MIT/LCS/TR-326

ORPHAN DETECTION IN THE ARGUS SYSTEM

Edward Franklin Walker

This blank page was inserted to preserve pagination.

Orphan Detection in the Argus System

by

Edward Franklin Walker

June 1984

This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Orphan Detection in the Argus System

by

Edward Franklin Walker

Abstract

In a distributed system, an activity running at one node can request another node to perform some service. This request results in an activity being created at the latter node to perform the requested service. The former node may then crash, destroying the activity that requested the service, but leaving behind the activity performing the service. Such surviving activities are known as *orphans* [Nelson81]. Orphans are undesirable since they waste resources and can view inconsistent data.

This thesis presents an algorithm that detects and exterminates orphans before they can view inconsistent data. The algorithm has the desirable property that no non-orphans are mistakenly identified as orphans and exterminated. An underlying premise of the algorithm is that orphan detection and extermination should delay normal computation as little as possible. The algorithm works by piggybacking information concerning orphans on various messages that flow about the system.

The algorithm piggybacks an impractical amount of data on messages. The main contribution of this thesis is the development of a method called *deadlining*. This method works in conjunction with the algorithm to detect orphans before they view inconsistent data, while substantially reducing the amount of data piggybacked on messages. An analytic model is used to predict the actual performance of deadlining.

This report is a minor revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on May 25, 1984 in partial fulfillment of the requirements for the Degree of Master of Science.

Thesis Supervisor: Barbara H. Liskov
Title: Professor of Computer Science

Keywords: atomic actions, orphans, distributed systems, remote procedure call

Acknowledgments

Barbara Liskov, my thesis advisor, has my gratitude for suggesting an intriguing thesis topic, listening to my ideas, making many invaluable suggestions, and promptly reading the drafts of this thesis.

I am indebted to many of my fellow graduate students, for both their technical expertise and for making the Lab a more pleasant place to work. My roommates and fellow computer science graduate students, Tom Wanuga and Jim Restivo, have been steady sources of encouragement and friendship. Brian Oki has also been quite helpful and friendly. Jennifer Lundelius proved to be a valuable source of information about clock synchronization. Bill Weihl, Maurice Herlihy, and Bob Scheifler answered many of my questions about Argus. Brian Coan organized a most enjoyable ski trip. Everyone in John's, Barbara's, and Nancy's groups has my thanks.

Gail Rubin has my love for her affection and emotional support while this thesis was underway.

And I must not forget to thank my mother for everything over the years.

Table of Contents

Table of Figures

Figure 8-4: Counter-example snapshot four 115

Chapter One

Introduction

A distributed computer system is composed of a group of *nodes* connected by a communications network. Distinct nodes do not share memory; they can communicate with each other only by sending messages over the network. Components of such a system can fail -- nodes can crash and messages can be lost. A primary goal of distributed computing is that the system, as a whole, should be robust to such failures.

In a distributed system, an activity running at a node can request another node to perform some service. This request results in an activity being created at the latter node to actually perform the requested service. However, the former node can crash, destroying the activity that requested the service, but leaving behind the activity performing the service. Such surviving activities are known as *orphans* [Nelson81]. Orphans can be created in more subtle ways than we have indicated here; the body of the thesis contains a more detailed discussion.

Orphans cause two undesirable problems. First, they waste resources -- the work of the orphaned activity above is futile since the requesting activity that would benefit from this work has perished. Second, orphans can view inconsistent data, i.e., data in a state it could not be in if the activity in question were not an orphan. Permitting activities to view inconsistent data imposes a burden on programmers, who are then obliged to write programs that behave properly even in the presence of inconsistencies. Therefore both problems make it desirable to exterminate orphans. If the latter problem is to be completely remedied, orphans must be exterminated before they view inconsistent data.

This thesis presents an algorithm that detects and exterminates orphans before

they can view inconsistent data. This algorithm has the desirable property that no non-orphans are mistakenly identified as orphans and exterminated. An underlying premise of the algorithm is that orphan detection and extermination should delay normal computation as little as possible. The algorithm works by piggybacking information on various messages that flow about the system. This information is used to detect orphans, and is guaranteed to arrive in time to prevent orphans from viewing inconsistent data. Goree [Goree83] has verified the correctness of a portion of this algorithm.

The algorithm, in fact, piggybacks a large amount of data on messages. In order for the algorithm to be considered practical, it is necessary to devise some means for reducing this information flow. The main contribution of this thesis is the development of a method called *deadlining*. This method works in conjunction with the algorithm to detect orphans before they view inconsistent data, while reducing the amount of data piggybacked on messages. Deadlining also preserves the property that no non-orphans are exterminated mistakenly.

The rest of this chapter is devoted to a discussion of the organization of the subsequent chapters of this thesis.

In Chapter 2, Argus is discussed. The orphan detection algorithm discussed in this thesis was developed expressly for the Argus system, although we believe it can be used in other distributed systems as well. Argus [Liskov83] is a programming language designed to support applications that run in a distributed computer system. The Argus system is the extensive run-time support system required to run an Argus program. Argus has three features that suit the distributed programming task well: *long-lived data*, *remote procedure calls*, and *atomic actions*. Data objects in Argus can survive much longer than the duration of the execution of a program, unlike data in most common programming languages. Argus programs typically modify long-lived data when run; they typically neither create nor destroy such data. A remote procedure call is very much like a familiar procedure call, except that the called

procedure executes on a different machine from the caller's. Atomic actions have the property of either running successfully to completion or having no effect upon system state.

Chapter 3 contains a discussion detailing what orphans are and why they are a problem. The basic orphan detection algorithm is discussed in Chapter 4. Chapters 5 and 6 discuss deadlining. Chapter 7 presents an analytical analysis of the effectiveness of deadlining, in terms of reducing the amount of data added to messages.

A discussion of related work and our conclusions appear in Chapter 8. Others have encountered orphans in their proposed systems and formulated their own orphan detection algorithms. We are aware of no actual implementation of an orphan detection algorithm. Allchin [Allchin83] presents an orphan detection algorithm similar to ours, but his algorithm is incorrect, and he presents no mechanism like deadlining to make his algorithm practical. Nelson [Nelson81] discusses several orphan detection strategies, all of which seem practical. Nelson, however, does not share our premise that orphan detection should avoid delaying normal computation.

Chapter Two

Argus

Argus [Liskov82] [Liskov83] [Liskov84] is a programming language and system designed to support programs that run in a network of computers. The Argus system is the extensive run-time support system required for Argus programs. This chapter presents a discussion of Argus that provides the necessary background for the material in later chapters.

2.1 Guardians

In Argus, a distributed program is composed of a group of modules called *guardians*. A guardian runs at a single computer in a network. From this point onward, a computer in a network is referred to as a *node*. A node can contain several guardians, but any single guardian is completely contained at some node. A guardian encapsulates and controls access to one or more resources, e.g., databases or devices. Every guardian provides a set of operations called *handlers*. These handlers provide access to the encapsulated resources of a guardian. When a guardian wishes to access some other guardian's encapsulated resource, it can only do so by calling one of that guardian's handlers.

Handler calls and handler operations in Argus are semantically and syntactically similar to procedure calls and procedures, respectively, in more familiar programming languages. Each handler operation provides a set of formal input parameters and a set of formal output parameters. A handler call specifies the actual input parameters to be transmitted to the handler and what to do with the returned output values from the handler. The arguments (both input and output) of handler calls are passed by value; it is impossible to pass a reference to an object in a handler call. Since guardians usually reside at distinct nodes, handler calls usually involve

sending messages across the network.

Internally, a guardian contains data objects and processes. The processes execute handler calls (a new process is spawned for each incoming handler call) and perform background housekeeping tasks. Some of the data objects make up the *state* of the guardian; these objects are shared by the processes running in the guardian. Other objects are strictly local to some individual process and disappear when that process terminates.

A guardian's state consists of *stable* and *volatile* objects. Stable objects are maintained in volatile memory but are periodically recorded on stable storage devices. Stable storage devices survive node failures (with a very high probability); volatile memory does not. Volatile objects are maintained only in volatile memory. When a guardian's node crashes, the volatile objects are lost but the stable objects survive.

A guardian is capable of surviving crashes at its node given that an appropriate collection of its objects are stable. Although a guardian does lose the work in progress at the time of a crash, the results of past completed work are not lost. After a crash and subsequent recovery of a guardian's node, the Argus support system together with the guardian's user-written *recovery code* recreate the guardian's state using the stable objects as they were when last recorded on stable storage. Of course, this means that the volatile objects must be derivable from the stable objects. Once the guardian's state has been restored, the guardian can resume background tasks and can start processing new incoming handler calls.

In this thesis, the terms *local* and *remote* are used with respect to guardians. That is, "local" means "at the same guardian" and "remote" means "at some other guardian."

2.2 Atomic Actions

Although a distributed program might consist of a single guardian, more typically it will be composed of several guardians, and these guardians will reside at different nodes. In a system composed of many guardians, the state of the system is distributed -- it is partitioned among the different guardians. This distributed state must be maintained consistently in the presence of concurrent activities in the system and in spite of the fact that the hardware components on which the system runs can fail independently. To provide consistency of distributed data, Argus supports *atomicity*.

An activity in Argus attempts to examine and transform some objects in the distributed state from their current (initial) states to new (final) states, with any number of intermediate state changes. Two properties distinguish an activity as being atomic: *indivisibility* and *recoverability*. Indivisibility means that the execution of one activity never appears to overlap (or contain) the execution of any other activity. If the objects being modified by one activity are observed over time by another activity, the latter activity will either always observe the initial states or always observe the final states of those objects -- never the intermediate states. Recoverability means that the overall effect of the activity is all-or-nothing -- either all of the objects remain in their initial state, or all change to their final state. If a failure occurs while an activity is running, it must be possible either to complete the activity or to restore all objects to their initial states.

Such an atomic activity is called an *action*. An action may complete either by *committing* or *aborting*. When an action aborts, the effect is as if the action had never begun; all modified objects are restored to their initial states. When an action commits, all modified objects take on their new states.

Atomic objects are special objects that support the indivisibility and recoverability properties of actions. To prevent one action from observing or interfering with the intermediate states of another action, accesses to an atomic

object are synchronized via read-write locking. To permit the modifications of an action to an atomic object to be undone, multiple *versions* of an atomic object are maintained. Atomicity is guaranteed only when the objects shared by actions are atomic objects. In this thesis, atomic objects are referred to as "objects." When an object that is not atomic is spoken of in this thesis, it will be explicitly referred to as a "non-atomic object." In addition, we assume in this thesis that atomic objects are always stable objects.

Atomic objects are based on a fairly simple read-write locking model. Before an action uses an atomic object, it must acquire the object's lock in the appropriate mode. The usual locking rules apply -- multiple readers are allowed but readers exclude writers and a writer excludes readers and other writers. When a write lock is obtained, a *version* (i.e., a copy) of the object is made, and the action operates on this version. If ultimately the action commits, this version will be retained, and the old original version discarded. If the action aborts, its version will be discarded, and the old version retained. All locks on atomic objects acquired by an action are held until the completion of that action, a simplification of standard two-phase locking [Eswaren76], in order to avoid the problem of cascading aborts [Wood80].

Not all objects are atomic objects, since the properties of synchronization and recovery are somewhat expensive and are not required in many situations. For example, objects that are entirely local to a single action do not require these properties.

Actions provide a straightforward way to deal with hardware failures at a node -- a failure forces the node to crash, which in turn forces all the actions there to abort. As was mentioned above, the stable state of guardians is stored on stable storage devices. However, stable objects are not copied to stable storage until actions commit. Versions of an atomic object made for a running action and information about locks and processes are kept in volatile memory. When the node crashes this volatile information is lost, effectively terminating all actions running there, releasing

all locks, and discarding all versions.

2.3 Nested Actions

Actions have been presented thus far as monolithic entities. In fact, it is useful to break down actions into parts; to this end Argus provides hierarchically structured, *nested* actions. Such a nested action is also called a *subaction*. An action may contain any number of subactions. Similarly, a subaction itself can contain any number of subactions. This nesting can go arbitrarily deep.

An action that is contained in no other action is called a *topaction*. The term *action* from this point hence will refer to either a topaction or a subaction.

We apply the usual terminology for hierarchical relationships to nested actions. Hence we talk about the *parent* action of a given subaction, or the *children* of a given action. We can also refer to an action's *descendants* or *ancestors*. An action is defined to be its own ancestor and descendant. When we desire to refer to an action's descendants (or ancestors) excluding the action itself, we shall refer to that action's *proper* descendants (or *proper* ancestors).

The fact that a topaction might have several children subactions cannot be observed from the outside; i.e., the overall action still satisfies the atomicity properties. Also, subactions appear as atomic activities with respect to their sibling subactions. Subactions can commit and abort independently, and a subaction can abort without forcing its parent action to abort. However, the commit of a subaction is conditional -- even if a subaction commits, aborting its parent action will undo the results of the subaction. Further, object versions are written to stable storage only when topactions commit.

Subactions are a mechanism for coping with failures. Since a subaction aborts independently of its parent, the failure of a child can be confined to that child. If a child cannot perform its work for some reason and aborts, the parent can then take

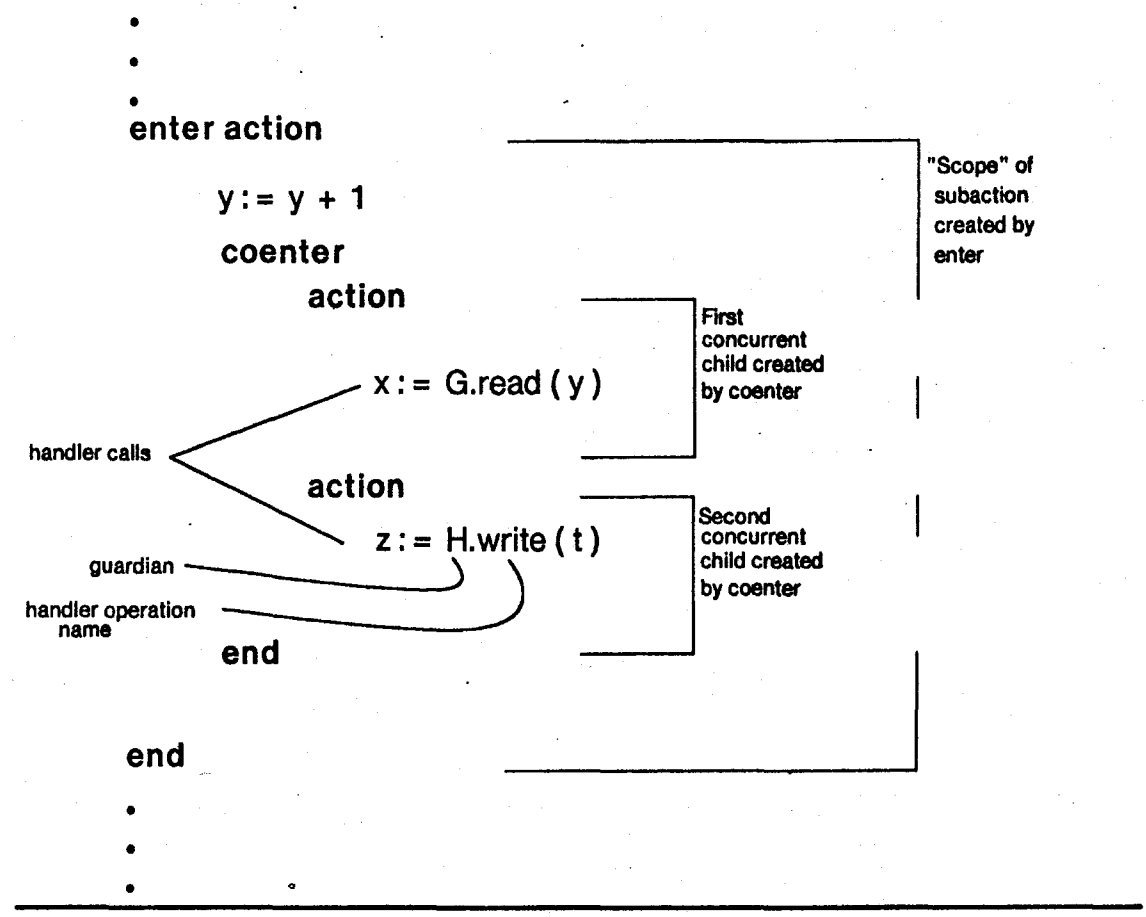
appropriate steps to try to work around the problem. This failure isolation provides the means to improve program robustness and to make error recovery more straightforward.

The locking rules are a bit more complicated for nested actions than for flat actions. To keep locking rules from getting too complex, a parent action is not allowed to run concurrently with its children -- a parent is suspended while it has active (i.e. non-completed) children. The rule for read locks is extended so that an action may obtain a read lock on an object if every action holding a write lock on that object is an ancestor. An action may obtain a write lock on an object provided every action holding a read or write lock on that object is an ancestor. When a subaction commits, its locks are inherited by its parent (i.e. the parent becomes the holder of the locks); when a subactions aborts, its locks and versions are discarded.

We say that an action B has *committed up to* ancestor action A if B and every ancestor of B up to but not necessarily including A has committed. This "committed up to" terminology will be used throughout this thesis.

There are three means of creating subactions in Argus. The first two are the **enter** and **coenter** statements. The **enter** statement is used to create a single child subaction of the action that executes the **enter**. The child runs at the guardian of the parent. The **coenter** creates some specified number of children subactions of the action that executes the **coenter**. The created children run concurrently with each other; each child runs at the guardian of the parent. The final means of creating a subaction is through handler calls. A handler call creates a child subaction of the action executing the handler call. The child runs at the called guardian. The handler call is the only means of creating a subaction that does not run at its parent's guardian. Handler calls are discussed in greater detail in the next section. An annotated piece of Argus code is given in Figure 2-1 that illustrates the use of **enter**, **coenter**, and handler calls.

An action resides at a single guardian in Argus -- an action is created, run, and



Subaction tree created by above code:

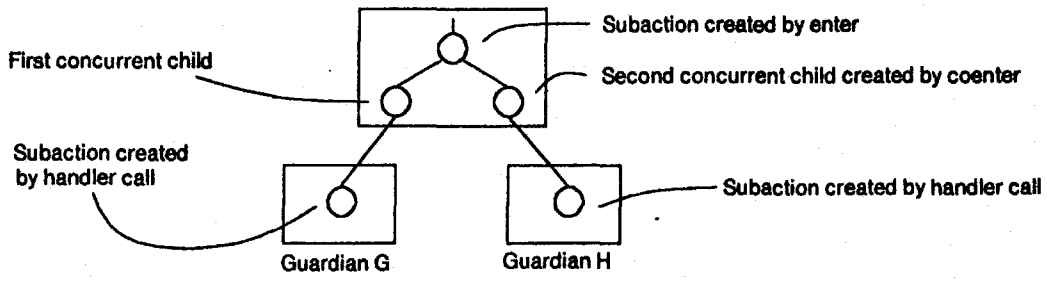


Figure 2-1: Argus subaction creation example

terminated all at a single guardian. Hence it makes sense to speak of an "action's guardian," since every action is intimately associated with a single guardian. A subaction need not run at its parent's guardian; such subactions are created by handler calls.

2.4 Handler Calls

The Argus system constructs and sends the *call* and *reply* messages needed to implement a handler call. A *call* message is sent from the calling action's guardian to that of the called guardian. Call messages contain the values of the actual parameters, the identity of the handler operation that is being called, and the identifier of the calling action. A *reply* message is sent from the called handler to the guardian of its caller when the handler completes. Reply messages contain the values of the output parameters being returned to the caller, and the identifier of the replying action.

Since handler calls are run as subactions, they have *at-most-once* semantics, namely that effectively either the call message is delivered and acted on exactly once at the called guardian, with exactly one reply received, or the message is never delivered.

A handler call actually creates two subactions. At the caller's guardian a *call action* is created. This action is a child of the caller. The call action handles the preparation of the call message and the receipt of the reply message. At the called remote guardian a *handler action* is created. This action is considered to be a child of the call action. The handler action executes the called handler operation.

2.5 Mutex Objects

Mutex objects are not atomic objects, even though they are shared by atomic actions. Mutex objects are used by programmers to implement their own atomic objects. Argus provides several "built-in" types of atomic objects; however, a programmer can achieve greater concurrency in some cases by building his own atomic object types. Mutex objects are similar to atomic objects in that they have locks, but there are two significant differences. Firstly, an action can release a lock it has acquired on a mutex object at any arbitrary time. An action cannot release a lock on an atomic object; locks on atomic objects are not released until after the action

holding them completes. Secondly, multiple versions are not maintained for mutex objects, i.e. modifications to mutex objects by aborted actions are not undone. Stable mutex objects modified by an action are written to stable storage after the action's topaction commits, as with stable atomic objects.

2.6 Implementation Details

The discussion thus far has basically focused on discussing the Argus language. This section is concerned with assorted details of the Argus system implementation.

2.6.1 Remote Lock Inheritance

When a handler action commits, the locks it obtained are inherited by its call action per the locking rules. However, to avoid the expense of including information about locks in reply messages, the information about these inherited locks is kept locally at the handler's guardian. The locks are still held; the call action becomes the absentee holder of the locks. In addition, the call action not only inherits the locks the handler subaction itself obtained, but also any locks the handler inherited and owns *in absentia*. Hence an action can hold several locks at several guardians distinct from its own. Furthermore, the system maintains no information at the action's guardian concerning the exact identities of these locks.

2.6.2 Two Phase Commit

The commit of a subaction is conditional. If the topaction a subaction has committed up to aborts, then the subaction should be aborted. On the other hand, if a topaction commits, all the results produced by subactions that committed up to the topaction should be committed. However, if one of these subaction's results have been wiped out by a crash, the topaction and all its committed descendants must be forced to abort.

To ensure that a topaction and all the subactions that committed up to it either

all commit or all abort, a standard two-phase commit protocol is carried out [Gray78]. In the first phase, an attempt is made to verify that all locks are still held, and to record the new state of each modified stable object on stable storage. This is done by sending a *prepare* message to each guardian where a handler call subaction ran that committed up to the topaction. Upon receipt of a *prepare* message, a guardian makes sure that the appropriate locks are still held and, if so, writes the appropriate objects to stable storage and then replies with a *prepared* message. If the first phase is successful, i.e. a *prepared* message is received from every guardian a *prepare* message was sent to, then in the second phase the locks are released, the recorded states become the current states, and the previous states are forgotten. If the first phase fails, the recorded states are forgotten and the topaction is forced to abort, restoring the objects to their previous states.

2.6.3 Granting Locks: Querying

When a subaction requests a lock, that lock might be held by an absentee lock holder. In this case, communication is necessary to discover if the lock can indeed be granted to the lock requester. This communication procedure is called *querying*.

In order for querying to be necessary, the subaction that obtained the lock in question must have committed. There are two possible cases -- either the lock was obtained by a relative or the lock was obtained by a non-relative.

In the case of a non-relative, the lock can be granted only if some ancestor of the lock obtainer has been aborted. In this case, the action that inherited the lock must have been aborted.[1] When an action is aborted, the system makes no attempt to locate and release any non-local locks the action might have inherited. Hence the system can find itself holding a lock for an action that does not exist, i.e. that has been aborted.

---

[1] Or should be aborted; actions with aborted ancestors are orphans.

To discover if some ancestor of the lock obtainer has aborted, the guardian of the lock requester first directs a *query message* to the lock-obtainer's topaction's guardian. If the topaction has completed two-phase commit or aborted, implying that some action that inherited the lock was aborted, a *query response* will be sent back indicating that the lock should be released, and any versions discarded. If this is not the case, a query response to the effect of "don't know" will be sent back. The requester's guardian can then direct queries to other guardians to attempt to discover if any ancestor the the lock obtainer has aborted.

Let us now consider the case where the lock in question was obtained by a relative of the requester. Consider the case where the lock requested is a read lock, and the lock held is a write lock. If the write lock has been inherited by an ancestor of the requester, the read lock can be granted to it according to the locking rules of Argus. In order to discover if this is the case, the requester's guardian directs a query to the guardian of the *least common ancestor*, or LCA, of the holder and requester. The LCA is defined as the closest ancestor any given set of related actions have in common. If the lock obtainer has committed up to the LCA, the LCA's guardian sends back a query response indicating that the LCA is indeed the absentee holder of the write lock. The read lock is granted to the requester once this message is received.

Similar events occur when the lock requested is a write lock. In this case queries will be directed to the guardians of each LCA of the requester and some particular obtainer of a read or write lock. Each query response indicates if the obtainer in question has committed up to the given LCA. If all the query responses indicate that all the locks have been inherited by ancestors of the requester, the write lock can be granted to the requester.

This discussion of querying has omitted many details; a full discussion can be found in [Liskov84]. This discussion is included here since query response messages have a role in orphan detection.

2.6.4 Action Identifiers

Each action has a unique identifier. A subaction's identifier consists of the identifier of its parent concatenated with the guardian identifier of the subaction's home guardian and a unique identifier. Guardian identifiers, incidentally, are unique fixed-length identifiers. A topaction's identifier consists of a unique identifier and its guardian's identifier. Thus an action identifier consists of a sequence of pairs of unique identifiers and guardian identifiers. Hence the identifiers of all a subaction's ancestors can be derived from that subaction's identifier. Note also that the identifiers of the guardians of all an action's ancestors can be derived from the action's identifier.

Chapter Three

Orphans

An orphan is an action that has had some ancestor perish or had the pertinent results of some relative action lost in a crash. This chapter discusses how orphans arise in Argus and identifies the problems that justify bothering to detect and exterminate them.

The reader should note that we define an orphan to always be an active action, i.e. an action that has neither committed nor aborted.

But a caveat here; note that orphans are not exclusively children actions whose parents (or ancestors) have been killed for one reason or another. This is just a warning that the typical preconceived notion about what constitutes an "orphan" is not totally correct.

Orphans arise in Argus due to crashes and explicit aborts. Orphans that arise due to explicit aborts will be discussed first, and then orphans that arise due to crashes.

3.1 Orphan Creation via Explicit Aborts

When a parent action is aborted, the active descendants it leaves behind become orphans. An active action with an explicitly aborted ancestor is called an *abort-orphan*.

3.1.1 Types of Explicit Aborts

There are two flavors of explicit aborts in Argus that can cause the creation of orphans. The first is the explicit abort of handler calls. Recall that a handler call

actually creates two subactions -- a call subaction at the caller's guardian and the handler subaction at the remote guardian. If the system judges that a handler call cannot be completed successfully due to a communications failure, etc., it aborts the call action. This orphans the handler subaction, if it indeed exists, and any of its descendants.

The second source of abort-orphans is explicit aborts initiated by the **coenter** statement. The **coenter** statement spawns concurrent sibling subactions, as previously discussed in Section 2.3. If any one of these concurrent siblings transfers control outside the textual scope of the **coenter** statement, the other active siblings are aborted before execution is allowed to proceed any further. If any of these aborted siblings happened to have made a handler call, the remote handler action and its descendants are orphaned. Figure 3-1 gives an annotated example of using the **coenter** statement to implement a handler call timeout. In the example, when the first subaction created by the **coenter** completes, the other is aborted. The example repeatedly makes a handler call until the handler call successfully returns within 60 seconds.

