatible with privileges already granted through existing tokens are revoked before the operation continues. The Token Manager is invoked by the VFS+ layer because both non-DFS File Exporters and locally executed system calls can perform various operations on local physical file systems that must be synchronized with the guarantees exported by the DFS File Exporter.

Using the Token Manager to control physical file and directory access rights suggests expanding the term ''client'' from simply being a remote user of files exported by a file server, to being any system (local or remote) that requests tokens on a file. This is a more flexible and appropriate definition of a client, because the potential clients of a Token Manager can range from the local UNIX kernel to any number of remote File Exporters.

49.3.4.2 Types of Tokens

DFS File Exporters support a number of different types of tokens. These different token types reflect the different types of access to files and directories that are required in a distributed computing environment. Because tokens reflect the type of file or directory access requested by a client, some types of tokens are incompatible. Before granting a new token, the Token Manager may have to revoke some existing tokens. The following types of tokens can be granted by a File Server machine's File Exporter:

- Data Tokens: Data Tokens grant the client the right to access a range of bytes in a file. Both read and write data tokens are available. A read data token allows the client to cache and use a copy of the relevant file data without repeatedly performing RPCs to the appropriate file server (either for validating the data or for re-reading it). A write data token allows the client to update the data in a cached copy of the file without storing the data back to the server or notifying the server.

- Status Tokens: Status Tokens allow the client to access the status information associated with a file or directory. Both read and write status tokens are available. A read status token allows the client to refer to cached copies of the status information without calling the server to check the status. A write status token allows the client to update its cached copy of the file's status without notifying the server. The Token Manager blocks other VFS+ functions from any access to status information for the cached file or directory while the client has its write token.

- Lock Tokens: Lock Tokens allow the client to set a lock on a particular range of bytes within a file. Both read and write lock tokens are available. With a lock token, the client is assured that the server does not attempt to set conflicting locks on the file without first revoking the token. If a client does not hold a lock token, it must get one before being able to operate on the data. Lock tokens behave in the same way as open tokens with respect to revocation; see the following.

- Open Tokens: Open Tokens grant the holder the right to open a file. Different types of open tokens are available, corresponding to different possible open modes: normal reading, normal writing, executing, shared reading (same as executing), and exclusive writing.

Different types of tokens can be held simultaneously for the same file because they refer to separate components of the file. Table 49-1 describes the compatibility matrix of open tokens. Tokens of the same type may also be incompatible with each other, as in the following examples:

- Read and write data tokens are incompatible if the byte ranges associated with those tokens overlap, but otherwise are compatible.
- All write tokens are incompatible with all others.
- Read and write lock tokens are incompatible if the byte ranges associated with those tokens overlap, but otherwise are compatible.

Table 49–1. Compatibility Between Open Tokens

Access	**Normal Reading**	**Normal Writing**	**Shared Reading**	**Shared Writing**
Open for normal reading	C	C	C	I
Open for normal writing	C	C	I	I
Open for shared reading or open for executing	C	I	C	I
Open for exclusive writing	I	I	I	I
C = Compatible I = Incompatible				

DFS tokens are managed by the VFS+ wrapper functions, as described previously. To summarize, the VFS+ wrapper functions modify all standard VFS functions to first call the Token Manager, obtaining the appropriate tokens for the operations they will perform before actually performing those operations. The functions that compose this part of the VFS+ interface are frequently referred to as the ‘‘glue layer.’’

When the Token Manager wishes to revoke a token, the File Exporter notifies the client that holds the token. If the token was granted for

- Reading (status or data), the client only has to return it.
- Writing, the client must write back any status or data that it modifies before returning the token.
- Locking or opening, the client cannot return the token if the file is still open, or if the corresponding lock is still held. This is the normal action if the client has already locked or opened the file.

For remote clients, token management requires two-way RPC communications. Clients must call File Server machines to access files and obtain tokens, and File Exporters must call clients to revoke tokens. Token revocation requests for file systems that are local to the client making the requests do not require RPC communications. When handling requests to local file systems, the Token Manager and VFS+ interface communicate directly, except for the use of the Host Module.

49.3.5 The Fileset Server

Filesets are the basic storage and administrative unit for data in the DFS. The abstraction provided by filesets is similar to that provided by UNIX disk partitions, although DCE LFS filesets are actually implemented as logical subsets of disk partitions (aggregates). The entire contents of a DCE LFS fileset must physically be located within a single aggregate, and its size is determined by a per-fileset quota. Non-LFS filesets are equal in size to the exported partition. Keeping filesets small compared to aggregates simplifies load balancing. Load balancing involves moving filesets from one aggregate to another when the first aggregate becomes nearly full or when equalizing fileset access across various File Server machines. It is easier to find room on other aggregates for a fileset if it is relatively small. DCE LFS filesets can be moved between aggregates on the same File Server machine, or to an aggregate on another server. The files stored together in a fileset form an entire subtree of the file system, which can be separately mounted and administered. Filesets are mounted so that they create the illusion of a seamless hierarchy of files, even though that hierarchy may actually be distributed across multiple File Server machines. Storing filesets on multiple File Server machines, or dynamically moving filesets between different machines, does not affect the transparency of user access.

A fileset header is associated with each fileset, and must be stored on the same aggregate as the fileset it describes. The fileset header contains the fileset name, fileset ID number (not guaranteed to be unique), unique identifier (which combined with the fileset ID produces a unique identification), type, and status. The fileset header also contains the anode indexes of all of the files and directories present in that fileset. The fileset label must be stored on the same aggregate as the fileset.

The Fileset Server allows system administrators to create, delete, duplicate (clone), move, back up, or restore entire filesets with a single operation. Each of these operations is carried out as a single operation that requires an exclusive lock on a specified fileset. The administration of DCE LFS filesets is covered in detail in the *OSF DCE Administration Guide*.

Creating, deleting, and moving filesets are standard administrative functions that are required to effectively manage any subset of a file system. Cloning allows an administrator to dynamically produce an inexpensive read-only copy of a DCE LFS fileset, recording its exact state at the time of the copy. Uses of fileset clones include serving as online backups from which users can retrieve a former version of a file they accidently change or delete, or as fileset replicas, which can be distributed to multiple File Server machines. Replication increases the availability of the contents of a fileset, and can reduce the overhead associated with accessing filesets that contain frequently used system binaries and files. When copies of these files are available at multiple locations, the requests for these files are distributed across the File Server machines on which these files are available. An incidental benefit of replicated filesets is that the File Exporter does not need to revoke tokens because (by definition) files in a read-only clone cannot change. Replication and cloning require features of DCE LFS and thus are not supported for non-LFS filesets.

It is important to understand the difference between cloning and replication. Cloning means to create a read-only copy of an existing fileset on the same aggregate. Replication, which uses the cloning operation as a primitive, means making an exact copy of a fileset, including all of the data blocks associated with the files and directories in that fileset. This copy may or may not be located on the same aggregate or File Server machine as the parent fileset.

Filesets can be cloned only to the aggregate on which the original fileset is located because the cloning operation initially copies only the anode indexes for the files on the specified fileset, rather than copying the files themselves. Once these are copied, the original indexes are updated to point to the clone's copied indexes, and a bit is set in each parent anode to indicate that this indirection is occurring. Subsequent write operations on parts of the fileset first check the status of the indirection bit for the anode associated with a file. If this bit is set, the original data must be copied to other blocks, the indirection must be undone, and the indirection bit must be cleared before the write actually occurs. This ensures that cloning a fileset initially involves copying the minimum amount of information. Initially, the parent fileset consists of a skeleton fileset that simply points to the blocks in the clone. As more writes occur in a cloned fileset, the parent fileset comes to own an increasing number of ''new'' blocks. Only the blocks that change are copied; a small change to a large file does not result in the entire file being copied.

Dumping and restoring files are standard administrative functions necessary for any computer system. Dumping a fileset refers to the process of copying a fileset to a data stream. The target of this data stream is usually some archival device, such as a tape drive. Restoring refers to the process of converting a data stream back into proper fileset format. The ability to dump and restore logical portions of disk storage is necessary to create backups for long-term storage on tape. Backups are done both as a precaution against loss of data due to hardware failure and as a way of protecting users against the accidental deletion of their own data. Backups are also routinely performed when deleting a fileset from the file system. Dumping and restoring can also be used when moving filesets between machines that store or represent filesets in different formats.

49.3.6 The Fileset Location Database and Server

The Fileset Location Database (FLDB) is a cell-wide replicated database that maps fileset names to the servers on which they are actually located. The FLDB is accessed only via a collection of Fileset Location Server processes, one at each machine on which the FLDB is replicated. These RPC server processes provide calls to examine and change information about the filesets located in the cell.

An individual fileset location entry, an FLDB entry, contains the fileset's name, its type, the File Server machines where the fileset is located, the numeric identifier of the aggregate on the specified File Server machine on which the fileset is located, any status flags specific to that file server location, any flags associated with the fileset, and the numeric identifiers of this and any associated filesets such as backups and read-only copies.

For more detailed information about legal fileset types and the contents of fileset location entries, see the *OSF DCE Administration Guide*.

49.3.7 The Replication Server

Note: Replication only operates on DCE LFS filesets; it cannot be applied to the UNIX File System.

The organization of DCE LFS filesets into aggregates provides a number of features not found on ordinary UNIX file systems. First, the concept of DCE LFS filesets as logical subsets of aggregates means that filesets are not tied to a physical location on the storage media and can therefore be manipulated independently. This allows filesets to be moved among partitions or moved from one server to another.

Similarly, a read-only copy of a fileset (a clone) can be created within the same aggregate where the original fileset is located. Fileset cloning can be used as an intermediate backup mechanism or as a part of the replication process. The administrative procedure of cloning a fileset and copying the clone(s) is called read/only replication of that fileset. Cloned copies of filesets cannot exist outside the aggregate on which the parent fileset is located, while replicas of filesets can be located on any file server machine. This is similar to the distinction between the **cp** (copy) and **ln** (link) commands in UNIX type operating systems; that is, files can be physically copied across partitions because the **cp** command creates a copy of both file and user data at the destination of the copy, but files can never be hardlinked across partitions. Similarly, filesets cannot be cloned outside the aggregate that holds their parent fileset because they directly share storage..

The DFS replication servers provide for both scheduled replication and release replication of filesets. In scheduled replication, a fileset can be copied and updated at specified intervals by the replication server. A replica is therefore maintained permanently and is guaranteed to be out-of-date by no more than an interval specified by the system administrator. For practical reasons, the existing implimentation is unable to keep up with short intervals of time (less than 10 minutes). However, system administrators can replicate a fileset at any time explicitly; this is called release replication.

Any client of a replicated fileset is guaranteed to always see a consistent snapshot of the fileset and is guaranteed that the data in the replica is never replaced by older data. When a replication server must update the replica, it obtains only those files that have changed during the replication interval from the master fileset.

Both cloning and replication are useful tools for system administration because the basic DFS backup unit is the fileset and not the aggregate. A system administrator can back up a fileset by first cloning it and then copying the clone to removable media whenever convenient. Cloning is also an efficient backup mechanism because of the small initial size of the cloned fileset, and because it initially copies only the fileset's metadata. Cloned filesets can continue to exist on disk indefinitely, during which time files can be directly restored from the cloned fileset. This can eliminate, in some cases, the most time-consuming part of restoring files from backups, which is the time required to mount and search removable backup media.

49.3.8 The BOS Server

The BOS Server (Basic OverSeer Server) runs on every DFS server machine and monitors the other DFS server processes on the machine. It restarts processes automatically if they fail, in the correct order. The BOS Server also provides an interface through which processes can be started, stopped, and monitored, and through which binaries for software can be maintained.

The programming interface to the BOS Server allows the creation, deletion, and modification of processes. Processes are managed by the use of bnodes, which keep track of process parameters, start times, and frequency of execution.

49.4 An Example of DFS File Access Synchronization

This section provides an example of the interaction between the various DFS support modules, showing the roles of the Token Manager and File Exporters in synchronizing access to a file. The file in this example is stored in a conventional UNIX file system, and is being written by both a local user (Userl), who is issuing both read and write system calls, and a remote user (User2), who is attempting to access the same file through a client Cache Manager.

Initially, User2's remote application issues a write system call to the file. This operation is handled by the client's Cache Manager, which requires a token guaranteeing that it is permitted to update the cached copy of the file locally. This token is requested from the File Exporter of the File Server machine where the master copy of the file is located. The File Exporter registers the client with the File Server machine's Token Manager as having a data write token. Once User2 receives the data write token, the client can handle all remote writes to the cached copy of the file without contacting the File Server machine further.

At some point Userl, through a process local to the File Server machine and therefore accessing the file locally (not through a DFS File Exporter), decides to read some data from the master copy of the file. Before reading the master copy, the VFS+ interface calls the local Token Manager, requesting a read data token for the file. Because there is a conflicting write data token already granted to User2 by the File Exporter, the read token for Userl cannot be granted immediately; User2's incompatible token must be revoked before Userl's local process can be granted its own token.

At this point, the Token Manager attempts to resolve the access conflict by requesting that the File Exporter revoke the conflicting token. As part of the revocation procedure, the File Exporter makes an RPC back to User2's client Cache Manager, asking it to return the write token. Before the client returns the write token, it also stores its modifications back to the fileset on the File Server machine in a separate RPC. Once this occurs, the File Exporter returns from the revocation call made by the Token Manager. The new read data token can be granted to the VFS+ interface call. Once the new token is granted, the VFS+ interface can then proceed with Userl's read operation by calling the original virtual file system's read data function.

The read data token can be returned at any time after the read data call has completed, although it is usually held as long as possible to avoid unnecessary RPCs. Remote clients always hold on to tokens as long as they can to avoid unnecessary RPCs. If User2 issues another call to write the specified data, the client system must contact the File Server machine again to get another guarantee, and the File Exporter must call the local Token Manager again to obtain another write data token. Thus, the VFS+ layer provides a consistent file system image regardless of the dispersion of clients making reference to any one piece of it.

Chapter 50

General Cache Manager Operations

This chapter describes the operations you can perform with the DFS Cache Manager. The Cache Manager is the process on the client machine that maintains information about filesets that have already been fetched from the server. For more information about the general operation of the Cache Manager, see the introductory chapter of this guide, or the *OSF DCE Administration Guide*. For more information about the functions described in this chapter, see the *OSF DCE Application Development Reference*.

The Cache Manager is the part of DFS that is responsible for the local caching of file and directory data on file system clients. Once a file or directory is requested by a client, the data is kept (cached) until either it is flushed or the cache storage is recycled. DFS provides functions to access cached data, manipulate related fileset information, and control the behavior of the Cache Manager. These function calls are described in this chapter.

At the end of this chapter is a summary of the syntax and parameters for the various calls.

Most Cache Manager management operations are done through the **ioctl**( ) and **pioctl**( ) calls. ACLs are manipulated through the **afs_syscall**( ) system call. The **ioctl**( ) call is a standard (BSD) UNIX call; **pioctl**( ) and **afs_syscall**( ) are DFS extensions. All three calls take several arguments, one of which is the identifier of the specific call to apply.

50.1 Extensions to the ioctl( ) System Call

The Cache Manager provides one additional **ioctl()** call, **VIOCIGETCELL.** This call finds the cell name associated with an open file descriptor. This is the recommended method of determining whether a file is stored in the DCE Distributed File System; if it is not, then it has no associated cell.

50.2 Using the pioctl( ) System Call

The syntax for **pioctl**( ) is as follows:

```
long pioctl (
      char *pathname,             /* in */
      int command,              /* in */
      struct afs_ioctl *ioDesc,    /* inout */
      intfollow_links)           /* in */
```

The **pioctl**( ) call is very similar to the **ioctl**( ) call. One important difference between the two is that for **pioctl**( ), the first argument, which refers to the file or directory to which the call applies, is passed via a pathname rather than via a file descriptor. (The **p** in **pioctl**( ) stands for pathname.) Using pathnames is necessary because pathnames, unlike file descriptors, can refer to files or directories that do not actually exist. Some calls, in fact, use the first argument to supply a directory and another argument to supply the name of a file in that directory; if the file does not exist but the directory does, some actions can still be taken, while a descriptor cannot be obtained on a nonexistent file.

The second argument, as with **ioctl**( ), is the specific call to be issued. Constants are defined for these values; you should use those constants rather than specific numeric values. The syntax summary at the end of this chapter contains these constants and other parameter information for each **pioctl**( ) call. Further information is also provided in the *OSF DCE Application Development Reference*.

As with **ioctl**( ), the third argument is a pointer to the block of data to be manipulated by the call. This data block contains an input buffer and an output buffer, one, both, or neither of which may be used for any particular call. The exact nature of this data depends on which call is being made. More information is given later in this chapter.

The fourth argument is unique to **pioctl( )**. It determines what file should be used if the file specified is a symbolic link. If this parameter is 1, the symbolic link is followed to its destination and the call is applied to that destination file. If the parameter is 0 (zero), the call is applied to the symbolic link itself. Note that this fourth parameter applies to a symbolic link only if the link is the last component of the pathname; links encountered in intermediate components are always followed.

Not all **pioctl( )** calls affect files. In those cases, a null pointer may be used for the pathname argument (in other words, **(char *)0**).

All calls either return 0 (zero), indicating successful execution, or return -1 and set **errno** to an error flag. Output, when provided, is placed in the output buffer, which is the third argument to **pioctl( )**.

The **pioctl( )** calls described in this chapter are only meaningful for operations on the DFS; in particular, they do not operate on files in the local file system.

The following **pioctl( )** calls require root privileges to perform:

- **VIOC_AFS_SYSNAME**
- **VIOC_CLOCK_MGMT** (setting only)
- **VIOC_EXPORTAFS** (setting only)
- **VIOCGETREQUEST**
- **VIOCSENDRESPONSE**
- **VIOCSETCACHESIZE**
- **VIOC_SETCELLSTATUS**
- **VIOCSETVOLSTAT**

50.2.1 System, Cell, and Fileset Operations

The calls in the following subsections have to do with getting information and performing operations on the system (workstation), cell, and filesets. Some information is related to the system in general (and not DFS in particular), some is related to the cell as a whole, and some is related to filesets. Anyone with access to the right directories, as determined by Access Control Lists on those directories, can examine this information, but usually only system administrators or root can change it. You should make sure you know the implications of what you are doing before you alter things at the system or cell level, as you will be affecting many other people. For more information about these implications, see the *OSF DCE Administration Guide*.

50.2.1.1 System Information

It is often useful to be able to check to see which file servers are up before starting expensive operations. This sort of test can be much faster than an attempt to contact a server that is not available. The **pioctl**( ) call **VIOCCKSERV** can be used to get this information. It returns a list of socket addresses of servers that are down (or none, if no servers are down). It can operate in one of two modes. The first (the ''quick'' mode) uses cached information about server status, which is updated periodically. The second (the ''thorough'' mode) performs the check when the call is made, so the information is more accurate, but at a price. The fast check is the recommended one unless your application really needs up-to-the-minute information. The thorough check is potentially expensive, as a query to a server that is down will have to wait to time out. Cached information about which servers are up is usually updated every 10 or 15 minutes; this time is controlled by parameters set by the system administrator.

The **pioctl**( ) call **VIOC_AFS_SYSNAME** is used to set or query the string that the *@sys* variable uses for pathname expansion. This variable, when used in a path, is expanded by the kernel; if you set it to different values on different hardware platforms, you can, for instance, use *@sys* to point to platform-dependent binaries while still using only one path in the code that calls them. The user command **cm sysname** checks the current platform's type.

It is not generally necessary to set the pathname variable, as DFS usually knows what its hardware and operating system are.

The input to this call contains a flag that indicates whether to query or set the value, and if setting a value, which value to use.