3.1.2 Can Abort-Orphan Creation be Avoided?

Abort-orphans are inevitable in Argus, due to a design decision that Argus should provide "quick" aborts. If abort-orphans are to be avoided, the abort of an action must be delayed until all of its active descendants are tracked down and aborted. Of course, many of its descendants might be remote, so tracking them down typically involves communicating across the network. Such a delay is not compatible with the design goal of "quick" aborts.

3.1.3 Problems Caused by Abort-Orphans

One might question what problems an abort-orphan could possibly cause. Since an abort-orphan does not commit up to its topaction, it does not participate in two phase commit. Hence the results it eventually produces are not committed to

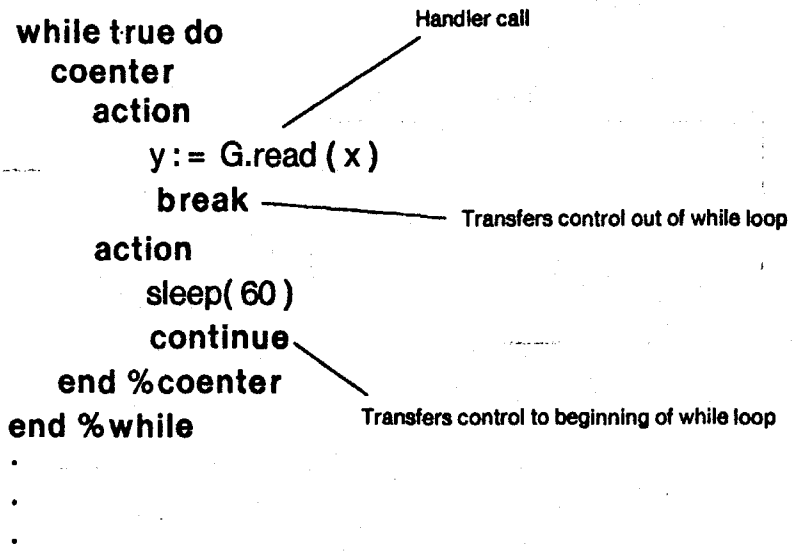


Figure 3-1: Coenter example

stable storage; these results will be discarded when it is discovered through querying (or orphan detection) that they were produced by an orphan. Hence an abort-orphan's results are undone. So one could argue that abort-orphans are harmless in that they have no observable effect on system state. Hence why bother detecting and aborting them?

One problem with abort-orphans is that they waste resources. An abort-orphan's results are undone; the resources used to produce these results are wasted. But even though resource wastage is an unfortunate consequence of abort-orphans, is it so grave a problem that it justifies bothering to detect and exterminate abort-orphans? Since the orphan detection scheme that this thesis presents is rather costly, one could argue that going to the bother just on account of wasted resources is not justified.

A significant problem does arise, however, when abort-orphans are permitted to run unchecked. An abort-orphan can encounter a situation where it views data in an inconsistent state. By this it is meant that an abort-orphan's data can get into an

"impossible" state -- one that violates the semantics of atomicity. The idea that an action should always view consistent data has been formalized by Goree as *view-serializability* [Goree83].

An example is now presented that illustrates how an abort-orphan can view inconsistent data. The crux of the example is two guardians named **GX** and **GY**. **GX** and **GY** each contain a single atomic object, x and y, respectively. There is an invariant between x and y, namely x = y. We can suppose **GX** and **GY** implement the copies of a replicated data base. Suppose that x = y = 0 initially. In the example, an abort-orphan will come to view x ≠ y, violating the consistency constraint.

In this example, the creation of call actions is ignored. Remember that a handler call causes two subactions to be created -- a call subaction at the caller's guardian and a handler subaction at the called guardian. But since these call actions have no interesting role in this example, we ignore them.

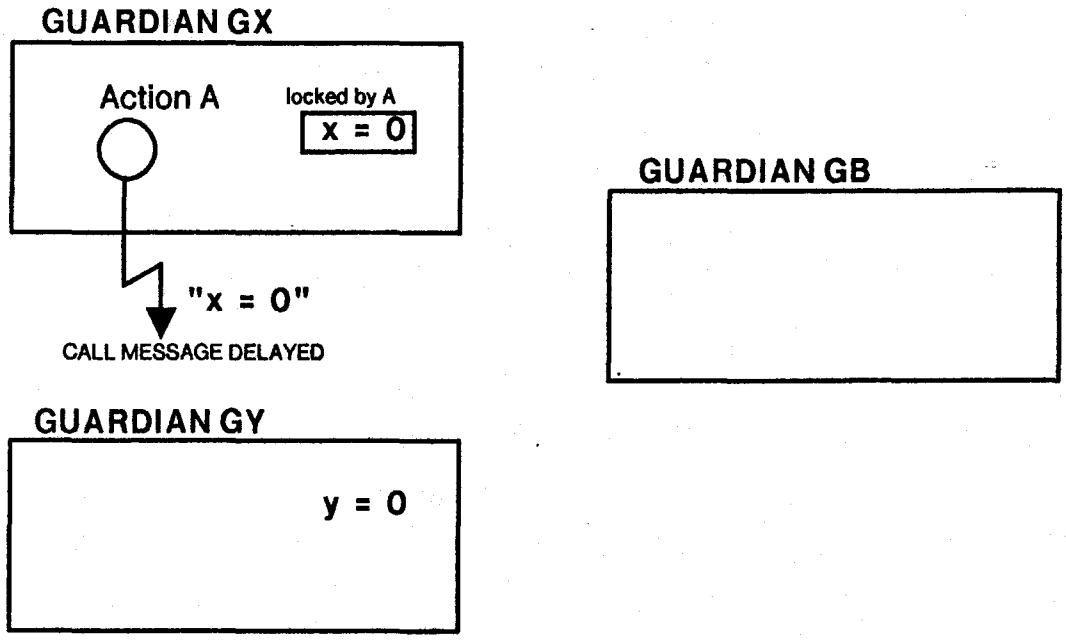


Figure 3-2: Abort-orphan example snapshot one

Suppose an action A is created at guardian **GX**. A reads x, which has the value

of zero, and then makes a handler call to **GY** passing the information in the arguments of the call that x is zero. But suppose the call message is delayed in the network. Figure 3-2 illustrates the resulting situation.

Suppose the system then judges that it cannot complete the handler call successfully and aborts it. (It does this by aborting the call action.) A then resumes execution but gives up and aborts itself. This causes the lock on x to be released.

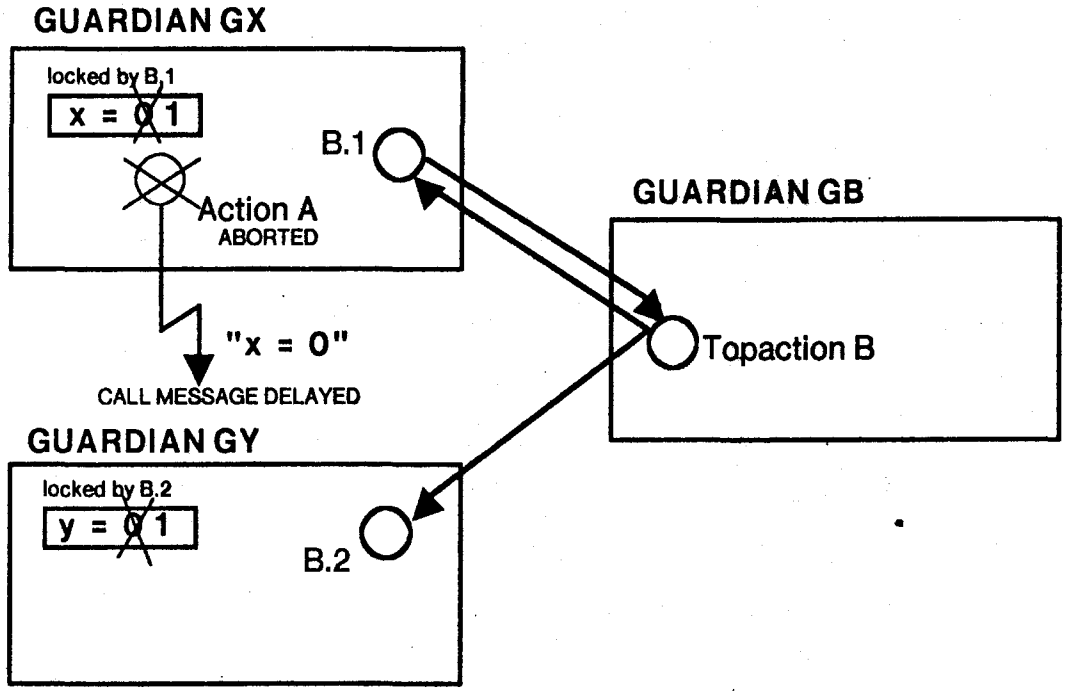


Figure 3-3: Abort-orphan example snapshot two

Suppose then a topaction B residing at guardian **GB** makes a handler call to **GX** creating subaction B.1. B.1 changes the value of x to one and commits to B. Then B makes another handler call, this time to guardian **GY**, creating action B.1. B.1 changes the value of y to one. Figure 3-3 illustrates the resulting situation.

Suppose that B.2 commits to B. Then B itself commits and two phase commit is completed successfully. This causes the locks on x and y to be released and their new values to be assumed. Then suppose the delayed handler call made by aborted

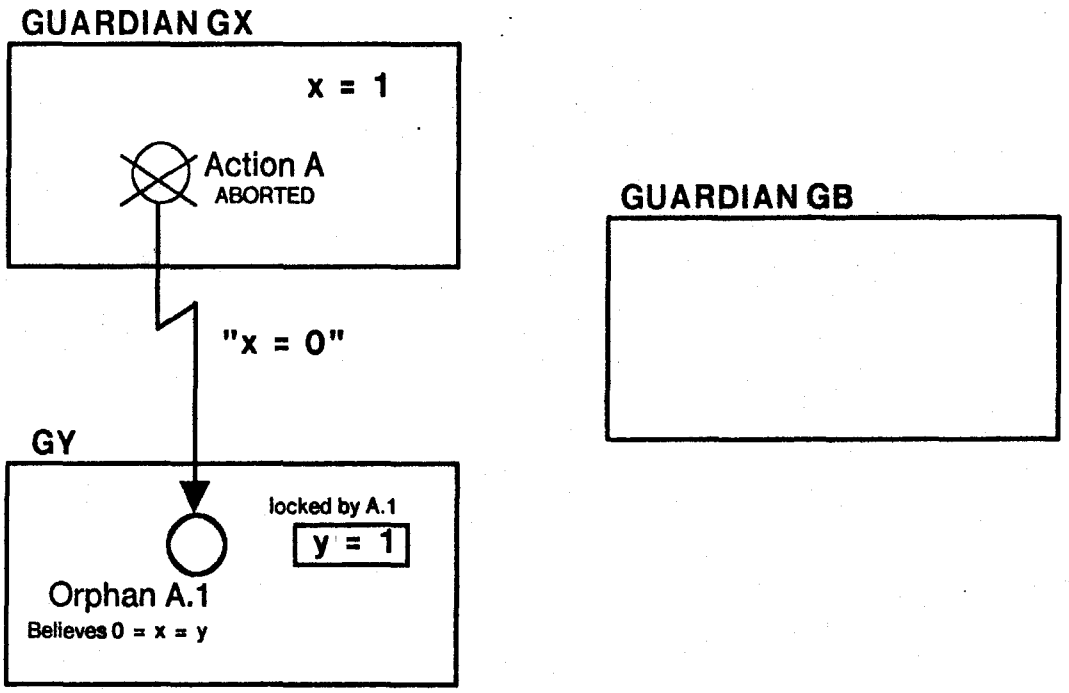


Figure 3-4: Abort-orphan example snapshot three

action A finally arrives at **GY**, creating subaction A.1. A.1 reads y, and finds that y is one, contrary to the information A.1 received through its arguments that indicates y is zero. The invariant x = y has been violated in the view of A.1. Figure 3-4 illustrates this final situation.

Let us now discuss the negative aspects of permitting an abort-orphan to view inconsistent data. An abort-orphan is just a piece of Argus code written under the assumption that the data it views is consistent. Thus an abort-orphan might behave erratically when this proves not to be the case. This bizarre behavior could be realized by going into an infinite loop, terminating with an unhandled exception, or perhaps producing garbage on a perplexed user's terminal. In these cases, a programmer's confidence in his code would be shaken since it would not be known to the programmer whether the action was indeed an abort-orphan when it displayed its erratic behavior.

It was stated previously that an orphan's results are always undone. This is

true insofar as atomic data is concerned, but is not true of non-atomic data. In particular, an orphan's modifications to mutex objects are never undone. In Argus, a programmer can use non-atomic mutex objects together with atomic objects to construct his own user-defined atomic objects. Actions share non-atomic data in such implementations of user-defined atomic objects; these implementations must be written with extreme care in order to work correctly. However, these programs are still written under the assumption that an action's view is consistent. When an orphan views inconsistent data, it could hopelessly corrupt shared non-atomic mutex objects. Such modifications are never undone. After mutex objects are corrupted, other non-orphaned actions that share the corrupted objects would then behave erratically and possibly corrupt other data.

In addition, an abort-orphan might have some interaction with the physical world that cannot be taken back. The example above of printing garbage on a user's terminal falls into this category.

Let us analyze how an abort-orphan can come to view inconsistent data. Consider an ancestor whose abort creates an abort-orphan. This ancestor could have passed information concerning the states of data it examined or modified down to the abort-orphan. This information reaches the abort-orphan by filtering down the descendant chain starting from the ancestor. The information is transmitted from one action to the next in the descendant chain either through the arguments of handler calls when the next descendant in the chain is remote, or through shared objects when the next descendant is local. But in any case, the abort-orphan receives information originating from the ancestor concerning the states of its local data. But this information becomes invalid after the ancestor aborts, since the ancestor's modifications are undone and its locks are released, permitting some other action to modify data the ancestor examined. It is this invalidated information that can lead the abort-orphan to view inconsistent data.

3.2 Orphan Creation via Crashes

When a guardian crashes, all active actions with an ancestor at the crashed guardian become orphans. Additionally, any active action with a descendant that ran at the crashed guardian becomes an orphan, provided this descendant committed up to the action in question. We call orphans created by crashes *crash-orphans*. Crash-orphans fall into two categories -- *orphaned-children* and *uprooted-actions*. A crash-orphan is an *orphaned-child* if it was orphaned due to an ancestor perishing in a crash. A crash-orphan is known as an *uprooted-action* when it is orphaned by crash of a non-ancestor's guardian.

3.2.1 Problems Caused by Orphaned-Children Crash-Orphans

An orphaned-child crash-orphan has had some ancestor perish in a crash. But perishing in a crash has the same effect as an explicit abort of the ancestor -- the ancestor's execution is terminated, its locks are released, and its versions are thrown away, as discussed in Section 2.2. Thus orphaned-child crash-orphans get into exactly the same type of trouble as abort-orphans do, for exactly the same reasons. All of the previous discussion concerning abort-orphans applies to orphaned-child crash-orphans as well.

3.2.2 Problems Caused by Uprooted-Action Crash-Orphans

An uprooted-action is either an action that has had a descendant's guardian crash, provided the descendant committed up to the action, or any descendant of such an uprooted-action. Hence an uprooted-action has suffered the crash of a some relative's guardian; note that this relative need not be a descendant. Uprooted-actions get into the same sorts of trouble as the other types of orphans that have been discussed -- they waste resources and they can view inconsistent data.

The results produced by an uprooted-action are discarded. Assuming an uprooted-action commits up to its topaction, two phase commit for the topaction will fail since it will be discovered that the locks obtained by the relative whose guardian's

crash created the uprooted-action have been released prematurely. Thus uprooted-actions waste resources, since their work to produce results is futile.

Uprooted-actions can also view inconsistent data. This has the same negative ramifications as permitting an abort-orphan or orphaned-child crash-orphan to view inconsistent data.

An example is now presented that illustrates how an uprooted-action can view data in inconsistent state. The uprooted-action in this example is created through the crash of a committed child's guardian. This example takes place in a scene like that of the previous example -- there are two guardians **GX** and **GY**, each containing an atomic object, x and y, respectively. The uprooted-action in this example will view the invariant "x = y" violated.

Again, in this example we ignore the existence of call subactions.

Suppose a topaction A is created at guardian **GA**. A does a handler call to **GX** creating subaction A.1 at **GX**. Action A.1 reads x and discovers it has the value of zero. A.1 then commits to A returning information in its return arguments that x is zero. Figure 3-5 illustrates the resulting situation.

Then guardian **GX** crashes and recovers. This causes the lock obtained by action A.1 and inherited by action A to be released. Topaction A is now an uprooted-action. Then suppose a topaction B at guardian **GB** does a handler call to **GX** creating subaction B.1. B.1 changes the value of x to one and then commits to B. B then does a handler call to **GY** creating subaction B.2. B.2 changes the value of y to one. Figure 3-6 illustrates the resulting situation.

Then B.2 commits to B. B commits, and two phase commit is done for B and succeeds. This causes the locks held on x and y to be released. Then topaction A does a handler call to **GY** passing information in the arguments of the call that x is zero. Recall that A's subaction A.1 passed it this information before the crash. This

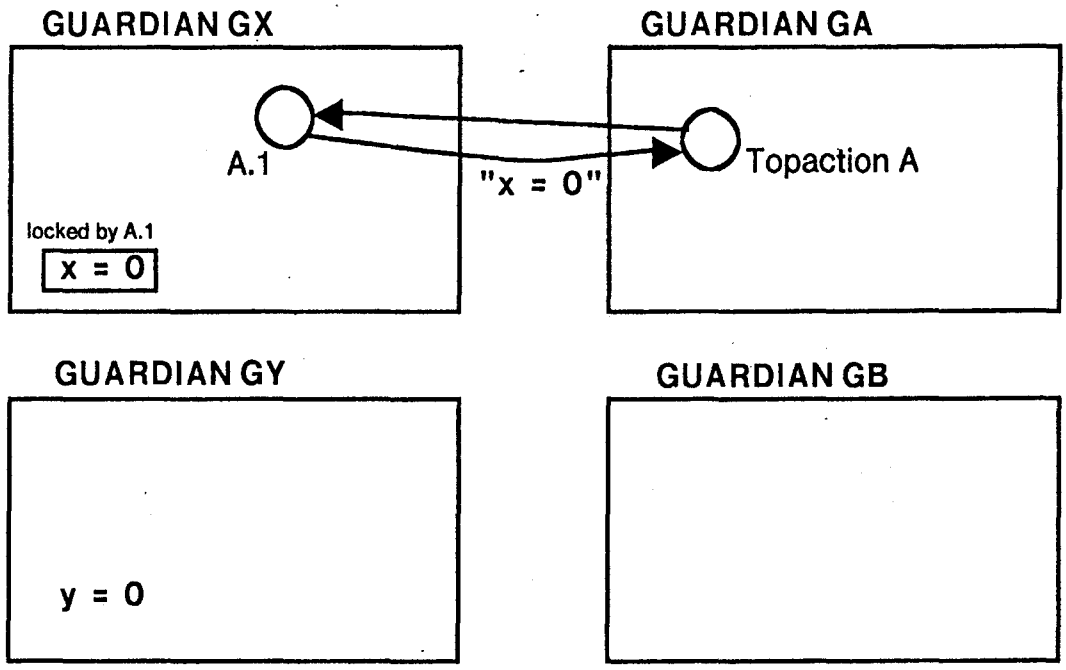


Figure 3-5: Crash-orphan example snapshot one

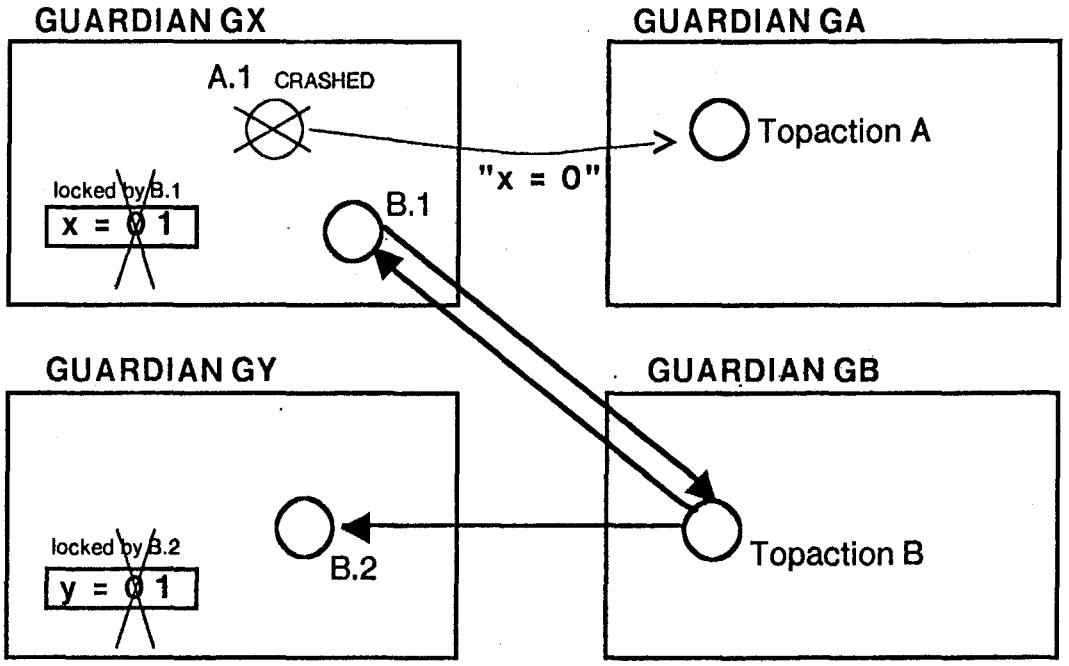


Figure 3-6: Crash-orphan example snapshot two

handler call causes the creation of subaction A.2 at GY. A.2 reads y and observes

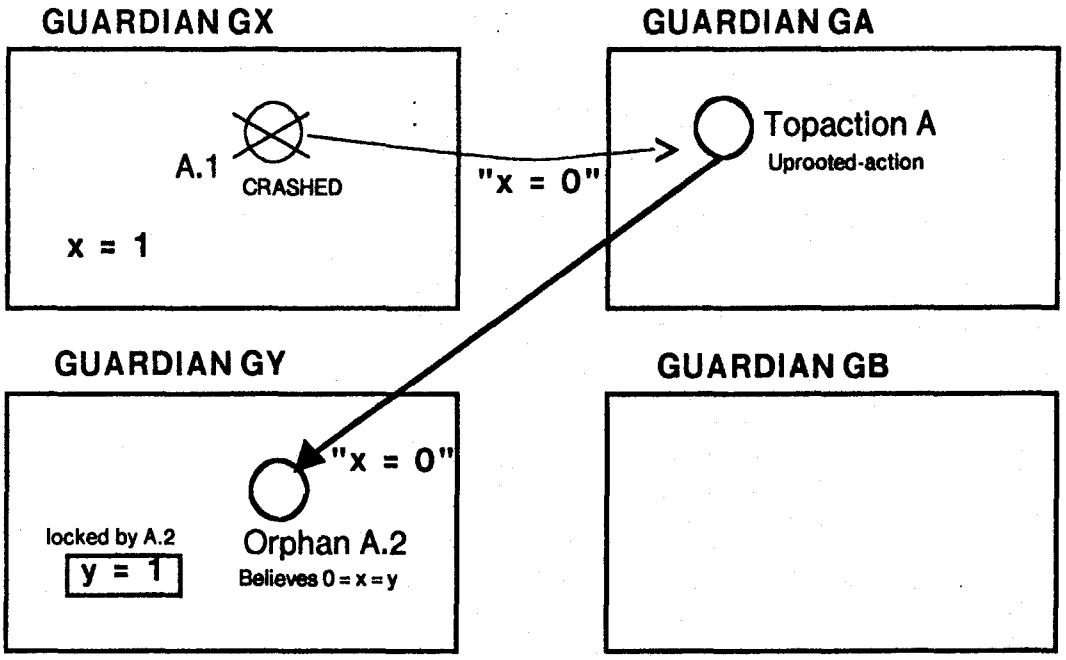


Figure 3-7: Crash-orphan example snapshot three

that the invariant x = y has been violated. Figure 3-7 illustrates this final situation.

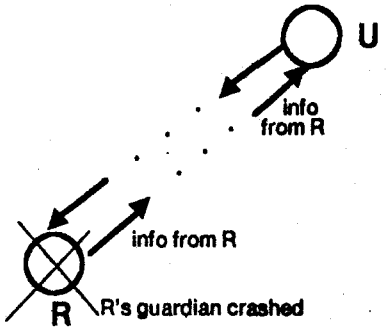


Figure 3-8: Uprooted-action created by crash of committed descendant

Let us analyze how uprooted-actions can come to view inconsistent data. Let us first consider uprooted-actions orphaned by the crash of a descendant's guardian. This descendant must have committed up to the uprooted-action. Figure 3-8 illustrates this case. Subaction **U** is an uprooted-action created by the crash of **R**'s guardian sometime after **R** committed. **R** has passed information concerning the

states of data it examined or modified up to U. This information was transmitted from one action to the next in the action ancestor chain starting from R and ending at U either through the return values of handler replies when the next ancestor in the chain was remote or through shared objects when the next ancestor was local. The crash causes R's modifications to objects to be undone and the locks it obtained to be released, permitting some other action to modify the data it examined. Since the information passed to U about the state of data at R's guardian no longer conforms to the true state of data there, U can find itself in a situation of viewing data in an "impossible" state.

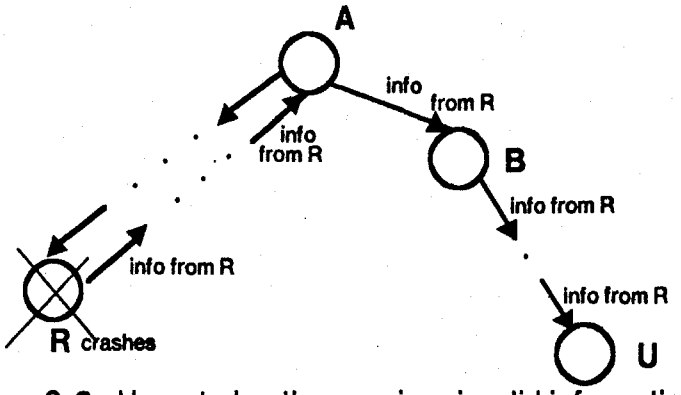


Figure 3-9: Uprooted-action receives invalid information

Let us now consider how uprooted-actions created by the crash of a non-descendant's guardian can view inconsistent data. In this case, the uprooted-action is a descendant of some other uprooted-action that has suffered the crash of a committed descendant's guardian. Figure 3-9 illustrates this situation. U is an uprooted-action created by the crash of R's guardian. (A and B are also uprooted-actions). In the case illustrated, subaction B was spawned after R committed up to A. As the illustration shows, information originating from subaction R has been passed up to A and then down to U. The crash of R's guardian invalidates this information, however, leading U to perhaps view inconsistent data.

Let us now consider the case where subaction B is spawned before R commits up to A. Figure 3-10 illustrates this situation. In this case, subaction B is a

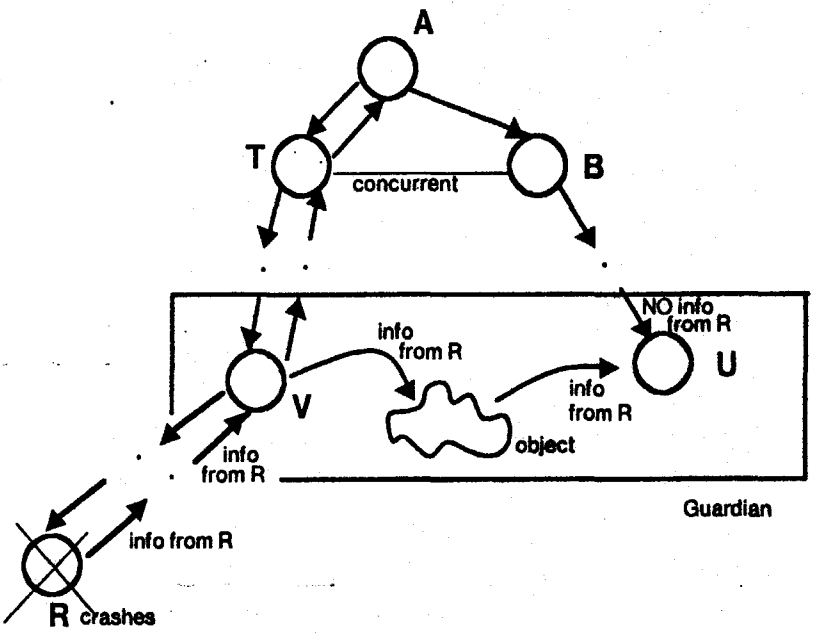


Figure 3-10: Uprooted-action receives no invalid information from parent or child

concurrent sibling of subaction T, R's ancestor that is a child of A. R has passed information concerning the states of data it examined or modified up to subaction V. Subaction V has in turn embedded this information in an object it modified. When V commits up to A, A inherits the lock on this object. At this point, U can obtain a lock on this object. If U does not reside at A's guardian, this involves querying, as discussed in Section 2.6.3. When U obtains the lock and examines this object, it obtains information concerning the states of data at R's guardian. This information is invalidated by the crash of R's guardian, however, leading U to perhaps view inconsistent data. Note that this case is unique in that invalid information was passed to the orphan neither by its parent nor child, unlike all the other cases involving abort-orphans and crash-orphans that have been examined previously.

Chapter Four

The Orphan Detection Algorithm

This chapter presents an orphan detection algorithm. This algorithm has a number of attractive features. First, the algorithm does not falsely accuse an action of being an orphan; only orphans are detected as such by the algorithm. Second, the algorithm detects an orphan before it can view any inconsistent data. Since an orphan is benign (except for wasting resources) until it views inconsistent data, the algorithm works well enough to keep orphans from getting into trouble.

The algorithm as it is presented in this chapter is obviously inefficient and impractical. At the end of the chapter, several minor inefficiencies in the algorithm are addressed. Subsequent chapters deal with correcting the major impractical aspects of the algorithm.

4.1 Introduction to the Algorithm

This section presents the fundamental workings of the orphan detection algorithm. In order to keep the presentation from becoming bogged down in detail, the discussion of many aspects of the algorithm is deferred until the next section. The discussion of precisely how an orphan is exterminated once detected is also deferred.

The orphan detection algorithm works by piggybacking information about orphans onto the messages that flow about the system. The algorithm can be divided into two halves. One half of the algorithm handles abort-orphans and the other half handles crash-orphans. A discussion of each half follows.

4.1.1 Detecting Abort-Orphans

Detection of abort-orphans is based upon a data structure called *done*. Done is a list of action identifiers of aborted actions. Each guardian maintains its own done data structure. When an action is aborted, its identifier is added to its guardian's done. The presence of an action's identifier in done is interpreted as meaning that all descendants[2] of that action are orphans.

Whenever a message is sent out from a guardian, the guardian piggybacks its current value of done onto the message. A guardian receiving a message uses the piggybacked done to detect local orphans. Any action running at the receiving guardian that has the identifier of an ancestor appearing in the piggybacked done is an orphan and is aborted. Additionally, if the message is among those that are sent on behalf of a particular action (e.g. handler call and reply messages), the sending action could be an orphan itself, or even have been aborted since the message was sent. The sending action's identifier is included in such messages. The receiving guardian checks the identifier included in the message against its own done to detect this condition. If the sending action is a descendant of an action whose identifier is in done, the received message is discarded. The receiving guardian also updates (unions) its own done with the piggybacked done.

A guardian that receives a message can start "normal" processing of the message only after all the steps above relating to orphan detection have been completed. For example, a guardian receiving a call message must complete all the above steps before a handler action is created to run the call.

Whenever a guardian participates in two phase commit, it records its current value of done on stable storage. This ensures that done is restored to a proper state after a crash.

---

[2] Recall from Chapter 2 that an action is always its own descendant, according to our definition of "descendant." Also recall that "ancestor" is similarly defined.

The workings of abort-orphan detection shall now be illustrated by taking the first example from the last chapter and adding orphan detection. Again, guardians GX and GY each contain a single atomic object, x and y, respectively. The invariant between x and y is x = y. Suppose initially that x = y = 0, and that every guardian's done is empty.

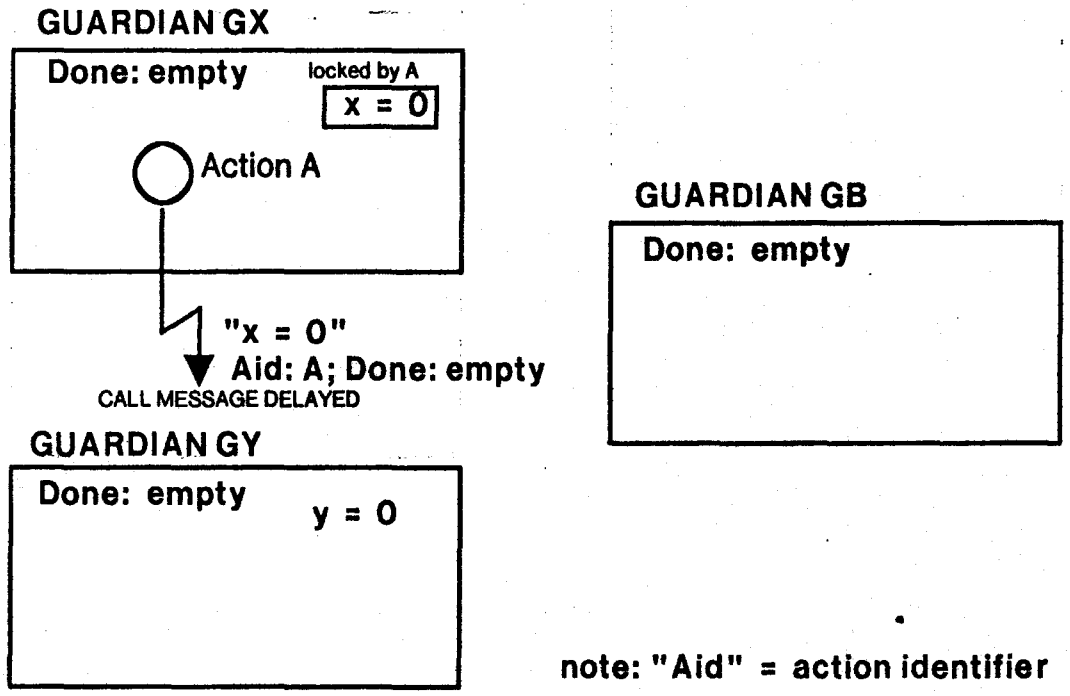


Figure 4-1: Abort-orphan detection example snapshot one

Suppose action A at guardian GX reads x, discovering that it has a value of zero. Then A does a handler call to guardian GY passing the information that x is zero in the arguments of the call. A's action identifier and GX's done are piggybacked onto the call message. But the call message gets delayed in the network. Figure 4-1 illustrates the resulting situation.

The system then judges that the handler call cannot be completed successfully and aborts the handler call. (It does this by aborting the call action. In this example we omit the detail of call action existence.) A then itself aborts, causing its action identifier to be added to GX's done. Then topaction B at guardian GB does a handler

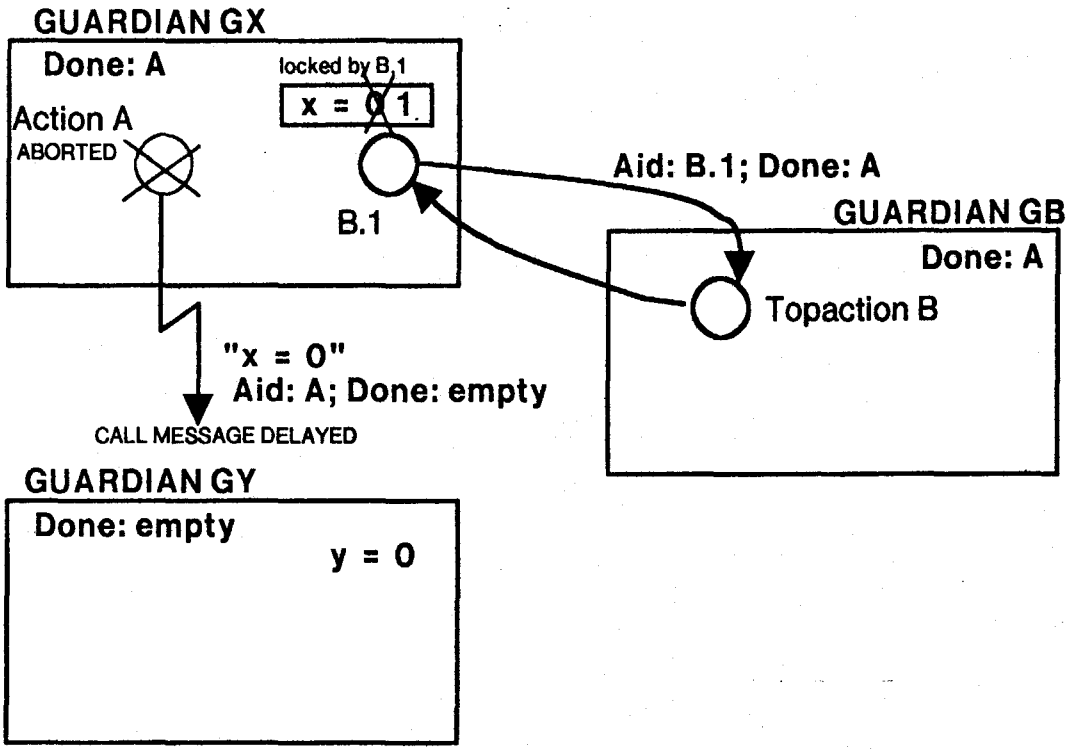


Figure 4-2: Abort-orphan detection snapshot two

call to GX creating subaction B.1 at GX. B.1 changes the value of x to one and commits to B. The reply message from B.1 to B includes GX's done which contains A's action identifier. When GB receives this message, it sees that the sent done contains an action identifier its done does not -- namely A's. Hence GB adds A's action identifier to its own done. Figure 4-2 illustrates the resulting situation.

Topaction B then makes a handler call to GY. GB's done, containing A's action identifier, is piggybacked on this message. When GY receives this message, it adds A's action identifier to its own done and creates subaction B.2 to run the handler call. Subaction B.2 changes the value of y to one. Figure 4-3 illustrates the resulting situation.

Subaction B.2 then commits to B and B subsequently itself commits. Two phase commit for topaction B finishes successfully, causing the locks on x and y to be released. Finally, the delayed handler call message from aborted action A arrives

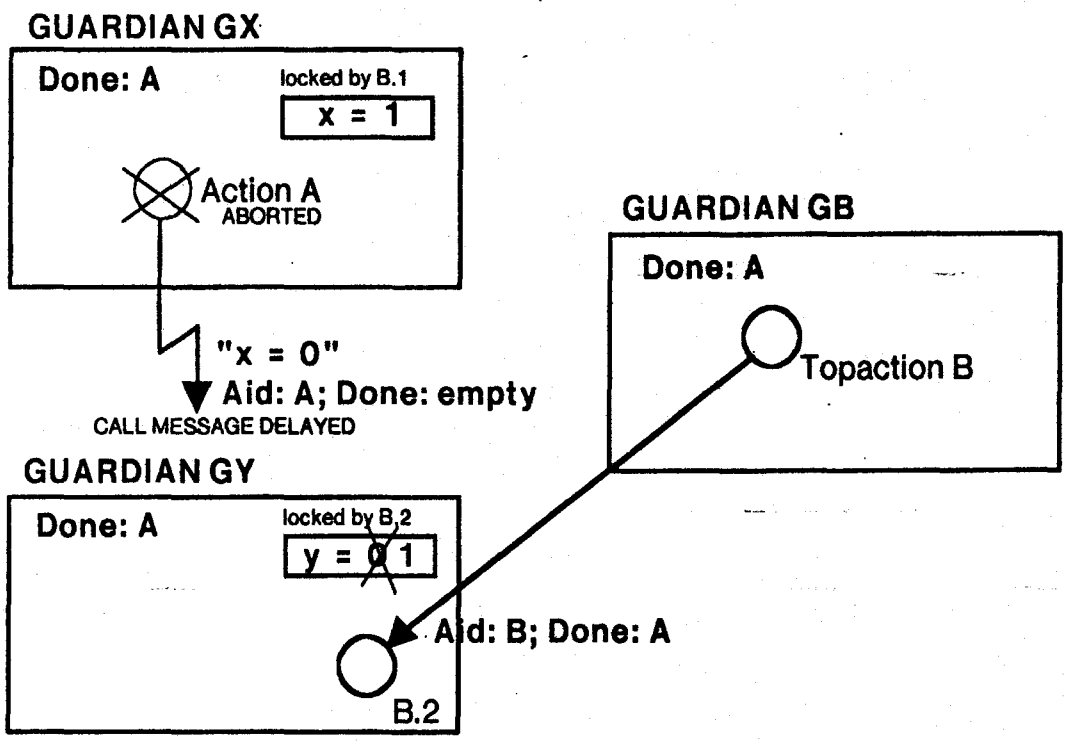


Figure 4-3: Abort-orphan detection snapshot three

at GY. This message carries the invalid information that x is zero. This message is discarded, however, since it is from A and A's action identifier appears in GY's done. Figure 4-4 illustrates the resulting situation.

4.1.2 Detecting Crash-Orphans

Detecting crash-orphans is somewhat more complicated than detecting abort-orphans. Every guardian maintains a counter in stable storage called a *crash count*. Whenever a guardian recovers from a crash, it increments its crash count. Every guardian also maintains a data structure called *map*. Map is a table that associates guardian identifiers with crash counts. Any given guardian's map represents the beliefs that guardian has about the number of crashes that have occurred at other guardians. A guardian's map contains an entry for itself, which is always up-to-date.

A data structure called the *d-list-map* also plays an important role in the

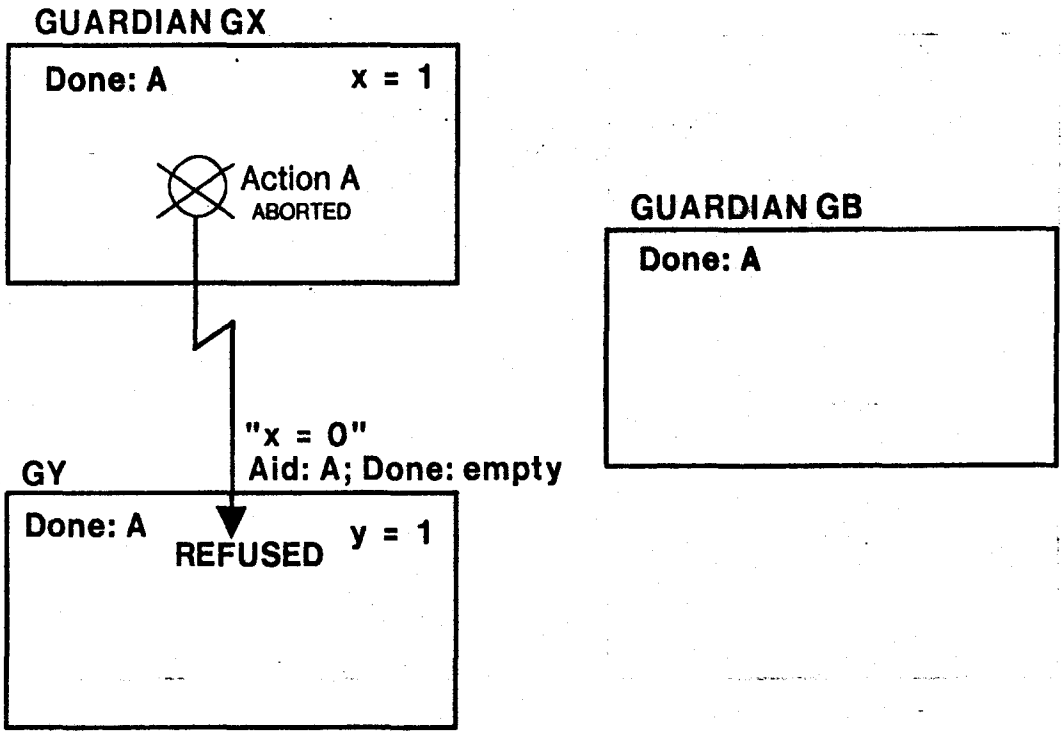


Figure 4-4: Abort-orphan detection snapshot four

detection of orphans created by crashes. The d-list-map is a table that associates guardian identifiers with crash counts, like map. Every action has a d-list-map associated with it. An action's d-list-map contains the identifiers of all the guardians whose crash would cause the action to become an orphan. These guardians include those where some ancestor of the action is running or where some descendant that committed up to the action ran. For each such guardian, the d-list-map associates the crash count of the guardian at the time the relative ran there with the guardian's identifier.

Whenever a message is sent out from a guardian, the guardian piggybacks its map onto the message. This piggybacked map is used to detect orphans at the guardian that receives the message. Any action at the receiving guardian is an orphan if its d-list-map is *out-of-date* according to the piggybacked map, i.e. if the crash count associated with some guardian identifier in the action's d-list-map is lower than the crash count associated with the same identifier in the piggybacked

map. All such orphans detected are aborted.

A piggybacked map is also used to *update* the map of a receiving guardian, as follows. Any entries for guardians in the piggybacked map not appearing in the receiving guardian's map are added to it. In addition, any entry for a particular guardian in the receiving guardian's map that has a lower crash count than the corresponding entry for the same guardian in the piggybacked map is adjusted to reflect the higher crash count.

For a message sent on behalf of a particular action (e.g. handler call and reply messages), that action's d-list-map is piggybacked onto the message. This d-list-map is used at the receiving guardian to detect if the sending action[3] is an orphan. The sending action is an orphan if its d-list-map is out-of-date according to the receiver's map. If the sending action proves to be an orphan, the receiver discards the message.

The d-list-map piggybacked on call messages is used to initialize that of the handler action created to run the call. The handler's d-list-map is initially the piggybacked d-list-map with an entry added for the handler's guardian. Similarly, the d-list-map piggybacked on reply messages is merged into the d-list-map of the action the reply is directed to. A topaction's d-list-map initially contains just a single entry for the topaction's guardian.

A guardian can only start "normal" processing of a received message after all the steps above relating to orphan detection for a received message are completed.

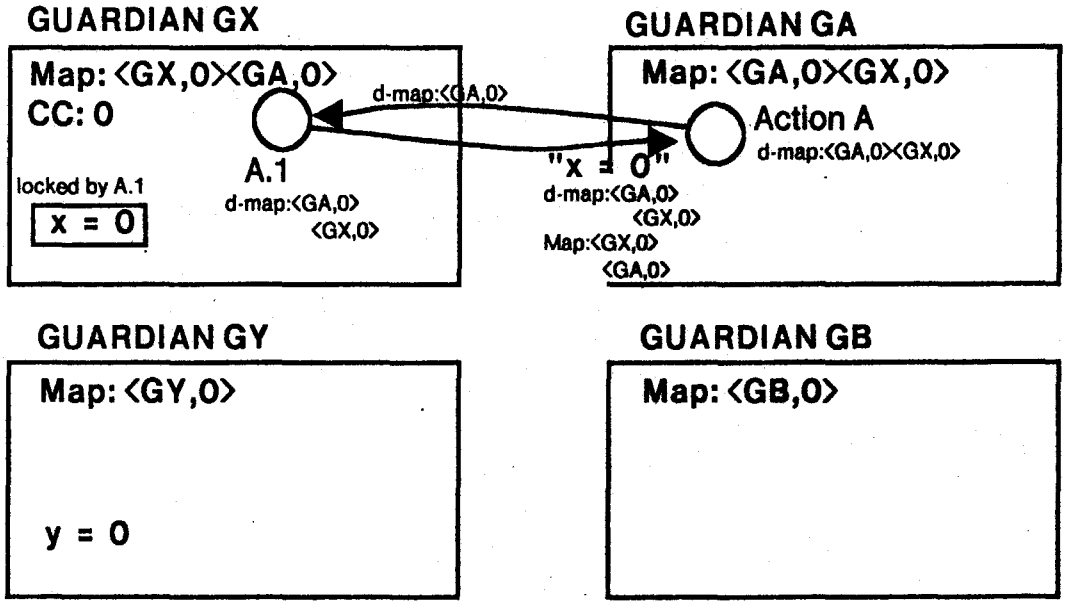
Whenever a guardian participates in two phase commit, it records its map on stable storage. This ensures that map is restored to a proper state after a crash.

Crash-orphan detection is now illustrated by taking the second example from

---

[3]The reader should note that actions never actually send or receive messages; only guardians send or receive messages, though sometimes acting as an agent of a particular action.

the last chapter and adding orphan detection. The crux of this example, like the one presented for abort-orphans, is two guardians named **GX** and **GY**. **GX** and **GY** each contain a single atomic object, x and y, respectively. The invariant between x and y is x = y. Suppose initially that x = y = 0, and that all guardian's crash counts are zero.



note: "d-map" = d-list-map

Figure 4-5: Crash-orphan detection example snapshot one

Suppose action A at guardian **GA** makes a handler call to **GX**, creating subaction A.1 at **GX**. A.1 reads x, discovering that it has the value of zero. A.1 then commits to A, returning information in the reply message that x is zero. A.1's d-list-map, which includes the entry <**GX**,0>, is piggybacked on the message. When the message arrives, the entry <**GX**,0> is added to A's d-list-map and **GA**'s map. The resulting situation if shown in Figure 4-5.

GX then crashes and recovers causing the lock on x to be released. Action A is now an uprooted-action. Then topaction B at guardian **GB** does a handler call to guardian **GX** creating subaction B.1. B.1 changes the value of x to one and commits to B. **GX**'s map piggybacked on the reply message to B includes the entry <**GX**,1>.

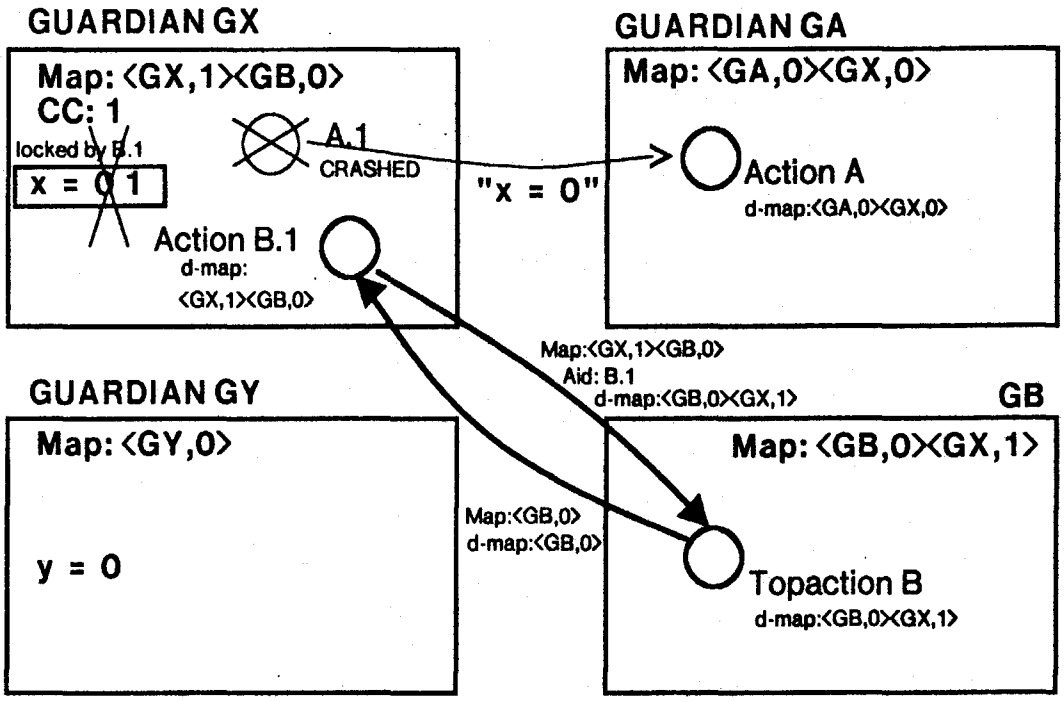


Figure 4-6: Crash-orphan detection example snapshot two

Upon receiving this message, **GB** updates its own map with the sent map resulting in **GB**'s map having the entry **<GX**,1>. The resulting situation is shown in Figure 4-6.

Topaction B at **GB** then does a handler call to guardian **GY**. **GB**'s map piggybacked onto the call message contains the entry **<GX**,1>. After the message is processed at **GY**, **GY**'s map contains **<GX**,1> and subaction B.2 is created to run the handler call. B.2 changes the value of y to one. The resulting situation is shown in Figure 4-7.

Subaction B.2 then commits to B. Topaction B then itself commits and two phase commit successfully completes, resulting in the locks on x and y being released. Then action A makes a handler call to guardian **GY** passing the invalid information that x is zero. A's d-list-map piggybacked on the call message contains the entry **<GX**,0>. When this message arrives at **GY**, it is refused since **GY**'s map contains the entry **<GX**,1>. Figure 4-8 illustrates this final situation.