Because so many Cache Manager operations are tied to the time, there is a **pioctl( )** call to examine and change the setting of the local machine's clock. This call, **VIOC_CLOCK_MGMT**, also fetches and stores kernel values that describe the clock's accuracy. This feature is typically used only on client-only machines to set the machine's clock based on the time as set on a local File Server machine. This feature is initialized with **dfsd** (see the *OSF DCE Administration Guide*).

The clock data block contains a number of items, which are described in the *OSF DCE Application Development Reference*.

50.2.1.2 Cells

There are several types of information that you may want to get about a cell.

The **pioctl( )** call **VIOCGETCELL** provides a list of the known Fileset Location Servers (FL Servers) for a cell. The call provides the addresses of FL Servers known to the workstation on which the call is made. The way to use this function is to keep calling it, incrementing the cell number (starting with 0 (zero)), until you reach the end of the list of cells, at which point the call returns an error.

Sometimes a client application needs to be able to determine the name of the cell that the machine on which it is running belongs to. This information is provided by the **VIOC_GET_WS_CELL pioctl( )** call. A machine, of course, can access many cells; the cell under discussion here is the primary cell to which users of the machine authenticate by default. Programs such as the login authentication call make this call once, at startup.

VIOC_GETCELLSTATUS and **VIOC_SETCELLSTATUS** obtain and set the status information for a named cell. Currently, the only information available is a status bit that indicates whether the named cell is the local one.

50.2.1.3 Mount Points

A fileset is connected to the root directory of other filesets through a DFS mount point. Each mount point connects to the root directory of a fileset. The mount point looks like an ordinary subdirectory, but it is actually a special interface to a fileset. A mount point's name is the name of the subdirectory. Mount points are created after the filesets themselves are created.

Unlike Network File System (NFS) mount points, DFS mount points are stored in the file system. They are thus persistent, meaning that they survive system restarts.

Given the name of a mount point, it is possible to find the name of the fileset to be mounted at that place. Mount point and fileset names can, and should, be different. A fileset's name should be more verbose than the name that users of the system see. Users want to see terse names, but system administrators often want verbosity to make tasks such as accounting and backups easier. For example, the fileset corresponding to the mount point **jones** in the user tree may be **user.jones**, indicating that this fileset corresponds to a user directory. Users browsing through the directory already know that this fileset is in the user space because a directory named something like **user** or **usr** is above it in the tree, so the information should not be repeated in the mount point name itself.

The **pioctl**( ) call **VIOC_AFS_STAT_MT_PT** maps mount point names to fileset names. In addition to the fileset name, this call provides the type of the mount point. (The call actually produces a single string that begins with the character representing the type.) Mount points come in three types:

- The first type (regular), indicated by the # (number sign) character, tells the Cache Manager to use the latest read-only replica (fileset copy) that corresponds to the named fileset, if the mount point is part of a read-only tree and there is a read-only version of the fileset.
- The second type, indicated by the % (percent sign) character, forces the Cache Manager to use the read/write copy.
- The third type, indicated by the ! (exclamation point) character, indicates a mount point for the global namespace itself.

The third type is typically only used for diskless support. Using regular mount points is more efficient than forcing the use of the read/write copy, but there are times when you want to force access to the original fileset. Replicas are discussed in more detail in Chapter 49 of this guide.

Anyone who has delete access to the directory can delete a mount point. The **pioctl**( ) call **VIOC_AFS_DELETE_MT_PT** accomplishes this. If you want to completely remove a fileset and its mount points, you should use the fileset manipulation functions described in Chapter 51 of this guide.

50.2.1.4 Filesets and Files

The **pioctl**( ) call **VIOCWHEREIS** locates copies of a fileset, given the name of any file in it. It returns the addresses of all servers that have copies of the fileset. This call allows you to make the correspondence between a server being down and a directory access timing out. If you give **VIOCWHEREIS** a file in a read-only fileset, it returns the read-only servers. If you give it a file in a read/write fileset, it returns the read/write servers.

Given a filename, you may want to find out additional information about the fileset and cell in which it is stored. **VIOCWHEREIS** is useful for finding copies, but if you want other information, such as the fileset's cell or the file ID number of a particular file in the fileset, you need to use the **pioctl**( ) call **VIOCGETFID**. This call, given a filename, gets the file handle, which is a string representing that file's cell identifier, fileset identifier, file slot number, and a uniquifier that, combined with the other information, ensures uniqueness.

The cell and fileset identifiers are unique IDs assigned to the cell and fileset, respectively. The file slot number is the number of the file in the fileset. These numbers can be reused when files are deleted and others are created; you can think of a fileset as being an array of files (referenced by index number), and it is most efficient to fill the array from beginning to end. It is therefore preferable to put a new file in the earliest open slot in the array, rather than extending the array and giving the new file the highest position assigned to date.

Because of this scheme, a uniquifier is needed to distinguish among files that occupy, or once occupied, a particular file slot. This is simply an integer specifying which file this is; when a file slot is first occupied the file is given a uniquifier of 1, and for each subsequent occupant of that file slot, the uniquifier is incremented.

If all you want is the cell name for a file, you can call **VIOC_FILE_CELL_NAME**, which returns the name of the cell in which a file is stored. This is easier than calling **VIOCGETFID** and extracting the cell data. Note, however, that while **VIOCGETFID** returns the cell ID, **VIOC_FILE_CELL_NAME** returns the cell name.

To summarize, **VIOCWHEREIS** finds copies of a fileset, **VIOCGETFID** gets the file handle for a particular file within a fileset, and **VIOC_FILE_CELL_NAME**, given a filename, finds the name of the cell in which that file is stored.

50.2.1.5 Fileset and File Access

The **VIOC_SETCELLSTATUS** and **VIOC_GETCELLSTATUS** **pioctl**( ) calls control and report status information about a cell. Currently, the only status information available is whether a cell is the local one.

The calls **VIOCGETVOLSTAT** and **VIOCSETVOLSTAT** control status information about a particular fileset, rather then the cell. There are two status flags for filesets. The first controls whether the **setuid** and **setgid** bits are to be honored for the fileset. The second controls whether special device files (those in **/dev**) can be seen from workstations other than the local one.

To determine if a caller has access rights to a file, according to the DFS ACL, use the **pioctl**( ) call **VIOCACCESS**. When making this call, you supply the name of the file or directory and a mask specifying the types of access you want. The call fails if any of the requested permissions are denied, but it does not provide a way of finding out which one failed (short of iterating over all of the access rights). Because the denial of any single access right causes the entire call to fail, you should make sure you request only the rights you actually need to check.

VIOCACCESS does not actually retrieve the file or change the last modification time; it just checks that you are able to access the file. You may want to check this in advance if, for example, you will be running a job later that requires access.

The Cache Manager allows clients to fetch files before they are actually needed in order to save time later. This is done to take advantage of idle time, or to prepare for accesses that must happen very quickly for application-specific reasons. The **pioctl()** call **VIOCPREFETCH** prefetches the desired file into the local cache, so that the first file access is treated like a later access. If the caller does not have access rights to the file, the prefetch is unsuccessful.

50.2.2 The Cache

There are several **pioctl()** calls that directly manipulate the local cache and its parameters. It is possible to manipulate cache parameters on only a single machine at a time; changes that you make on one machine do not propagate to any other. Further, these calls cannot affect remote machines; you have to run the application on each machine separately to make changes on multiple machines. If you want to permanently change cache parameters, it is probably better to do this through the standard system administration procedures, described in the *OSF DCE Administration Guide*. These calls are intended primarily for use by applications that need to make temporary or localized changes.

50.2.2.1 Manipulating Cache Parameters

There is currently only one cache parameter that can be set or monitored: the cache size. The larger the cache, the more information you are able to store without performing unnecessary remote accesses. (If the cache fills up, the oldest entries are discarded first.) You need to balance this convenience against your other disk space needs, such as swap space and a local file system. More information about suggested sizes can be found in the *OSF DCE Administration Guide*.

To change the size of the cache, use the **pioctl()** call **VIOCSETCACHESIZE**. Size is allocated in 1024-byte units; you supply the number of these units to allocate. If you reduce the size of the cache, data is immediately deleted, oldest first, to make room.

If you attempt to reduce the cache size to an unrealistic value, the cache is not resized. (See the *OSF DCE Administration Guide* for guidelines on appropriate values.) If you specify a size of 0 (zero), it is reset to its default value.

The **pioctl** call **VIOCGETCACHEPARMS** can be used to find out how many blocks of data are currently allocated to the cache, and how many of those are in use.

There is no way to find out what is actually in the cache; you can only find out how much space it is using.

50.2.2.2 Manipulating Pending File Writes

Sometimes the Cache Manager attempts to write a file and cannot do so because the fileset is over quota. This type of write failure is different from others because quota problems can be corrected more easily than, for example, protection problems. Thus, the Cache Manager will remember that it is trying to write the file and keep trying, rather than rejecting the write entirely, in the hope that room will be cleared for the file. By default, the Cache Manager repeats these attempts approximately once per minute.

If you want to flush the queue of pending writes, you can use the **VIOCRESETSTORES** call. There are no inputs or outputs. By the time a write gets to this point, the error [**EDQUOT**] has already been returned to the caller, so it is not sent again. At this stage, no guarantees are made that the write will actually succeed, but efforts are made anyway. Thus, canceling these efforts does not significantly change expectations held by the application.

To see what is in this queue, use **VIOCLISTSTORES**. This call produces a count of the files waiting and an array describing the filesets to which they are to be written.

50.2.2.3 Flushing Information from the Cache

As discussed earlier, when the Cache Manager needs to make room to cache something, it flushes the oldest data it has. Sometimes your cache fills quickly causing the Cache Manager to flush something you still need. Thus, it is sometimes useful to be able to explicitly flush data from the cache to reclaim space, to prevent flushing data you still want, or to repair a corrupted cache.

The most general way to flush data from the cache is to use the **pioctl( )** calls **VIOCFLUSH** and **VIOC_FLUSHVOLUME**. The former flushes the cache entry for a single file, and the latter for an entire fileset. The next time that file or fileset is accessed, the server is queried and new data is retrieved.

The **pioctl( )** call **VIOCCKBACK** flushes the Cache Manager's memory of the mappings between fileset names and IDs, and between the ID of the read/write fileset and its replicas, if any. This information must be flushed periodically to allow for the fact that filesets can be moved. This flushing is normally done automatically once every hour.

Before flushing data, these functions try to write any local modifications to the server. You should not rely on this, however, as they may not succeed; *do not* use these functions in place of the **fsync( )** call.

50.2.3 Other Operations

The following subsections describe other Cache Manager **pioctl( )** operations.

50.2.3.1 Exporting the LFS to NFS

The **pioctl( )** call **VIOC_EXPORTAFS** exports a DFS Local File System (LFS) to the Network File System (NFS). It can be used to query and set the export status of the DCE LFS. (See the *OSF DCE Application Development Reference* for details.)

50.2.3.2 Using Nonkernel Helper Functions

Because DCE is very large, most of the code cannot reside in the kernel. When the kernel needs to call a routine or access data that is not resident in the kernel, it obtains its access through a helper function that is resident in user space. The user space process that makes the call is sometimes called a slave process.

The **VIOCGETREQUEST** and **VIOCSENDRESPONSE pioctl( )** calls are used by a user process to communicate with the kernel. The process announces its readiness to accept kernel calls with **VIOCGETREQUEST**, and the process then waits until the kernel returns from the call. When the kernel returns, the process should then take the contents of the output buffer and issue the call described therein. Once it has done this, it calls **VIOCSENDRESPONSE** to communicate the results back to the kernel. At this point it may call **VIOCGETREQUEST** again to repeat the process.

50.3 Using the afs_syscall( ) System Call

File and directory protection is handled by DCE Access Control Lists (ACLs), which are manipulated through the **afs_syscall( )** system call. An ACL specifies the types of access users and groups have to a file or directory. The use of ACLs is described fully in the *OSF DCE User's Guide and Reference*; the following subsections describe only the programming interface to them. DFS provides calls to set, retrieve, and copy ACLs, and to try to access files or directories.

All ACL calls are issued through the **afs_syscall( )** call. The first argument selects a DFS component; all ACL system calls use **AFSCALL_VNODE_OPS** for this argument. The second argument to **afs_syscall( )** is the name of the particular call to issue; it is analogous to the second argument to **pioctl( )** or **ioctl( )**. There are also additional call-specific arguments.

50.3.1 Retrieving ACLs

The **afs_syscall() VNX_OPCODE_GETACL** retrieves an ACL for an object. It takes four additional arguments:

- The pathname of the file or directory whose ACL is requested
- Memory in which to store the ACL (a string)
- Memory in which to store the length of the ACL
- An indication of which ACL is requested.

A directory can have up to three ACLs:

- One for its own use (**VNX_ACL_REGULAR_ACL**)
- One for files created in it (**VNX_ACL_INITIAL_ACL**)
- One for directories created in it (**VNX_ACL_DEFAULT_ACL**)

A regular file can have only a **VNX_ACL_REGULAR ACL** set.

The ACL is an array of bytes, and is very complex. The full description of the format is contained in the **afs_syscall(2dfs)** reference page in the *OSF DCE Application Development Reference*.

50.3.2 Setting ACLs

The **afs_syscall() VNX_OPCODE_SETACL** sets an ACL for an object. Like **VNX_OPCODE_GETACL**, it takes four additional arguments:

- A pathname, an ACL (represented as a string)
- The length of the string
- Which ACL to set

The ACL types are the same as for **VNX_OPCODE_GETACL**. It is an error to try to set any ACL type other than **VNX_ACL_REGULAR_ACL** for files.

50.3.3 Copying ACLs

The **VNX_OPCODE_COPYACL** call copies an ACL from one object to another. It takes four additional arguments:

- The destination pathname
- The source pathname
- Which ACL to set in the destination
- Which ACL to copy from the source.

As with **VNX_OPCODE_SETACL**, files can only have one of the three ACL types set.

50.4 Syntax Summary

The following subsections provide a summary of the syntax and parameters for the **ioctl()**, **pioctl()**, and **afs_syscall()** system calls.

50.4.1 The ioctl( ) Call

The syntax for a call to **ioctl( )** follows:

```
#include<ioctl.h>

long ioctl(
      char *pathname,          /* in */
      int command,             /* in */
      struct afs_ioctl *ioDesc) /* inout */
```

The following table shows the specific call implemented in DCE.

Call	Contents of Input Buffer	Contents of Output Buffer
VIOCIGETCELL	none	string

50.4.2 The pioctl( ) Call

The syntax for a call to **pioctl**( ) follows:

```
#include<ioctl.h>

long pioctl (
      char *pathname,          /* in */
      int command,             /* in */
      struct afs_ioctl *ioDesc, /* inout */
      int follow_links)        /* in */
```

The **pioctl**( ) calls are invoked not by name but by opcode number or by a defined constant. The following table contains these constants and the input and output types for each call.

Call	Contents of Input Buffer	Contents of Output Buffer
VIOCACCESS	long	none
VIOC_AFS_DELETE_MT_PT	string	none
VIOC_AFS_STAT_MT_PT	string( )	string
VIOC_AFS_SYSNAME	long + string	long + string
VIOC_CLOCK_MGMT	(see text)	(see text)
VIOCCKBACK	none	none
VIOCCKSERV	long [+ string]	long + sockaddrs
VIOC_EXPORTAFS	long	long
VIOC_FILE_CELL_NAME	none	string
VIOCFLUSH	none	none
VIOC_FLUSHVOLUME	none	none
VIOCGETCACHEPARMS	none	array of longs
VIOCGETCELL	long	long + sockaddrs + string
VIOC_GETCELLSTATUS	string	long
VIOCGETFID	none	afsFid
VIOCGETREQUEST	none	long + bytes
VIOCGETVOLSTAT	none	long
VIOC_GET_WS_CELL	none	string
VIOCLISTSTORES	none	long + long + array of afsHyper
VIOCPREFETCH	none	none
VIOCRESETSTORES	none	none
VIOCSENDRESPONSE	bytes	none
VIOCSETCACHESIZE	long	none
VIOC_SETCELLSTATUS	2 longs + string	none
VIOCSETVOLSTAT	long	none
VIOCWHEREIS	none	long + array of sockaddrs + string

50.4.3 The afs_syscall( ) Call

The syntax for the DFS part of **syscall** follows:

```
#include<syscall.h>
#include<aclint.h>

int afs_syscall(
     AFSCALL_VNODE_OPS,     /* in */
     VNX_OPCODE_GETACL,     /* in */
     char *pathname,        /* in */
     char *acl,             /* out */
     int *length,           /* out */
     int whichacl)          /* in */

int afs_syscall(
     AFSCALL_VNODE_OPS,     /* in */
     VNX_OPCODE_SETACL,     /* in */
     char *pathname,        /* in */
     char *acl,             /* in */
     int length,            /* in */
     int whichacl)          /* in */

int afs_syscall(
     AFSCALL_VNODE_OPS,     /* in */
     VNX_OPCODE_COPYACL,    /* in */
     char *dest_pathname,   /* in */
     char *source_pathname, /* in */
     int dest_whichacl,     /* in */
     int source_whichacl)   /* in */
```

Chapter 51

Manipulating Filesets

This chapter describes the three sets of functions available for manipulating filesets. These sets are:

- Volume Call (**VC...()**)
- Fileset Location Database (**VL...()**)
- Fileset Server (**FTSERVER...()**)

The **VC...()** functions meet most fileset manipulation needs. These are high-level, general-purpose functions for maintaining filesets. (They are not, technically, RPC calls, but they build on the **VL...()** and **FTSERVER...()** RPC calls.) These functions automatically keep the Fileset Location Database (FLDB) up-to-date, and handle errors gracefully. Failing to keep the FLDB and actual filesets in sync with each other could leave the system in an inconsistent state, as explained in the following paragraph. The **VC...()** functions are implemented using the **VL...()** and **FTSERVER...()** functions. They also provide consistency guarantees that naive use of the other two sets cannot provide.

At times, you may need to perform very specific operations that are not covered by the general **VC...()** functions. In those cases, you can use the **VL...()** and **FTSERVER...()** functions to alter the FLDB entries and actual data, respectively. If you use these functions instead of the **VC...()** functions, you have to do your own consistency checking when you alter filesets or FLDB entries to ensure that the two stay in sync. Changing either the data or FLDB entry alone, without changing the other, leads to inconsistency that can eventually eliminate your change; that is, if the wrong thing, database or data, is synchronized with the other. You need to be careful to make both changes at the same time. If you are creating new filesets or copies of filesets, you should make the copy first and then create the FLDB entry because if you have an entry and no fileset, applications that try to access the fileset before you create it get errors. When you are making a change, such as renaming a fileset, there is no good rule about which to change first, the data or the database entry. Just make sure you do one immediately after the other.

In addition, you need to make sure that if one of the two changes you make (to the fileset or FLDB entry) gets an error that prevents the change from actually being made, you do not make the other change anyway or leave the disk or database in an intermediate state.

All of the functions in this chapter operate, directly or indirectly, through remote procedure calls. You must write your own codc to obtain and manage RPC connections. (See Part 3 of this guide for information on doing this.)

All of the functions in this chapter operate on DFS filesets, and not on files resident on the native UNIX File System (UFS) that are not exported to DFS. An exported UFS is treated as a DFS fileset.

51.1 DCE and DFS API Terminology Differences

There are some differences in terminology between DCE documentation and the DFS API. In the DFS API, a *fileset* is referred to as a *volume*, an *aggregate* as a *partition*, and the *FLDB* as the *VLDB*. Although the DCE documents have been changed to use fileset terminology, the DFS API has not always been changed. Thus, calls, parameter lists, and structures often include the *vol*, *volume*, and *vldb* strings, and refer to *partitions*. In the DFS API, *volume* always means *fileset*, and *partition* means *aggregate* unless a UFS partition is explicitly being referred to.

51.2 Parameters, Types, and Return Values

Many UNIX functions either return 0 (zero) if the function succeeds, or return -1 and set a global variable (**errno**) to a status code. The functions described in this chapter do not do this; they always return a status code, or zero if there is no error.

Because the functions return error codes, the problem of how to return other data arises. Because these calls are issued over a network, rather than on the local machine, the functions cannot simply return pointers to modified data. Thus, you must allocate space for this additional data in advance, and pass pointers to this space in the calls. You simply need to allocate and manage the memory for these parameters using standard UNIX memory management techniques (the **malloc( )** and **free( )** calls, or stack variables).

All DFS functions with RPC interfaces have one argument in common, the RPC handle. It is always the first argument to a function and is of type **rpc_binding_handle_t**, as defined in Part 3 of this guide. This handle identifies the particular remote client, and is supplied when the remote connection is first established.

For the purpose of the examples in this chapter, we assume that you have created a function to obtain an RPC handle and set the variable *Rpc_Handle* to its output. For information about creating RPC handles, see Part 3 of this guide.

51.3 Data Types

Most of the structures described in this chapter have been defined in **typedefs** as well as structures. For example, **struct afsHyper** and **afsHyper** are the same. For reasons of extra clarity, the structure notation is used in describing the syntax of these functions. In the unlikely event that an application programmer needs to use the IDL interface for application development, only the latter format is acceptable in that environment.

Likewise, most of the fields of these structures are of unsigned types, including characters. These are explicitly labeled as unsigned only in cases where ambiguity could result otherwise.

51.4 Authorization Requirements

The Fileset Location Server (FL Server) and the Fileset Server (FT Server) each have administrative lists stored in *dcelocal***/var/dfs**. The Fileset Location Server for a cell runs on some subset of the machines in the cell, and the Fileset Server runs on every machine configured as a file server.

FL Server administrators can create file server entries in the FLDB that are owned by specific groups. People in the group owning a server entry can create, delete, and modify FLDB entries for filesets that are housed on the declared file server. Specifically, the caller has to be in the group owning the file server entry for all file servers referenced by the FLDB fileset entry, either before or after any proposed modification. This is transparent to the application programmer because the caller uses authenticated DCE RPC and the permissions are granted on the basis of that authentication.

FL Server administrators can make changes to the FLDB. All functions in this chapter that require altering the FLDB, including those in the **VC...()** set that create, delete, and move filesets, can be called only by those administrators whose principals are in the security group that is the ``owner'' of an FLDB server machine entry.

The administrative list for the Fileset Server should be the union of the FL Server administrative list, and the members of the group declared as owning the server entry in the FLDB. People on this list can alter any filesets on the affected servers. Callers of functions in this chapter that alter filesets themselves must be in the appropriate groups. (See the *OSF DCE Administration Guide* for more information about these administrative lists.)

51.5 The VC Functions: General Fileset Operations

The Volume Call (**VC...()**) functions obtain information and perform actions on filesets. These functions allow you to manipulate filesets and their corresponding Fileset Location Database (FLDB) entries; for example, you can read them, delete them, move them around, get status information from them, and so forth.

All of the **VC...()** functions except **VC_VolumeStatus()** require authorization.

51.5.1 Parameters

Because filesets are accessed through RPC calls, almost all functions in this section take an RPC connection and a socket address as arguments. These are necessary in order to tell the file server where to look for the fileset you are manipulating. When you first establish the remote connection, you should save the information about the connection and simply pass it to these functions. (See Part 3 of this guide for more information.)

A socket address is needed in addition to an RPC connection because the RPC connection is a connection to a fileset server and the socket address represents a file server machine to the fileset location server. The socket address can be obtained with the standard UNIX function **gethostbyname( )**.

Most of these functions also take an aggregate identifier, which identifies the aggregate containing the fileset. This identifier is unique only within a single file server. This information is available from the Fileset Location Database (see Section 51.6).

Filesets have both names and identifiers. The name is a string. Because system administrators and programmers create these names, there is no guarantee that the names are unique, although they are checked for collisions. The identifier is numeric, is generated by the system, and is unique.

51.5.2 Creating and Deleting Filesets

VC_CreateVolume( ) creates a fileset. Only DCE LFS filesets can be created with this function. It takes a fileset name and other identifying information, creates the fileset, creates an FLDB entry for the fileset, and returns the fileset's identifier. You should save this identifier for future references to this fileset.

The fileset name is a string of up to 111 characters, plus a null terminator as the 112th character.

There are two functions that can be used to delete a fileset. The first, **VC_DeleteVolume( )**, removes the fileset and updates the FLDB. The second, **VC_VolumeZap( )**, removes the fileset, but does not update the FLDB. You may want to use the latter if the database entry is already corrupted or nonexistent, or if you have multiple copies of a fileset and some of those copies are corrupted or out of date. You can use **VC_SyncVldb()**, described later in this chapter, to correct the FLDB entry for the fileset. Until you update the FLDB manually, the entry for this fileset is still there, and this could interfere with other operations on the fileset.

Both **VC_DeleteVolume()** and **VC_VolumeZap()** take four input parameters: an RPC connection, a socket address, an aggregate identifier, and a fileset identifier.

Note that the fileset name is not used in these calls. Both of these functions operate only on DCE LFS filesets.

51.5.3 Moving, Renaming, and Backing Up Filesets

There are three functions for modifying filesets. These are **VC_MoveVolume, VC_RenameVolume**, and **VC_BackupVolume**.

51.5.3.1 Moving Filesets

VC_MoveVolume() moves a fileset from one aggregate to another. Both the source and destination must be DCE LFS filesets; moving a UFS fileset, which by definition occupies an entire disk partition, would not be helpful. It takes five arguments: a fileset identifier, a socket address for the source server, the source aggregate, a socket address for the destination server, and the destination aggregate. The source and destination socket addresses can, of course, be on the same server, but they cannot be the same aggregate.

VC_MoveVolume() moves the given fileset to the new aggregate. It deletes any backups that existed on the old aggregate, but does not make new backups. If an error occurs (for example, if the fileset or the destination aggregate does not exist), an error is returned and any partially completed actions are undone.

As a general guideline, if there is a system failure you should check the state of the system to find out what the FLDB believes to be true and what is actually true (which filesets exist where). This enables you to recover appropriately.

51.5.3.2 Renaming Filesets

VC_RenameVolume( ) changes the name of a fileset. It also renames all associated backup and replicated copies of the fileset. Only the fileset *name* is changed; the fileset stays in the same aggregate and retains the same ID and mount point. To move a fileset from one aggregate to another, use **VC_MoveVolume**( ). If you want to change both a fileset's name and aggregate, you need to call both functions.

VC_RenameVolume( ) takes three parameters: an FLDB entry for the fileset, the old name, and the new name. The FLDB entry must be obtained using the function **VL_GetEntryByID**( ) or **VL_GetEntryByName**( ) (see Section 51.6).

51.5.3.3 Backing Up Filesets

VC_BackupVolume( ) makes a copy of a fileset and registers it as a backup in the FLDB. Backups have the name *fileset*.**backup**, where *fileset* is the name of the source fileset. Backups are important for recovering to a previous state, in the event that the fileset becomes corrupted due to programmer, user, or system errors. There is (by default) only one backup copy of a fileset; if one already exists, **VC_BackupVolume**( ) brings it up to date. Because **VC_BackupVolume**( ) uses cloning, it can be called only on DCE LFS filesets.

VC_BackupVolume( ) has four parameters: an RPC connection, a socket address, an aggregate identifier, and a fileset identifier.

51.5.4 Saving and Restoring Changes to Filesets

There are two fairly general functions for dumping and restoring filesets. These dumps can either be full or incremental; the changes since the last dump (if incremental) are written to or read from a pipe that you manipulate.

51.5.4.1 Dumping Changes

VC_DumpVolume( ) is used for dumping fileset changes to a file. This function is meant primarily for short-term dumping, such as moving filesets around by hand. For long-term storage, you should use the DCE backup facility (see the *OSF DCE Administration Guide*).

In addition to the parameters described previously, **VC_DumpVolume**( ) takes a source aggregate identifier and the reference date, which is the date from which changes should be written. This date is a structure containing fields for the date, the version number, and the mask. (You do not need to supply all three of these.) You must specify the mask, which is used to determine how **VC_DumpVolume**( ) should decide what changes to write. A value of 1 means to use the date only (so the version number does not need to be supplied). A value of 2 means to use the fileset's version number only (so the date does not need to be supplied). A value of 0 (zero) means to use neither of these, and just do a complete dump of the contents of the fileset. Keep in mind that any of these, but especially the complete dump, could require quite a bit of disk space. How much disk space you need depends on the amount of data you are dumping.

You can cause **VC_DumpVolume**( ) to write to standard output (**stdout**) by passing a **NULL** pointer as the filename.

51.5.4.2 Restoring Changes

The function **VC_RestoreVolume**( ) is used to restore data changes that you have recorded using **VC_DumpVolume**( ). The fileset must be a DCE LFS fileset. The function takes an RPC connection, a socket address, a fileset identifier, the destination aggregate (for the fileset), a new fileset name, and a pointer to a filename where the dump is stored. It also takes an option that tells the function what to do with any preexisting copy of the named fileset. If the value is 1, the old copy is overwritten; if the value is 0 (zero), the restoration is canceled if a fileset with that name already exists. **VC_RestoreVolume**( ) restores from standard input (**stdin**) if the fileset name is a **NULL** pointer.

When restoring from incremental dumps, be sure to do the restorations in the same order in which the dumps were made, starting with the full dump, if applicable.

51.5.5 Setting Fileset Quotas

The function **VC_SetQuota**( ) is used to change the disk quota for a DCE LFS fileset. You supply a fileset ID and a new quota. If the new quota is not sufficient to hold the current contents of the fileset, the quota is set anyway and any future write to the fileset will fail until the usage is reduced below the new limit.

The quota is measured in units of 1024 bytes.

51.5.6 Synchronizing the Database and File Server

It is possible for the Fileset Location Database (FLDB) to become inconsistent with the actual state of the system. Sometimes filesets will be deleted without the FLDB being updated (for example, **VC_VolumeZap**( ) does this); sometimes ''garbage'' like incomplete filesets will be generated and not cleaned up. For these reasons, there are two functions for synchronizing the FLDB and the actual contents of the aggregates.

VC_SyncVldb( ) operates on a specific aggregate on a server, or all aggregates on one server. It makes sure that everything in the aggregate(s) is represented in the FLDB, creating or updating entries as necessary. This function should always be called before **VC_SyncServer**( ), or you risk throwing away data.

VC_SyncServer( ) synchronizes the server with the FLDB. Filesets on the aggregate and not in the given FLDB entries are deleted, unless they are corrupted, in which case they are removed. Once deleted, there is no way to recover this data (except from backups), so make sure you *really* want to do this and have first called **VC_SyncVldb**( ).

VC_SykcServer( ) either operates on a single aggregate or, if the aggregate ID parameter is -1, operates on all aggregates on a server.

51.5.7 Getting Information About Filesets

Several functions are available for obtaining various types of information about filesets. **VC_ListVolumes**( ) lists all of the filesets in a given aggregate. **VC_VolumeStatus**( ) reports the status of a particular fileset. **VC_VolserStatus**( ) reports the status of the fileset server.

51.5.7.1 Listing Filesets in an Aggregate

VC_ListVolumes( ) provides a list of all filesets in an aggregate. It takes an RPC connection, a socket address, an aggregate identifier, and an option as input parameters, and a pointer to an array of results structures and the size of that structure as output parameters. One structure is returned for each fileset in the aggregate.

The result of a call to **VC_ListVolumes**( ) is an array of **ftserver_status** structures, one per fileset. Some of the fields of this structure include the fileset identifier, the type of fileset (read-only, read/write, or backup), the identifiers and dates of the last backup, the last clone and other copies, the fileset's creation date and last update date, the number of files in the fileset, information about disk quotas, and so on. For detailed information about this structure, see the *OSF DCE Application Development Reference*.

51.5.7.2 Getting the Status of One Fileset

VC_VolumeStatus( ) returns the same fileset information as **VC_ListVolumes**( ), but for only one (specified) fileset. It takes an RPC connection, a socket address, an aggregate identifier, and a fileset identifier as input parameters, and returns the status (a **ftserver_status** structure) as an output parameter. (See the previous section and the *OSF DCE Application Development Reference* for information about this structure.)

This is an unprivileged call.

51.5.7.3 Getting the Status of the Fileset Server

Currently active operations (''transactions'') are the basis for all file server operations, such as manipulating processes or filesets. When an operation is started a transaction is opened, and at the end of the operation the transaction is closed. Transactions are explained further in Section 51.7.

VC_VolserStatus( ) returns a list of all active transactions on the fileset server. These transactions cover all file server operations, such as rename, delete, create, move, and copy.

The transaction information is represented by an array of **flserver_transStatus** structures. The structures contains information such as the transaction identifier, the aggregate identifier, the fileset identifier, a descriptor of the open fileset, the time the transaction was started, and the time the transaction was last active. For more information, see the *OSF DCE Application Development Reference*.

51.5.8 Syntax Summary

The **VC...**( ) functions are as follows:

- **VC_BackupVolume**( ): Makes a backup copy of a fileset (DCE LFS filesets only)
- **VC_CreateVolume**( ): Creates a new fileset (DCE LFS filesets only)
- **VC_DeleteVolume**( ): Deletes a fileset (DCE LFS filesets only)
- **VC_DumpVolume**( ): Dumps recent changes to a fileset
- **VC_ListVolumes**( ): Gets a list of filesets on a particular aggregate
- **VC_MoveVolume**( ): Moves a fileset from one aggregate to another (DCE LFS filesets only)
- **VC_RenameVolume**( ): Renames a fileset
- **VC_RestoreVolume**( ): Restores previously saved changes to a fileset (DCE LFS filesets only)
- **VC_SetQuota**( ): Sets the disk space quota for a fileset (DCE LFS filesets only)

- **VC_SyncServer( )**: Removes filesets from an aggregate that are not listed as being there in the Fileset Location Database
- **VC_SyncVldb( )**: Synchronizes the Fileset Location Database with the actual state of an aggregate
- **VC_VolserStatus( )**: Reports the status of a fileset server
- **VC_VolumeStatus( )**: Reports the status of a particular fileset
- **VC_VolumeZap( )**: Deletes a fileset without updating the Fileset Location Database (DCE LFS filesets only)

A syntax summary of each VC function follows.

```
#include <param.h>
#include <sysincludes.h>
#include <stds.h>
#include <common_data.h>
#include <rpc.h>
#include <pthread.h>
#include <cma_exception.h>
#include <compat.h>
#include <nubik.h>
#include <fldb_proc.h>
#include <flserver.h>
#include <flclient.h>
#include <fcntl.h>
#include <ftserver_proc.h>
#include <queue.h>
#include <volume.h>
#include <fldb_data.h>
#include <ftserver.h>
#include <ftserver_trans.h>
#include <aggr.h>
#include <volc.h>
```

```
long VC_BackupVolume(
     rpc_binding_handle_t RpcBinding,      /* in */
     struct sockaddr *servAddrp,           /* in */
     unsigned long aggrId,                 /* in */
     struct afsHyper *filesetIDp)          /* in */
```

```
long VC_CreateVolume(
     rpc_binding_handle_t RpcBinding,      /* in */
     struct sockaddr *servAddrp,        /* in */
     unsigned long aggrId,           /* in */
     char *filesetNamep,            /* in */
     struct afsHyper *filesetIDp)       /* out */

long VC_DeleteVolume(
     rpc_binding_handle_t RpcBinding,      /* in */
     struct sockaddr *servAddrp,        /* in */
     unsigned long aggrId,           /* in */
     struct afsHyper *filesetIDp)       /* in */

long VC_DumpVolume(
     rpc_binding_handle_t RpcBinding,      /* in */
     struct sockaddr *servAddrp,        /* in */
     struct afsHyper *filesetIDp,       /* in */
     unsigned long fromAggrId,         /* in */
     struct ftserver_Date *dumpDatep,     /* in */
     char *filename)              /* in */

long VC_ListVolumes(
     rpc_binding_handle_t RpcBinding,      /* in */
     struct sockaddr *servAddrp,        /* in */
     unsigned long aggrId,           /* in */
     int allFlag,              /* in */
     struct ftserver_status **resultPtr,    /* out */
     long *sizep)              /* out */

long VC_MoveVolume(
     struct afsHyper *filesetIDp,       /* in */
     struct sockaddr *fromservAddrp,      /* in */
     unsigned long fromaggrId,         /* in */
     struct sockaddr *toservAddrp,       /* in */
     unsigned long toaggrId)          /* in */

long VC_RenameVolume(
     struct vldbentry *entryp,         /* in */
     char *oldNamep,             /* in */
     char *newNamep)              /* in */
```

```
long VC_RestoreVolume(
     rpc_binding_handle_t ToRpcBinding,     /* in */
     struct sockaddr *toservAddrp,        /* in */
     unsigned long toaggrId,            /* in */
     struct afsHyper *filesetIDp,         /* in */
     char *filesetNamep,              /* in */
     int override,                 /* in */
     char *filename)                 /* in */

long VC_SetQuota(
     rpc_binding_handle_t RpcBinding,      /* in */
     struct sockaddr *servAddrp,          /* in */
     unsigned long aggrId,             /* in */
     struct afsHyper *filesetIDp,         /* in */
     unsigned long quota)              /* in */

long VC_SyncServer(
     rpc_binding_handle_t RpcBinding,      /* in */
     struct sockaddr *servAddrp,          /* in */
     unsigned long aggrId)             /* in */

long VC_SyncVldb(
     rpc_binding_handle_t RpcBinding,      /* in */
     struct sockaddr *servAddrp,          /* in */
     unsigned long aggrId)             /* in */

long VC_VolserStatus(
     rpc_binding_handle_t RpcBinding,      /* in */
     struct sockaddr *servAddrp,          /* in */
     ftserver_transEntries *rstatusp)      /* out */

long VC_VolumeStatus(
     rpc_binding_handle_t RpcBinding,      /* in */
     struct sockaddr *servAddrp,          /* in */
     unsigned long aggrId,             /* in */
     struct afsHyper *filesetIDp,         /* in */
     struct ftserver_status *statusp)      /* out */
```

```
long VC_VolumeZap(
    rpc_binding_handle_t RpcBinding,    /* in */
    struct sockaddr *servAddrp,         /* in */
    unsigned long aggrId,               /* in */
    struct afsHyper *filesetIDp)        /* in */
```

51.6 The VL Functions: Interacting with the Fileset Location Database

The Volume Location (**VL...(**)) functions are used by remote clients to interact with the Fileset Location Server. (To interact with the filesets themselves, use the **FTSERVER...(**) functions.) As described in Chapter 49, each fileset has one entry in the FLDB, and this entry is accessed by the Fileset Location Server.

The **VL...(**) functions are the basic means by which DFS Cache Managers locate resources, primarily filesets, in a DCE cell. The locations of filesets are listed in the Fileset Location Database (FLDB), which is primarily read and updated by the Fileset Location Server.

The **VL...(**) functions fall into several categories:

- Fileset location: These functions determine the location of a fileset given either its assigned unique identifier (''uniquifier'') or its name.
- Fileset maintenance: These functions create, delete, and change attributes of filesets, enumerate entries in the Fileset Location Database, report statistics, and change attributes of fileset-bearing servers.
- Obtaining configuration information: This function allows an application to get configuration information about a cell from the Fileset Server.