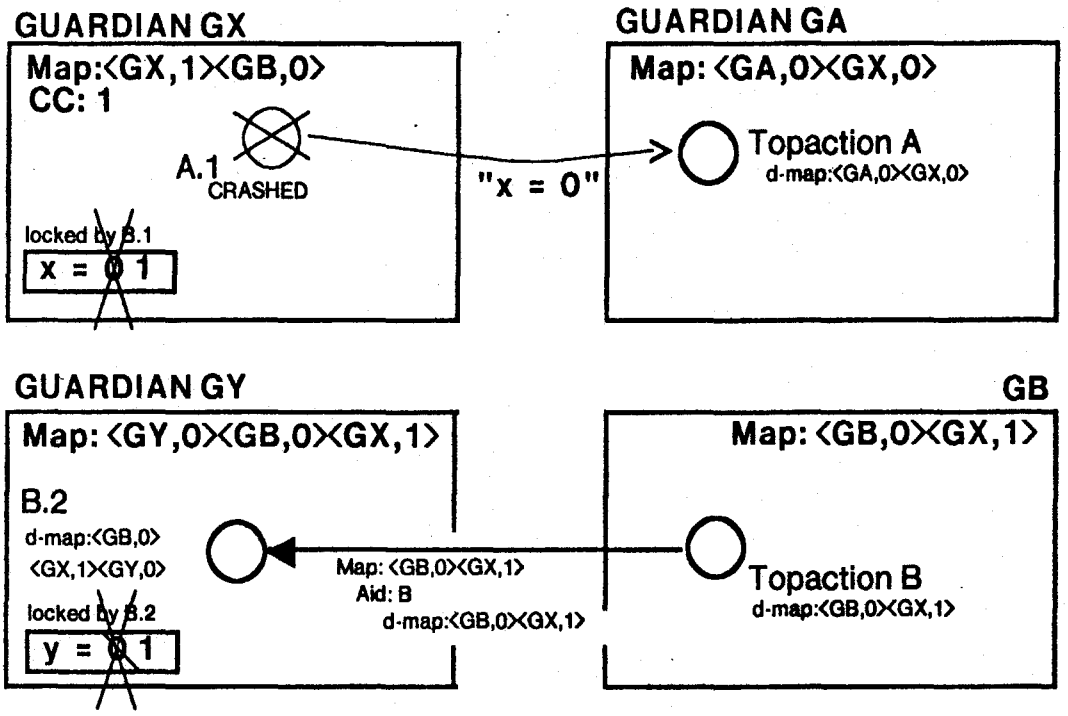


Figure 4-7: Crash-orphan detection example snapshot three

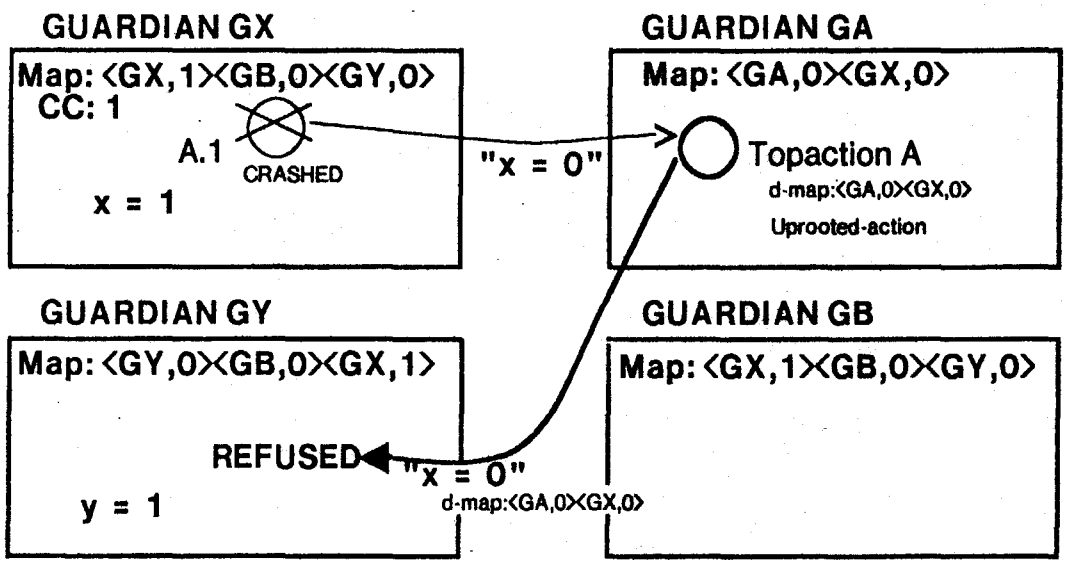


Figure 4-8: Crash-orphan detection snapshot four

4.2 Details of the Orphan Detection Algorithm

Several important details about the orphan detection algorithm were not mentioned in the previous section in the interest of preventing that discussion from becoming cluttered. This section presents the orphan algorithm in all its detail.

The algorithm is presented in this section by considering individually those situations that require some sort of activity by the orphan detection mechanism.

In the last section, it was categorically stated that done and map are piggybacked on *all* messages. This is actually not the case. Done and map are only piggybacked on messages discussed below.

4.2.1 Recovery

Upon recovery from a crash, a guardian must increment its crash count on stable storage. It must also restore its map and done from the copies last written on stable storage. Its map must also be updated to reflect its new crash count. The guardian can only start accepting handler calls when all these tasks are completed.

4.2.2 Action Abort

When an action running at a guardian G is aborted, its action identifier must be added to G's done. Also, all descendants of the action running at G must be aborted. Both of these tasks must be completed before any of the action's locks are released or versions discarded.

This rule is applied in a recursive manner, so an action is never actually aborted until all its descendants running at its guardian are first aborted. This results in no aborted action leaving behind any local active descendant actions; an abort creates no local abort-orphans.

4.2.3 Handler Call

A handler call causes the creation of a remote subaction. Suppose action A running at guardian G is doing a handler call to guardian H. The following items are piggybacked on the call message that is sent to H: G's done, G's map, and A's d-list-map. In addition, the call message contains A's identifier.

When the call message is received at H, H must perform several tasks. Let the done, map, d-list-map, and identifier of the calling action included in the call message be denoted as m.done, m.map, m.d-list-map, and m.aid, respectively. H first checks to see if the sending action is an orphan. If H's done contains the action identifier of some ancestor of m.aid or if a comparison of m.d-list-map and H's map shows that m.d-list-map is out-of-date, then A is an orphan. In this case, a *refusal* message is sent back to G. Refusal messages are discussed later. After this, H uses m.done and m.map to detect and abort any local orphans. H then updates its own done and map from m.done and m.map. After all these tasks are completed, and if the call was not refused, a handler action can be created to run the handler call. The handler action's d-list-map is initialized to be m.d-list-map with an entry added for H.

4.2.4 Reply

When a handler action commits, a reply message is sent back to guardian of the action that did the handler call. Suppose handler action A.C at guardian H is committing to call action A at guardian G. The following items are piggybacked on the reply message for A.C: H's done, H's map, and A.C's d-list-map. In addition, the reply message contains A.C's identifier.

When G receives the reply message, it must perform several tasks. Let the done, map, d-list-map, and the identifier of the replying action included in the reply message be denoted as m.done, m.map, m.d-list-map, and m.aid, respectively. First G ascertains if the replying action is an orphan by checking m.d-list-map and m.aid against its map and done. If the replying action proves to be an orphan, the reply message is discarded. M.done and m.map are then used to detect and abort local

orphans running at G. Then m.done and m.map are used to update G's own done and map. If the reply was not discarded, the sent d-list-map is merged into A's d-list-map. This is done by just adding to A's d-list-map any entry for a guardian that appears in the piggybacked d-list-map but not in A's d-list-map. Only after these tasks are completed can A start processing the reply.

4.2.5 Refusal Messages

Whenever a handler action is aborted due to orphan detection, a refusal message is sent to the guardian of the call action.

The sending guardian's done and map are included in a refusal message. The guardian receiving a refusal message uses the sent done and map to detect and abort local orphans and to update its own done and map.

4.2.6 Topaction Creation

When a topaction is created, its d-list-map is initialized to have a single entry consisting of the identifier of its guardian paired with its guardian's current crash count.

4.2.7 Local Subaction Creation

When a subaction is created that runs at the same guardian as that of its parent, the subaction's d-list-map is initially a copy of its parent's.

4.2.8 Local Subaction Commit

When a subaction commits to a parent and both run at the same guardian, the subaction's d-list-map is merged into the parent's d-list-map.

4.2.9 Prepare Messages

When a topaction commits, two phase commit is performed. The done and map of the topaction's guardian are piggybacked on the prepare messages of the two phase commit protocol. When a guardian receives a prepare message, it uses the sent done and map to detect local orphans and to update its own done and map. After done and map are updated, they must be written to stable storage before a prepared message can be sent back.

4.2.10 Local Lock Propagation

Before a lock inherited by an action is granted to some local descendant of that action, the d-list-map of the former action must be merged into that of the latter.

4.2.11 Query Responses

When an action desires to obtain a lock acquired by a committed relative, a query message is directed towards the guardian of their closest common ancestor, as described in Section 2.6.3. Suppose the query response indicates that the relative has committed up to the ancestor in question, signifying that the lock can be granted to the action. This query response message must include the sending guardian's done and map, and also the d-list-map of the closest common ancestor. The guardian receiving the query response uses the sent done and map to detect local orphans and update its own done and map. The sent d-list-map is also merged into the lock-requesting action's d-list-map. These tasks must be completed before the lock can be granted.

In other cases, a query response message may indicate that all locks acquired by an action should be released and its versions discarded. Such a query response can result due to either a relative or non-relative attempting to acquire a lock obtained by the action. Such a query response message must include the sending guardian's map and done. The receiving guardian uses the sent map and done to detect local orphans and to update its own map and done. These tasks must be

accomplished before the locks in question are released.

4.3 Unwanted Committed Subactions

A committed action is never an orphan, since an orphan is always an active action. However, the orphan detection algorithm can also detect unwanted committed subactions. Recall that the commit of a subaction is conditional; if some ancestor of the subaction aborts, the subaction's results become unwanted. A committed subaction's results also become unwanted if some ancestor becomes orphaned or some guardian in the committed subaction's d-list-map crashes. In these cases, the committed subaction's results will never be committed as a result of two phase commit; its results will be discarded eventually. The locks acquired by a committed subaction are held until two phase commit for the subaction occurs or a query response message is received indicating that the locks should be released. One would like the locks acquired by an unwanted committed subaction to be released as soon as possible to avoid delaying actions that might want one of these locks.

A guardian can use its map and done to detect local unwanted committed subactions at any convenient time. A committed subaction is known to be unwanted if some ancestor's action identifier appears in the sent done or its d-list-map is out-of-date. Each detected unwanted committed subaction has its locks released and versions discarded.

The above discussion assumes that an action's d-list-map is kept even after the subaction commits; this is not strictly necessary. In this case, only the committed action's identifier is available to check against done.

4.4 Simple Improvements to the Orphan Detection Algorithm

The orphan detection algorithm presented above is both inefficient and impractical. Some inefficiencies in the algorithm will be addressed in this section. The greatest impractical aspect of the algorithm, however, is the size of done and map. Every guardian's done grows without bound since the algorithm never removes any identifier from done. In some imaginable systems, each guardian's map would be enormous -- perhaps containing on the order of a thousand entries. Piggybacking such large dones and maps onto messages increases communication costs to a ludicrous level. The problem of the large size of done and map is not easily remedied. Later chapters present a scheme for cutting down the sizes of these data structures.

4.4.1 Done

There are several ways that the growth of done can be reduced. Each of the modifications to the orphan detection algorithm proposed here is inexpensive in terms of time and uses no additional space.

First of all, the identifier of an action can be deleted from a guardian's done that also contains the identifier of one of the action's ancestors. The presence of the ancestor's identifier in done implies that all its descendants are orphans. Of course, the ancestor's descendants are a superset of any of its descendant's descendants.

Secondly, in some cases it is clearly not necessary to add the identifier of an aborted action to done. One such case is when some ancestor's identifier is already in done. Another more significant case is when the aborted action has no active remote children.

Thirdly, when the second phase of two phase commit is ready to begin for a topaction, its identifier is added to done, given that done contains some descendant's identifier. This is advantageous since the topaction's identifier might replace several of its descendant's identifiers in done; also the topaction's identifier

is shorter than that of any of its descendants. Note that the use of this strategy means that one must be slightly more careful about detecting unwanted committed subactions based on information in done than in the presentation of Section 4.3. Committed subactions that have completed the first phase of two phase commit are not (necessarily) unwanted even though the identifier of their topaction might appear in done.

Fourthly, an orphan detected and aborted as a result of information in map does not necessarily need its identifier added to done. If the map entry that caused the orphan to be detected is for a guardian of one of the orphan's ancestors, then the orphan's identifier need not be added to done. Every one of the orphan's descendants has an out-of-date entry for the ancestor in its d-list-map. Since map and done are always transmitted together, the entry in map for the ancestor's guardian suffices to "catch" all the orphan's descendants.

4.4.2 Limiting the Growth of Done

The above modifications to the algorithm reduce the growth rate of done, but they do not address the problem of done's unbounded growth. Information in map can be used to "garbage collect" information in done and thereby effectively bound its growth. As we shall see, however, such a scheme does not work well enough; done does not stay at a reasonable size.

Recall that every action identifier contains the guardian identifiers of all its ancestor's guardians, as discussed in Section 2.6.4. In this scheme, action identifiers are modified to also include the crash counts of these guardians. An action's identifier thus contains the entries for ancestor's guardians appearing in the action's d-list-map. A guardian, using its map, can eliminate from its done any identifier that contains an out-of-date crash count associated with some guardian's identifier.

To understand why this is the case, consider the orphans whose detection can be caused by the presence of an action identifier A in the done of guardian G. Every

such orphan is a descendant of the action named by A, so every such orphan's d-list-map contains an entry for each of the guardian identifiers in A. This is true since every action's d-list-map contains an entry for each ancestor's guardian. Furthermore, the crash counts associated with the guardian identifiers in A and the d-list-maps of these orphans are the same. Suppose then that A can be deleted from G's done according to an entry for guardian H in G's map. Every orphan detected by A then has an out-of-date crash count associated with H in its d-list-map. Hence every such orphan will be detected by G's map entry containing the more up-to-date crash count for H. Since map and done are always piggybacked on messages together, the identifier A in G's done is redundant, insofar as orphan detection is concerned.

Assuming that every guardian crashes regularly, this scheme solves the problem of done's unlimited growth. Guardians are assumed, however, to crash infrequently. Hence done will still tend to be too large to consider piggybacking it on messages as practical. Since this modification to the algorithm increases the size of action identifiers, it is probably best left unimplemented, since it does not adequately limit done's growth.

4.4.3 D-list-map

The crash counts in d-list-maps are not needed. Every entry in an action's d-list-map also appears in the map of that action's guardian. Furthermore, corresponding entries for the same guardian in the d-list-map and map agree on crash counts. Hence the crash count associated with a guardian identifier in an action's d-list-map can be determined by looking up the guardian identifier in the map of the action's guardian. Note that every time in the algorithm when an action's d-list-map is piggybacked on a message, the map of the action's guardian is also piggybacked on the message. Thus a guardian can determine the crash counts in a received d-list-map from the map received in the same message. Hence the d-list-map can be shortened to a d-list, i.e. just the d-list-map without crash counts.

Actually, this modification to the algorithm becomes invalid when the scheme for controlling the size of map is discussed in a later chapter, since guardians neither maintain nor transmit a copy of the entire map.

4.4.4 Local Lock Propagation

In the algorithm, when an action acquires a lock inherited by a local ancestor, the action's d-list-map must be extended so that it contains all the entries in the ancestor's d-list-map (Section 4.2.10).

However, we now outline a scheme that lowers the cost of lock acquisition by not requiring any d-list-map manipulation when acquiring a lock, if no querying is required. This scheme has two parts. Firstly, when a concurrent child commits to its parent, its d-list-map is merged into that of every one of the parent's local descendants. This step is valid since if the parent is orphaned by a crash of a guardian in its d-list-map, its descendants are also all orphaned. If none of these descendants acquire locks inherited by the parent from the committed child, this step just results in the parent's descendants being detected as orphans possibly sooner than in the algorithm as presented above. Secondly, when a handler action is created, its d-list-map must have the d-list-maps of every local ancestor merged in. This must be done since the handler action (or any of its local descendants) could acquire a lock inherited by some local ancestor.

4.5 Orphan Extermination

This section's discussion is divided into two parts. First, the details involved in actually aborting an orphan are discussed. Second, *stranded actions* are discussed. Stranded actions are created by aborting orphans, and must also be aborted.

4.5.1 How to Kill an Orphan

Aborting orphans when detected is usually a quick and simple matter. Typically, aborting an orphan just involves immediately terminating the action's execution, aborting all its local descendants (youngest first), releasing its locks, and discarding its versions. However, orphan extermination can sometimes be more complicated than this due to the existence of mutex objects, introduced in Section 2.5.

An action holding a lock on a mutex object cannot have its execution abruptly terminated and its mutex lock released, since doing so could leave the mutex object in an inconsistent state. There are two options when faced with the need to abort an orphan that holds a lock on a mutex object. First, the orphan's guardian can be crashed. This will abort the orphan, as well as every other action at the guardian. The recovery process will restore the mutex object the orphan had locked to a consistent state. Second, the extermination of the orphan can be delayed until it releases the lock on the mutex object. Of course, there is no guarantee of when, if ever, the orphan will release the lock. If the orphan does not release the lock within a "reasonable" period, the first option is always available. Recall that the "normal" processing of a message cannot commence until all orphan processing associated with the message is completed -- including the abortion of detected orphans, so waiting for an orphan to release a lock slows the progress of other actions.

4.5.2 Stranded Actions

A child subaction can be an orphan while its parent is not. This can only be true if the child is an uprooted-action. In any case, when the child is detected and exterminated, the parent can be left *stranded*.

Let us first consider the case where the orphaned child and non-orphaned parent both run at the same guardian. Furthermore, suppose that the child is not a call action. Then the child must have been created by an **enter** or **coenter** statement. First consider the case where the child was created by an **enter**

statement. When this child is detected and aborted, the parent cannot be restarted. The child has some *specification* that it is supposed to satisfy. When the parent is restarted after the child terminates, the parent expects data shared with the child to obey this specification. Note that this data could include non-atomic objects. The orphaned child is detected and terminated at an arbitrary point in its execution; hence, there is no guarantee that this specification is satisfied after the child is aborted. The parent cannot be safely restarted, since the parent's proper behavior depends on the child fulfilling its specification. One could imagine somehow signaling the parent that the child has been aborted by the system, indicating that the child's "normal termination" specification has not necessarily been satisfied, but this approach is not taken in Argus. Thus the parent cannot be restarted; it is *stranded*.

Consider the case where the orphaned child was spawned by a **coenter** statement. Again, the same statements about the child not fulfilling its specification apply when the child is detected and abruptly aborted by the system. However, aborting the child need not leave the parent stranded if other concurrent siblings are still active. If one sibling completes by transferring control out of the **coenter**, the parent can be safely restarted, since then all uncommitted siblings are aborted at an arbitrary point anyway. However, if this does not occur, the parent is left stranded.

Let us now consider the case where the orphaned child is a call action. In this case an **unavailable** exception can be signaled to the parent, so the parent can be safely restarted. The **unavailable** exception signals the parent that the handler call could not be completed for some reason. Hence the parent is not left stranded in this case.

When the orphaned child is a handler action, a refusal message should be sent back to the parent's guardian, following the procedure in Section 4.2.5. Otherwise the call action would be left hanging waiting for a reply message from the handler action. If the call action is not an orphan, the call could be attempted again or an **unavailable** exception could be signaled to the parent. Thus neither a call action

nor its parent are left stranded by aborting an orphaned handler action.

Stranded actions should be aborted. If a stranded action holds a mutex lock, it can only be aborted by a crash of its guardian. Furthermore, the abort of a stranded action might also leave its parent stranded.

In the Argus implementation, the position is taken that whenever an orphan is aborted, its closest ancestral handler action or topaction is also assumed stranded and aborted, resulting in the abort of all the handler's or topaction's local descendants. One arrives at this stance by assuming that whenever a child created by a **coenter** is aborted, its parent is left stranded.

Chapter Five

Controlling the Size of Done: Deadlining

One of the greatest impracticalities of the orphan detection algorithm presented in the preceding chapter is the potentially large size of done. Done is piggybacked onto many messages; the communication overhead this entails when done is large is unacceptable. This chapter presents a scheme for keeping done down to a "reasonable" size, called *deadlining*. The actual performance of deadlining depends on several parameters and will be analyzed in a later chapter. However, under reasonable conditions, deadlining does reasonably well.

Deadlining requires *approximately synchronized clocks*. That is, every node must have its own clock and these node clocks must be all approximately synchronized with each other. The greatest possible difference between the readings on any two node clocks at any instant of real time must be bounded; let this upper bound be denoted as ε. In other words, at any given instant of real time, there is a node clock with the lowest reading and a node clock with the highest reading; these readings must not be more than ε seconds apart. Every guardian is assumed to have access to its node's clock, which will be referred to as that guardian's clock.

The magnitude of ε required for deadlining to perform adequately determines if having approximately synchronized clocks is indeed feasible. We envision that an ε on the order of a few minutes in magnitude is acceptable. For networks where communication delays have a small upper bound, a clock synchronization algorithm such as that of Marzullo [Marzullo83] can be used. In networks where message delay is extremely arbitrary, the N.B.S. time dissemination service provided by a satellite and accurate to within 1 ms anywhere in North America could be used to synchronize clocks. This satellite has been used to obtain synchronized clocks in ARPAnet hosts

for the purpose of gathering performance measurements [Seitz83]. The magnitude of ε required by deadlining seems to be realistic.

Before proceeding any further, some terminology is introduced. A *purely local descendant* of an action A is any action B that (1) runs at the same guardian as A, and (2) has no ancestor X such that X does not run at A's guardian but is a descendant of A. See Figure 5-1. An action is considered to be one of its own purely local descendants. The *local root action*, or simply *local root*, of an action A is an action P where (1) P is a handler action or topaction, and (2) A is a purely local descendant of P. A handler action's or topaction's local root is itself. In addition, handler actions and topactions are collectively referred to as *local root actions*.

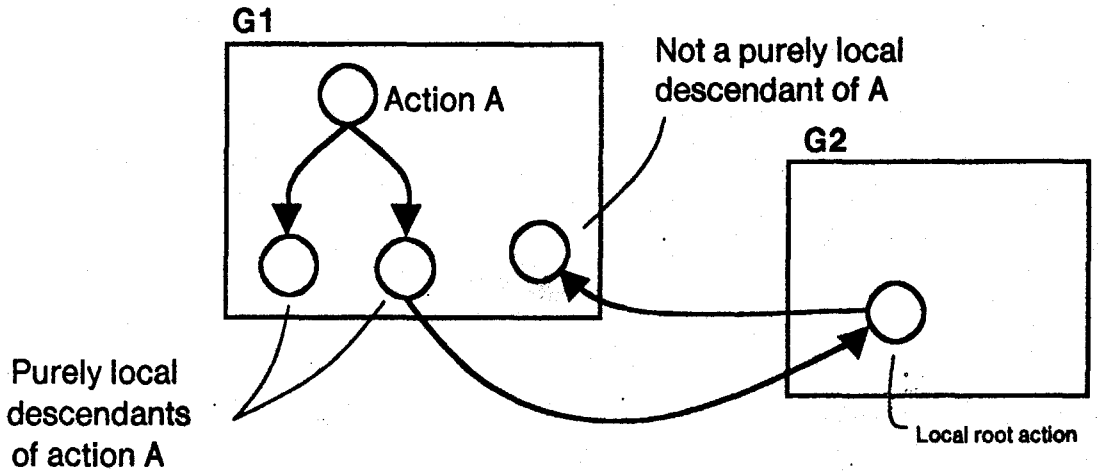


Figure 5-1: Purely local descendants

5.1 Deadlining

The idea behind deadlining is to establish a limit on the amount of time an abort-orphan can survive before being aborted. Then an action identifier need not stay in any guardian's done longer than this time.

In *deadlining*, every local root action is assigned a *deadline time*. The deadline time assigned to a topaction is some arbitrary future time. The deadline time assigned to a handler action is included in its call message; this deadline is the same

as that of the calling action's local root action. Thus a topaction and all its descendant handler actions have the same deadline time.

We will say a local root's deadline *arrives* when a local root's deadline time equals or exceeds the reading on its guardian's clock. An action whose deadline has arrived is also said to be *expired*.

Expired local root actions are aborted, but not necessarily promptly at the time they expire. The abort of an expired local root is postponed until a message with a piggybacked done arrives at the local root's guardian -- with one exception discussed later. To implement this, a guardian checks for expired actions during the orphan detection processing that occurs when a message with a piggybacked done is received. More precisely, when a guardian compares a local root action's identifier against those in a received done to check if the local root is an orphan, it also checks if the local root has expired. If the local root is orphaned or expired, it is aborted. Note that the abort of a local root leads to the abort of all its purely local descendants, as dictated by the procedure for aborting actions in Section 4.2.2.

An expired local root is not aborted when a reply message is received, provided that the reply message is directed to one of the local root's descendants. This is the one exception alluded to in the above paragraph.

A guardian discards any call messages it receives that include a deadline time that has passed, according to its clock.

5.2 Deleting Identifiers From Done

Deadlining's goal is to permit identifiers to be deleted from done within a "reasonable" period. This section discusses how the deadlines associated with actions can be used towards this end.

First of all, the done data structure must be modified somewhat. Done must

now be a set of *tagged* action identifiers. When an action's identifier is added to done, it is tagged with the deadline time of the aborting action's local root. These tags are ignored when identifiers in done are used by the orphan detection algorithm of the last chapter.

An identifier is deleted from a given guardian's done ε seconds after its tagged time passes, according to that guardian's clock. The tag associated with an identifier in done thus indicates when the identifier can be deleted -- ε seconds after the tagged time. In an implementation, guardians need not delete identifiers promptly at the time indicated by their tags, but can wait until any convenient time.

Let us now informally examine the correctness of the above rule for deleting identifiers from done. The orphan detection algorithm, together with this rule and deadlining, will be referred to in the following as the "deletion rule algorithm." First assume that the abort-orphan detection algorithm from the last chapter is correct, i.e. it detects orphans before they can view inconsistent data. The deletion rule algorithm is valid if every orphan is detected at least as soon as the plain abort-orphan detection algorithm would detect it. Note that our notion of "correctness" is restricted to the property that orphans are detected before they can view inconsistent data -- we do not consider the notion that a healthy action should not be aborted due to orphan detection. The deletion rule algorithm clearly violates this notion.

An orphaned and expired local root action is aborted when the first message with a piggybacked done arrives at the local root's guardian, unless this message is a reply to one of the expired local root's descendants. First consider the case where the message is not a reply. In the plain abort-orphan detection algorithm, the piggybacked done might or might not have contained an identifier of one of the orphaned local root's ancestors. Since the orphaned and expired local root is aborted in either case, it is detected as soon or sooner than the plain orphan detection algorithm would.