The **VL...(**) functions have, in addition to the RPC handle that is common to all RPC functions, a second common argument, which is always the last output parameter. This parameter contains the RPC error codes, if any RPC errors occurred during the call. It is of type **error_status_t**, which is defined in Part 3 of this guide.

51.6.1 The Fileset Location Database Entry

An FLDB entry contains the following information:

- Fileset name
- Fileset type (read/write, read-only, or backup)
- Information about all of the servers on which the fileset is replicated
- Information used for replicating and cloning the fileset
- The DCE principal for the fileset

This information is represented by the **vldbentry** structure, which is defined as follows. It is discussed fully in the *OSF DCE Application Development Reference*.

```
struct vldbentry{
    char name[MAXNAMELEN];
    unsigned long volumeType;
    unsigned long nServers;O
    struct afsNetAddr siteAddr[MAXNSERVERS];
    unsigned long sitePartition[MAXNSERVERS];
    unsigned long siteFlags[MAXNSERVERS];
    unsigned long sitemaxReplicaLatency[MAXNSERVERS];
    struct kerb_princ_name sitePrincipal[MAXNSERVERS];
    struct afsUUID siteOwner[MAXNSERVERS];
    struct afsUUID siteObjID[MAXNSERVERS];
    struct afsHyper VolIDs[MAXVOLTYPES];
    unsigned long VolTypes[MAXVOLTYPES];
    struct afsHyper cloneId;
    unsigned long flags;
    unsigned long maxTotalLatency;
    unsigned long hardMaxTotalLatency;
    unsigned long minimumPounceDally;
    unsigned long defaultMaxReplicaLatency;
    unsigned long reclaimDally;
    unsigned long WhenLocked;
    unsigned long spare1;
    unsigned long spare2;
    unsigned long spare3;
    unsigned long spare4;
    char LockerName[MAXLOCKNAMELEN];
    char charSpares[50];
  }
```

The fields of this structure are

name The string naming the fileset.

volumeType

The fileset type, one of the following:

- **VOLTIX_TO_VOLTYPE(RWVOL)** (read/write)
- **VOLTIX_TO_VOLTYPE(ROVOL)** (read-only)
- **VOLTIX_TO_VOLTYPE(BACKVOL)** (backup)

The macro **VOLTIX_TO_VOLTYPE** takes one of the three types as its argument and produces the actual type representation, which is internal.

nServers The number of servers that contain this fileset, up to **MAXNSERVERS.**

siteAddr An array of server addresses; a value higher than the value of **nServers** is meaningless.

sitePartition An array listing aggregate identifiers for the fileset on each server; a value higher than the value of **nServers** is meaningless.

siteFlags Flags for each server, as described in the *OSF DCE Application Development Reference* under **VL.**

sitemaxReplicaLatency

The maximum age, in seconds, that a fileset replica can be without the Replication Server attempting to update it.

sitePrincipal

The name of the DCE principal.

siteOwner The UUID of the authentication group that can modify site information.

siteObjId The UUID of the site.

VolIDs The fileset IDs for all related filesets.

VolTypes The types of those related filesets. This array and the previous one are paired.

cloneId IDs of cloned copies of a fileset.

flags General flags, as opposed to site flags, as described in the *OSF DCE Application Development Reference* under **VL**.

maxTotalLatency
The maximum age, in seconds, a cached copy of data from a fileset can be before the DFS Cache Manager seeks to refresh it.

hardMaxTotalLatency
The fileset age at which the Cache Manager will refuse to use cached data.

minimumPounceDally
The amount of time, in seconds, for the Replication Server to wait before attempting to retrieve a new token after losing one.

defaultMaxReplicaLatency
The age, in seconds, a replica can be before the replication server will not trust it.

reclaimDally
The time, in seconds, between keep-alive messages sent by the Cache Manager.

WhenLocked
The time at which this entry was locked, if it is currently locked.

LockerName
The name of the user holding the current lock on the entry, if any.

Caution: While DFS gives you the ability to alter any of this information, doing so could have severe ramifications throughout the file system. You should not attempt to alter any of this information directly unless you are fully aware of the consequences. (See the *OSF DCE Administration Guide* for further information.)

In addition to the **vldbentry** structure, the **compactvldbentry** structure holds a subset of this information. The latter, being much smaller, is more efficient to pass around. (See Section 51.6.2 for a discussion of this.)

The standard fileset types are obtained with a call to the **VOLTIX_TO_VOLTYPE** macro with one of the following three values:

RWVOL	Read/write fileset
ROVOL	Read-only fileset
BACKVOL	Backup fileset

51.6.2 Fileset Location

There are two ways to get the FLDB entry for a fileset: by the fileset's name or by its unique identifier.

VL_GetEntryByID( ) and **VL_GetEntryByName**( ) return a full FLDB entry for the given fileset. This is the most straightforward way to get fileset information from the FLDB.

If **VL_GetEntryByID**( ) is being used, there are two input parameters: a fileset ID (an **afsHyper** structure) and the fileset type (a number). A fileset has a number of IDs (one per distinct type); you just need to supply one of them. The type is used to distinguish the various copies of a fileset from the read/write version because you could be asking for information about any of these and they share the same name. The type is used as a hint to locating the correct entry. If **VL_GetEntryByName**( ) is being used, there is a single input parameter, a string.

In addition to **VL_GetEntryByID**( ) and **VL_GetEntryByName**( ), DFS provides two functions to get ''compact'' information from the FLDB. These functions, **VL_GetCEntryByID**( ) and **VL_GetCEntryByName**( ), take the same arguments and have the same behavior as the **VL_GetEntryByID**( ) and **VL_GetEntryByName**( ) functions, except that instead of returning structures, they return **compactvldbentry** structures.

The compact entry has the following fields from the original structure:

- **name**
- **volumeType**
- **nServers**
- **sitePartition**

- **siteFlags**
- **sitemaxReplicaLatency**
- **volIDs**
- **VolTypes**
- **cloneId**
- **flags**
- **maxTotalLatency**
- **hardMaxTotalLatency**
- **minimumPounceDally**
- **defaultMaxReplicaLatency**
- **reclaimDally**
- **WhenLocked**
- **LockerName**

There is also one additional field, **siteCookies**. The contents of this field can be used to generate detailed site information; more information on this is given later in this section.

The FLDB stores information about the servers on which a fileset is available, such as addresses, server partition numbers, and so on. Depending on your configuration, a fileset may exist on only one server or on a large number of servers. Because there is no way to guess how much server data will have to be stored for a fileset, **VL_GetEntryById( )** and **VL_GetEntryByName( )** may not be able to return all of the server information that is available in a single call. An FLDB entry can contain up to 16 addresses for a fileset, but each server can have up to four addresses, so a fileset could have as many as 64 addresses.

In order to get this additional information, you must use the **VL_GetNextServersByID( )** and **VL_GetNextServersByName( )** functions. As with the functions described previously, the only difference between the two is the means of identifying the fileset.

The **VL_GetNextServersByID( )** and **VL_GetNextServersByName( )** functions are used to get a list of all of the file servers that contain a given fileset. Call them repeatedly until you have received all of the server information. Each produces an output value whose low-order bit is set to 1 if the results from this call end the list. Otherwise, this bit is set to 0 (zero).

This iteration is done by feeding the functions a start position. This indicates how far through the list of servers you have gotten; the initial value is 0 (zero). **VL_GetNextServersByID( )** and **VL_GetNextServersByName( )** will return, along with the expected data, a new start number to use in the next call. You simply keep calling the functions with the new start numbers until you run out of servers, which is signalled by the functions returning a special error code (**VL_ENDOFLIST**). The iterator is not used except as input to the subsequent call.

If the next call to the function would return **VL_ENDOFLIST**, the low-order bit will be set in the output flags parameter.

The first time **VL_GetNextServersByID( )** or **VL_GetNextServersByName( )** is called, the data looks like the output from **VL_GetEntryByID( )** and **VL_GetEntryByName( )**. If you know that you are going to want to retrieve all of the server information, use the server functions instead of **VL_GetEntryByID( )** and **VL_GetEntryByName( )**.

The inputs to **VL_GetNextServersByID( )** and **VL_GetNextServersByName( )** are analogous to those for **VL_GetEntryByID( )** and **VL_GetEntryByName( )**; the former takes a fileset ID and a type, and the latter takes a fileset name (a string).

The following code fragment obtains a list of all servers that contain the **user.jones** fileset.

```
char *fileset = "user.jones";
unsigned long startIterator, newIterator, flags;
struct vldbentry *entry;
error_status_t error;
long nextServ = 0;
int j;

while (nextServ != VL_ENDOFLIST && ((nextServ != 0) || (flags & j) == 0))
  {
    nextServ = VL_GetNextServersByName(Rpc_Handle, fileset,
        startIterator, &newIterator, &entry, &flags, &error);
    if(error)
      printf("Got an error! \n");
    else
      {
       startIterator = newIterator;
       printf("Servers: \n");
       for(j=0; j<16 && j < entry->nServers; j++)
         /* net address is in two pieces, a short and a string */
         printf("  %2d: %d%s\n", j, entry[j]->type, entry[j]->data);
     }
}
```

As with **VL_GetEntryByID**( ) and **VL_GetEntryByName**( ), there are also analogs for the server functions that produce compact FLDB entries. These functions are **VL_GetCNextServersByID**( ) and **VL_GetCNextServersByName**( ), and they behave as the similarly named function pair described previously.

The ''compact'' functions, as mentioned earlier, store a compacted version of the server information instead of the larger version found in the **vldbentry** structure. This compacted format is only useful if it is easy to convert from this to an expanded format. **VL_ExpandSiteCookie**( ) does exactly this. It takes the cookie field from a **compactvldbentry** structure and returns a **siteDesc** structure, which contains network addresses and the DCE principal.

51.6.3 Fileset Entry Maintenance

Most of the **VL...**( ) functions relate to maintaining the FLDB. This maintenance includes routine activities like creating, deleting, and modifying entries, obtaining various information, changing the servers associated with filesets, and locking filesets to block other access to them. This maintenance is usually initiated by system administrators, but applications may need to create and maintain filesets and their associated data as well.

You should lock fileset entries before modifying them or their filesets; this lock will serve as a signal to others who may wish to operate on it. Locks are discussed in Section 51.6.3.2.

In order to modify FLDB entries, you must have administrator privileges on the FLDB or, in some cases, be in the owner group for any file server entries that your action refers to.

51.6.3.1 Creating and Deleting Fileset Entries

The functions **VL_CreateEntry**( ) and **VL_DeleteEntry**( ) do what their names imply, they create and delete FLDB entries.

VL_CreateEntry( ) is used to create a new FLDB entry; creating the actual fileset is a separate issue. You have to supply a full **vldbentry** structure, as described earlier. You must fill in all fields except the name of the user who last locked the entry.

The FLDB entry that you supply to **VL_CreateEntry**( ) includes a fileset ID. Before creating an entry, you need to allocate this ID. You really need to allocate three such IDs: one for the source, one for the backup, and one for a replica.

The functions **VL_GetNewVolumeId**( ) and **VL_GetNewVolumeIds**( ) allocate fileset IDs for future use. **VL_GetNewVolumeId**( ) allocates one ID; **VL_GetNewVolumeIds**( ) takes an argument that specifies the number of identifiers to reserve.

Avoid allocating more IDs than you actually need because these IDs are kept forever once they are allocated. There is no way to reclaim them. While the limit on the total number of IDs is large, it is still finite, and it is beneficial to keep the numbers representing IDs as low as possible. The best approach is to allocate IDs right before you need them, rather than allocating a large block and then using only a few of them later.

VL_DeleteEntry( ) deletes an entry from the FLDB. As with other functions in this set, you must separately remove the actual fileset. **VL_DeleteEntry**( ) takes two arguments: the fileset ID and fileset type.

Because there is only one FLDB entry per fileset, deleting the entry automatically deletes information about backups, clones, and replicas. The *filesetType* parameter to **VL_DeleteEntry**( ) is used as a hint; if you know the type of fileset, you should supply this information so that the function can locate the entry more easily.

If you do not know the type, use a value of -1.

51.6.3.2 Locks

In order to reserve an FLDB entry for your exclusive use, such as to write to it, you must first lock it. There is no way to lock a fileset directly; all locks are handled through the FLDB.

Locking an FLDB entry alerts the FLDB that the fileset is reserved for writing by one particular client. Applications are free to bypass locks and alter the fileset directly; locks are purely advisory. You should make sure your applications attempt to set locks before modifying FLDB entries or filesets; applications must never make modifications without setting locks first. Locks are there to help you: that is, to protect you from other applications (including user commands) that might attempt to manipulate the same filesets that you are manipulating. However, locks do not *enforce* anything, and if any application bypasses the use of locks, the potential for incompatible changes exists.

The function used to set a lock is **VL_SetLock( )**. It takes three arguments: a fileset ID, fileset type, and a set of options (represented as a number; the following names are the names of constants). The options indicate the reason for the lock, such as **VLOP_MOVE** for moving the fileset and **VLOP_DELETE** for deleting the fileset. You should set the correct options so that other applications attempting to access the FLDB entry know why the lock is in place.

A list of the possible options follows:

- **VLOP_MOVE**
- **VLOP_RELEASE**
- **VLOP_BACKUP**
- **VLOP_DELETE**
- **VLOP_DUMP**
- **VLOP_RESTORE**
- **VLOP_ADDSITE**

Any client that tries to lock a locked fileset entry will be informed of the existing lock, and its attempt to obtain a lock will fail.

You should make sure you release the lock as soon as you are done with it, so that the fileset entry will be available to others who need to write it.

VL_ReleaseLock( ) releases a fileset lock acquired by **VL_SetLock( )**. It takes three arguments: the fileset ID, fileset type, and a mask for the FLDB lock options to clear.

51.6.3.3 Modifying Fileset Entries

Sometimes it is necessary to change information in an FLDB entry, such as a fileset name or server information, to reflect changes on the file servers. For example, backup facilities may need to do this. You can change any field of an FLDB entry, although as discussed before, it can be dangerous to change them without being aware of the consequences.

VL_ReplaceEntry( ) replaces one entry in the FLDB with another. As with **VL_CreateEntry( )**, you are responsible for making sure the new entry is complete. The function takes four arguments: the fileset ID for the entry you wish to change, the fileset type, the new entry, and the lock fields to be cleared at the end of the operation.

Calling **VL_ReplaceEntry( )** is functionally equivalent to calling **VL_CreateEntry( )** with the new entry then calling **VL_DeleteEntry( )** with the old entry; however, actually doing this is prohibited because you cannot have two filesets with the same name or ID.

51.6.3.4 Modifying Server Addresses

A machine can have several network addresses, generally corresponding to different networks for which the machine has an interface. Because network addresses sometimes change, and network interfaces may be added and dropped from server machines, and addresses specified when the server is first added to the file system may become out-of-date and need to be changed. The functions described in this section are used to be added, remove, change, and look up these addresses.

VL_AddAddress( ) declares another network address for a server. It takes two arguments: any current address for the server (to identify it) and the address to be added. All known addresses can be found by the **VL_GetSiteInfo( )** call.

VL_RemoveAddress( ) removes a network address from the list of addresses for a server. It takes two arguments, the associated RPC connection and the address to be removed.

VL_ChangeAddress( ) changes one of the network addresses for a file server to a new value. It is comparable to the **VL_AddAddress**(*OldAddr, NewAddr*) sequence followed by **VL_RemoveAddress**(*OldAddr*), but does not require that the set of addresses have space available for a new entry.

The maximum number of addresses that a server can have is **ADDRSINSITE**.

In order to get all of the addresses for a server, use the **VL_GetSiteInfo**( ) function, which returns all known server addresses and the DCE principal, given any one address. It produces a **siteDesc** structure as output. This structure contains several fields:

- **Addr**, an array of four site addresses
- **KerbPrin**, a string identifying the principal
- **Owner**, the UUID for the owning authentication group
- **ObjID**, the UUID for the server itself
- **CreationQuota**, the maximum number of filesets allowed (0 means unlimited)
- **CreationUses**, the number of filesets that currently exist

51.6.3.5 Modifying the Set of Servers

In addition to modifying addresses and principals for existing servers, DCE provides a way to add and change servers themselves in the FLDB.

The **VL_CreateServer**( ) function adds a server to the FLDB. It takes a **siteDesc** structure, described previously. Before you can create entries on a server or modify the server's addresses, you must create it in the FLDB. Merely having a File Exporter running on the server is not sufficient; if the FLDB is not told about the server explicitly, it will not be able to locate filesets on the servers in question. This is an operation that should be performed only once. If the FLDB contains servers marked as deleted, one of these is reused to save space.

The **VL_AlterServer( )** function makes changes to an existing server. It takes an address (any single address by which the server is known can be used) and a **siteAlter** structure, which specifies the properties to change. The **siteAlter** structure is defined as follows:

```
struct siteAlter {
    unsigned long Mask;
    char KerbPrin[MAXKPRINCIPALLEN];
    afsUUID Owner;
    afsUUID ObjID;
    unsigned long CreationQuota;
    unsigned long CreationUses;
    unsigned long spare1;
    unsigned long spare2; }
```

The mask is the bitwise OR of the options for the fields to be altered; the new values for those fields are taken from the rest of this structure. (Any other values that are filled in are ignored.) The options are

- **SITEALTER_PRINCIPAL**
- **SITEALTER_OWNER**
- **SITEALTER_OBJID**
- **SITEALTER_CREATIONQUOTA**
- **SITEALTER_CREATIONUSES**
- **SITEALTER_DELETEME**

CreationQuota is the maximum number of filesets that can be created on the server. If it is 0 (zero), there is no limit.

CreationUses is the current number of entries in the FLDB that point to this server. *Do not* modify this field.

51.6.3.6 Manipulating Other FLDB Information

If **SITEALTER_DELETEME** is specified, the server is deleted from the FLDB. It is an error to combine this bit with any other, or to delete a server that is still in use. Once a site is deleted, there is no way to retrieve it. The memory used to store the data is zeroed and reused by the next site creation.

It is sometimes useful to be able to get a list of all FLDB entries, so that you can perform some global actions or just survey the existing filesets. The **VL_ListEntry**( ) function provides a list of all entries in the FLDB. The **vldbentry** structure for each is provided; this structure lists a variety of information about the entry, and was described earlier in this chapter.

You should be sure that you really want information about the *entire* FLDB before issuing this call repeatedly. The database could be quite large.

VL_ListEntry( ) provides FLDB entries one at a time by using an iterator, as described earlier. The function returns, along with an entry, an entry number saying where to start on the next call. On subsequent calls, you pass in that number and it starts there, rather than at the beginning. On the first call, pass 0 (zero) for this value. This allows you to get all of the information, although it generally takes many calls. When that number is returned as 0 (zero), all of the entries have been supplied.

This number is the only input argument to **VL_ListEntry**( ). Three values are placed into the output buffer: an estimate of the number of entries remaining, the start number for the next call, and the returned entry.

VL_ListByAttributes( ) addresses two limitations in the **VL_ListEntry**( ) mechanism: that each FLDB entry is returned in individual calls, and that no selection of FLDB entries is done at the database location itself. **VL_ListByAttributes**( ) uses a selector structure that describes the subset of the FLDB entries that should be returned. The results are returned in an array (up to **MAXBULKLEN** entries).

The selector is a **VldbListByAttributes** structure.

```
struct VldbListByAttributes {
    unsigned long Mask;
    struct afsNetAddr site;
    unsigned long partition;
    unsigned long volumetype;
    struct afsHyper volumeid;
unsigned long flag;
unsigned long spare1;
unsigned long spare2;
unsigned long spare3;
unsigned long spare4;
unsigned long spare5; };
```

The fields are as follows:

site The network address of the server.

partition The aggregate numeric ID.

volumetype The fileset type, one of the following:

- **(VOLTIX_TO_VOLTYPE(RWVOL)**
- **VOLTIX_TO_VOLTYPE(ROVOL)**
- **VOLTIX_TO_VOLTYPE(BACKVOL)**

volumeid The fileset ID.

flag The options.

Mask A value for the whole FLDB, indicating the fields about which information is desired. Possible mask values are

Site	**VLLIST_SITE (0x1)**
Partition	**VLLIST_PARTITION (0x2)**
Fileset type	**VLLIST_VOLUMETYPE (0x4)**
Fileset ID	**VLLIST_VOLUMEID (0x8)**
Flags	**VLLIST_FLAG (0x10)**

To generate a mask, take the bitwise OR of the constants listed previously for the fields you want to see. (The numeric values are provided for convenience; you should always use the constants.)

For example, to select all filesets on a particular server and aggregate, use a mask value of 0x3 and fill in the **site** and **partition** fields with the particular site and aggregate in which you are interested.

More information about this data structure is provided in the *OSF DCE Application Development Reference*.

As with **VL_ListEntry( )**, iteration is used to get all of the entries, but up to **MAXBULKLEN** entries are returned on each call.

VL_ListByAttributes( ) takes two input parameters: the structure described previously and the iteration value (starting with 0). It returns four values: the number of entries returned, the entries, the next iterator value, and a flag. Only the low order bit of this flag is defined; if it is set, then this return includes the last matching entry.

The following code obtains all read/write entries on a particular server; the ID of this server is presumed to be stored in the global variable **This_Server**.

```
struct VldbListByAttributes attrib;
unsigned long iterator, newIterator, numEntries, flags;
bulkentries *entries;
error_status_t error;
/* The size of the following array depends on how many
   filesets you expect to match. */
struct vldbentry AllEntries[256];
int lastFilledEntry = 0, j;

attrib.volumeType = VOLTIX_TO_VOLTYPE(RWVOL);
attrib.Mask = VLLIST_SITE | VLLIST_VOLUMETYPE;
bcopy(&This_Server, &attrib.site, sizeof(afsNetAddr));

flags = 0;
iterator = 0;

while (flags == 0)
  {
    VL_ListByAttributes(Rpc_Handle, &attrib, iterator,
                        &numEntries, &entries,
                        &newIterator, &flags, &error);
    if(error)
      printf("Got an error! \n");
    else
      {
        vldbentry *entry;

        iterator = newIterator;
        entry= &entries.bulkentries_val[0];
```

```
          for(j=0; j<numEntries; j++, entry++,
              lastFilledEntry++)
            AllEntries[lastFilledEntry] = entry;
        }
  }
```

It is possible to get information about the FLDB itself, such as when servers started running, how many requests of what types have been made, and so on. **VL_GetStats**( ) returns two data structures that describe both the specific operational statistics that have been gathered by this Fileset Location Server instance and the basic information describing the FLDB as a whole.

The operational statistics, represented by the **vlstats** structure, include the following:

- **start_time**, when the server started up
- **requests**, the number of requests of each type of RPC procedure
- **aborts**, the number of aborts of each type

These last two are arrays; the index values for each RPC are listed in the *OSF DCE Application Development Reference* under the **VL_GetStats(3dfs)** reference page.

The basic FLDB information, a **vital_vlheader** structure, contains frequently used global values and general information such as internal statistics.

If you do not want all of the preceding information, you can get just a list of all active file servers known to the FLDB (in other words, any servers mentioned in it and not marked as deleted) by using the **VL_GenerateSites**( ) function. This function uses an iterator, and provides **siteDesc** structures (see **VL_GetSiteInfo**( )) compacted together into a **bulkSites** structure, nine at a time.

You can also check for communications problems involving the Fileset Location Server with the **VL_Probe**( ) function. There are no error conditions specific to the Fileset Location Server, although the underlying communications mechanism may encounter errors and signal the errors in the final output parameter.

51.6.4 Obtaining Configuration Information

There is a single function to get cell-wide information from the Fileset Location Server. **VL_GetCellInfo**( ) allows a DFS Cache Manager to learn all the configuration information about a cell simply by contacting one of the Fileset Location Server instances for that cell. It takes no input (other than the RPC connection) and returns a **vlconf_cell** structure containing the information.

The **vlconf_cell** structure contains fields for the name of the cell, the cell's ID, the number of database servers on which Fileset Location Servers are running, and the network addresses and names of those servers. There is a limit on the size of these last two values (see the structure definition, described in the *OSF DCE Application Development Reference*); if the number of servers is greater than the limit imposed by this size, then there is no way to get the information about the additional servers.

51.6.5 Syntax Summary

The functions in this set are

- Fileset Location
  - — **VL_ExpandSiteCookie**( ): Expands compacted site representation
  - — **VL_GetCEntryByID**( ): Gets the compact FLDB entry corresponding to a fileset ID
  - — **VL_GetCEntryByName**( ): Gets the compact FLDB entry corresponding to a fileset name
  - — **VL_GetCNextServersByID**( ): Returns the next set of servers from a compact FLDB entry for a fileset (using the ID for lookup)
  - — **VL_GetCNextServersByName**( ): Returns the next set of servers from a compact FLDB entry for a fileset (using the name for lookup)
  - — **VL_GetEntryByID**( ): Gets the FLDB entry corresponding to a fileset ID
  - — **VL_GetEntryByName**( ): Gets the FLDB entry corresponding to a fileset

- — **VL_GetNextServersByID( )**: Returns the next set of servers for a fileset (using the ID for lookup)
- — **VL_GetNextServersByName( )**: Returns the next set of servers for a fileset (using the name for lookup)

- Fileset Maintenance
  - — **VL_AddAddress( )**: Declares another network address for a server
  - — **VL_AlterServer( )**: Alters server information in the FLDB
  - — **VL_ChangeAddress( )**: Alters a network address for a server
  - — **VL_CreateEntry( )**: Creates a new FLDB entry
  - — **VL_CreateServer( )**: Declares another server machine to the FLDB
  - — **VL_DeleteEntry( )**: Deletes an entry from the FLDB
  - — **VL_GenerateSites( )**: Lists all file servers known to the FLDB
  - — **VL_GetNewVolumeId( )**: Allocates one fileset ID
  - — **VL_GetNewVolumeIds( )**: Allocates several fileset IDs
  - — **VL_GetSiteInfo( )**: Finds out a server's addresses and principal
  - — **VL_GetStats( )**: Gets FLDB and server statistics
  - — **VL_ListByAttributes( )**: Returns selected FLDB entries
  - — **VL_ListEntry( )**: Lists the contents of the FLDB
  - — **VL_Probe( )**: Checks whether the Fileset Location Server is reachable
  - — **VL_ReleaseLock( )**: Releases a previously held lock on an FLDB entry
  - — **VL_RemoveAddress( )**: Removes a network address for a server
  - — **VL_ReplaceEntry( )**: Replaces an FLDB entry for a fileset
  - — **VL_SetLock( )**: Marks an FLDB entry as locked

- Obtaining Configuration Information
  - — **VL_GetCellInfo( )** Gets configuration information for the server's cell

A syntax summary for each **VL...( )** function follows.

```
#include <param.h>
#include <sysincludes.h>
#include <stds.h>
#include <common_data.h>
#include <rpc.h>
#include <pthread.h>
#include <cma_exception.h>
#include <compat.h>
#include <nubik.h>
#include <fldb_proc.h>
#include <flserver.h>
#include <flclient.h>

long VL_AddAddress(
     rpc_binding_handle_t rpcBinding,      /* in */
     struct afsNetAddr *OldAddr,          /* in */
     struct afsNetAddr *AddrToAdd,         /* in */
     error_status_t *theCommStatus)        /* out */

int VL_AlterServer(
     rpc_binding_handle_t rpcBinding,      /* in */
     struct afsNetAddr *Addr,            /* in */
     struct siteAlter *Attrs,         /* in */
     error_status_t *theCommStatus)        /* out */

long VL_ChangeAddress(
     rpc_binding_handle_t rpcBinding,      /* in */
     struct afsNetAddr *OldAddr,          /* in */
     struct afsNetAddr *NewAddr,          /* in */
     error_status_t *theCommStatus)        /* out */

long VL_CreateEntry(
     rpc_binding_handle_t rpcBinding,      /* in */
     struct vldbentry *new,           /* in */
     error_status_t *theCommStatus)        /* out */

long VL_CreateServer(
     rpc_binding_handle_t rpcBinding,      /* in */
     struct siteDesc *FullSiteInfo,       /* in */
     error_status_t *theCommStatus)        /* out */
```

```
long VL_DeleteEntry(
     rpc_binding_handle_t rpcBinding,      /* in */
     struct afsHyper *filesetID,          /* in */
     unsigned long filesetType,          /* in */
     error_status_t *theCommStatus)        /* out */

long VL_ExpandSiteCookie(
     rpc_binding_handle_t rpcBinding,      /* in */
     unsigned long Cookie,               /* in */
     struct siteDesc *FullSiteInfo,        /* out */
     error_status_t *theCommStatus)        /* out */

long VL_GenerateSites(
     rpc_binding_handle_t rpcBinding,      /* in */
     unsigned long startHere,            /* in */
     unsigned long *nextStartP,           /* out */
     struct bulkSites *TheseSites,         /* out */
     long *nSites,                 /* out */
     error_status_t *theCommStatus)        /* out */

long VL_GetCellInfo(
     rpc_binding_handle_t rpcBinding,      /* in */
     struct vlconf_cell *MyCell,          /* out */
     error_status_t *theCommStatus)        /* out */

long VL_GetCEntryByID(
     rpc_binding_handle_t rpcBinding,      /* in */
     struct afsHyper *filesetID,          /* in */
     unsigned long filesetType,          /* in */
     struct compactvldbentry *entry,       /* out */
     error_status_t *theCommStatus)        /* out */

long VL_GetCEntryByName(
     rpc_binding_handle_t rpcBinding,      /* in */
     char *filesetName,                /* in */
     struct compactvldbentry *entry,       /* out */
     error_status_t *theCommStatus)        /* out */
```

```
long VL_GetCNextServersByID(
        rpc_binding_handle_t rpcBinding,      /* in */
        struct afsHyper *filesetID,          /* in */
        unsigned long filesetType,           /* in */
        unsigned long startHere,             /* in */
        unsigned long *nextStartP,            /* out */
        struct compactvldbentry *entry,       /* out */
        unsigned long *flags,              /* out */
        error_status_t *theCommStatus)        /* out */

long VL_GetCNextServersByName(
        rpc_binding_handle_t rpcBinding,      /* in */
        char *filesetName,                /* in */
        unsigned long startHere,             /* in */
        unsigned long *nextStartP,            /* out */
        struct compactvldbentry *entry,       /* out */
        unsigned long *flags,              /* out */
        error_status_t *theCommStatus)        /* out */

long VL_GetEntryByID(
        rpc_binding_handle_t rpcBinding,      /* in */
        struct afsHyper *filesetID,          /* in */
        unsigned long filesetType,           /* in */
        struct vldbentry *entry,             /* out */
        error_status_t *theCommStatus)        /* out */

long VL_GetEntryByName(
        rpc_binding_handle_t rpcBinding,      /* in */
        char *name,                      /* in */
        struct vldbentry *entry,             /* out */
        error_status_t *theCommStatus)        /* out */

long VL_GetNewVolumeId(
        rpc_binding_handle_t rpcBinding,      /* in */
        unsigned long BumpCount,              /* in */
        struct afsNetAddr *serverAddr,         /* in */
        struct afsHyper *NewfilesetID,        /* out */
        error_status_t *theCommStatus)        /* out */
```

```
long VL_GetNewVolumeIds(
     rpc_binding_handle_t rpcBinding,      /* in */
     unsigned long numWanted,            /* in */
     struct afsNetAddr *serverAddr,      /* in */
     struct bulkIds *newIDs,           /* out */
     error_status_t *theCommStatus)       /* out */

long VL_GetNextServersByID(
     rpc_binding_handle_t rpcBinding,      /* in */
     struct afsHyper *filesetID,         /* in */
     unsigned long filesetType,         /* in */
     unsigned long startHere,           /* in */
     unsigned long *nextStartP,          /* out */
     struct vldbentry *entry,           /* out */
     unsigned long *flags,            /* out */
     error_status_t *theCommStatus)       /* out */

long VL_GetNextServersByName(
     rpc_binding_handle_t rpcBinding,      /* in */
     char *name,                   /* in */
     unsigned long startHere,           /* in */
     unsigned long *nextStartP,          /* out */
     struct vldbentry *entry,           /* out */
     unsigned long *flags,            /* out */
     error_status_t *theCommStatus)       /* out */

long VL_GetSiteInfo(
     rpc_binding_handle_t rpcBinding,      /* in */
     struct afsNetAddr *OldAddr,         /* in */
     struct siteDesc *FullSiteInfo,      /* out */
     error_status_t *theCommStatus)       /* out */

long VL_GetStats(
     rpc_binding_handle_t rpcBinding,      /* in */
     struct vldstats *stats,           /* out */
     struct vital_vlheader *vital_header,  /* out */
     error_status_t *theCommStatus)       /* out */
```

```
long VL_ListByAttributes(
      rpc_binding_handle_t rpcBinding,        /* in */
      struct VldbListByAttributes *attributes, /* in */
      unsigned long cookie,               /* in */
      unsigned long *nentries,            /* out */
      bulkentries *blkentries,            /* out */
      unsigned long *nextcookiep,            /* out */
      unsigned long *flagp,               /* out */
      error_status_t *theCommStatus)         /* out */

long VL_ListEntry(
      rpc_binding_handle_t rpcBinding,        /* in */
      unsigned long previous_index,        /* in */
      unsigned long *count,               /* out */
      unsigned long *next_index,           /* out */
      struct vldbentry *entry,            /* out */
      error_status_t *theCommStatus)         /* out */

long VL_Probe(
      rpc_binding_handle_t rpcBinding,        /* in */
      error_status_t *theCommStatus)         /* out */

long VL_ReleaseLock(
      rpc_binding_handle_t rpcBinding,        /* in */
      struct afsHyper *filesetID,          /* in */
      unsigned long filesetType,          /* in */
      long ReleaseType,                          /* in */
      error_status_t *theCommStatus)         /* out */

long VL_RemoveAddress(
      rpc_binding_handle_t rpcBinding,        /* in */
      struct afsNetAddr AddrToRemove,         /* in */
      error_status_t *theCommStatus)         /* out */

long VL_ReplaceEntry(
      rpc_binding_handle_t rpcBinding,        /* in */
      struct afsHyper *filesetID,          /* in */
      unsigned long filesetType,          /* in */
      struct vldbentry *new,              /* in */
      long ReplaceType,                          /* in */
      error_status_t *theCommStatus)         /* out */
```

```
long VL_SetLock(
      rpc_binding_handle_t rpcBinding,      /* in */
      struct afsHyper *filesetID,           /* in */
      unsigned long filesetType,            /* in */
      long fsOper,                          /* in */
      error_status_t *theCommStatus)        /* out */
```

51.7 The FTSERVER Functions: Interacting with the Fileset Server

The Fileset Server (**FTSERVER...( )**) functions are used to perform basic operations on filesets, such as creating, deleting, moving, and replicating them. Unlike the Volume Call (**VC...( )**) functions, these functions do not change the corresponding entries in the Fileset Location Database. To do that, use the **VL...( )** functions, described in the previous section. Unless you need the extra flexibility that the **FTSERVER...( )** functions provide, you should use the **VC...( )** functions instead; the **VC...( )** functions do more for you automatically.

All of the **FTSERVER...( )** functions require authorization except **FTSERVER_ListAggregates( )**, **FTSERVER_ListVolumes( )**, **FTSERVER_AggregateInfo( )**, and **FTSERVER_GetOneVolStatus( )**.

Each **FTSERVER...( )** operation is carried out as a transaction, which can be logically viewed as an ''open fileset'' operation with an exclusive lock on the affected fileset. The term ''transaction'' is somewhat of a misnomer here; Fileset Server transactions are not atomic. The Fileset Server does not use a 2-phase commit mechanism, nor does it attempt to undo changes on abort. Instead, the Filset Server's transactions are simply a convenient way of denoting state information being held by the Fileset Server itself.

There is a large amount of state associated with holding a fileset open. That state is not returned to the client. So, if you open a transaction and do not complete it, the FLDB and file server are not returned to the states they were in before the transaction was opened.

Fileset Server transactions do prevent simultaneous access to a fileset. Opening a transaction signals to other servers that the fileset in question is unavailable while the Fileset Server is manipulating it.

Before calling any function except **FTSERVER_CreateVolume**( ) or **FTSERVER_GetOneVolStatus**( ), you must first open a transaction by calling **FTSERVER_CreateTrans**( ), or the function fails. (**FTSERVER_CreateVolume**( ) opens its own transaction automatically.) You then must explicitly close every transaction, even those created by **FTSERVER_CreateVolume**( ), with **FTSERVER_DeleteTrans**( ), or the affected fileset remains locked even after you are done making changes to it, and other processes will not be able to access it.

Transactions do not time out so long as they are still actively referenced, such as in a long-running operation like a dump. Once they are no longer being actively referenced, they time out after 10 minutes, and the filesets are restored to service. However, applications should not rely on timing out; you should call **FTSERVER_DeleteTrans**( ) as soon as you no longer need a transaction. Timing out is provided to prevent filesets from becoming permanently unavailable if the failure of a machine or other problem interrupts a transaction before **FTSERVER_DeleteTrans**( ) is called.

51.7.1 Basic Transaction Functions

The functions in this section are used to create and delete transactions on filesets. No other functions in this set, except **FTSERVER_CreateVolume**( ) and **FTSERVER_GetOneVolStatus**( ) can be called before a transaction has been opened for the fileset in question.

FTSERVER_CreateTrans( ) creates a new fileset transaction and returns its unique transaction identifier. You must use this ID in all calls to **FTSERVER...**( ) functions that operate on that fileset.

The function accepts a fileset ID, an aggregate ID (where the fileset lives), and a set of options. The options represent the operations that will be performed while the fileset is open and the status codes to report to other processes attempting to access the same fileset. The options are combined via bitwise OR in the options variable.

The following options represent operations that will be performed on the fileset during the transaction:

- **FTSERVER_OP_DELETE( )**
- **FTSERVER_OP_DUMP( )**
- **FTSERVER_OP_RESTORE( )**
- **FTSERVER_OP_CLONE( )**
- **FTSERVER_OP_RECLONE( )**
- **FTSERVER_OP_GETFLAGS( )**
- **FTSERVER_OP_SETFLAGS( )**
- **FTSERVER_OP_SETSTATUS( )**
- **FTSERVER_OP_GETSTATUS( )**

The following options represent the error codes that can be reported:

- **VOLERR_PERS_LOWEST**: The lowest bound for persistent errors.
- **VOLERR_PERS_DELETED**: The fileset has already been deleted/moved.
- **VOLERR_PERS_BADDUMPOPCODE**: A bad opcode was passed to the dump.
- **VOLERR_PERS_BADDUMP**: A bad dump format.
- **VOLERR_PERS_BADFTSOPSVECTOR**: A bad ftserver ops vector was passed in.
- **VOLERR_PERS_REPDESTROY**: The fileset was deleted by the repserver.
- **VOLERR_PERS_DAMAGED**: The fileset is inconsistent.
- **VOLERR_PERS_BADVOLOPSVECTOR**: A bad fileset ops vector was passed in.
- **VOLERR_TRANS_LOWEST**: The lowest bound for transient errors.
- **VOLERR_TRANS_DELETE**: The fileset is being deleted/moved.
- **VOLERR_TRANS_DUMP**: The fileset is being dumped.
- **VOLERR_TRANS_RESTORE**: The fileset is being restored.