Now consider the case where the message is a reply from a descendant of the

local root. In the plain algorithm, the piggybacked done on the reply would not contain an identifier of one of the local root's ancestors. If this done did contain such an identifier, the replying handler action would never have been allowed to complete and reply in the plain algorithm; it would have been detected and aborted. Thus the plain algorithm would not have detected the orphan at this point. Hence orphaned and expired actions are detected just as quickly as in the plain algorithm.

Let us now consider whether an orphaned local root is detected properly before it expires. Could a piggybacked done be received at an orphaned and unexpired local root action's guardian that would have contained an ancestor's identifier in the plain algorithm, but that does not in the deletion rule algorithm? This could occur only if an identifier of one of the orphaned local root's ancestors was deleted from some guardian's done before the local root expired. Suppose that this indeed occurred; this assumption will now be shown to lead to a contradiction. Let the orphaned local root's deadline be denoted as τ. The tag on the deleted identifier must have been τ, since all related local root actions share the same deadline. Hence some guardian's clock read $\tau + \varepsilon$ before the clock where the orphaned local root is running read τ, since some guardian deleted the local root's ancestor's identifier before the local root expired. Suppose the local root's guardian's clock read τ-σ (σ>0) at the moment the identifier was deleted. Hence two clocks differed by $\varepsilon + \sigma$ at some instant, violating the assumption that clock differences are bounded by ε. Thus an orphaned and unexpired action is aborted no sooner or later than by the plain orphan detection algorithm.

The above argument is not quite complete -- there is still the issue of call messages originating from orphaned actions to consider. The issue here is whether or not a call message that would be refused in the plain orphan detection algorithm could be accepted in the deletion rule algorithm. A call message including a deadline time that has passed according to its receiver's clock is discarded outright. Thus the danger here is a call message being accepted that would have been refused by the plain algorithm when the time on the receiver's clock is less than the deadline

time in the call message. Suppose that this occurred. Let τ denote the deadline time included in the call message. As in the case above, some guardian must have deleted an identifier tagged with τ before the receiving guardian's clock read τ. Hence again there are two clocks that are more than ε seconds apart at some instant. Thus call messages from orphaned actions are properly handled by the deletion rule algorithm.

The reader should note that orphaned actions are not always detected just as quickly as they would be by the plain algorithm. This is due to the fact that call messages from expired orphans (or non-orphans) are ignored. In the plain algorithm, such a call message could result in a refusal message being sent back. When received, this refusal message could result in an orphan being detected; since the deletion rule algorithm does not send a refusal message back in this case, this orphan is not detected as soon as in the plain algorithm. But this does not detract from the correctness of the deletion rule algorithm. We could simply pretend that refusal messages are always lost; then orphans are detected by the deletion rule algorithm at least as soon as they are by the plain algorithm.

This argument concerning the correctness of the deletion rule algorithm implicitly makes the assumption that clocks are never set back. Due to running a clock synchronization algorithm, etc., it might be occasionally necessary to decrease the reading on a guardian's clock. However, there is a trivial patch to make the deletion rule algorithm work correctly even when clocks can be set back: before setting a clock back, all expired actions must be aborted. A guardian also must make a note of the time just before setting back its clock; any call messages coming in with a deadline time less than this noted time are discarded. Once the guardian's clock exceeds this noted value, it can unnote the value; the guardian no longer needs to keep track of it.

Clock wrap around can be viewed as setting a clock back. Note that the above is an unacceptable way to handle clock wrap around, however, since a guardian

would never accept call messages again after its clock wraps around. But we assume that the values read from clocks and timestamps contain sufficient bits to keep clock wrap around from occurring in practice.

5.3 Deadline Extension

Aborting any action after it expires can lead to the abort of healthy non-orphans -- an unpalatable situation. To prevent healthy actions from being aborted due to expired deadlines, we now present a scheme for *extending*, i.e. increasing, the deadlines of actions that are not abort-orphans.

Basically, deadline extension works as follows. When a local root action nears its deadline, its guardian attempts to extend its deadline by sending a message to the guardian of the local root's parent. This message is propagated up the call chain to the topaction's guardian. Along the way, the health of the action's ancestors are checked. Then a message indicating that the local root's deadline can be extended is propagated back down to the guardian, if all the action's ancestors proved to be healthy.

The deadline extension protocol is based upon three types of messages: **orphaned?**, **not-orphaned**, and **orphaned** messages.

When a handler action nears its deadline, its guardian sends an **orphaned?** message to the guardian of the handler action's parent. This **orphaned?** message includes the identifier of the handler action and its deadline as well. The deadline is included so that the protocol properly handles lost, delayed, and duplicated **orphaned?** messages.

When a topaction nears its deadline, its guardian increases the topaction's deadline to some arbitrary future time. A topaction can never be an abort-orphan. **Not-orphaned** messages are then sent to the guardians running handler actions for call actions among the topaction's purely local descendants. Each **not-orphaned**

message contains the identifier of a call action and the new deadline of the topaction. One **not-orphaned** message is sent for each such call action.

A guardian that receives an **orphaned?** message takes the following steps. Let the handler action whose identifier is included in the message be denoted as m.handler; let the deadline included in the message be denoted as m.deadline. Let m.handler's parent, a call action, be denoted as m.call. If m.call's local root is not active, an **orphaned** message, containing m.call's local root's identifier, is sent back to m.handler's guardian. If the local root is active, the receiver then examines the deadline included in the **orphaned?** message. There are two possible cases at this point -- either m.deadline is less than the deadline of m.call's local root or these deadlines are equal.

Let us first consider the former case. In this case, first a **not-orphaned** message is prepared -- but is not actually sent quite yet. This **not-orphaned** message contains m.call's identifier and the deadline value of m.call's local root. The health of m.call is then ascertained; if m.call has been aborted, an **orphaned** message is sent back to m.handler's guardian and processing of the **orphaned?** message terminates. This **orphaned** messages includes m.call's identifier. Otherwise, the guardian sends the **not-orphaned** message it previously prepared to m.handler's guardian.

In the case that m.deadline equals that of m.call's local root, the health of m.call is ascertained. If m.call has been aborted, an **orphaned** message, containing m.call's identifier, is sent to m.handler's guardian. Otherwise, the deadline extension procedure starts for m.call's local root -- an **orphaned?** message is sent to its parent's guardian, etc. Of course, it is possible that the deadline extension procedure has already previously started for m.call's local root; no action needs to be taken in this case.

An **orphaned** message contains an action identifier of an action. When a guardian receives an **orphaned** message, it aborts all descendants of the named

action. In addition, the receiver sends an **orphaned** message to every guardian running a remote handler action for any call action aborted due to receiving the **orphaned** message. Each **orphaned** message sent contains the identifier of one of these call actions.

A guardian that receives a **not-orphaned** message takes the following steps. A **not-orphaned** message contains a deadline value and the identifier of a call action whose child the message is directed to; let these be denoted as m.deadline and m.call, respectively. If m.deadline is less than or equal to that of m.call's child, the **not-orphaned** message is old and is ignored. Otherwise, the deadline of the m.call's child is set to m.deadline, and **not-orphaned** messages are sent to all of the guardian's running a handler action for some purely local descendant call action of m.call's child. A **not-orphaned** message is sent for each such call action; each message contains the identifier of one of these actions and m.deadline. Of course, the above discussion assumes that m.call's child has not terminated at the time the **not-orphaned** message arrives; the **not-orphaned** message is ignored if this is the case.

Let us now consider the correctness of deadline extension -- is the rule for deleting identifiers from done based upon tag values still valid? First note that a local root's deadline is greater than or equal to any deadline associated with any of its descendants. A local root's deadline is increased only when an appropriate **not-orphaned** message is received. Such a message is sent only after the deadlines associated with all of the local root's proper ancestors that happen to be local root actions have been increased to the deadline value included in the message. Also note that before a **not-orphaned** message directed to a particular handler action is actually sent, the health of the handler's call action is checked. If it is aborted, the message is not sent. Thus when an action's identifier is added to done and tagged with τ, all of the action's proper descendants that happen to be local roots have deadline values no greater than τ. Hence all these descendants will expire before the identifier is removed from any guardian's done, showing that deadline extension

works properly.

Let us now discuss the issue of lost messages. In order to guard against lost **orphaned?** messages, a guardian should retransmit a **orphaned?** message if it has not received a response within a "reasonable" period. To protect this retransmitted message from being lost, an acknowledgment could be requested. A lost **orphaned** message is not harmful; a properly received **orphaned** message merely prevents its recipient from retransmitting **orphaned?** messages. A lost **not-orphaned** message could cause a healthy action to be aborted. The retransmission of **orphaned?** messages, however, makes the protocol resilient to lost **not-orphaned** messages.

5.4 When to Start Deadline Extension

Thus far, it has been said that deadline extension should be undertaken when a local root action "nears" its deadline. This section discusses just when deadline extension should actually be undertaken.

The amount of time allotted to extend the deadline for a local root action should be based on the depth of the local root action in the call chain, i.e. it should be based upon the number of ancestors the local root action has that are handler actions. In the deadline extension protocol, **orphaned?** messages must propagate up the call chain and then **not-orphaned** messages must propagate back down. The time required to successfully extend a local root action's deadline is therefore proportional to its number of handler action ancestors.

A local root might be so deep in a call chain, however, that there is not enough time to propagate messages up and down the call chain before its deadline arrives. This problem is addressed in a later section.

In this scheme, a local root action and its purely local descendants are permitted to run while the deadline extension protocol is being run on their behalf. Thus, an action is permitted to make a handler call even as its local root nears its

deadline. Unfortunately, if the action making a call is deep in the call chain and its local root is close to its deadline, there might not be sufficient time to propagate a **not-orphaned** message down to the handler action created by the call before it expires. In order to prevent the creation of handler actions that are unlikely to successfully have their deadlines extended, a handler call made by an action whose local root is "close" to its deadline should be delayed until the local root's deadline is extended. The call message generated by such a handler call is queued at the caller's guardian until the deadline is extended; if the deadline is not extended the message is discarded.

In the deadline extension protocol, the propagation of **orphaned?** messages up a call chain is not crucial; the propagation of **not-orphaned** messages down a call chain actually causes deadlines to be extended. **Orphaned?** messages are propagated up to a topaction's guardian in order to force it to start propagating **not-orphaned** messages back down while there is still sufficient time for these messages to reach the lowest extents of the call chains. We now suggest a scheme that reduces the need of using **orphaned?** messages to stimulate a topaction's guardian. This is desirable since then the deadline extension procedure only need start for a local root in sufficient time for a **not-orphaned** message to propagate down to it, as opposed to in sufficient time to both propagate an **orphaned?** message up and then a **not-orphaned** message down. This can be accomplished if a topaction's guardian "predicts" the length of the topaction's longest associated call chain, and starts the process of propagating **not-orphaned** messages in sufficient time for these messages to propagate down a call chain of that length. The most straightforward means of implementing this idea is to have a guardian always "predict" the same length. A typical value might be five. Then suppose it takes Q seconds to propagate a **not-orphaned** message down a call chain of this fixed "predicted" length. Suppose a guardian then extends a topaction's deadline and sends out the appropriate **not-orphaned** messages $Q+\varepsilon$ seconds before the topaction's deadline arrives. Then there is absolutely no need to transmit any **orphaned?** messages if a call chain is within this "predicted" length -- assuming that

no **not-orphaned** messages are lost. To guard against lost **not-orphaned** messages, each local root transmits an **orphaned?** message when it has not received a **not-orphaned** message in an appropriate amount of time. If a call chain is actually longer that this fixed "predicted" value, deadline extension must start for the deeper actions in the chain in sufficient time for **orphaned?** messages to propagate up and then **not-orphaned** messages to propagate down the call chain.

Note that this prediction of call chain length also lightens the restriction on an action making handler calls while its local root is "close" to its deadline. An action can make a handler call no matter how close it is to its deadline as long as the handler call does not extend the call chain length beyond the "predicted" value.

One might question why the topaction's guardian starts the deadline extension process at $Q + \varepsilon$ instead of just Q seconds before the deadline arrives. This is done to insure that the **not-orphaned** message starts its journey in enough time to reach a guardian with a clock that grossly disagrees with that of the topaction's guardian when ε is large relative to Q. Figure 5-2 illustrates what could happen if deadline extension was only started Q seconds before deadlines. Consider what could occur if a topaction was running at the "slow" guardian and some descendant at the "fast" guardian.

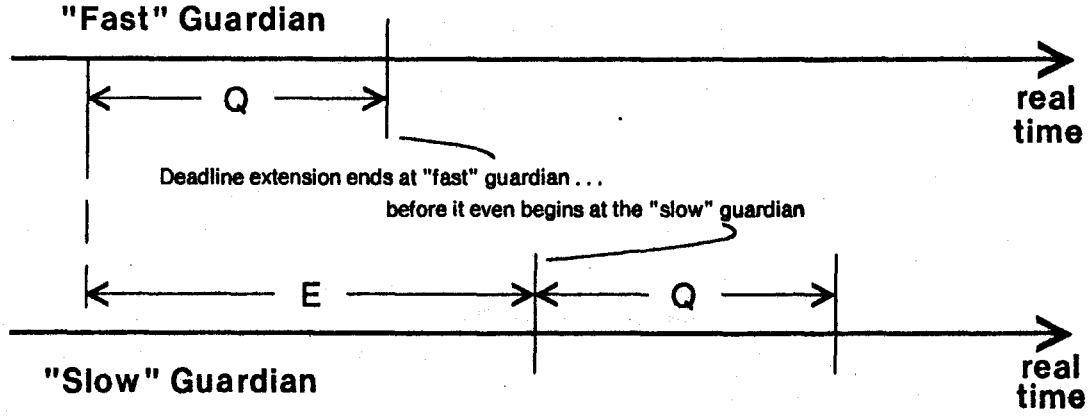


Figure 5-2: Why deadline extension starts at $Q + \varepsilon$ seconds before deadline

5.4.1 Guardian Isolation

Consider a guardian that is trying to extend a local root action's deadline, but is unable to do so before the deadline arrives. Does the guardian really need to abort the action when a message with a piggybacked done arrives? Actually, the guardian does have an alternative.

Instead of aborting such an action, the guardian could postpone all processing of incoming messages with piggybacked dones until the action's fate was ascertained; note, however, that reply messages from descendants of the action can be processed as normal. After the local root expires, any action identifiers indicating that the action is an orphan potentially have been deleted from many guardians' dones. Hence any incoming message, except in the case of certain reply messages, might come from a guardian that did delete such an action identifier from done.

This is a rather poor method since it delays the progress of many actions in the system on account of a single local root action and its purely local descendants. But if a guardian judges that perhaps deadline extension could be completed for an action in just a small amount of additional time, then perhaps it is worthwhile.

One might believe that just suspending a local root action and all its purely local descendants after the local root expires is an acceptable method of allowing the deadline extension process to continue. Processing of incoming messages proceeds as normal while the expired action and its purely local descendants are suspended. However, this can lead to a rather subtle problem. Some other action created by an incoming handler call and passed information that the suspended action was remotely aborted could "see" the suspended orphan; the action could "see" an action that was supposedly aborted.

The following illustrates the danger of just suspending an action after it expires. Suppose parent P at guardian **GP** makes a handler call to guardian **G**, creating action P.1 at **G**. Then the handler call is aborted at **GP**. P.1 is now an orphan. Suppose orphan P.1 locks atomic object O at **G**. Then say that the action

identifier indicating P.1 is an orphan is deleted from GP's done. P.1 is suspended at this time. Suppose then that P then makes another handler call to G, creating action P.2. P.2 sees that the lock on O is still held. But suppose that only guardian GP is the only guardian in the system that ever makes handler calls to G and furthermore that GP never allows more than one handler call at a time to be active at G. Thus P.2 expects to find the lock on O free, and hence has evidence that an orphan is lurking about when it finds the lock held.

5.5 Deadline Extension for Deeply Nested Calls

There is a problem with the deadline extension scheme presented in the preceding sections. A call chain could conceivably get so long that there would only be just enough time to propagate **orphaned?** messages up and **not-orphaned** messages back down the call chain before a newly established deadline arrived. The action at the bottom of this long call chain would not be permitted to make any handler calls. Hence, a limit has been effectively placed upon how deep a call chain can become. This limit depends upon the choice of the amount of time between deadlines.

But is this actually a problem in practice? Experience shows that call chains do not get very deep unless recursion is present. Recursion does not appear to be a practical programming technique in the Argus environment, since substantial overhead is associated with each such remote recursive call. Hence it seems that in any practical case, there will be plenty of time to do deadline extension with any reasonable choice of the time between deadlines. Also note that in many conventional programming language implementations, the permissible depth of calls is bounded due to a fixed stack size. Thus the fact that call depth is limited does not seem to be of practical significance; however, we subsequently explore ways of alleviating this problem.

The first method for coping with deeply nested recursive calls involves

increasing the inter-deadline time for any topaction with such a deeply nested call chain. The second method is a modification of the deadline extension protocol that can vastly reduce the number of messages and the amount of time needed to do deadline extension for actions in a long call chain.

5.5.1 Increasing the Time Between Deadlines

A topaction's deadline is extended by some arbitrary amount. Typically, this amount should be more than enough to permit the deadline extension procedure to complete for non-recursive calls. However, for extremely deeply nested calls, this might not be the case. An action might be so deeply nested that there is insufficient time to propagate **orphaned?** messages up the call chain and then **not-orphaned** messages back down to the action before its deadline arrives.

To alleviate this problem, a topaction's guardian needs to take into account lengths of outstanding call chains when establishing a new deadline for a topaction. Unfortunately, such information is not normally available at the topaction's guardian. The following discusses who passes this information up to the topaction's guardian and when.

First of all, let us make the rule that guardians never decrease the duration between deadlines for a particular topaction. In other words, if a topaction ran for τ seconds between its last two deadlines, it will run for at least τ seconds before its next deadline.

Suppose an action's handler call is postponed since the system judges that the handler action so created would not have a good chance of successfully completing the deadline extension process. If this happens when the calling action is "close" to its deadline, it just means that the action chose a poor time to do a handler call. If this occurs when the calling action is "far" from its deadline, however, it means that the action is at the end of an extremely long call chain. In this case, the duration between successive deadlines should be increased to allow the call chain to increase

its length. This can be done by having such an action communicate with its topaction's guardian, informing it of the call chain's length. The topaction's guardian can then set the next deadline for the topaction appropriately.

Also, for very deep call chains, increasing the Q values for the local root actions in the chain would be beneficial. The time needed to extend deadlines for local roots in a long call chain could be reduced by increasing the Q values used for these actions. News about a new Q value can then be propagated down the call chain on **not-orphaned** messages.

5.5.2 Short-Circuiting Deadline Extension Protocol

We now present a deadline extension protocol that can drastically reduce the number of messages needed to do deadline extension for recursive handler calls. In practice, recursion is the sole source of deeply nested calls. This scheme is called *short-circuiting*. In the worst case, this scheme does no worse that the plain deadline extension protocol, in terms of the number of messages sent.

Short-circuiting is an embellishment to the plain deadline extension protocol presented in the previous section. **Orphaned?** messages are still sent out basically as before. The major change is the information tacked onto **not-orphaned** messages.

In the plain deadline extension protocol, a **not-orphaned** message only indicates that a single local root action is not an abort-orphan and can have its deadline extended. But in short-circuiting, a single **not-orphaned** message potentially indicates that several local root actions at a guardian can have their deadlines extended. This is done by placing additional information on **not-orphaned** messages.

Consider the recursive call chain depicted in Figure 5-3. The call chain repeatedly loops through guardians **GA**, **GB**, and **GC**. The deadline extension

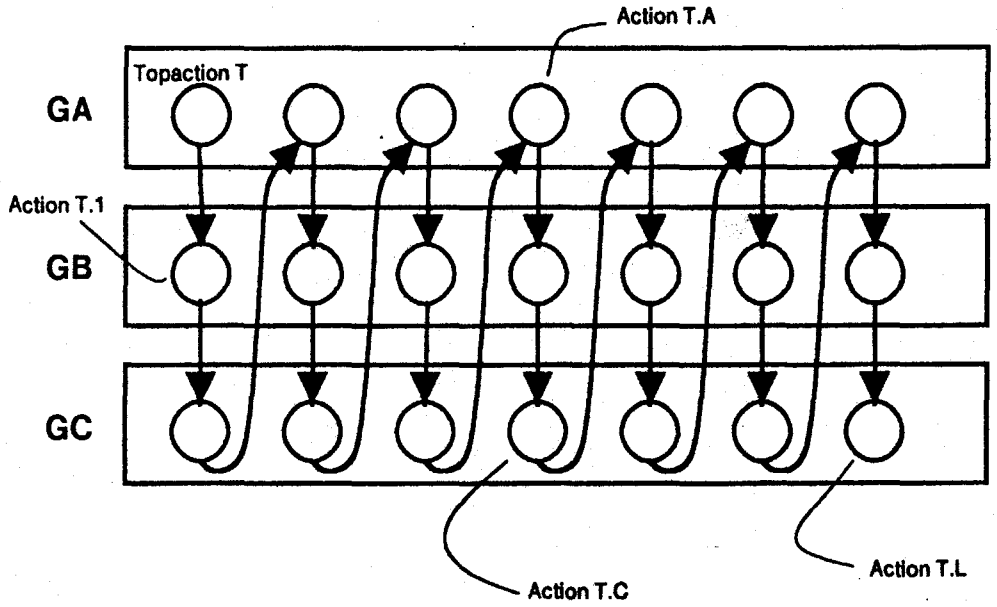


Figure 5-3: Recursion example

protocol of the previous section propagates a **not-orphaned** message all the way down this call chain, repeatedly looping through the three guardians. The short-circuiting protocol, on the other hand, only propagates a **not-orphaned** message completely around the loop once. The **not-orphaned** message in short-circuiting propagates from **GA** to **GB**, to **GC**, back to **GA**, and finally to **GB**.

In short-circuiting, each **not-orphaned** message carries a history of the guardians that have propagated it. This history takes the form of a sequence (i.e., ordered list) of guardian identifiers. Each guardian that propagates a given **not-orphaned** message adds its guardian identifier to the end of this sequence.

In short-circuiting, a single **not-orphaned** takes the place of several **not-orphaned** and **orphaned** messages of the plain protocol. A **not-orphaned** message is directed to all the descendants of a given topaction at a guardian, instead of just a particular handler action. Therefore, each **not-orphaned** message carries the identifier of a topaction, instead of the identifier of a call action as in the plain deadline extension protocol. In Figure 5-3, the **not-orphaned** messages propagated down the illustrated call chain contain T's identifier.

Information concerning aborted actions is included in **not-orphaned** messages. This information is in the form of a set of identifiers of aborted actions. Each guardian that propagates a given **not-orphaned** message adds to this set all the action identifiers in its done that belong to actions descended from the topaction whose identifier is included in the message. It is this information concerning aborted actions that permits a single **not-orphaned** message to take the place of several messages the plain protocol would send.

When a guardian receives a **not-orphaned** message, it extends the deadline of any local root action descended from the topaction whose identifier is included in the message, given that the local root satisfies the following two conditions. First, no identifier of one of the local root's ancestors appears in the set of aborted actions included in the message. Secondly, each one of the local root's ancestors either ran at a guardian whose identifier appears in the guardian identifier sequence included in the message, or ran at the receiving guardian itself.

There is a problem with short-circuiting as it has been presented thus far. A **not-orphaned** message pretends to carry all information about aborts of descendants of a given topaction at the guardians that have propagated the message. But, in reality, this is not the case; some action can abort at one of these guardians after the **not-orphaned** message is propagated. Consider Figure 5-3 again. Suppose no descendant of T has aborted. Topaction T nears its deadline, guardian **GA** extends T's deadline, and sends a **not-orphaned** message to guardian **GB**. Guardian **GB** extends only the deadline of action T.1, and propagates the message to **GC**. **GC** then extends the deadline of every descendant of T running locally. But note that at this point the deadlines of several actions at **GC** are greater than those of several of their ancestors. Consider action T.C, for example; its deadline is greater than that of its ancestor T.A. If T.A were to abort at this point, its identifier would be deleted from done before T.C's deadline expired. In order to remedy this problem, a *done-tag* is associated with every local root action.

In short-circuiting, when an action is aborted, its identifier in done is tagged with its local root's *done-tag*, instead of its local root's deadline as before. A local root action's done-tag is initially set to be the same as its deadline. Local root actions still have a deadline associated with them. When a **not-orphaned** message is propagated by a guardian, it increases the done-tag of all descendants of the appropriate topaction to the deadline value included in the message. Then in the situation recounted above, action T.C's deadline would indeed be greater than that of its ancestor, T.A, but T.A's done-tag equals T.C's deadline. Hence if T.A aborted at this point, its identifier would not be deleted from any guardian's done until after T.C expires.

There still remains a similar problem caused by call messages that arrive at a guardian after a **not-orphaned** message is propagated. After a **not-orphaned** message including fields m.top, a topaction identifier, and m.deadline, a deadline value, is sent out from a guardian, any descendant of m.top that runs at the guardian must have its identifier stay in done until m.deadline if aborted. For descendants running at the guardian at the time the message is sent, this is accomplished by upping their done-tags to m.deadline. This does not properly handle handler actions related to m.top created at the guardian after the message is sent out. In order to properly set these handlers' done-tags, a *done-tag-set* is maintained by each guardian. The done-tag-set is a set of *done-tag-entries*. A done-tag-entry has two fields -- a topaction identifier and a done-tag value. When a **not-orphaned** message is received by a guardian, a new done-tag-entry is created with its fields set to m.top and m.deadline. This done-tag-entry is then added to the done-tag-set. The done-tag-set is maintained so that it never contains two done-tag-entries for the same topaction. A done-tag-entry with a lower done-tag is eliminated in favor of a done-tag-entry for the same topaction with a higher done-tag. Any done-tag-entry can be deleted from the done-tag-set at the done-tag value. Whenever a call message arrives at a guardian, it searches through its done-tag-set for a done-tag-entry with the caller's topaction's identifier. If there is such an entry, the created handler action has its done-tag set to the value in the done-tag-entry -- unless the deadline value

included in the call message is greater, in which case it is set to this deadline value.

Let us now recount all the events that occur when a guardian receives a **not-orphaned** message. Let the new deadline, topaction identifier, guardian identifier sequence, and action identifier included in the message be denoted as in.deadline, in.top, in.gseq, and in.aborts. Firstly, in.aborts is merged into the guardian's done and any actions that are descendants of those in in.aborts are aborted. Then the done-tag of each local handler action descended from in.top is increased to in.deadline, unless its done-tag is already greater than in.deadline. This latter check is needed so that old **not-orphaned** messages have no effect. Then the deadlines of all local handler actions descended from in.top are changed to in.deadline if the following two conditions are met. First, in.deadline must be greater than the handler action's current deadline -- again, this is in the interests of ignoring repeated **not-orphaned** messages. Second, the set composed of all the handler action's ancestor's guardians must be a subset of the guardians appearing in in.gseq together with the handler action's guardian.