- **VOLERR_TRANS_CLONE**: The fileset is being cloned.
- **VOLERR_TRANS_RECLONE**: The fileset is being recloned.
- **VOLERR_TRANS_LIST**: Lists the filesets.
- **VOLERR_TRANS_GETSTATUS**: Gets the status of a fileset.
- **VOLERR_TRANS_CREATEFILESET**: Creates a new fileset.
- **VOLERR_TRANS_RELEASE**: Releases a fileset.
- **VOLERR_TRANS_SETQUOTA**: Sets the quota on a fileset.
- **VOLERR_TRANS_FILESETEXISTS**: A fileset already exists.
- **VOLERR_TRANS_SETFLAGS**: Sets the options on a fileset.
- **VOLERR_TRANS_MOVE**: The fileset is being moved.
- **VOLERR_TRANS_SETSTATUS**: Sets the status on a fileset.
- **VOLERR_TRANS_COPYCLONE**: Copies the clone to a new location.
- **VOLERR_TRANS_FORWARD**: Copies the fileset to a different server or aggregate.
- **VOLERR_TRANS_BACKUPDUMP**: Dumps a fileset to tape.
- **VOLERR_TRANS_BACKUPRESTORE**: Restores a fileset from tape.
- **VOLERR_TRANS_REPGETSTATUS**: The repserver gets the status on a fileset.
- **VOLERR_TRANS_REPSETSTATUS**: The repserver sets the status on a fileset.
- **VOLERR_TRANS_REPSETFILESETVERSION**: The repserver sets the fileset version.
- **VOLERR_TRANS_REPCLONE**: The repserver clones a fileset.
- **VOLERR_TRANS_REPEDITSTATUS**: The repserver edits fileset status.
- **VOLERR_TRANS_REPCLEARSTATUS**: The repserver clears fileset status.
- **VOLERR_TRANS_REPUNCLONE**: The repserver unclones the fileset.

- **VOLERR_TRANS_REPSWITCHFILESETS**: The repserver switches filesets.
- **VOLERR_TRANS_REPGETFILESETCHANGES**: The repserver gets fileset changes.
- **VOLERR_TRANS_REPFORWARD**: The repserver forwards the fileset.
- **VOLERR_PERS_NOMEM**: No more memory available.
- **VOLERR_PERS_IO**: Pipe I/O failure.
- **VOLERR_TRANS_QUOTA**: The fileset is over quota.
- **VOLERR_TRANS_HIGHEST**: The upper bound for transient errors.

Additional **VOLERR** codes are allocated as spares to provide for future expansion.

If no option is set, the fileset is available.

Any number of operation options can be specified, but only one error code may be given. The operation options are used by DFS to determine what operations may occur concurrently with this one; the error code is returned to any process that attempts to open the fileset for an incompatable operation.

FTSERVER_DeleteTrans( ) takes one argument (in addition to the RPC connection): a transaction ID, and deletes the transaction associated with it. Once you delete a transaction, you will not be able to perform **FTSERVER...**( ) operations on the fileset until you create another transaction.

51.7.2 Creating, Deleting, and Cloning Filesets

The functions in this section create, delete, clone, and update clones of filesets. They all apply only to DCE LFS filesets. All except **FTSERVER_CreateVolume**( ) require that a transaction be open for the fileset.

FTSERVER_CreateVolume( ) is used to create a fileset. Like **VC_CreateVolume**, it takes an RPC connection, an aggregate ID, a fileset name, and a fileset ID. In addition, you must specify the fileset type and the ID of the parent fileset. This parent fileset has nothing to do with where the fileset will eventually be mounted on the file server; rather, this refers to relationships with other filesets. For example, a copy of a fileset has that fileset as its parent. If you are creating a copy, use the fileset ID for the source as the parent; if you are creating a new fileset, pass a 0 (zero) ID. Finally, **FTSERVER_CreateVolume**( ) takes an option argument like the one taken by **FTSERVER_CreateTrans**( ).

FTSERVER_CreateVolume( ) returns a transaction ID, which should be treated like transaction IDs returned by **FTSERVER_CreateTrans**( ).

The types of filesets are as follows:

VOLTIX_TO_VOLTYPE(RWVOL): read/write fileset

VOLTIX_TO_VOLTYPE(ROVOL): read-only fileset

VOLTIX_TO_VOLTYPE(BACKVOL): backup fileset

The fileset will not actually be accessible until **FTSERVER_SetFlags**( ) or **FTSERVER_SetStatus**( ) is called to bring it online and the transaction ends. Be sure to call **FTSERVER_DeleteTrans**( ) to close the transaction, or the fileset will remain unavailable until the transaction times out.

The following example shows the creation of a read/write fileset.

```
char *fileset = "foo";          /* arbitrary name */
long AggID = 0;                 /* arbitrary disk no. */
struct afsHyper filesetID, parentID;
unsigned long transactionID;
int code;                       /* return code */
error_status_t error;

bzero(&filesetID, sizeof(filesetID));
bzero(&parentID, sizeof(parentID));
bzero(&transactionID, sizeof(transactionID));

/* PARENT FILESET */

/* get ID */
VL_GetNewVolumeId(Rpc_Handle, 1, &filesetID, &error);
if(error)
  printf("Got an RPC error while getting fileset ID! \n");
```

```
/* create fileset */
code = FTSERVER_CreateVolume(Rpc_Handle, AggID, fileset,
                             VOLTIX_TO_VOLTYPE(RWVOL),
                             &parentID,&filesetID,
                             &transactionID);
if(code != 0)
  printf("Got an RPC error while creating parent fileset! \n");

/* bring fileset online */
code = FTSERVER_SetFlags(Rpc_Handle, transactionID, VOL_RW);
if(code != 0)
  printf("Got an RPC error while bringing fileset online!\n");

/* delete transaction */
code = FTSERVER_DeleteTrans(Rpc_Handle, transactionID);
if(code != 0)
  printf("Got an RPC error while deleting transaction! \n");
```

FTSERVER_DeleteVolume( ) deletes a fileset from the aggregate. It takes two arguments: an RPC connection and the transaction ID that was obtained from **FTSERVER_CreateTrans( )** or **FTSERVER_CreateVolume( )**.

FTSERVER_Clone( ) creates a clone (read-only or backup copy) of a read/write fileset. It takes an RPC connection and transaction ID and the ID, type, and name of the copy.

To make administration easier, the name of the clone should be related to the name of the source fileset. For example, a backup clone of fileset **foo** could be named **foo.backup**.

Once you have made a clone, you may wish to update it from time to time. This is more efficient than making new clones and then deleting the old ones, as only the information that has changed will need to be updated.

FTSERVER_ReClone( ) brings a clone up to date with the source fileset. It takes three arguments: an RPC connection, the transaction ID of the source fileset, and the fileset ID of the clone.

There is no explicit ‘‘unclone’’ function to delete a clone. **FTSERVER_DeleteVolume( )**, if given a clone as a parameter, removes that clone only.

51.7.3 Getting and Modifying Fileset Status

The functions in this section are used to find and modify the status of filesets; for instance, to move them online or offline, or to examine or change fileset header fields.

You must explicitly set the status of a fileset when you create it, and manually change it, when using the **FTSERVER...()** functions. The status determines the conditions under which the fileset can be accessed, and what types of operations can be done.

FTSERVER_SetFlags() is used to set the current status of a fileset.

The status is a bitwise OR of values from the following list:

- **VOL_RW** indicates that the fileset allows both read and write access.
- **VOL_READONLY** indicates that the fileset is read-only. Writes to this fileset will fail.
- **VOL_DELONSALVAGE** indicates that the fileset is not usable; it will be deleted upon reboot. This is an intermediate state used to guarantee correctness for large operations such as cloning a fileset.
- **VOL_OFFLINE** indicates that the fileset is offline. Reads and writes to this fileset will fail.
- **VOL_BUSY** indicates that the fileset is temporarily unavailable.
- **VOL_OUTOFSERVICE** indicates that the fileset is not usable; it will not be deleted upon reboot.
- **VOL_DEADMEAT** indicates that the fileset is in the process of being deleted.
- **VOL_OK** is the bitwise negation of **VOL_BUSY, VOL_OFFLINE, VOL_DELONSALVAGE,** and **VOL_OUTOFSERVICE.** It is used to mark a fileset as usable.
- **VOL_REPFIELD** is used for replication and should never be modified by applications.

FTSERVER_GetFlags() gets the current availability of a fileset. It takes two arguments: an RPC connection and the transaction ID of the fileset in question, and returns a mask derived from the options described previously for **FTSERVER_SetFlags()**.

In addition to the current availability, there are other types of status information that can be set. **FTSERVER_SetStatus**( ) sets this status information. It takes four arguments: an RPC connection, the transaction ID for the fileset, a mask, and a structure representing the status to set. The status is an **ftserver_status** structure.

```
struct ftserver_status {
    ftserver_status_static vss;
    ftserver_status_dynamic vsd;
    }

struct ftserver_status_static {
    struct afsHyper volId;
    struct afsHyper parentId;
    struct afsTimeval cloneTime;
    struct afsTimeval vvCurrentTime;
    struct afsTimeval vvPingCurrentTime;
    unsigned long type;
    unsigned long accStatus;
    unsigned long accError;
    unsigned long states;
    unsigned long reclaimDally;
    long static_spare1;
    long static_spare2;
    long static_spare3;
    long static_spare4;
    long static_spare5;
    long static_spare6;
    long static_spare7;
    char volName[FTSERVER_MAXFSNAME];
    char static_cspares[16];
    }

struct ftserver_status_dynamic {
    struct afsTimeval creationDate;
    struct afsTimeval updateDate;
    struct afsTimeval accessDate;
    struct afsTimeval backupDate;
    struct afsTimeval copyDate;
    struct afsHyper volversion;
    struct afsHyper backupId;
struct afsHyper cloneId;
struct afsHyper llBackId;
struct afsHyper llFwdId;
long fileCount;
long maxQuota;
long minQuota;
long size;
long owner;
long unique;
long index;
long rwindex;
long backupIndex;
long parentIndex;
long cloneIndex;
long dynamic_spare1;
long dynamic_spare2;
long dynamic_spare3;
long dynamic_spare4;
long dynamic_spare5;
long dynamic_spare6;
char statusMsg[128];
char dynamic_cspares[16];
}
```

The fields of these structures are as follows:

- **ftserver_status_static**
  - **volId**: The ID of the fileset, which should be unique throughout the cell.
  - **parentId**: The fileset ID of the read/write source, if this is a read-only or backup clone.
  - **cloneTime**: The time the last clone was made of this fileset.
  - **vvCurrentTime**: The most recent time that the fileset version number was known to be current on the read/write site. Applications should not modify this field.
  - **vvPingCurrentTime**: The most recent time that a read-only site tried to contact a read/write site to determine how current the fileset version number there is. Applications should not modify this field.

- **type**: The fileset's type.
- **accStatus**: The access status on the fileset; the status will be one of the codes described under **FTSERVER_CreateTrans( )**. This field should never be modified by an application program.
- **accError**: The access error on a fileset; this will be one of the codes described under **FTSERVER_CreateTrans( )**. This field should never be modified by an application program.
- **states**: The status flag, as set by **FTSERVER_SetFlags( )**.
- **reclaimDally**: The time, in seconds, between keep-alive messages sent by the Cache Manager.
- **static_spare1** through **static_spare3**: Spares reserved for future use.
- **volName**: The name of the fileset.
- **static_cspares**: Reserved for future use.

- **ftserver_status_dynamic**
  - **creationDate**: The date of the fileset's creation.
  - **updateDate**: The timestamp of the last modification by a user.
  - **accessDate**: The last access time by a user.
  - **backupDate**: The date at which the last backup clone was made.
  - **copyDate**: The time that this copy of the fileset was created.
  - **volVersion**: The current version number of the fileset.
  - **backupId**: The fileset ID of the latest backup version of this fileset.
  - **cloneId**: The fileset ID of the clone.
  - **llBackId** and **llFwdId**: Links all related filesets in a doubly linked list; used to find the correct fileset to unclone when deleting a fileset.
  - **fileCount**: The number of files in the fileset.
  - **maxQuota**: The maximum amount of disk space the fileset is allowed to take up, expressed in units of 1024 bytes. A value of 0 (zero) means there is no limit.

— **minQuota**: The amount of disk space guaranteed to be available for the fileset, expressed in units of 1024 bytes. Other filesets on the partition are prevented from growing if their growth would impinge on this limit.

— **size**: The size of the fileset, in units of 1024 bytes.

— **owner**: The DFS ID of the owner of the fileset.

— **unique**: The uniquifier, a value combined with other IDs to guarantee uniqueness (*do not* modify this).

— **index**, **rwindex**, **backupIndex**, **parentIndex**, and **cloneIndex**: Information about where other related filesets are located within an aggregate; for the remote application user they are meaningless. They cannot be set remotely.

— **dynamic_spare1** through **dynamic_spare6**: Spares reserved for future use.

— **statusMsg**: A message field used by low-level code to transmit error messages back up to the caller (across the FTServer-to-kernel interface). Applications should not use this field.

— **dynamic_cspares**: Reserved for future use.

The mask is the bitwise OR of identifiers for the fields to be set. The fields are defined by the following constants:

- **VOL_STAT_VOLNAME**
- **VOL_STAT_VOLID**
- **VOL_STAT_VERSION**
- **VOL_STAT_UNIQUE**
- **VOL_STAT_OWNER**
- **VOL_STAT_TYPE**
- **VOL_STAT_STATES**
- **VOL_STAT_ACCSTATUS**
- **VOL_STAT_BACKUPID**
- **VOL_STAT_PARENTID**
- **VOL_STAT_CLONEID**

- **VOL_STAT_LLBACKID**
- **VOL_STAT_LLFWDID**
- **VOL_STAT_CREATEDATE**
- **VOL_STAT_UPDATEDATE**
- **VOL_STAT_ACCESSDATE**
- **VOL_STAT_COPYDATE**
- **VOL_STAT_FILECOUNT**
- **VOL_STAT_MAXQUOTA**
- **VOL_STAT_MINQUOTA**
- **VOL_STAT_SIZE**
- **VOL_STAT_INDEX**
- **VOL_STAT_BACKVOLINDEX**
- **VOL_STAT_STATUSMSG**
- **VOL_STAT_CLONETIME**
- **VOL_STAT_VVCURRTIME**
- **VOL_STAT_VVPINGCURRTIME**
- **VOL_STAT_ACCERROR**
- **VOL_STAT_BACKUPDATE**
- **VOL_STAT_RECLAIMDALLY**

FTSERVER_GetStatus( ) returns the status information listed previously for a fileset. It takes two arguments: an RPC connection and the transaction ID for the fileset in question, and returns a **ftserver_status** structure containing the information.

FTSERVER_GetStatus( ), like most calls in this section, requires that **FTSERVER_CreateTrans**( ) and **FTSERVER_DeleteTrans**( ) be called, thus locking the fileset while the status check is in progress. An easier solution for those cases where you wish to quickly check the status of a single fileset is **FTSERVER_GetOneVolStatus**( ). This function does *not* require the two transaction calls, and **does not** require any special authentication to use. It takes an RPC handle, the identifier for the fileset to check, and an aggregate ID, and returns the same status as **FTSERVER_GetStatus( )** does in the output buffer.

51.7.4 Dumping, Restoring, and Moving Filesets

This section describes functions for dumping and restoring filesets and moving dumps around.

Dumping refers to the process of copying a fileset, or portion thereof, to a byte stream, which is suitable for storage on tape or as an intermediate format when moving filesets between file server machines that store filesets in different formats. Restoring refers to conversion of a byte stream back into the appropriate fileset format and placement back into the file system. Dumping is different from backing up because the result is not a mountable fileset. In order to use a backed up fileset, you merely need to bring it online, which is a fast operation. In order to use a dumped fileset, you need to restore it. However, dumping is necessary for offline storage.

FTSERVER_Dump( ) dumps a fileset as a byte stream. It takes four arguments: an RPC connection, the transaction ID for the affected fileset, a specification of what to dump, and an RPC pipe to accept the output. The date is represented as an **ftserver_Date** structure, described previously.

The **ftserver_Date** structure contains a mask slot that indicates which of the other two fields, the date or version number, to use. A mask of 1 means to use the date, and a mask of 2 means to use the version. A mask of 0 means to use neither. All other values are undefined.

The dump data itself is returned as an output pipe parameter.

A fileset dump consists of a fileset header dump followed by a set of individual vnode dumps, followed by an end-of-fileset opcode. Individual vnode dumps consist of a vnode start opcode, multiple vnode descriptor opcodes, and an end-of-vnode opcode.

Complete information about these opcodes is given in the *OSF DCE Application Development Reference*. The information that is represented follows:

- Maximum uniquifier, the value of the highest uniquifier that has been used.

- Maximum quota, the maximum number of 1024-byte units that the fileset can grow to (0 (zero) means no limit).
- Minimum quota, the minimum number of 1024-byte units that the fileset has reserved (0 (zero) means no reservation).
- Actual disk usage, in 1024-byte units.
- The number of files in the fileset.
- Fileset owner ID.
- Creation date, when the fileset ID was first used by this fileset.
- Last access date.
- Last update date.
- Message of the day, a string that gives human readable information about the particular state of this fileset; that is, why the fileset is unavailable, when it will be back, and so on.
- Volume version.
- Fileset type from which the dump was made.
- Vnode type.
- Vnode link count.
- Vnode data version.
- Vnode fileset version.
- Vnode access time.
- Vnode modify time (from the client rather than from server).
- Vnode change time.
- Vnode true modify time (from the server rather than from client).
- Vnode group owner.
- Vnode author (the person who last modified the fileset).
- Vnode owner.
- Vnode UNIX mode bits.
- Vnode access control lists.

- Vnode size.
- Vnode data.

If you are planning to immediately restore the dumped fileset elsewhere, for efficiency reasons, you should call **FTSERVER_Forward()** instead of calling **FTSERVER_Dump()** followed by **FTSERVER_Restore()**.

FTSERVER_Restore() restores a DCE LFS fileset from a dump. It takes three arguments: an RPC connection, the transaction ID for the fileset, and a third argument that is reserved for future use (use 0 (zero)).

Dumping and restoring filesets are CPU- and network-intensive processes, so you should be careful about their use.

FTSERVER_Forward() dumps a fileset from one file server and restores it on another. The destination must be a DCE LFS fileset. It takes five arguments: an RPC connection, a transaction ID for the source fileset, an **ftserver_Date** structure describing what kind of dump to make (this is the same as the corresponding argument to **FTSERVER_Dump()**), the address of the destination machine in an **ftserver_dest** structure, and a transaction ID for the fileset on the destination machine.

You must first create a fileset on the destination machine, and then open transactions on both the source and destination filesets.

51.7.5 Enumerating Filesets, Aggregates, and Transactions

The functions in this section supply information about filesets, aggregates, and current transactions. Because these functions do not directly manipulate filesets, they do not require open transactions.

FTSERVER_ListAggregates() returns the names, devices, IDs, and types of all valid aggregates on a server. It takes an RPC connection on the server in question. It returns an array of **ftserver_aggrList** structures. This structure has four main fields that correspond to the four pieces of information returned for each aggregate. The array contains up to **FTSERVER_MAXAGGR()** structures.

FTSERVER_ListAggregates() uses iterators to return the aggregates.