After taking the above steps, the **not-orphaned** message receiver itself sends out **not-orphaned** messages. Let the information included in these outgoing messages be denoted as out.top, out.aborts, out.gmap, and out.deadline. Out.top and out.deadline are the same as in.top and in.deadline. Out.aborts is in.aborts with any identifiers of descendants of in.top in done added. Out.gmap is in.gmap with the receiver's guardian identifier concatenated onto the end. The guardians that are candidates to receive a **not-orphaned** message are those running a remote child of a purely local descendant of a local root action that just had its deadline changed to in.deadline above. However, not all these guardians are sent a message; they are screened as follows. A **not-orphaned** message is not sent to guardian DG if there exist sequences of guardian identifiers X and Y such that out.gseq = X || DG || Y, where every guardian that appears in Y appears in X and "||" denotes concatenation. If this test is satisfied, the set of aborted action identifiers included in the message

has not grown any[4] since DG last received the message.

Let us now detail the events that occur when a topaction nears its deadline or its guardian receives an appropriate **orphaned?** message (that is not old). The topaction's guardian first determines and sets the topaction's new deadline. It then changes the done-tag of every local handler action descended from the topaction to the new deadline value. Then a **not-orphaned** message is sent to every guardian that is running a remote subaction whose parent is some purely local descendant of the topaction. Every one of these **not-orphaned** messages contains the following information. The guardian sequence included in these messages consists of a single guardian -- that of the topaction. The set of action identifiers consists of all action identifiers in the topaction's guardian's done belonging to descendants of the topaction. Also, the topaction's identifier and new deadline are included in the messages.

As in the plain deadline extension scheme, when a local root nears its deadline, an **orphaned?** message is sent to its parent's guardian. However, if there are several related local roots at a guardian whose call actions all reside at the same guardian, an **orphaned?** message need only be sent on behalf of the eldest. **Orphaned?** messages still serve as insurance against lost **not-orphaned** messages. If a local root has not had its deadline extended within a "reasonable" period after nearing its deadline, it retransmits an **orphaned?** message.

A lost **not-orphaned** message should cause its sender to eventually receive an **orphaned?** message. The appropriate response to this **orphaned?** message is the lost **not-orphaned** message. Therefore, a guardian must somehow remember any **not-orphaned** message it transmits so that it can be retransmitted if necessary. To do this, whenever a guardian sends a **not-orphaned** message, the guardian

---

[4]Actually, this set might have grown some, but DG still does not need to learn of these additional aborted actions since their identifiers in done are tagged with the new deadline value contained in the message.

associates the message with every local root action that had its deadline changed due to receiving the message. This **not-orphaned** message replaces any such message previously associated with any one of these local roots.

When a guardian receives an **orphaned?** message it checks if the deadline included in the message is less than that of the local root of the parent of the handler action the message was sent on behalf of. If this is the case, a **not-orphaned** message transmitted previously has been lost or delayed, and the **not-orphaned** message associated with the appropriate local root is retransmitted. Otherwise, this local root is considered to have "neared" its deadline and the appropriate steps are taken, i.e. an **orphaned?** message is sent, etc.

Chapter Six

Controlling the Size of Map: Deadlining

There are two impractical aspects to the orphan detection algorithm of Chapter Four -- the large sizes of done and map. This chapter presents a deadlining scheme, somewhat similar to that previously presented for done, that keeps the size of map small.

6.1 Map Deadlining

In map deadlining, every local root action has a second deadline associated with it. In order to differentiate this deadline from the one associated with local root actions for the purpose of controlling the size of done, the former deadline will be known as a *map-deadline* and the latter as a *done-deadline*.

The map-deadline assigned to a topaction is some future time; however, the map-deadline *period* must be the same for all topactions in the system. The map-deadline period is the time between a topaction's creation and its map-deadline.

Call messages include the map-deadline value of the calling action's local root. A handler action's map-deadline is set to the value included in its call message. Hence a topaction and its descendant handler actions all have the same map-deadline time.

When a local root action's map-deadline arrives, that action is said to be *map-expired*, or simply expired. When a local root action becomes map-expired, it is aborted along with all its purely local descendants. The local root need not be aborted the exact instant its map-deadline arrives, but it must be aborted before any message that arrives with a piggybacked map can be processed.

A guardian discards any call message it receives if the map-deadline included in the message has passed, according to the guardian's clock.

We envision the map-deadline period as being a relatively large value; virtually no action should ever find itself closing in on its map-deadline.

6.2 Deleting Entries From Map

As in the deadlining scheme for done, map entries are tagged with a time stamp. These tags are ignored insofar as the orphan detection algorithm of Chapter Four is concerned.

When a guardian recovers from a crash, it places an updated entry for itself in its map. The guardian tags this entry with the current time plus the map-deadline period. We assume that clocks do not fail during crashes; they keep on ticking reliably even while their node is down.

An entry in a given guardian's map can be deleted ε seconds after the entry's tagged time, according to that guardian's clock. The entry need not be deleted promptly; the guardian can wait until a convenient time.

In map deadlining, the abbreviation of d-list-maps to d-lists, as proposed in Section 4.4.3, is no longer valid. Due to the above rule for deleting entries from map, a guardian's map is no longer necessarily a superset of each of its local action's d-list-maps.

Let us now consider the correctness of this scheme. The orphan detection algorithm from Chapter 4, together with map-deadlines and the above map entry deletion rule, is referred to in the following as the "deletion rule algorithm." The question is whether or not a crash-orphan is detected by the deletion rule algorithm as quickly as it would be by the plain orphan detection algorithm.

First consider the case of a map-expired local root action with a crash-

orphaned purely local descendant. The local root and all its purely local descendants, including the crash-orphan, will be aborted by the time the first message with a piggybacked map arrives. Hence the plain algorithm detects a crash-orphan with a map-expired local root no faster than the deletion rule algorithm.

Now consider the case of a crash-orphan whose local root has not yet expired. Suppose a piggybacked map arrives that would have contained an entry identifying the crash-orphan as such in the plain algorithm, but that does not in the deletion rule algorithm. For this to be true, some entry for a guardian appearing in the crash-orphan's d-list-map was deleted from some guardian's map. We proceed to show that this leads to a contradiction. Suppose this deleted entry was for guardian G, and was tagged with the time τ; thus the crash-orphan's d-list-map contains an entry for G with an out-of-date crash count. Since the clock of the guardian that deleted the entry must have read at least $\tau + \varepsilon$ at the time of the deletion, the crash-orphan's guardian's clock must read at least τ when the piggybacked map missing the entry arrives. Since the crash-orphan's local root has not expired when the message arrives, its map-deadline must be greater than τ. Since a topaction and all its descendant local roots share the same map-deadline, the map-deadline of the crash-orphan's topaction must also be greater than τ. But then the topaction must have been created after G recovered. G recovered at τ-P, where P denotes the map-deadline period. The topaction was created after τ-P, since its map-deadline is greater than τ. But if the topaction was created after G recovered, none of its descendant's d-list-maps can possibly contain an entry for G with an old crash count, contradicting the fact that the crash-orphan's d-list-map does indeed contain such an entry.

This correctness argument is not quite complete until call messages are considered. Call messages carrying an expired map-deadline are discarded. The danger is that a call message carrying a map-deadline that has not expired might be accepted when in the plain orphan detection algorithm it would have been refused. An argument similar to the one above can be made showing that for such an anomaly

to occur, the calling action's topaction must have been created after the guardian whose entry was deleted from map recovered, and hence the d-list-map included in the call message could not possibly contain an outdated entry for the deleted guardian.

6.3 Map-Deadline Extension

Since we assume crashes occur infrequently, the map-deadline period can be quite large while still keeping map at a reasonable size. Very few actions should ever map-expire, if any. Hence map-deadline extension is a somewhat less critical issue than is done-deadline extension. In any case, we now present a map-deadline extension scheme.

Map-deadline extension works basically as follows. When a local root action nears its map-deadline, its guardian queries all the guardians appearing in the local root's d-list-map and those appearing in the d-list-maps of its purely local descendants. If the local root's guardian discovers that none of these guardians have crashed, the local root's map-deadline is increased by the map-deadline period. As mentioned earlier, the map-deadline period is a constant, and must be uniform across all guardians in a system.

Let us now discuss the map-deadline extension procedure in detail. When a local root action nears its map-deadline, its guardian first constructs an *e-map* and associates it with the local root action. The *e-map* is a table that associates guardian identifiers with either crash counts or the special value **null**, and is used to keep track of guardians' responses to queries. The *e-map* is constructed by taking the union of the local root's d-list-map with those of all its purely local descendants, and then mapping each guardian into the special value **null**. Once the e-map has been constructed, the local root's guardian queries each guardian in the e-map for its current crash count. Of course, the local root's guardian can immediately update the entry for itself in the e-map. As the guardian acquires information about these

guardian's crash counts, it updates entries in the e-map.

The protocol used to query guardians of their crash count is quite straightforward. Each guardian appearing in the e-map is sent a **crashed?** message. A **crashed?** message contains the identifier of the local root action and its current map-deadline.

When a guardian receives a **crashed?** message, it immediately sends back a **status** message. A **status** message contains the replying guardian's identifier, its crash count, and also the action identifier and map-deadline included in the **crashed?** message.

When a guardian receives a **status** message, it first checks that the map-deadline included in the message equals that of the local root action whose identifier is included in the message. If this is not the case, the **status** message is discarded. Otherwise, the crash count in the **status** message is used to update the local root's e-map.

In order to guard against lost **crashed?** and **status** messages, the local root's guardian times out at some point after sending out the first round of **crashed?** messages, but before the local root's map-deadline arrives, and retransmits **crashed?** messages to any guardians in the e-map still mapped into **null**. The guardian then can set another time-out to occur before the local root expires to again check and retransmit any **crashed?** messages if necessary.

When a local root action's map-deadline expires, its map-deadline is extended as follows. If the d-list-map of the local root or those of any of its purely local descendants is not strictly a subset of the e-map, then the local root's deadline is not extended -- the local root is aborted instead. If this is not the case, the local root action's map-deadline is increased by the map-deadline period and its e-map is discarded. When a local root's map-deadline expires, its guardian must attempt to extend the local root's map-deadline before any messages with a piggybacked map

that happen to arrive can be processed.

The time an entry stays in map needs to be increased, in order for map-deadline extension to work correctly. This has to do with the fact that an action orphaned by the crash of a guardian can survive longer than a single deadline period after the guardian recovers. This is true since the interval between receiving successive **status** messages from a particular guardian directed at the same action can exceed one map-deadline period. If this guardian crashes and recovers immediately after sending out the first **status** message, the crash-orphaned action survives until it receives the second **status** message, and thus the crash-orphan survives for more than a single map-deadline period.

An upper bound on the amount of time an entry must spend in map is two deadline periods. Since the map-deadline period is a relatively large value, doubling the magnitude of map entry tags would be detrimental to the performance of deadlining, in terms of keeping map small. One can improve the state of affairs by establishing an amount of time, denoted C, that is less than the map-deadline period, and restricting guardians to starting map-deadline extension for any given local root only when the current time is less than C seconds away from the map-deadline. Then new map entries need only be tagged with the current time plus the map-deadline period plus C.

Our description of map-deadline extension is not quite complete. There remains a problem to be addressed concerning reply and lock-granting query response messages. These messages cause some action's d-list-map to grow. If one of these messages is received while map deadline extension is going on, should the local root's e-map be modified? If deadline extension is not going on, is there any problem if the action sending the message has a map-deadline less than that of the receiver? We explain what occurs in these cases below.

A handler's map-deadline is included in its reply message. When a guardian receives a reply message, it first ascertains if there is an e-map for the appropriate

call action's local root. If there is, the local root is undergoing map-deadline extension. There are two possible cases -- either the map-deadline in the reply is greater than the local root's map-deadline or it is not. In the former case, the e-map is updated using the piggybacked d-list-map. That is, any entries in the d-list-map but not the e-map are added to the e-map, and any guardians mapped into **null** in the e-map that appear in the d-list-map are mapped into the crash count given by the d-list-map. In the latter case, any guardians in the piggybacked d-list-map not appearing in the e-map are added to the e-map, but these guardians are mapped into **null**.

If there is not e-map for the call action's local root, two cases are again possible: either the map-deadline in the reply message is less than that of the local root or it is not. In the former case, before the reply can be processed, the guardians of non-ancestors in its d-list-map must be queried. Alternatively, the reply message could be discarded. In the latter case, the reply message is processed as normal.

While the deadline extension process is taking place for a handler action, the handler action is not permitted to complete and thereby cause a reply message to be sent to its parent's guardian; its completion is delayed until its map-deadline is extended. This is done since there might not be enough time for a guardian to extend the deadline of a local root if some handler that has several guardians in its d-list-map committed up just as the local root got very close to its deadline. This situation cannot be avoided entirely, however, due to the fact that reply messages can be delayed.

Query responses that include an action's d-list-map also include that action's map-deadline. Such messages with a piggybacked map-deadline are treated in the same manner as reply messages. However, one can discard query response messages whenever convenient. Again, a query directed towards an action's relative while that action is undergoing map-deadline extension is postponed until after the map-deadline is extended. This avoids the reception of query responses with map-

deadlines less than that of the receiving action.

Again as in the done deadlining scheme, a purely local descendant of a local root action should not be permitted to make a handler call if there is only a slim chance of the created handler action being able to successfully extend its map-deadline before it expires.

Chapter Seven

Performance Analysis of Deadlining

Deadlining's goal is to keep the sizes of done and map both down to a "reasonable" size. How well does deadlining do this? A performance analysis is presented in this chapter that shows how well deadlining achieves this goal. This chapter first examines the performance of done deadlining, and then the performance of map deadlining.

7.1 Performance Analysis of Done Deadlining

The parameter that affects the performance of done deadlining the most is the done-deadline *period*, i.e. the amount of time between successive done-deadlines for any given action. In this analysis, this is assumed to be a fixed value and is denoted by P.

There is a tradeoff involved with the value of P. As one makes P smaller, the average size of done becomes smaller. On the other hand, the smaller P is, the shorter the interval between deadlines for every action, and hence deadline extension is necessary more often. One can make the average size of done arbitrarily small by appropriately setting P. The interesting performance issue is how small the average size of done can be while still having only a "reasonable" amount of deadline extension.

7.1.1 Modelling Deadline Extensions per Topaction

In this section, a simple model is formulated for the total number of times any given topaction will undergo deadline extension. The analysis of the model developed in this section is deferred until after done is modeled in the next section.

In order to model the total number of deadline extensions a topaction undergoes, it is first necessary to model the length of a topaction's lifetime. Let the random variable L denote the length of a topaction's lifetime. Topactions are assumed to have exponentially distributed lifetimes with mean $1/\lambda$. That is, we assume the time from a topaction's creation to its completion is exponentially distributed, and that this time amounts to $1/\lambda$ seconds on average. The probability distribution and density functions for L are given by Equations 7-1 and 7-2, respectively. The shape of L's density function is illustrated in Figure 7-1.

$$F_L(x) = P[L < x] = 1 - e^{-\lambda x} = \int_0^x f_L(t)\,dt \qquad (7\text{-}1)$$

$$f_L(x) = \lambda e^{-\lambda x} \qquad (7\text{-}2)$$

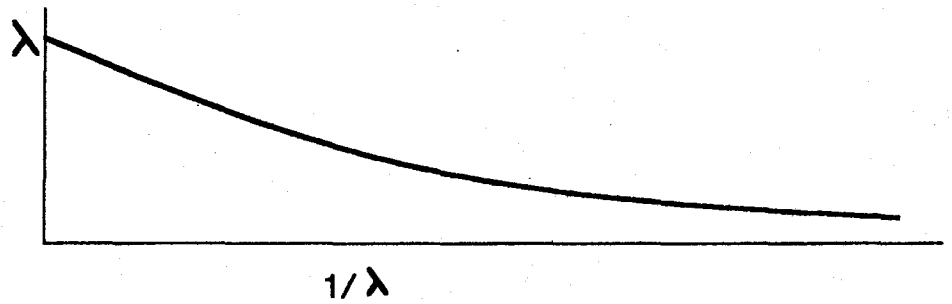


Figure 7-1: Exponential density function

One might question why it is assumed that topaction lifetimes are exponentially distributed. Why not some other distribution? First of all, since Argus has not been implemented as of this writing, there is no data to debunk this assumption. As it turns out, the exponential distribution has proved itself quite versatile in modelling phenomena somewhat analogous to action lifetimes. For a discussion of the exponential distribution, the reader is directed to [Kleinrock75]. However, no broad claim is made about the applicability of the exponential assumption to the case of topaction lifetime; it is only hoped that this assumption is not too unreasonable. Part of the attractiveness of this choice is that it somewhat simplifies the mathematics of this chapter.

We now proceed to model the number of deadlines a topaction reaches. Let the random variable **D** denote the total number of deadlines a topaction reaches over the course of its lifetime. A topaction reaches no deadlines if it lives for less than P seconds, showing that $P[D = 0] = P[L < P]$. A topaction encounters exactly one deadline over its lifetime if it lives for more than P seconds but less than 2P seconds, demonstrating that $P[D = 1] = P[P \leq L < 2P]$. The general case is given by Equation 7-3.

$$P[D = n] = P[nP \leq L < (n+1)P] \tag{7-3}$$

Since we have assumed that L is exponentially distributed, Equation 7-3 can be rewritten as Equation 7-4. The derivation is given in Appendix Section A.1.

$$P[D = n] = [1 - e^{-\lambda P}] e^{-\lambda nP} \tag{7-4}$$

Let $\overline{D}$ denote the mean number of deadlines a topaction reaches, i.e. $\overline{D}$ denotes the mean of D. $\overline{D}$ is given by Equation 7-5; the derivation appears in Appendix Section A.2.

$$\overline{D} = 1 / (e^{\lambda P} - 1) \tag{7-5}$$

Preferably no topaction ever reaches its first deadline. Deadline extension causes additional communication traffic in a system. When an action's deadline is extended, that action's health is also jeopardized -- even when the action is not an abort-orphan, there is always some chance that deadline extension for the action will not complete successfully, thus causing the action to be aborted. An important measure of how well deadline extension is avoided is $P[D=0] = 1 - e^{-\lambda P}$.

7.1.2 Modelling the Size of Done

The model of done size presented in this section is based upon the M/G/∞ queue [Kleinrock75]. An abstract M/G/∞ queue is illustrated in Figure 7-2. When a new action identifier is added to a guardian's done, this is modeled as a new "customer" coming in for "service" at the queue. An M/G/∞ queue has an unlimited number of "servers", so the customer does not spend any time waiting for

service. The time an action identifier spends in a guardian's done is modeled as the "service time" of the customer in the queue. When a customer completes its service time in the model, it leaves the queue, corresponding to the deletion of an action identifier from done. Customers coming into an M/G/∞ queue constitute a Poisson process. That is, the time between customer arrivals to the queue is exponentially distributed. Thus we must assume that the time between adding two successive action identifiers to done is exponentially distributed. No such assumption needs to be made about the distribution of service times; the distribution of service time in a M/G/∞ queue is arbitrary. With this simple model, the average size of a guardian's done corresponds to the average number of customers receiving service in the queue at any given time, which is the product of the average arrival rate and the average service time.

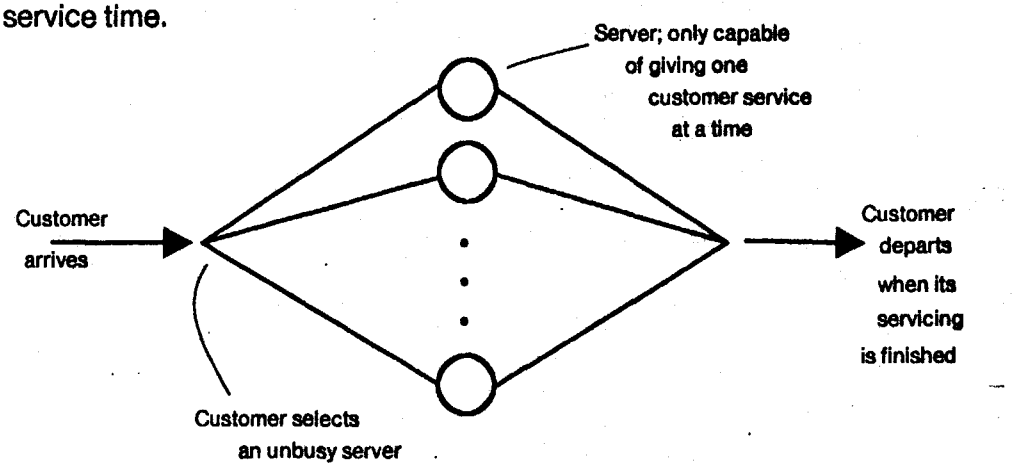


Figure 7-2: M/G/∞ queue

Let us first formulate the arrival rate of action identifiers to a guardian's done. First consider action identifiers that a guardian adds to its done due to the abort of some local action. Let us ignore for now the action identifiers a guardian adds to its done as a result of merging in some other guardian's done that was piggybacked on a message. Let us assume that local action aborts at a guardian that cause a new action identifier to be added to done occur at rate α. Furthermore, the inter-add time is assumed to be exponentially distributed. All guardians are assumed to be homogeneous in this respect.

When an ancestor's identifier is added to done, all its descendant's identifiers can be deleted. In this analysis, however, only the deletion of identifiers due to deadlining is considered. This other source of deleted identifiers is ignored by restricting α to be the rate topaction identifiers are added to done locally. Recall that a topaction's identifier is added to done when the topaction completes, given that there is some descendant's identifier in the topaction's guardian's done. Then as done propagates about the system, the topaction's descendant's identifiers are replaced by the topaction's identifier. Restricting α to topactions ignores the transient effects of identifiers being added to done and later being replaced by their topaction's identifier.

Now let us consider action identifiers added to a guardian's done as a result of merging in a sent done. Let us assume that there are N guardians in the distributed system. Also, we assume that each guardian communicates with every other guardian, directly or indirectly. Hence every guardian eventually receives, in a piggybacked done, any action identifier any other guardian adds to its done due to a local abort. Then the rate new topaction identifiers are added to any particular guardian's done as a result of merging in sent dones is $(N-1)\alpha$, since the other N-1 guardians in the system each produce identifiers of local topactions at rate α. Again, the effects of non-topaction identifiers in done are ignored. Thus the rate topaction identifiers are added to a guardian's done from both remote and local sources is $(N-1)\alpha + \alpha = N\alpha$.

Let us now turn our attention towards determining the "service time" of action identifiers in the $M/G/\infty$ model. A topaction's identifier is tagged with the topaction's deadline when first added to done. Guardians can delete the identifier at the tagged time plus ε. Hence the time a topaction identifier needs to stay in done is dependent upon the difference between the time of the topaction's completion and its deadline. Let the random variable S denote the amount of time a topaction identifier spends in done. The distribution of S can be derived from the assumption that L, topaction lifetime, is exponentially distributed.

For the M/G/∞ model, only the mean of **S**, denoted $\overline{S}$, is of interest. The derivation of $\overline{S}$ is given in Appendix Section A.3. The assumption that **L** is exponentially distributed makes this derivation quite straightforward, since this distribution has the "memoryless property." Equation 7-6 shows the result. We assume that ε is small; an addend of ε is ignored in Equation 7-6.

$$\overline{S} = \{ F_L(P)[\lambda P - 1] + Pf_L(P) \} / \lambda F_L(P) \qquad (7\text{-}6)$$

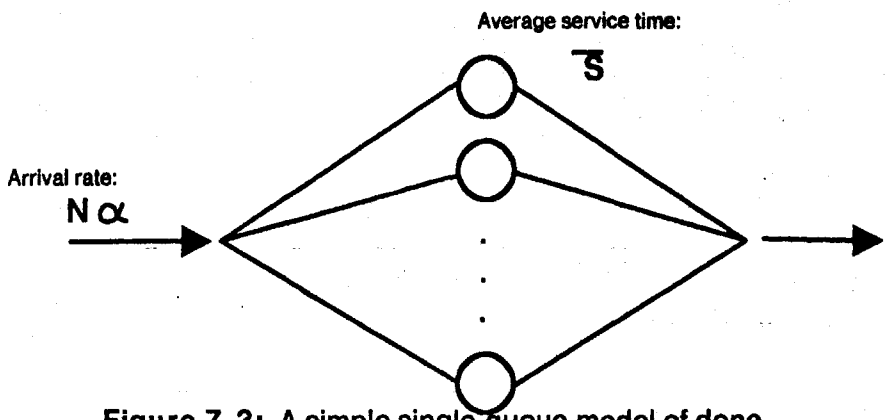


Figure 7-3: A simple single-queue model of done

Figure 7-3 illustrates a simple M/G/∞ model of a guardian's done. Action identifiers come into done at the rate of $N\alpha$ per second. Each such identifier then receives "service" for $\overline{S}$ seconds and then leaves the queue. The average size of done, denoted $\overline{\textbf{done}}$, is the average number of identifiers in service at any given time in the model. The equation for $\overline{\textbf{done}}$ is given by Equation 7-7:

$$\overline{\textbf{done}} = \overline{S}N\alpha. \qquad (7\text{-}7)$$

The model of Figure 7-3 is a bit too simple, however. When a topaction's identifier is added to its own guardian's done, the identifier does indeed spend an average of $\overline{S}$ seconds there. But identifiers of non-local actions added to the same guardian's done will have "aged" some as they were propagated to the guardian in piggybacked dones. These identifiers spend less time than $\overline{S}$ on average in the guardian's done. The above model suggests that identifiers of completed topactions are broadcast to all guardians and immediately added to their dones; this is certainly

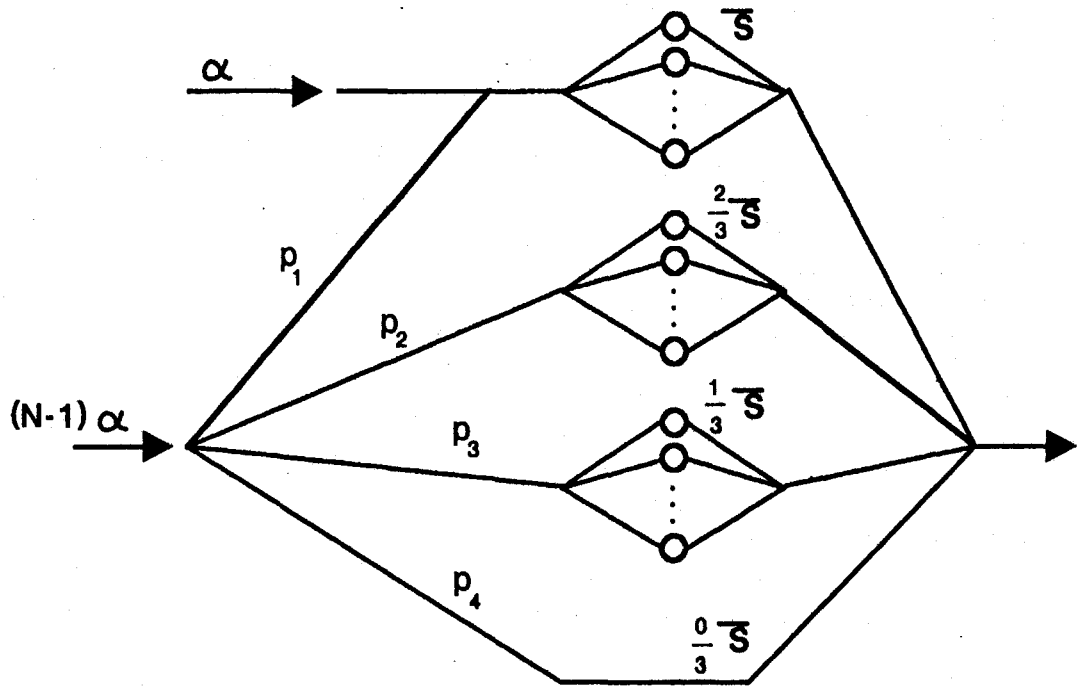


Figure 7-4: Multiple M/G/∞ queue model of done

not the case. A more accurate model of a guardian's done can be obtained by using several M/G/∞ queues, as illustrated in Figure 7-4. This particular model uses four M/G/∞ queues. In the model, $p_1 + p_2 + p_3 + p_4 = 1$; also $0 \leq p_1, p_2, p_3, p_4 \leq 1$. The value p_1 represents the proportion of guardians whose aborted local action identifiers reach the modeled guardian's done very quickly, and hence spend $\overline{S}$ seconds on average in done before being deleted. The value p_4, on the other hand, represents the proportion of guardians whose aborted local action identifiers take such a long time to reach the modeled guardian's done that they spend no time in it at all -- they are in fact deleted from all other guardians' dones before they ever reach the modeled guardian's done. The values p_2 and p_3 represent cases in between the latter two extremes. The p_i's are called *branching probabilities*. When a customer enters the model from the customer stream with the $(N-1)\alpha$ rate, the customer takes the topmost branch with probability p_1, the next branch with probability p_2, etc. Note that in this model, the stream of local identifiers added to done is distinguished from the stream of remote identifiers added to done. In this model, the average size of the modeled done is given by Equation 7-8.

$$\alpha\overline{S} + p_1(N-1)\alpha\overline{S} + p_2(N-1)\alpha(2/3)\overline{S} + p_3(N-1)\alpha(1/3)\overline{S}. \quad (7\text{-}8)$$

The four-queue model of Figure 7-4 above can be generalized to an n-queue model of done. Figure 7-5 illustrates the general model. The average size of done is denoted by $\overline{\text{done}}$ and is given by Equation 7-9. Again, $\sum_{i=1}^{n} p_i = 1$, and $0 \leq p_i \leq 1$. Also, $0 \leq f_i \leq 1$, and $f_1 = 1$.

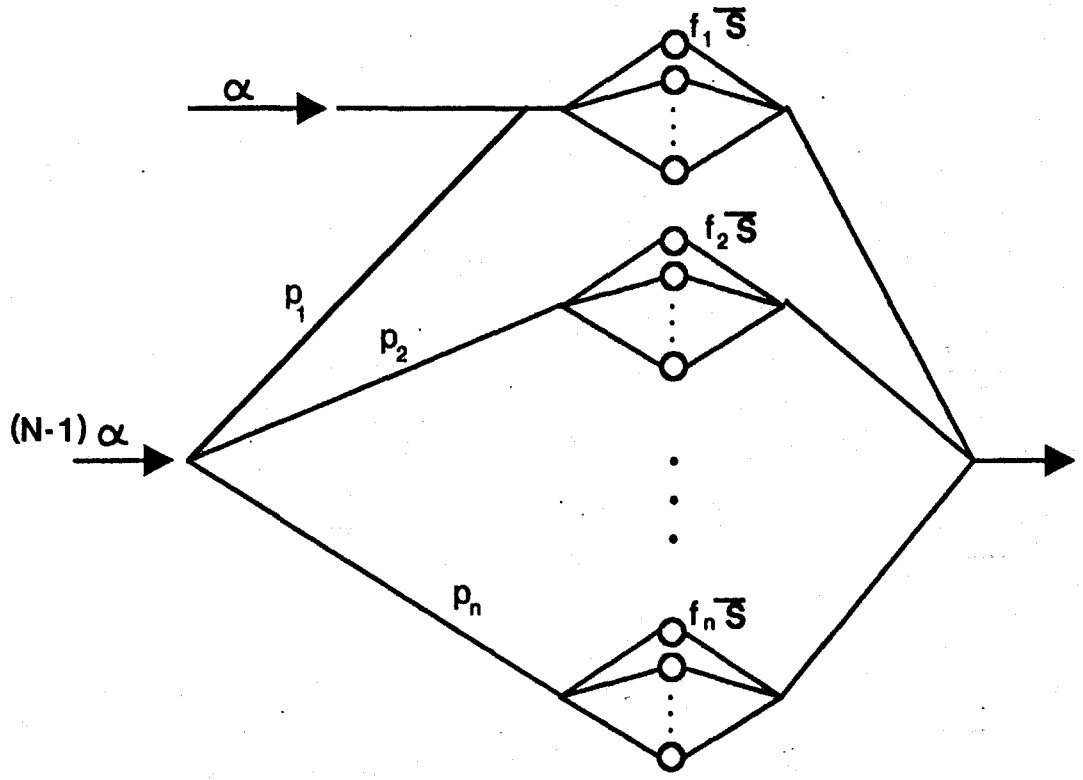


Figure 7-5: General model of done

$$\overline{\text{done}} = \alpha\overline{S}[1 + p_1(N-1)] + \sum_{i=2}^{n} p_i(N-1)\alpha f_i\overline{S} \quad (7\text{-}9)$$

7.1.3 The Performance of Done Deadlining

Does deadlining keep done at a reasonable size while still not causing excessive amounts of deadline extension to occur? In this section, we attempt to answer this question based upon the modelling machinery developed in the past two sections.

The major parameter of deadlining is P, the deadline period. Let us first examine the performance question of how large P must be to avoid excessive amounts of deadline extension.

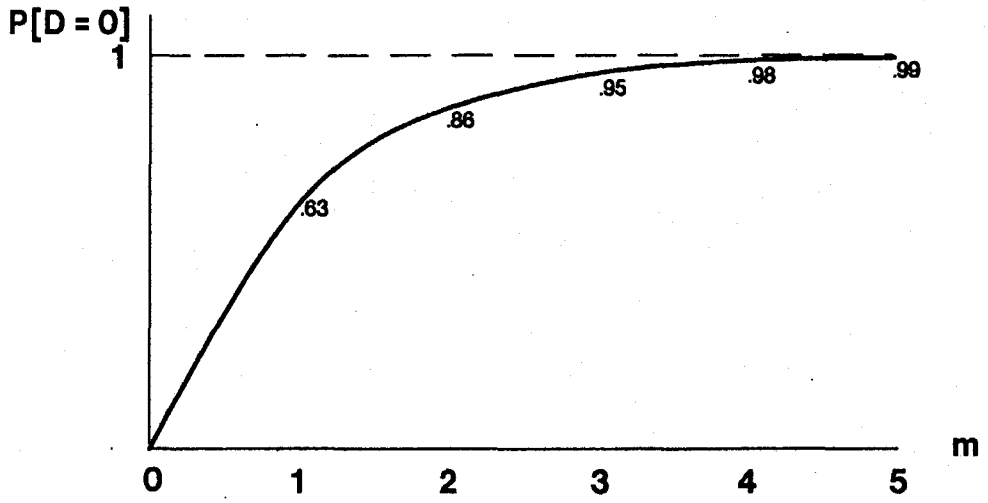


Figure 7-6: Probability that D = 0 as a function of m

Let P = m(1/λ), i.e. let m denote the proportionality constant between P and 1/λ, the average topaction lifetime. Equation 7-10 shows the result of substituting m(1/λ) for P in Equation 7-4. Figure 7-6 is a graph of P[D = 0] as a function of m. From the graph, we see that if P is three times larger than the average topaction lifetime, then only about 5% of all topactions ever hit a single deadline. If P is five times larger than the average topaction lifetime, then 99% of all topactions never hit a single deadline.

$$P[D = n] = [1 - e^{-m}]\, e^{-nm} \qquad (7\text{-}10)$$

Figure 7-7 similarly shows $\overline{D}$, the average number of deadline extensions per topaction, as a function of m. From the graph, we see that a topaction only encounters a significant number of deadlines if P is less than the average topaction lifetime. Equation 7-11 is obtained by substituting m(1/λ) for P in Equation 7-5.

$$\overline{D} = 1 / (e^{m} - 1) \qquad (7\text{-}11)$$

From the above analysis, it appears that setting P at least three times larger

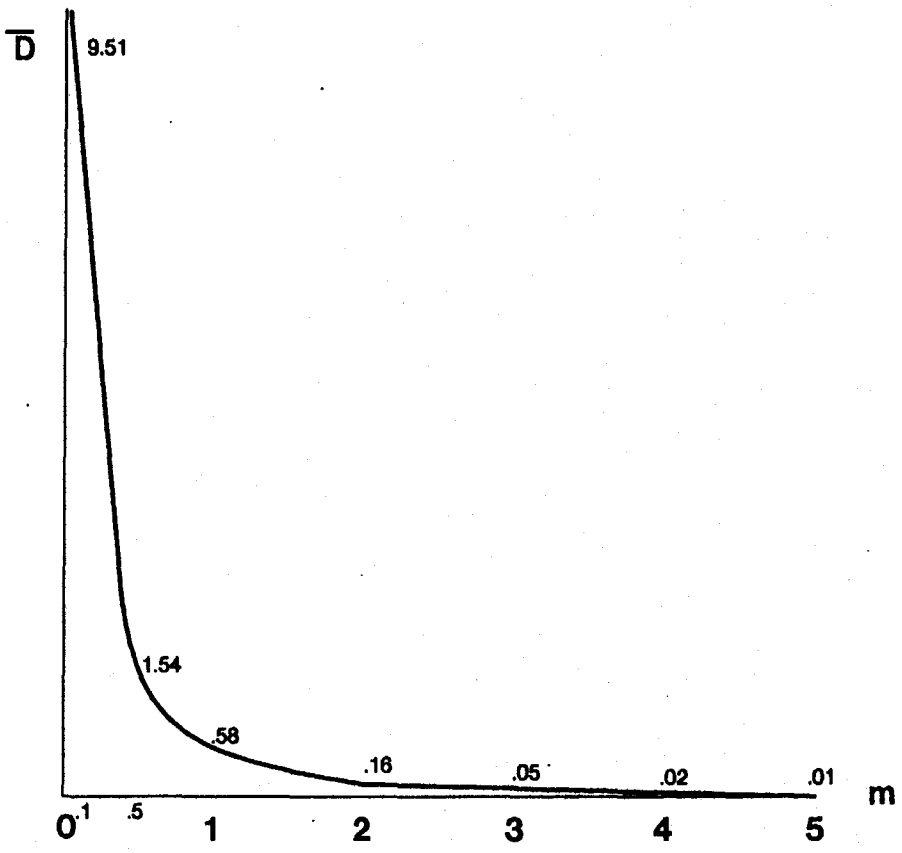


Figure 7-7: Graph of $\overline{D}$ as a function of m

than the average topaction lifetime should lead to acceptable performance with respect to the amount of deadline extension that occurs in a system. One could even make a case that setting P only twice as large as the average topaction lifetime yields acceptable performance.

Let us now examine the impact of P upon the size of done. Our first analysis is based upon the simple single-queue model of Figure 7-3. Let $\beta = N\alpha$. Then β is the overall rate that topaction identifiers are added to the modeled done; β's dimensionality is "identifiers per second." Let $P = m(1/\lambda)$, as before. Furthermore, let n be defined by $\beta = n\lambda$. Suppose every topaction's identifier is added to done when it terminates. Then the value n represents the degree of parallelism of topactions in the distributed system. If $n = 1$, then only one topaction tends to be

running anywhere in the distributed system at any given time. If n = 2, then exactly two topactions tend to be running in the system at any given time; etc.

Let $\overline{\textbf{done}}$ denote the average size of done. For the single-queue model, $\overline{\textbf{done}} = \beta\overline{S}$. Substituting $n\lambda$ for β and $m(1/\lambda)$ for P into Equations 7-7 and 7-6 leads to Equation 7-12:

$$\overline{\textbf{done}} = n \{ (1 - e^{-m})(m - 1) + me^{-m} \} / (1 - e^{-m}) \qquad (7\text{-}12)$$

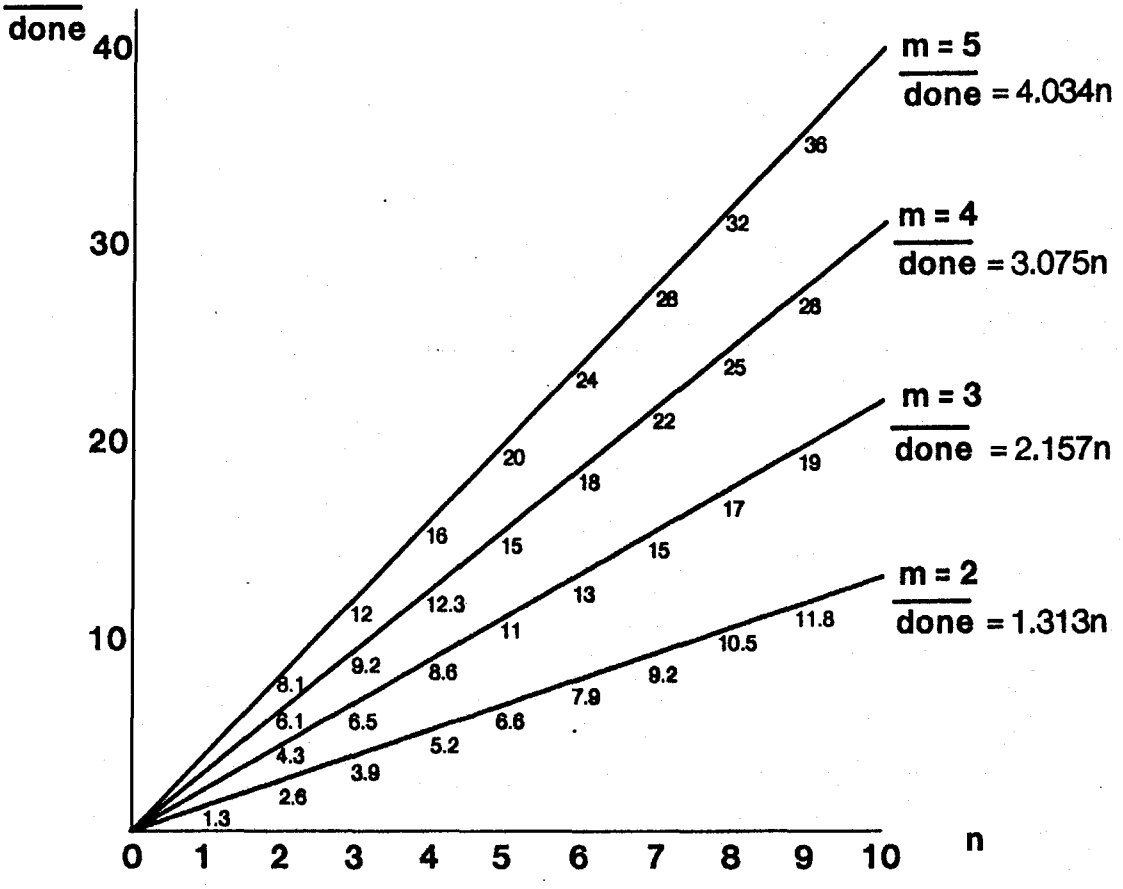


Figure 7-8: $\overline{\textbf{done}}$ graphed as a function of n

Figure 7-8 shows Equation 7-12 graphed as a function of n for several different values of m. From the graph, we see that if m = 3 -- i.e., that if P is thrice the average topaction lifetime, the size of done is just about twice the degree of topaction

parallelism in the distributed system.

In evaluating this result, we must consider the type of system the single-queue model "fits" best. The single-queue model fits a system where information about an abort spreads about quickly to all the guardians that make up the system. It seems that this implies that such systems must be small. Also note that the single-queue model overstates the size of done; information about aborts never spreads throughout a system instantaneously as the model suggests. Hence the above result provides an upper bound on the average number of topaction identifiers for done in small systems. For large systems, the single-queue model still provides an upper bound, but not a very tight one.

Let us now repeat the above analysis using a four-queue model of done. This analysis will be more complicated due to the many parameters of the multiple-queue model of done.

Suppose that the guardians in a system can be divided into three categories with respect to the modeled guardian's done -- "fast," "medium," and "slow." Identifiers added to done by a "fast" guardian reach the modeled guardian's done with very little delay. On the other hand, the identifiers of "slow" guardians take a while to reach the modeled guardian's done. Let p_2, p_3, and p_4 denote the fractions of the N-1 guardians that fall into the fast, medium, and slow categories, respectively. Let $p_1 = 0$. This takes care of determining the branching probabilities of a four-queue model (Figure 7-5).

The more guardians a system has, the larger it would seem p_4 is. Large systems (on the order of 500 guardians) are probably organized into several subsystems. Guardians within a subsystem frequently communicate with each other, but rarely with guardians outside the subsystem. A small system is composed of a single subsystem; all guardians communicate frequently with each other, so p_4 is almost zero. A large system, on the other hand, is composed of many subsystems, so information concerning aborts takes a long time to travel from one subsystem to

another. Hence p_4 has a significant value; many guardians fall into the "slow" category.

Again, in this analysis let us assume that all topaction identifiers are added to done. Let n be defined such that $\alpha = n\lambda$. Then n is the degree of local multiprogramming of topactions. Note that this differs from the previous definition of n to be the global degree of multiprogramming. For example, if $n = 1$, then there tends to be one topaction running at each guardian at any point in time.

Also, let m be defined such that $P = m(1/\lambda)$, as before.

The "delay factors" f_2, f_3, and f_4 need to be determined (Figure 7-5). It seems that these delay factors should be dependent upon n -- the higher the level of activity at each guardian the more frequent it seems inter-guardian communication should occur. Hence the higher n is, the closer the f_i's should get to 1.

Suppose every c_2^{th} topaction at a guardian in the "fast" category communicates with the guardian whose done we are modelling. Then it seems that $d_2 = c_2 / n\lambda$ is a respectable estimate of the amount of time an identifier ages before reaching the modeled done from a fast guardian. Hence $f_2 = (\overline{S} - d_2) / \overline{S}$. f_3 and f_4 are defined similarly. Equation 7-13 gives the expression for f_i in terms of m, n, and c_i.

$$f_i = 1 - \{ c_i(1 - e^{-m}) / n[(1 - e^{-m})(m - 1) + me^{-m}] \}, \text{ if greater than 0.} \qquad (7\text{-}13)$$

Fix the value of N at 50, the value of c_2 at 1, c_3 at 5, and c_4 at 20. Also fix the value of m at 3. Figure 7-9 graphs **done** as a function of n based on different choices of the branching probabilities.

From the above analysis, it appears that deadlining keeps done to a reasonable size when the local degree of multiprogramming is around one topaction. In the above analysis, this implies a system-wide degree of multiprogramming around 50 -- i.e., at any given time 50 topactions are running in the distributed system, on

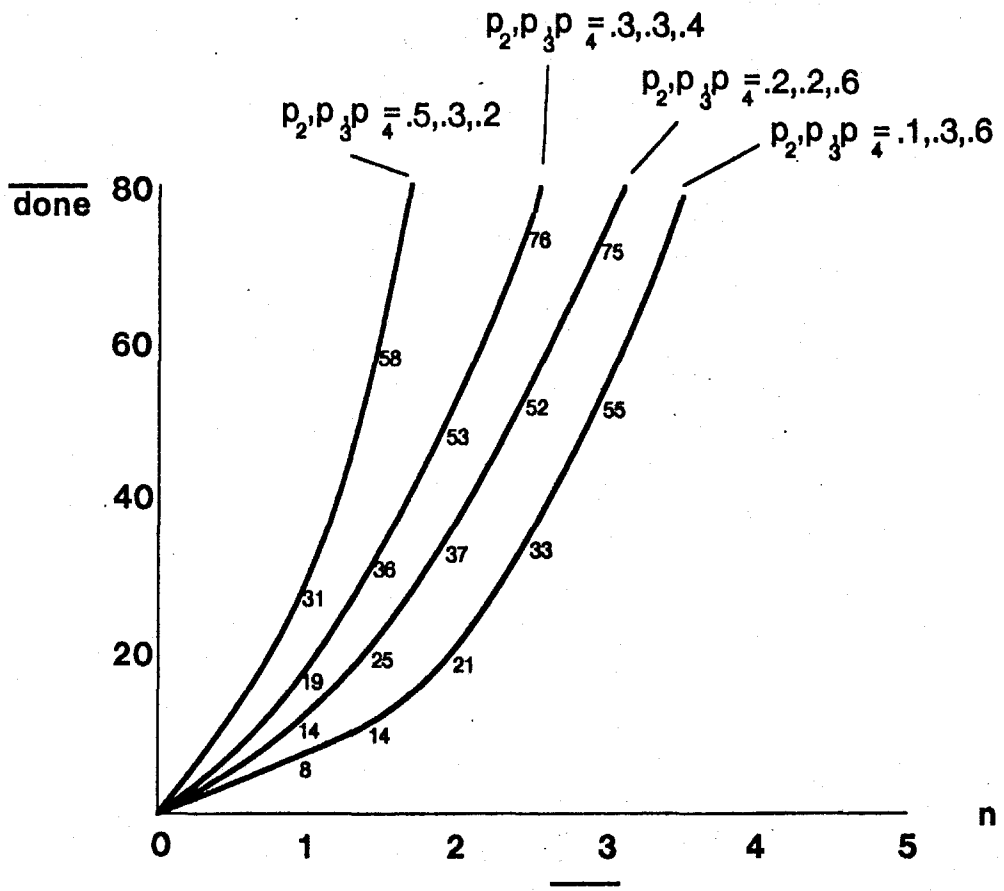


Figure 7-9: Graph of $\overline{done}$; N = 50

average. Recall that it was assumed that all topaction identifiers are added to done. In reality, there are probably many topactions that when run cause no identifiers to be added to done. Hence the actual degree of multiprogramming can be much higher than one while still having done stay at a reasonable size. In addition, many guardians in a typical system do not run topactions, so the actual number of guardians in a system with performance comparable to that predicted by the model will probably be much greater than 50.

In any case, we suspect that the actual local degree of topaction multiprogramming in a typical system is fairly small -- around one. Under these conditions, the above analysis has demonstrated that deadlining performs adequately.

In this analysis, we have so far ignored the "transient" effect of non-topaction identifiers in done. However, the presence of these identifiers in done does cause the actual average size of done to be larger than that indicated by our analysis above. The question is exactly how much larger.

We make the assumption that only one identifier belonging to a descendant of any particular topaction is in any given guardian's done at any particular moment. This assumption can be made true (for all practical purposes) by adding any committing action's identifier to done, when one of its descendants identifiers is already in done. If this is done, then identifiers in done can no longer be used to detect unwanted committed subactions, as explained in Section 4.3. Our analysis below depends on this assumption being true; if this assumption is not true, actual system performance could be much worse than that predicted. However, it is not clear if this assumption need be made true to obtain acceptable performance in practice.

Let the random variable T denote the amount of time any identifier of any one of a given topaction's descendants can be found in some guardian's done, under the assumption that some descendant of a topaction is aborted very shortly after the topaction is created. Then the mean of T is given by Equation 7-14. Since E[D], the mean of D, is approximately zero for the magnitudes of P we consider practicable, E[T] is approximately P.

$$E[T] = P\,(E[D] + 1) \qquad (7\text{-}14)$$

We now examine the result of adding non-topaction identifiers to the single-queue model of done. To have this model take transient identifiers into account, one just changes the mean service time to E[T]. This works due to the assumption that only one identifier of an action descended from a particular topaction is in any one guardian's done. When a topaction is created, one of its descendant's identifiers is very quickly (i.e. instantaneously) added to done at some guardian, by assumption. This is modeled as the identifier being a "customer" coming in for service at the

queue at the time the topaction is created. The "customer" leaves the queue E[T] seconds later, modelling the deletion of the appropriate topaction's identifier from done. While it is in service, this "customer" might represent several different identifiers of a topaction's descendants. But since there are never two or more identifiers belonging to the same topaction's descendants in any particular done, by assumption, this one "customer" suffices to represent any identifier of the appropriate topaction's descendants that finds its way into done. Then the approximate average size of done is n(m) when taking non-topaction identifiers into account, where $\beta = n\alpha$ and $P = m(1/\lambda)$. From examining Figure 7-12, we can see that the size of done is less than double the size predicted by the analysis that only takes topaction identifiers into account. For m = 2, for example, the average size of done is about 52% larger than the topaction-identifier-only average size. (Recall that P, the deadline period, is m times the average topaction lifetime). For m = 4, the "true" average size of done is about 30% larger. Hence it appears that even though our analysis ignoring non-topaction identifiers understates the size of done, this does not result in a gross underestimate.

7.2 Performance of Map Deadlining

The modeling machinery developed in previous sections to analyze the performance of done deadlining also can be used to predict the performance of map deadlining.

Let P denote the map-deadline period. Figures 7-6 and 7-7 apply immediately to map deadlining. Here the random variable D denotes the number of map-deadlines a topaction encounters over its lifetime. Again, m is defined by $P = m(1/\lambda)$. From Figure 7-6, it can be seen that if P is five times larger than the average action lifetime, then 99% of all topactions never hit a single map-deadline.

Let α denote the rate at which guardians crash. Then a guardian produces a map entry for itself at rate α. We can then use the models of Figures 7-5 and 7-3 to

model the size of map. The service time, $\overline{S}$, of map entries is simply twice P. (We ignore ε and the restriction proposed in the last section concerning C.)

Let n be defined by $\alpha = (1/n)\lambda$. Then n is the ratio of a topaction's lifetime to a guardian's inter-crash time. In any proper system, the value of n should be fairly large; the inter-crash time should be much larger than the average topaction lifetime.