To get more specific information about a single aggregate, use **FTSERVER_AggregateInfo()**. This function returns detailed information about a single aggregate, unlike **FTSERVER_ListAggregates()**, which returns general information about all of them. **FTSERVER_AggregateInfo( )** takes two arguments: an RPC connection and an aggregate ID, and returns a **ftserver_aggrInfo** structure. The structure contains the following information:

- The name of the partition on which the aggregate resides
- The new device name on which the aggregate is mounted
- The type of aggregate (for example, UFS or DCE LFS)
- The number of usable 1024-byte units allocated
- The total space free, in 1024-byte units
- The reserved space, in 1024-byte units

If you want to list the contents of an aggregate, use **FTSERVER_ListVolumes( )**. This function returns a list of all filesets on a given aggregate. It takes three arguments: an RPC connection, an aggregate ID, and an iterator. It returns two values: the iterator to use for the next call and the actual information. As with other functions that use an iterator, you should give an initial value of 0 (zero) and use the output iterator as the next input iterator value until you have all of the data.

This operation has substantial overhead because it reads the fileset headers for all the filesets on that aggregate on the disk. If you are really only interested in a small number of filesets, you would be much better off using **FTSERVER_GetStatus( )** or **FTSERVER_GetOneVolStatus( )** to get information on those specific filesets only. For information about the information returned, see Section 51.7.3.

Finally, **FTSERVER_Monitor( )** is provided to return a list of all active transactions involving a given file server. This function allows processes to monitor the progress of a transaction. The function takes one argument, an RPC connection for the server, and returns an array of **ftserver_trans** structures containing the results.

The following information about each transaction is returned:

- The transaction ID
- The open fileset's aggregate
- The open fileset's ID
- The descriptor of the open fileset

- The time the transaction was last active
- The time the transaction started
- The transaction error code, if any errors have occurred
- The transaction's status bits

51.7.6 Syntax Summary

The **FTSERVER...()** functions are as follows:

- Transactions
  - **FTSERVER_CreateTrans()**: Opens a transaction on a fileset
  - **FTSERVER_DeleteTrans()**: Closes a transaction on a fileset
- Creating, Deleting, and Cloning (LFS filesets only)
  - **FTSERVER_Clone()**: Creates a read-only copy of a fileset
  - **FTSERVER_CreateVolume()**: Creates a new fileset
  - **FTSERVER_DeleteVolume()**: Deletes a fileset
  - **FTSERVER_ReClone()**: Brings a clone up to date with the read/write copy
- Modifying Fileset Status
  - **FTSERVER_GetFlags()**: Gets the current availability of a fileset
  - **FTSERVER_GetOneVolStatus()**: Gets the full status of a fileset without requiring explicit transactions
  - **FTSERVER_GetStatus()**: Gets the full status of a fileset
  - **FTSERVER_SetFlags()**: Sets the current availability of a fileset
  - **FTSERVER_SetStatus()**: Sets the various status information for a fileset
- Dumping, Restoring, and Moving
  - **FTSERVER_Dump()**: Dumps a fileset to a character stream
  - **FTSERVER_Forward()**: Dumps a fileset from one file server and restores it to another (DCE LFS filesets only)

- — **FTSERVER_Restore( )**: Restores a fileset from a character stream (DCE LFS filesets only)

- Enumerating Filesets, Aggregates, and Transactions
  - — **FTSERVER_AggregateInfo( )**: Provides specific information about an aggregate
  - — **FTSERVER_ListAggregates( )**: Identifies all valid aggregates on a server
  - — **FTSERVER_ListVolumes( )**: Lists the filesets on a file server
  - — **FTSERVER_Monitor( )**: Identifies all active transactions involving a given file server

A syntax summary for each **FTSERVER...( )** function follows.

```
#include <param.h>
#include <fcntl.h>
#include <compat.h>
#include <ftserver_proc.h>
#include <queue.h>
#include <sysincludes.h>
#include <fldb_data.h>
#include <flserver.h>
#include <ftserver.h>
#include <ftserver_trans.h>
#include <rpc.h>

long FTSERVER_AggregateInfo(
      rpc_binding_handle_t RpcBinding,      /* in */
      unsigned long aggrID,                 /* in */
      struct ftserver_aggrInfo *aggrDesc)   /* out */

long FTSERVER_Clone(
      rpc_binding_handle_t RpcBinding,      /* in */
      unsigned long transID,                /* in */
      unsigned long newType,                /* in */
      char *newName,                        /* in */
      struct afsHyper *newfilesetID)        /* inout */
```

```
long FTSERVER_CreateTrans(
    rpc_binding_handle_t RpcBinding,      /* in */
    struct afsHyper *filesetIDp,          /* in */
    unsigned long aggregateID,            /* in */
    unsigned long flags,                  /* in */
    unsigned long *transID)               /* out */

long FTSERVER_CreateVolume(
    rpc_binding_handle_t RpcBinding,      /* in */
    long aggregateID,                     /* in */
    char *name,                           /* in */
    unsigned long type,                   /* in */
    unsigned long flags,                  /* in */
    struct afsHyper *parentID,            /* in */
    struct afsHyper *filesetID,           /* inout */
    unsigned long *transID)               /* out */

long FTSERVER_DeleteTrans(
    rpc_binding_handle_t RpcBinding,      /* in */
    unsigned long transID)                /* in */

long FTSERVER_DeleteVolume(
    rpc_binding_handle_t RpcBinding,      /* in */
    unsigned long transID)                /* in */

long FTSERVER_Dump(
    rpc_binding_handle_t RpcBinding,      /* in */
    unsigned long transID,                /* in */
    struct ftserver_Date *dumpDate,       /* in */
    struct pipe_t *dataPipeP)             /* out */

long FTSERVER_Forward(
    rpc_binding_handle_t RpcBinding,      /* in */
    unsigned long fromTrans,              /* in */
    struct ftserver_Date *fromDate,       /* in */
    struct ftserver_dest *destAddress,    /* in */
    unsigned long destTrans)              /* in */
```

```
long FTSERVER_GetFlags(
    rpc_binding_handle_t RpcBinding,      /* in */
    unsigned long transID,                /* in */
    unsigned long *flags)                 /* out */

long FTSERVER_GetOneVolStatus(
    rpc_binding_handle_t RpcBinding,      /* in */
    struct afsHyper *filesetID,           /* in */
    unsigned long aggrId,                 /* in */
    unsigned long spare1,                 /* in */
    ftserver_status *status)              /* out */

long FTSERVER_GetStatus(
    rpc_binding_handle_t RpcBinding,      /* in */
    unsigned long transID,                /* in */
    struct ftserver_status *status)       /* out */

long FTSERVER_ListAggregates(
    rpc_binding_handle_t RpcBinding,      /* in */
    ftserver_iterator *inCookie,          /* in */
    ftserver_iterator *outCookie,         /* out */
    struct ftserver_aggrEntries *aggrEnts)  /* out */

long FTSERVER_ListVolumes(
    rpc_binding_handle_t RpcBinding,      /* in */
    unsigned long aggrID,                 /* in */
    ftserver_iterator *inCookie,          /* in */
    ftserver_iterator *outCookie,         /* out */
    struct ftserver_statEntries *resultEnts[]) /* out */

long FTSERVER_Monitor(
    rpc_binding_handle_t RpcBinding,      /* in */
    struct ftserver_transEntries *transP)  /* out */

long FTSERVER_ReClone(
    rpc_binding_handle_t RpcBinding,      /* in */
    unsigned long transID,                /* in */
    struct afsHyper *cloneID)             /* inout */
```

```
long FTSERVER_Restore(
     rpc_binding_handle_t RpcBinding,      /* in */
     unsigned long transID,             /* in */
     unsigned long flags,             /* in */
     pipe_t *dataPipeP)               /* in */

long FTSERVER_SetFlags(
     rpc_binding_handle_t RpcBinding,      /* in */
     unsigned long transID,             /* in */
     unsigned long flags)             /* in */

long FTSERVER_SetStatus (
     rpc_binding_handle_t RpcBinding,      /* in */
     unsigned long transID,             /* in */
     unsigned long mask,              /* in */
     struct ftserver_status *status,       /* in */
     unsigned long spare1)             /* in */
```

Chapter 52

The BOS Server: Monitoring File Server Processes

The BOS Server interface (**BOSSVR...**( )) functions maintain all necessary DFS processes and software on DFS File Server machines. At the time the File Server machine is initially configured, a system administrator sets parameters for these operations, probably interactively using the BOS command suite; the functions in this chapter allow the manipulation of this information. Care must be taken in doing so, however, because such changes impact everyone who uses the server machines on which the operations are performed.

The BOS interface is useful for writing administrative monitoring tools, such as monitoring whether a server is running or shutdown is reported. In addition, you can obtain statistics about the time of last restart and other similar information, which could be used in system monitoring applications.

You can use this interface to write a graphical command interface, wherein an administrator may use a point-and-click interface to describe what servers should be running on which machines.

The functions in this set fall into several categories:

- Process Monitoring: Monitoring processes and restarting them upon failure.

- Server Key Maintenance: Maintaining the set of authentication keys used by the various DFS servers.

- Binary Maintenance: Installing new binaries on the server machines and scheduling server restarts to pick up the newly installed binaries.
- Authorization: Ensuring that only authorized users reconfigure the servers. Authentication of these users is handled by the DCE Security Service.

Most of the functions in this set require that the caller be on the server's BOS user list. The ones that *do not* are

- **BOSSVR_GetCellName( )**
- **BOSSVR_GetStatus( )**
- **BOSSVR_EnumerateInstance( )**
- **BOSSVR_GetInstanceInfo( )**
- **BOSSVR_GetCellName( )**
- **BOSSVR_GetInstanceParm( )**
- **BOSSVR_ListSUsers( )**
- **BOSSVR_GetDates( )**
- **BOSSVR_GetRestartTime( )**
- **BOSSVR_GetLog( )**

Note: There are situations in which **BOSSVR_GetLog( )** does require the caller to be on the BOS user list; see the discussion of this function later in this chapter for more information.

52.1 Common Arguments

As with the other RPC interfaces, the functions in this set have two common arguments: an RPC handle that is supplied when the remote connection is first established, and an output parameter giving RPC error status.

For the purpose of the examples in this chapter, we assume that the programmer has created a function to obtain an RPC handle and set the variable *Rpc_Handle* to its output. For information about creating RPC handles, see Part 3 of this guide.

All structure types described in this chapter are also defined as types; for example, **struct bossvr_key** and **bossvr_key** are equivalent. The former form is used in this document.

52.2 Configuration Files

The BOS Server uses a configuration file *dcelocal*/**var/dfs/BosConfig**, which records all of the processes to be run when the BOS Server starts up. The BOS process itself should *not* be included in this file; doing this will result in process recursion and as a result performance penalties and possible system failures.

Some of the functions described in this chapter, and some of the commands in the BOS command suite, modify the **BosConfig** file. This file should not be edited manually; use one of the two interfaces (application or command suite) to do this. The file format is internal to the BOS Server and is subject to change.

52.3 Process Monitoring

The process monitoring functions are responsible for bnode maintenance. These functions create and delete bnodes, examine and change their properties, and start and stop process execution on a server.

52.3.1 Bnodes

A bnode is a structure that gives the static characteristics of a process; that is, the characteristics that apply to every instance of a process with that name. Bnodes have types, and for any given type of bnode, the BOS Server has embedded code that defines how those types of processes are managed. There are two types of bnodes: simple bnodes and cron bnodes.

All bnodes are parameterizable at bnode creation time. Bnode parameters are character strings, and each type of bnode has its own specific set of parameters.

A simple bnode manages a single process, and manages it with a simple goal: to keep the process running. If a process managed by a simple bnode crashes, any resulting core file is saved for future reference and the process is restarted. If the process restarts too often in a given period, the bnode instance is marked as damaged and is shut down pending an operator's intervention. Most bnode instances in a File Server machine installation are simple bnodes.

A simple bnode has only one parameter, which is the command string to be used to start the process. The command string is parsed by **execve**, and thus cannot contain the usual shell-provided expansions, such as those containing asterisks, but otherwise resembles a typical UNIX command. This command is reexecuted whenever the process needs to be restarted.

A cron bnode manages a single process that is to be run either exactly once or periodically. Essentially, this is a generalization of the simple bnode that runs a process, but leaves it shut down after it exits until the next time that the process is scheduled to be executed. After a process's last scheduled execution (assuming that it *has* a last execution), a cron bnode deletes itself; this generally only occurs with once-only bnodes.

A cron bnode has two parameters. The first is the command to be used to start the process to be managed, and the second is a string that can be parsed as a periodically recurring date, giving the times that the process should be run.

The date format is one of the following (all are strings):

- **never**
- **now**
- A day of the week and a time, separated by spaces, such as ''wed 16:00.'' Such processes run once per week. The day must come first and begin with a lowercase character. Use either the entire day name or the first three letters.
- A time, in 24-hour notation. The process runs once per day.

52.3.2 Creating and Deleting Bnodes

Because processes are represented by bnodes, and most of the time the BOS Server is used to manage processes, the most fundamental BOS functions are those to create bnodes.

You can use the function **BOSSVR_CreateBnode()** to create a bnode. This function creates a bnode instance that represnts a new process to be run. The name you give that instance is used to refer to this bnode until you delete it. In addition to the name, you specify the type (**simple** or **cron**) and further parameters, the nature of which depends on the type.

The following two examples create two bnodes, a simple one called **test1** and a cron one called **test2**. The simple bnode simply runs a program called **test1**; the cron bnode runs a program called **test2** every Saturday at 23:00.

```
handle_t Rpc_Handle;
error_status_t error;
int bos_error = 0;
char *errorStringP = "";

/* assign Rpc_Handle */

bos_error = BOSSVR_CreateBnode(Rpc_Handle, "simple",
                               "test1",
                               "/afs/tr/usr/bin/test1",
                               "", "", "", "", "", &error);
if(error != error_status_ok)
  printf("Got an error! (test1)\n\tError text: %s\n",
         error_text(error));
 else printf("First call completed.\n");

bos_error = BOSSVR_CreateBnode(Rpc_Handle, "cron",
                               "test2",
                               "/afs/tr/usr/bin/test2",
                               "sat 23:00", "", "", "",
                               "", &error);
if(error != error_status_ok)
  printf("Got an error! (test2)\n\tError text: %s\n",
         error_text(error));
 else printf("Second call completed.\n");
```

```
if ((error == error_status_ok) && (bossvr_error == 0)) {
    printf("Test Completed OK\n");
   }
```

To delete a bnode, first shut down all its processes (described later in this chapter) and then call **BOSSVR_DeleteBnode( )**. This function takes the instance name, which you supplied to **BOSSVR_CreateBnode**. If you only want to kill processes, and do not want to permanently remove those processes from the BOS Server start-up file, *do not* use this function. Operations on processes are described later in this chapter.

52.3.3 Changing and Examining Bnode Instances

The BOS Server provides a number of functions for examining or changing various information associated with bnodes. This information includes:

- Current run status
- Permanent run status
- Bnode type (cannot be changed once bnode is created)
- Process start and stop times
- bnode parameters (cannot be changed once bnode is created)
- Lists of all known bnodes

The most commonly accessed information about a bnode is its run status. The run status is, as the name implies, the status of the process associated with the bnode; that is, running or stopped. A process could be stopped for a number of reasons deriving from system problems that are beyond the scope of this guide.

The function **BOSSVR_GetStatus( )** returns, in an output parameter, the run status of the named bnode. This status is a long integer; the following constants represent possible values:

- **BSTAT_SHUTDOWN**: The process is not currently running.
- **BSTAT_NORMAL**: The process is running normally.

- **BSTAT_STARTINGUP**: The BOS Server is still starting up the process; likely, checking again will result in a normal status.
- **BSTAT_SHUTTINGDOWN**: A kill signal has been sent to the process, but it has not exited yet.

The functions **BOSSVR_SetStatus( )** and **BOSSVR_SetTStatus( )** set the status of a bnode to one of the previously listed options. The difference between the two is that **BOSSVR_SetStatus( )** records the change in the BOS initialization file, so that the change will be in effect in future startups of the BOS Server, while **BOSSVR_SetTStatus( )** changes the status only in the current environment. (The **T** stands for temporary.) Of course, a change made with **BOSSVR_SetStatus( )** can be changed by a later call to that function, or can be temporarily overridden.

There is other useful information about a bnode, such as the bnode type and the parameters supplied when it was created. The function **BOSSVR_GetInstanceInfo( )** provides such information about a bnode. It provides two output parameters, the type and a **bossvr_status** structure. The **bossvr_status** structure contains a number of fields, including the following:

- **procStartTime:** The time this process last started
- **procStarts:** The total number of times this process has been started
- **lastAnyExit:** The last time this process exited
- **lastErrorExit:** The last time this process got an error
- **errorCode:** The last error code returned by a process

To only obtain the parameters that are associated with a bnode, the function **BOSSVR_GetInstanceParm( )** should be used instead of **BOSSVR_GetInstanceInfo( )**. **BOSSVR_GetInstanceParm( )** takes a bnode instance name and the parameter number (the first one is number 0 (zero)). Simple bnodes have only one parameter, while cron bnodes have two.

All of the preceding functions require the bnode instance name. One way to get a list of all names is to diligently keep track of all names as they are created. Another way is to use the function **BOSSVR_EnumerateInstance( )**. This function can be used to enumerate all instance names at a server; however, because the number of instances is unknown, this function cannot return all of them at once. Thus, a single call to this function produces one instance. You use this function by continuing to call it until there are no more instances, at which point the function signals this by returning an error code. You supply an instance number when calling the function; start with 0 (zero) and increment by one on each call.

The following code stores the names of all the bnode instances into the array **SavedInstances**.

```
handle_t Rpc_Handle;
long j;
bossvr_out_string instance;
char SavedInstances[256];
error_status_t error;
long code;

/* assign Rpc_Handle */

for(j=0; (code != BZDOM) && (error == 0)
          && (j < 256); j++)
   { code = BOSSVR_EnumerateInstance(Rpc_Handle, j,
                                      &instance, &error);
     if(error != error_status_ok)
        printf("Got an error!\nError text: %s\n",
               error_text(error));
     else {
       printf("Call completed.\n");
       strcpy(SavedInstances[j], instance.theString);}
    }

if (j == 256 && code != BZDOM)
   printf("Too many instances!  Give me more room! \n");
else printf("%d instances.\n", j);
```

52.3.4 Stopping and Starting Bnode Instances

The BOS Server provides functions to start, stop, and restart bnode instances. Most of these functions take a bnode instance name; some operate on all bnodes on the server. All of them take the standard two RPC arguments: the RPC handle and the RPC error status, which were described at the beginning of this chapter. Some of the following functions read the initialization file, and some take arguments specifying the processes to manipulate.

The following functions manage bnode instances:

- **BOSSVR_ShutdownAll( )** shuts down all of the bnode instances at a server. It does not change the BOS Server initialization file, so all BOS processes start up again when BOS is restarted.
- **BOSSVR_Restart( )** restarts a single bnode instance. If that instance is already running, the function stops it and restarts it. This function does not alter the initialization file.
- **BOSSVR_RestartAll( )** restarts all bnode instances on a server. Note that **BOSSVR_RestartAll( )** restarts all processes marked in the initialization file as runnable; processes that are currently running are shut down and restarted. This function does not alter the initialization file.

  BOSSVR_RestartAll( ) does not restart the BOS Server itself. To do that, you must call **BOSSVR_ReBossvr( )**.
- **BOSSVR_ReBossvr( )** restarts the BOS Server and then restarts all of the bnode instances at a server that are marked as runnable in the initialization file. It is like **BOSSVR_RestartAll( )** except that it also restarts the BOS Server itself. This is useful in cases where the BOS Server binary has changed and you want to start using the new version immediately.
- **BOSSVR_StartupAll( )** starts all of the bnode instances at a server that are not running and that are marked as runnable in the BOS Server's initialization file. This call does not change the initialization file, or do anything to those bnode instances marked in the file as not runnable. It also does not restart processes that are already running; if you want to do that, you must use **BOSSVR_RestartAll( )** or **BOSSVR_ReBossvr()**.

  It is not necessary to call **BOSSVR_ShutdownAll( )** before calling **BOSSVR_StartupAll( )**.
- **BOSSVR_WaitAll( )** waits until all of the bnode instances that are changing state from running to not running, or vice versa, complete their state changes. The call does not return until all state changes are complete. When it is important to have the status of all bnode instances up-to-date before proceeding, this function should be called immediately after all the status-changing calls.

- **BOSSVR_SetRestartTime**( ) and **BOSSVR_GetRestartTime**( ) are used to set and retrieve the restart times associated with BOS processes and the BOS Server itself. It is useful to occasionally restart BOS processes to pick up new versions of binaries (for example, ones that have been installed on the File Server machine since the BOS Server started running), or to clean up after core leaks.

52.4 Server Key Maintenance

The server key maintenance functions maintain the authentication keys used by DFS servers. More information about server keys can be found in Part 6 of this guide.

BOSSVR_AddKey( ) adds a key to the server key database. It takes a principal name, the key version number, the new password to use, and a flag that indicates whether the change is local or global. The password is a string.

The change can be either local to the BOS Server machine, or it can be exported to the Registry Server as well. If the value of this flag is nonzero, the change is local only.

You can also add a key by calling **BOSSVR_GenerateKey**( ), which generates a key randomly so that you do not have to provide one. There is no option for specifying a local or global change; all changes are global.

BOSSVR_ListKeys( ) returns the key given an iteration value. Like other iterative functions, you continue calling this, incrementing the iterator, until the end of the list is signaled. In this case, the error flag **BZDOM** will be returned when there are no more keys to return.

BOSSVR_ListKeys( ) returns the key version number, the key, and a checksum value for each key. If the server is not running in **NoAuth** mode, only the checksum (rather than the key) is returned.

BOSSVR_DeleteKey( ) removes the key with a given key version number from the key database.

BOSSVR_GarbageCollectKeys( ) removes all keys that are no longer used for a given principal. It operates on the local key file only.

52.5 Installing Binaries

There are several functions for managing system binaries and reverting to older versions when necessary.

The BOS Server maintains up to three copies of a program: the current version, a backup (**.BAK**) version, and an old (**.OLD**) version, which is older than the **.BAK** version. Generally, the **.BAK** version of a program is the previously installed version of the program. The **.OLD** version is created from the previous **.BAK** version, but only if this version is at least 7 days old or there is no already existing **.OLD** version. The BOS Server attempts to make sure that the **.OLD** version is at least a week old in case it is necessary to revert to a previous version of the software; if the BOS Server did not enforce this and several changes were made in a short amount of time, there would be no prior version to use.

The function **BOSSVR_Install**( ) installs a new version of a binary on a server. It takes the pathname (on the destination machine) of a program to install, a pipe to the source, and other information about the binary. **BOSSVR_Install**( ) moves the current **.BAK** file to the **.OLD** file if it is at least a week old. It then moves the current file to the **.BAK** file and installs the new file. Any processes running the old binary at the time must be stopped and restarted to get the new version. This can be done by the application unless special privileges are needed to start the processes.

The information you must supply is the destination pathname, the size of the binary, a set of flags (reserved for future use), the last modification date of the file, and a pipe that points to the actual file data. The pipe is of type **pipe_t**; pipes are discussed in more detail in Part 3 of this guide.

If the binary cannot be installed for some reason, such as disk space limitations, the following happens:

1. **BOSSVR_Install**( ) empties the input pipe to prevent RPC errors.
2. The function deletes the temporary file that was being used to collect the new binary.
3. The function returns the value **BZINSTALLFAILED**.

The **.BAK** and **.OLD** files are not moved around until the entire binary has been successfully received in the temporary file.

Sometimes it is necessary to undo an installation; for example, if something went wrong in the installation. In that case, use the function **BOSSVR_UnInstall( )**, which reverts to the previous version of a binary. It renames any existing **.BAK** file to be the currently installed program, and renames any existing **.OLD** file to be the new **.BAK** file. If there is no **.BAK** file but there is a **.OLD** file, it installs the **.OLD** file instead. (This situation can only exist if something unusual is done, such as manually operating on these files.) Note that the binary being uninstalled will be deleted from the server; if you want to keep a copy, you need to make the copy before calling this function.

The age of binaries can be checked with **BOSSVR_GetDates( )**, which provides the modification times for a binary, its **.BAK** version, and its **.OLD** version.

Occasionally, old versions of binaries on a server need to be removed. (Lack of disk space sometimes requires this.) The function **BOSSVR_Prune( )** can be used to delete prior versions and core files.

The way to specify pruning actions is to pass the bitwise OR of the following flags.

- **BOSSVR_PRUNEOLD( )** deletes all **.OLD** files.
- **BOSSVR_PRUNEBAK( )** deletes all **.BAK** files.
- **BOSSVR_PRUNECORE( )** deletes all saved core files.

Note that **BOSSVR_Prune( )** cannot be used to selectively remove files of a given type; the function removes *all* files of a given type (**.BAK**, **.OLD**, or core). If more selectivity is required, files must be removed manually.

52.6 Authorization Issues

The functions in this section relate to authorization issues. Specifically, they manipulate the administrative user lists.

All administrative list manipulation functions take a pathname argument that is the name of the user list file. If the filename contains any slashes, it is used as an absolute pathname. Otherwise, the path *dcelocal*/**var/dfs/** is prepended to the name and the resultant filename is used as the name of the file containing the user list.

All of these functions also have arguments that are reserved for future use. They are strings, so pass "" (the empty string) for these arguments.

BOSSVR_AddSUser( ) adds a user to the administrative user list contained in the named file. It takes a filename and a DCE Registry identity for the user to be elevated to administrative status.

BOSSVR_AddSUser( ) also takes a flag that indicates whether to create the file for the administrative user list if it does not already exist (in other words, if there are no other administrative users in the file specified). If the flag is 0 (zero), the file is not created and an error is returned; otherwise, the file is created.

BOSSVR_DeleteSUser( ) removes a user from a named administrative user list. The user's DCE principal (DCE Registry identity) is passed. This function also takes a flag that indicates whether the file should be deleted if this administrative user is the last one. A value of 0 (zero) means to keep the file; otherwise it is deleted.

BOSSVR_ListSUsers( ) enumerates the administrative users. It is an iterative function; keep calling it, incrementing the iterator, until the value **BZDOM** is returned to signal the end of the list.

52.7 Miscellaneous Functions

Some BOS functions cannot easily be categorized:

- **BOSSVR_GetCellName( )** provides the name of the current cell (a string). This is similar to the **pioctl** call **VIOCGETWSCELL** but the **pioctl** call finds the cell associated with the workstation on which the call is made, while **BOSSVR_GetCellName( )** instead finds the cell associated with the BOS Server.
- **BOSSVR_Exec( )** executes a shell command in a **bosserver** subprocess. The command runs as root. The call does not return until the subprocess exits. Any output generated by the subprocess is discarded.
- Most BOS Server processes write log files. **BOSSVR_GetLog( )** provides access to this data; it returns a pipe (see Part 3 of this guide) to which this output is fed. You must process the data in the pipe yourself. Logs are typically stored in *dcelocal*/**var/dfs/adm**. Because the BOS Server runs as root, logs in this directory are accessible to any BOS

Server process. If this function is called for a file not in this directory, however, the caller must be in the BOS Server's user list. See the *OSF DCE Administration Guide* for more information about the user list.

52.8 Syntax Summary

The BOS Server functions are as follows:

- Process Monitoring
  - — **BOSSVR_CreateBnode( )**: Creates a new process bnode instance
  - — **BOSSVR_DeleteBnode( )**: Deletes a bnode instance
  - — **BOSSVR_EnumerateInstance( )**: Enumerates all bnode instances on a server
  - — **BOSSVR_GetInstanceInfo( )**: Gets a basic bnode instance description
  - — **BOSSVR_GetInstanceParm( )**: Gets the parameters for a bnode instance
  - — **BOSSVR_GetRestartTime( )**: Gets the BOS Server restart process time
  - — **BOSSVR_GetStatus( )**: Gets the run status of a bnode instance
  - — **BOSSVR_ReBossvr( )**: Restarts all servers, including the BOS Server
  - — **BOSSVR_Restart( )**: Restarts a given BOS process
  - — **BOSSVR_RestartAll( )**: Restarts all BOS processes
  - — **BOSSVR_SetRestartTime( )**: Sets BOS Server process restart times
  - — **BOSSVR_SetStatus( )**: Sets the run status of a bnode instance
  - — **BOSSVR_SetTStatus( )**: Temporarily sets the run status of a bnode instance
  - — **BOSSVR_ShutdownAll( )**: Shuts down all BOS processes

  - **BOSSVR_StartupAll( )**: Starts all BOS processes
  - **BOSSVR_WaitAll( )**: Waits for all bnodes to stabilize their status
- Server Key Maintenance
  - **BOSSVR_AddKey( )**: Adds a new server key to the database
  - **BOSSVR_DeleteKey( )**: Deletes the named server key
  - **BOSSVR_GarbageCollectKeys( )**: Gets rid of obsolete keys
  - **BOSSVR_GenerateKey( )**: Generates and adds a key to the server key database
  - **BOSSVR_ListKeys( )**: Lists all known server keys
- Binary Maintenance
  - **BOSSVR_GetDates( )**: Gets the modification times of a program and its backups
  - **BOSSVR_Install( )**: Installs a server binary on a server
  - **BOSSVR_Prune( )**: Deletes old and unnecessary binaries
  - **BOSSVR_UnInstall( )**: Reverts to an older copy of a server binary
- Authorization
  - **BOSSVR_AddSUser( )**: Adds a user to the named administrative user list
  - **BOSSVR_DeleteSUser( )**: Deletes a user from the named administrative user list
  - **BOSSVR_ListSUsers( )**: Gets the list of all administrative users
  - **BOSSVR_SetNoAuthFlag( )**: Sets the flag that controls whether DFS servers check authorization
- Miscellaneous
  - **BOSSVR_Exec( )**: Executes a command from the BOS Server
  - **BOSSVR_GetCellName( )**: Gets the server's cell name
  - **BOSSVR_GetLog( )**: Retrieves a text log file

Full syntax for each BOS Server function follows.

```
#include<bbos_ncs_interface.h>

const BOSSVR_BSSIZE = 256;  /* bosserver string length */

long BOSSVR_AddKey(
    handle_t bosserverBinding,            /* in */
    char prinNameP[BOSSVR_BSSIZE],        /* in */
    long kvno,                    /* in */
    char passwdP[BOSSVR_BSSIZE],          /* in */
    long localOnly,                 /* in */
    error_status_t *theCommStatus)        /* out */

long BOSSVR_AddSUser(
    handle_t bosserverBinding,            /* in */
    char filename[BOSSVR_BSSIZE],         /* in */
    char typeStr[BOSSVR_BSSIZE],          /* in */
    char name[BOSSVR_BSSIZE],             /* in */
    char permStr BOSSVR_BSSIZE],          /* in */
    long createFile,                /* in */
    error_status_t *theCommStatus)        /* out */

long BOSSVR_CreateBnode(
    handle_t bosserverBinding,            /* in */
    char type[BOSSVR_BSSIZE],             /* in */
    char instance[BOSSVR_BSSIZE],         /* in */
    char p1[BOSSVR_BSSIZE],               /* in */
    char p2[BOSSVR_BSSIZE],               /* in */
    char p3[BOSSVR_BSSIZE],               /* in */
    char p4[BOSSVR_BSSIZE],               /* in */
    char p5[BOSSVR_BSSIZE],               /* in */
    char p6[BOSSVR_BSSIZE],               /* in */
    error_status_t *theCommStatus)        /* out */

long BOSSVR_DeleteBnode(
    handle_t bosserverBinding,            /* in */
    char instance[BOSSVR_BSSIZE],         /* in */
    error_status_t *theCommStatus)        /* out */
```

```
long BOSSVR_DeleteKey(
      handle_t bosserverBinding,           /* in */
      char prinNameP[BOSSVR_BSSIZE],        /* in */
      long kvno,                      /* in */
      error_status_t *theCommStatus)        /* out */

long BOSSVR_DeleteSUser(
      handle_t bosserverBinding,           /* in */
      char filename[BOSSVR_BSSIZE],         /* in */
      char typeStr[BOSSVR_BSSIZE],          /* in */
      char name[BOSSVR_BSSIZE],             /* in */
      long removeFile,                 /* in */
      error_status_t *theCommStatus)        /* out */

long BOSSVR_EnumerateInstance(
      handle_t bosserverBinding,           /* in */
      long instanceNum,                /* in */
      bossvr_out_string *result            /* out */
      error_status_t *theCommStatus)        /* out */

long BOSSVR_Exec(
      handle_t bosserverBinding,           /* in */
      char cmd[BOSSVR_BSSIZE],               /* in */
      error_status_t *theCommStatus)        /* out */

long BOSSVR_GarbageCollectKeys(
      handle_t bosserverBinding,           /* in */
      char prinNameP[BOSSVR_BSSTRING],        /* in */
      error_status_t *theCommStatus)        /* out */

long BOSSVR_GenerateKey(
      handle_t bosserverBinding,           /* in */
      char prinNameP[BOSSVR_BSSIZE],        /* in */
      long kvno,                      /* in */
      error_status_t *theCommStatus)        /* out */

long BOSSVR_GetCellName(
      handle_t bosserverBinding,           /* in */
      bossvr_out_string *nameP,             /* out */
      error_status_t *theCommStatus)        /* out */
```

```
long BOSSVR_GetDates(
     handle_t bosserverBinding,          /* in */
     char path[BOSSVR_BSSIZE],             /* in */
     long *newtime,                   /* out */
     long *baktime,                   /* out */
     long *oldtime,                   /* out */
     error_status_t *theCommStatus)        /* out */

long BOSSVR_GetInstanceInfo(
     handle_t bosserverBinding,          /* in */
     char instance[BOSSVR_BSSIZE],          /* in */
     bossvr_out_string *type,            /* out */
     struct bossvr_status *status        /* out */
     error_status_t *theCommStatus)        /* out */

long BOSSVR_GetInstanceParm(
      handle_t bosserverBinding,          /* in */
     char instance[BOSSVR_BSSIZE],          /* in */
     long num,                      /* in */
     bossvr_out_string *parm,             /* out */
     error_status_t *theCommStatus)        /* out */

long BOSSVR_GetLog(
     handle_t bosserverBinding,          /* in */
     char name[BOSSVR_BSSIZE],             /* in */
     pipe_t *thePipeP,                  /* out */
     error_status_t *theCommStatus)        /* out */

long BOSSVR_GetRestartTime(
     handle_t bosserverBinding,          /* in */
     long type,                      /* in */
     struct bossvr_netKTime *restartTime,   /* out */
     error_status_t *theCommStatus)        /* out */

long BOSSVR_GetStatus(
     handle_t bosserverBinding,          /* in */
     char instance[BOSSVR_BSSIZE],          /* in */
     long *status,                    /* out */
     bossvr_out_string *statdescrP,         /* out */
     error_status_t *theCommStatus)        /* out */
```

```
long BOSSVR_Install(
     handle_t bosserverBinding,          /* in */
     char path[BOSSVR_BSSIZE],           /* in */
     long size,                  /* in */
     long flags,                 /* in */
     long date,                  /* in */
     pipe_t *thePipeP,               /* in */
     error_status_t *theCommStatus)      /* out */

long BOSSVR_ListKeys(
     handle_t bosserverBinding,          /* in */
     char prinNameP[BOSSVR_BSSIZE],          /* in */
     long an,                    /* in */
     long *kvno,                   /* out */
     struct bossvr_key *key,           /* out */
     struct bossvr_keyInfo *keyinfo,       /* out */
     error_status_t *theCommStatus)      /* out */

long BOSSVR_ListSUsers(
     handle_t bosserverBinding,          /* in */
     char filename[BOSSVR_BSSIZE],         /* in */
     long an,                    /* in */
     bossvr_out_string *nameP,           /* out */
     bossvr_out_string *typeStrP,        /* out */
     bossvr_out_string *permStrP,         /* out */
     error_status_t *theCommStatus)      /* out */

long BOSSVR_Prune(
     handle_t bosserverBinding,          /* in */
     long flags,                 /* in */
     error_status_t *theCommStatus)      /* out */

long BOSSVR_ReBossvr(
     handle_t bosserverBinding,          /* in */
     error_status_t *theCommStatus)      /* out */

long BOSSVR_Restart(
     handle_t bosserverBinding,          /* in */
     char instance[BOSSVR_BSSIZE],         /* in */
     error_status_t *theCommStatus)      /* out */
```

```
long BOSSVR_RestartAll(
     handle_t bosserverBinding,             /* in */
     error_status_t *theCommStatus)            /* out */

long BOSSVR_SetNoAuthFlag(
     handle_t bosserverBinding,             /* in */
     long flag,                  /* in */
     error_status_t *theCommStatus)            /* out */

long BOSSVR_SetRestartTime(
     handle_t bosserverBinding,             /* in */
     long type,                  /* in */
     struct bossvr_netKTime *restartTime,  /* in */
     error_status_t *theCommStatus)            /* out */

long BOSSVR_SetStatus(
     handle_t bosserverBinding,             /* in */
     char instance[BOSSVR_BSSIZE],             /* in */
     long status,                  /* in */
     error_status_t *theCommStatus)            /* out */

long BOSSVR_SetTStatus(
     handle_t bosserverBinding,             /* in */
     char instance[BOSSVR_BSSIZE],             /* in */
     long status,                  /* in */
     error_status_t *theCommStatus)            /* out */

long BOSSVR_ShutdownAll(
     handle_t bosserverBinding,             /* in */
     error_status_t *theCommStatus)            /* out */

long BOSSVR_StartupAll(
     handle_t bosserverBinding,             /* in */
     error_status_t *theCommStatus)            /* out */

long BOSSVR_UnInstall(
     handle_t bosserverBinding,             /* in */
     char path[BOSSVR_BSSIZE],             /* in */
     error_status_t *theCommStatus)            /* out */
```

```
long BOSSVR_WaitAll(
    handle_t bosserverBinding,          /* in */
    error_status_t *theCommStatus)      /* out */
```

Index

Notes

Notes

Notes

Notes

OPEN SOFTWARE FOUNDATION™

INFORMATION REQUEST FORM

Please send to me the following:

( ) OSF™ Membership Information

( ) OSF™ DCE License Materials

( ) OSF™ DCE Training Information

Contact Name ______________________________

Company Name ______________________________

Street Address ______________________________

Mail Stop ______________________________

City ______________ State ____ Zip __________

Phone ______________ FAX ______________

Electronic Mail ______________________________

MAIL TO:

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

Attn: OSF™ DCE

For more information about OSF™ DCE call OSF Direct Channels at 617 621 7300.

OSF™DCE

OSF™DCE Application Development Guide

TITLES IN THE OSF™DCE SERIES:

Introduction to OSF™DCE

OSF™DCE User's Guide and Reference

OSF™DCE Administration Guide
— Introduction
— Core Components
— Extended Services

OSF™DCE Administration Reference

OSF™DCE Application Development Guide

OSF™DCE Application Development Reference

Application Environment Specification (AES)
Distributed Computing

Printed in the U.S.A.

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

Prentice Hall, Inc.

ISBN 0-13-643826-1

90000

9 780136 438267

DISTRIBUTED SYSTEMS