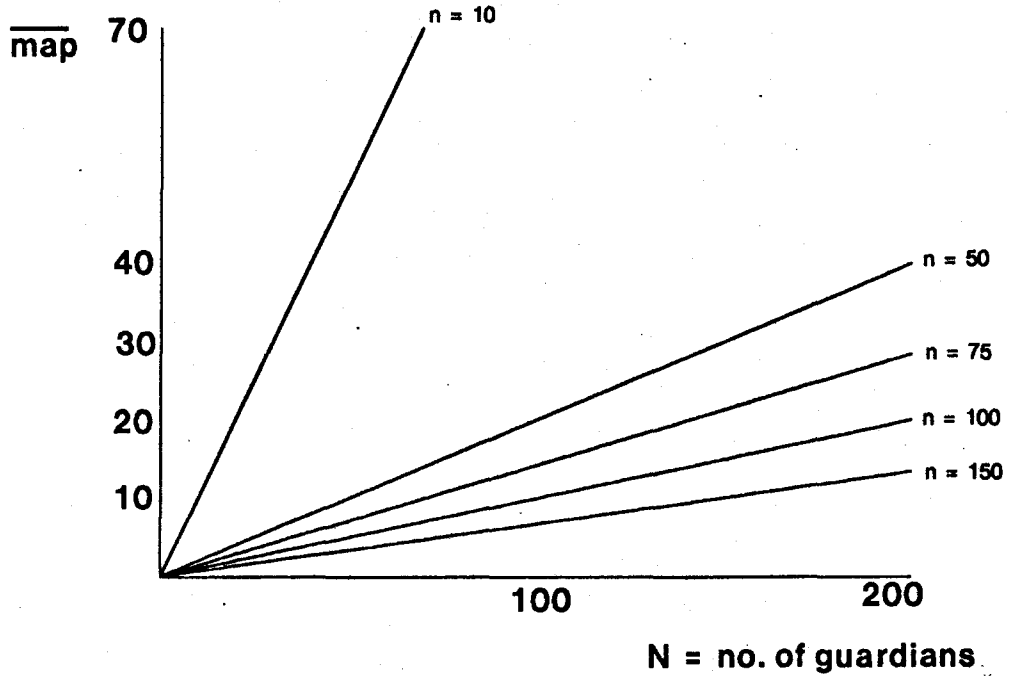


Figure 7-10: Average size of map, according to single-queue model; m = 5

Let us consider the simple, but inaccurate, single-queue model of map illustrated in Figure 7-3. The average number of map entries in this model is given by $\overline{map} = 2Nm(1/n)$, where N is the number of guardians in the system. Figure 7-10 shows $\overline{map}$ graphed as a function of N for several values of n with m fixed equal to 5. Since the single-queue model is only valid for systems with a "small" number of guardians, this graph looks quite encouraging. For example, if the inter-crash time is 100 times the length of topaction lifetime, $\overline{map} = (1/10)N$. So for a system with 100 guardians, map tends to only have 10 entries in it.

For extremely large systems, the single-queue model is not accurate; it

overestimates the size of map. But the preceding analysis has shown that the single-queue model indicates that map has a reasonable size for a system composed of a few hundred guardians, certainly not a small system. Since map entries are tagged with P + C instead of 2P, map-deadlining performs even better than our modelling here indicates. Hence the claim that map-deadlining keeps map "small" seems justified by this analysis.

Chapter Eight

Conclusion

This chapter is organized into two parts. First, related work on orphans is discussed. Second, the conclusions of this thesis are presented.

8.1 Related Work

Several others have proposed orphan detection strategies. These schemes are discussed and compared with the orphan detection scheme presented in the thesis.

8.1.1 Nelson's Thesis

Nelson [Nelson81] discusses orphans and orphan detection in the context of his remote procedure call scheme. Nelson's orphans are different from Argus's orphans in several respects. First of all, Nelson's orphans are created strictly by the crash of an ancestor. Secondly, Nelson's orphans are simple subprocesses; in Argus orphans are subactions. Finally, there is no notion of Nelson's orphans viewing inconsistent data; in Argus, on the other hand, this is the primary justification for orphan detection. Nelson justifies orphan detection by showing that it is needed to provide so-called *last-of-many* semantics for remote procedure calls.

The orphan detection schemes in Nelson's thesis are basically worked-out versions of schemes proposed by Lampson [Lampson81].

The first orphan detection scheme Nelson describes is called *extermination*. This scheme delays recovery from a crash until all orphans created by the crash are tracked down and destroyed. This scheme leads to unbounded recovery times.

Nelson then describes a scheme for relieving the problem of unbounded recovery times in his extermination scheme, called *expiration*. In this scheme, each process is assigned a time limit. A remote subprocess spawned via a remote procedure call inherits the time limit of its parent. If a process is still running when its time limit arrives, it is destroyed. As in the deadlining scheme described in this thesis, this scheme imposes a maximum bound upon the amount of time an orphan can exist before being destroyed. Expiration and extermination are used together to create an orphan detection scheme without unbounded recovery times. If for some reason extermination cannot complete normally at a recovering site, recovery at that site is simply delayed until all orphans created by the crash are certain to have been destroyed via expiration. Unfortunately, expiration can lead to the destruction of non-orphaned processes.

The final orphan detection scheme Nelson details is called *reincarnation*. This scheme is similar to the basic algorithm for detecting crash-orphans presented in this thesis in that it works by piggybacking information onto messages. In Nelson's scheme, each site maintains a crash-count, which he calls an *epoch*. When a site recovers from a crash, it increments its epoch number. The epoch number is piggybacked on every outgoing message from a site. When a site receives a message with a higher epoch number than its own, it destroys all its local processes and increases its own epoch number. Of course, this can result in non-orphans being destroyed. To correct this deficiency, Nelson proposes another scheme called *gentle reincarnation*. Gentle reincarnation works just like plain reincarnation does except when destroying processes at a site that has just received a higher epoch number on an incoming message. Instead of simply destroying processes, querying up the ancestor chain is done to ascertain if a process is indeed an orphan or not. Of the orphan schemes Nelson describes, this one is the closest to that of Argus in that it works by piggybacking information on messages.

Of all the orphan detection schemes presented in his thesis, Nelson advocates the combined expiration and extermination scheme as the best. The orphan

detection scheme presented in this thesis represents an improvement over this scheme since recovery need never be delayed waiting for orphans to perish. While a node is waiting for orphans to perish in Nelson's scheme, that same node would be up and running in our scheme. Also, our scheme does not cause non-orphans to be aborted.

8.1.2 Lampson's Orphan Detection Schemes

As Nelson points out in his thesis, his orphan detection schemes are worked-out versions of schemes proposed by Lampson [Lampson81]. Lampson also proposes an additional scheme to those detailed in Nelson's thesis, *deadlining*.

Deadlining is an enhancement of expiration, described in the last section. Instead of merely aborting a process when it reaches its time limit, querying is done up the ancestor chain to ascertain if the process is actually an orphan. If the process is not an orphan, its time limit is extended. Of course, deadlining is the method used in this thesis to age entries out of done and map. Lampson does not go into the details of deadlining. The communication required to extend deadlines is much simpler in Lampson's context than ours, since he must only communicate up to ancestors, whereas we must also communicate down to committed relatives. Lampson uses deadlining to establish a maximum on the amount of time crash recovery need be delayed due to orphan detection, whereas we use deadlining to trim the orphan information piggybacked on messages.

8.1.3 Allchin's Thesis

Allchin [Allchin83] presents a system based on nested atomic actions that is very similar to the Argus system. Orphans arise in Allchin's system from sources that are analogous to the sources of orphans in Argus. Allchin's orphans cause the same sort of problems as Argus's orphans -- they waste resources and can see inconsistent data. Allchin discusses the orphan problem and proposes an orphan detection algorithm. His algorithm is more efficient than that presented in Chapter 4,

but it is incorrect.

I have chosen to present Allchin's orphan detection scheme using the terminology of this thesis and within the context of Argus, rather than using his terminology.

Allchin's orphan detection algorithm is strikingly similar to the algorithm of Chapter 4. His algorithm can be separated into two halves, just as our algorithm -- an abort-orphan detection and crash-orphan detection. His abort-orphan detection scheme is basically the same as ours. His crash-orphan detection scheme is almost identical to ours, except in one vital respect -- he does not piggyback map on any message. Messages on which our algorithm piggybacks both a guardian's map and an action's d-list-map, he piggybacks only the d-list-map. On prepare messages we piggyback the map of the committing topaction's guardian -- he instead piggybacks the topaction's d-list-map. A message receiver uses the sent d-list-map in Allchin's algorithm at those times the sent map is used in our algorithm. That is, in Allchin's algorithm the sent d-list-map is used by the receiving guardian to update its own map and detect local orphans.

We now present a counter-example that demonstrates Allchin's crash-orphan detection algorithm can fail. In this example there are three guardians of interest: GX, GY, and GZ. Each of these guardians has a single atomic object -- x, y, and z, respectively. The consistency constraint is that $x > y > z$. Suppose that initially $x = 100$, $y = 99$, and $z = 98$. Also suppose each guardian's map just contains a single entry for itself.

Suppose topaction A is created at G1. Its d-list-map initially contains just the entry <G1,0>. Action A does a handler call to guardian GX creating subaction A.1. A.1's d-list-map is initialized to that of A piggybacked on the call message along with an additional entry for GX. A.1 reads x, discovering that it has the value of 100. A.1 then commits and passes the information that x is 100 to A. The d-list-map piggybacked on the reply is merged into A's d-list-map, resulting in A's d-list-map

note: "d-map" = d-list-map

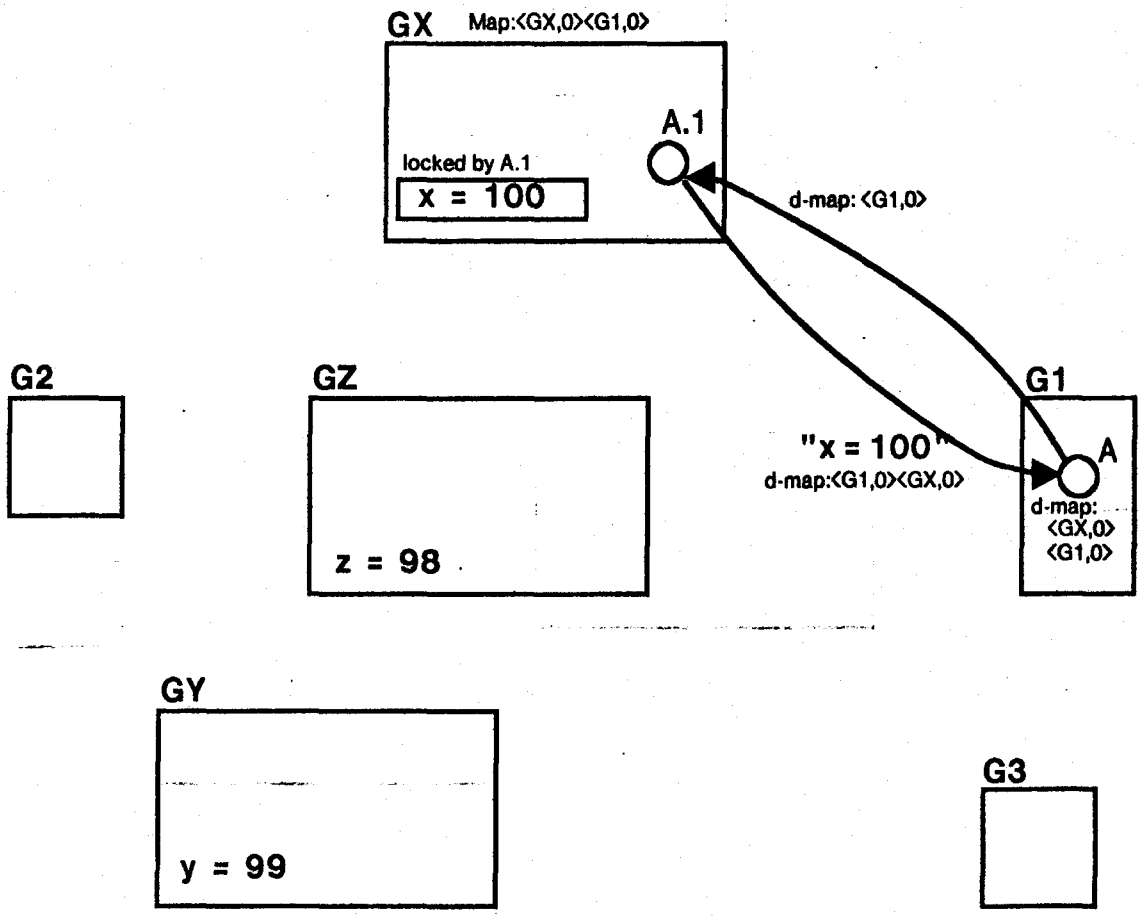


Figure 8-1: Counter-example snapshot one

acquiring the entry <GX,0>. Figure 8-1 illustrates the state of affairs at this point.

Then GX crashes and recovers. This causes the lock A.1 obtained on x to be released and GX's crash count to be incremented. Action A is now a crash-orphan. This counter-example will show that the information about GX's crash does not reach GZ in time to prevent A from making a handler call there and viewing inconsistent data.

Suppose a topaction B at guardian G2 makes a handler call to guardian GX after it recovers, creating subaction B.1. B.1 changes the value of x to 200 and commits to B, passing information to B that x is 200. Note that B's d-list-map

piggybacked on the reply message contains the entry <GX,1>. Thus after the sent d-list-map is merged with that of B, B's d-list-map contains the entry <GX,1>.

Topaction B then makes a handler call to GY, passing the information that x is 200. B's d-list-map is piggybacked on the associated call message. At GY, merging B's sent d-list-map into GY's own map results in its map acquiring the entry <GX,1>. Thus news about the crash of GX has reached GY at this point in the counter-example.

This call message creates subaction B.2 at GY. B.2 changes the value of y to 150, and checks to make sure that the consistency constraint x > y > z is still preserved by checking that the passed value of x, 200, is greater than the new value of y, 150, which is itself greater than the old value of y, 99. The consistency constraint is indeed preserved. Figure 8-2 illustrates the current situation.

Subaction B.2 then commits to topaction B. Then B itself commits and subsequently two phase commit for topaction B successfully finishes. This results in the release of the locks on x and y.

Then a topaction C is created at G3. C makes a handler call to GY, resulting in the creation of subaction C.1. C.1 reads the value of y, and discovers it has a value of 150. C.1 then commits to C, passing information that y is 150. But note that C.1's d-list-map only contains entries for G3 and GY. Hence the reply message carries no information about the crash of GX, even though the message itself carries information that is inconsistent with the state of GX before the crash.

C then makes a handler call to GZ, passing information that y is 150. This causes the creation of subaction C.2 at GZ. C.2 changes the value of z to 100. C.2 then checks that the consistency constraint x > y > z still holds by making sure the the new value of z, 100, is less than the value of y passed to it, 150. The consistency constraint is indeed preserved. Note that C.2's d-list-map only contains entries for G3, GY, and GZ. Figure 8-3 illustrates the current situation.

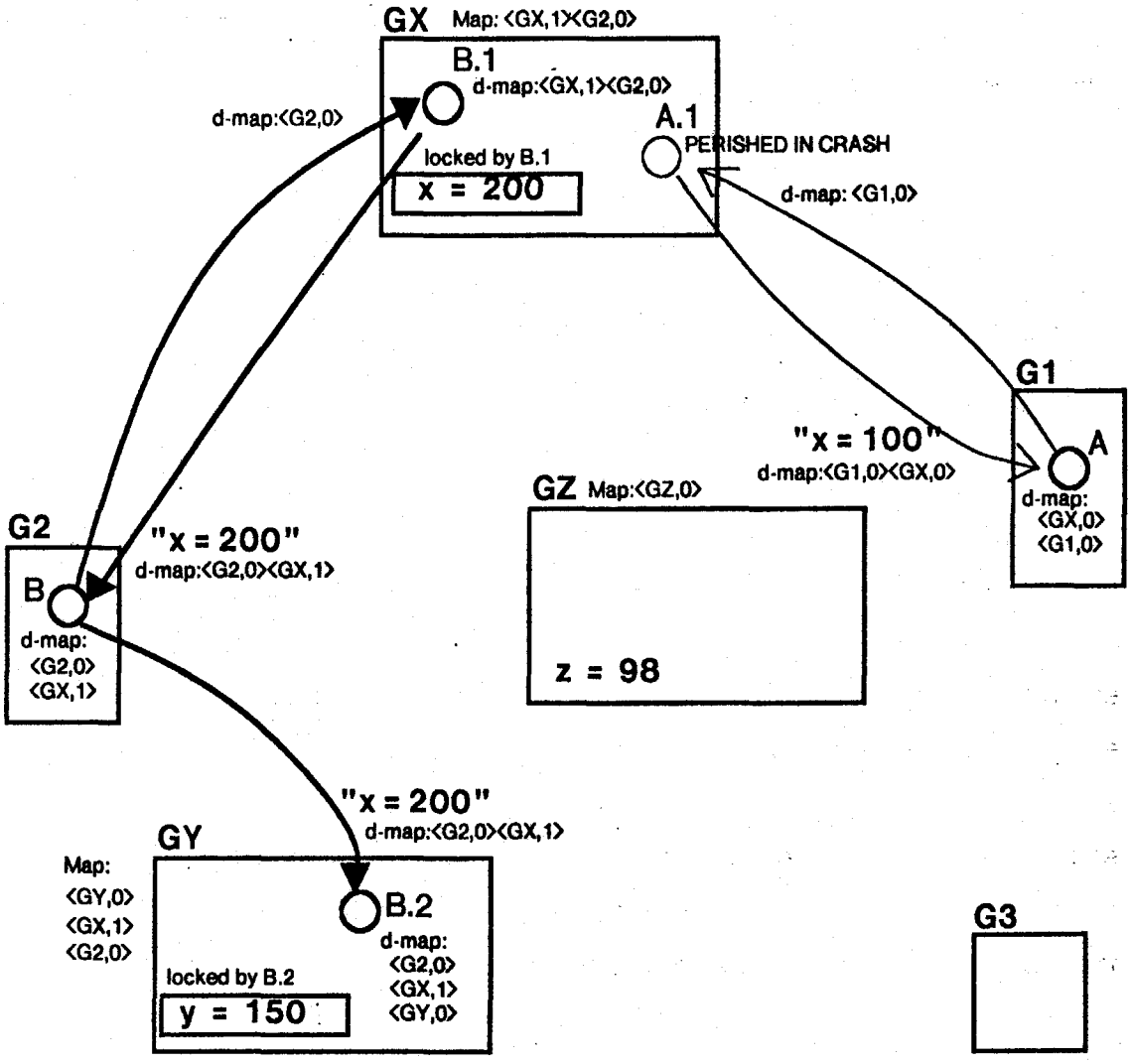


Figure 8-2: Counter-example snapshot two

Subaction C.2 then commits to C and C subsequently itself commits. Note that C's final d-list-map contains entries just for G3, GY, and GZ. Two phase commit for topaction C then successfully finishes. Note the information about the crash of GX has failed to reach GZ, although the state of GZ is inconsistent with the state of GX before the crash.

Finally, orphaned action A makes a handler call to GZ, passing the invalid information that x is 100. Since GZ's map contains no entry that is more up-to-date

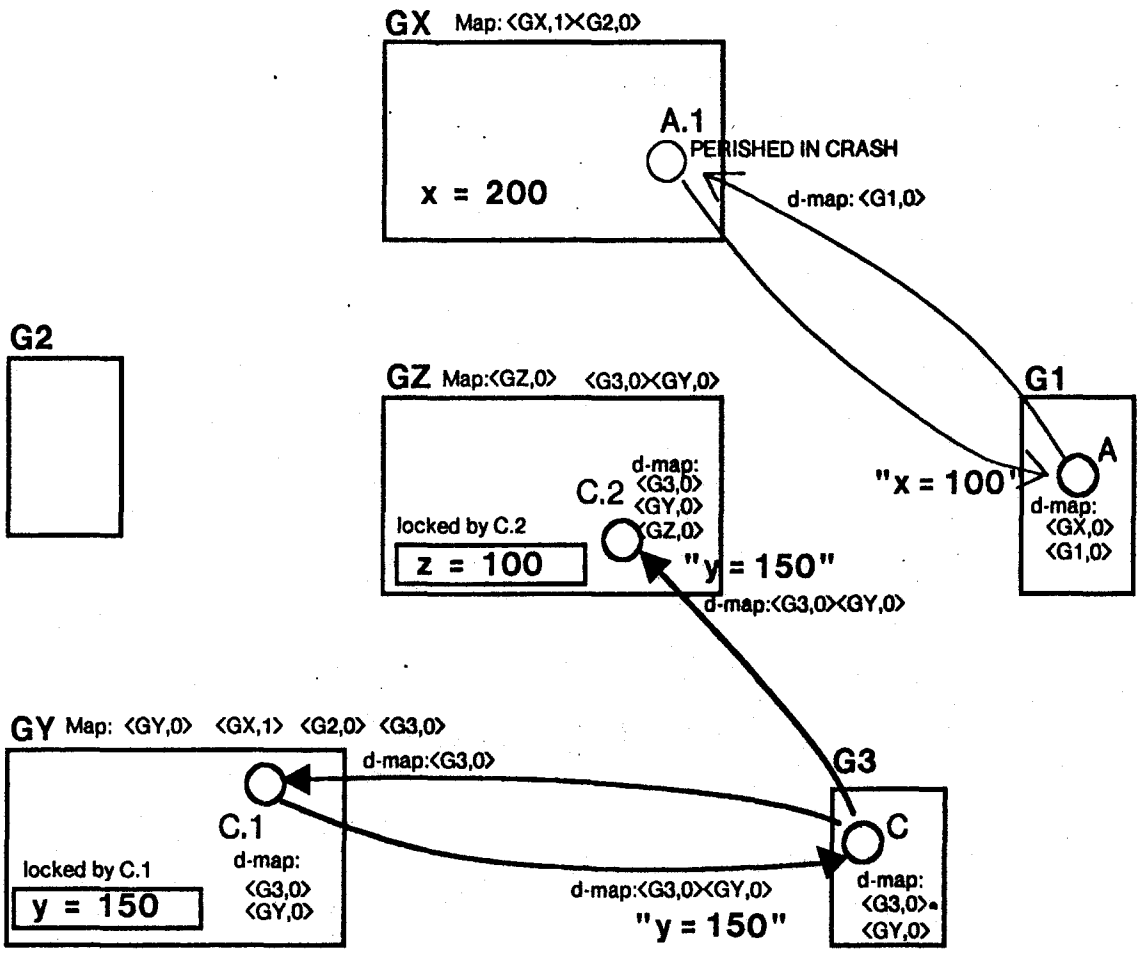


Figure 8-3: Counter-example snapshot three

than any entry in the call message's piggybacked d-list-map, the call is accepted. Subaction A.2 is created at GZ to run the call. A.2 then reads z and finds that the consistency constraint has been unexplicably violated. Figure 8-4 illustrates the final situation.

Allchin's orphan detection algorithm is more efficient than the one presented in this thesis, since it never piggybacks any guardian's entire map on any message. As the counter-example shows, however, this optimization does not always lead to correct results.

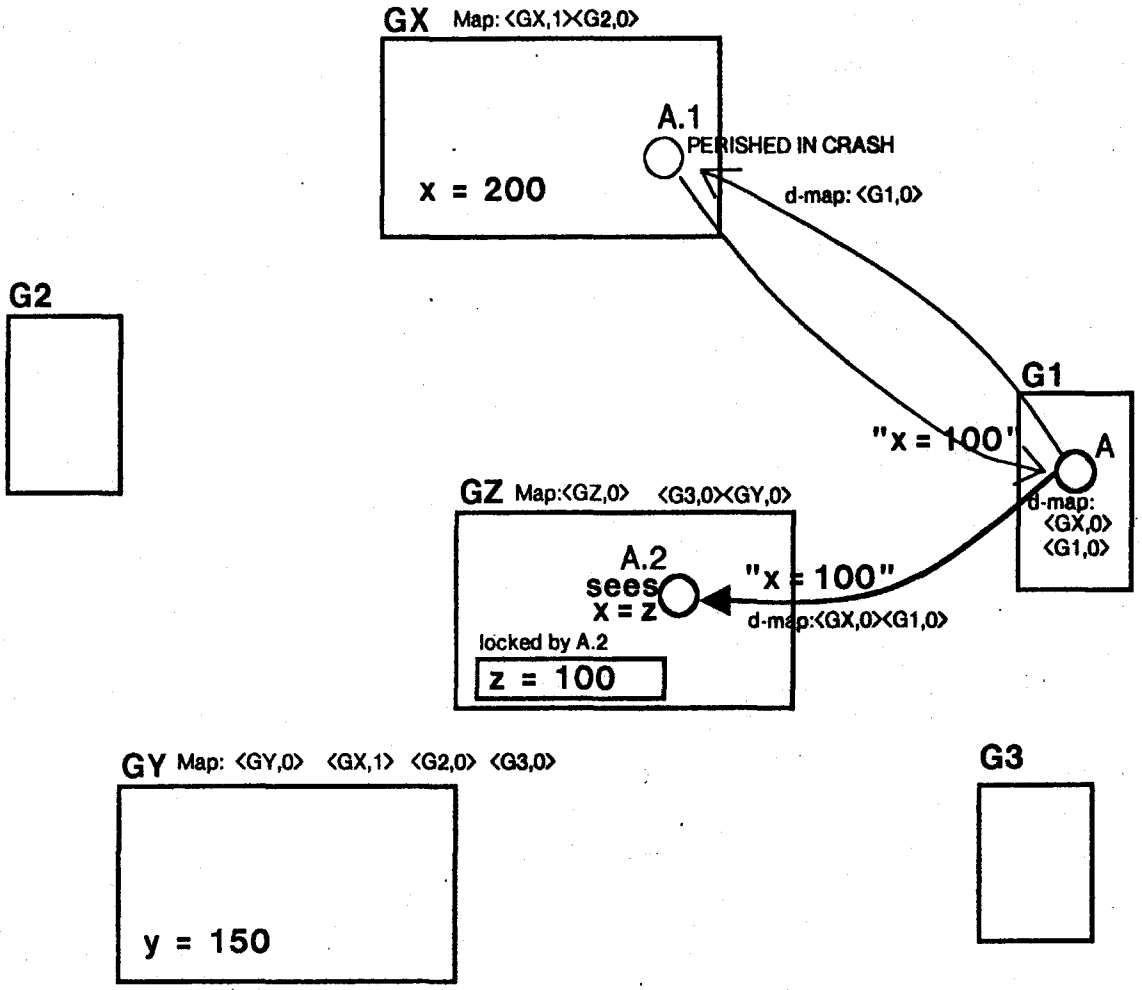


Figure 8-4: Counter-example snapshot four

8.2 Summary and Suggested Work

This thesis presented an orphan detection algorithm that worked by piggybacking two data structures named map and done on messages. Since map and done can be large, this algorithm is not practical.

A method called deadlining was introduced to trim the sizes of done and map. In this method, a map-deadline and done-deadline are associated with actions. When either of an action's deadlines arrives, the action is aborted unless it successfully completes the deadline extension procedure.

A performance analysis of deadlining was presented that predicted its performance. From this analysis, it was concluded that map and done deadlining both should work satisfactorily, but in heavily utilized systems it is important to avoid adding identifiers to done whenever possible.

More work to improve deadlining as presented in this thesis could be done. First of all, the map-deadline period is quite inflexible. A scheme should be developed that permits this period to be changed. A map-deadline extension protocol that works well in the presence of recursion could also be developed. Our protocol inundates the system with messages in this situation.

Also, methods to adjust the done-deadline period based on actual conditions in the system could be developed. For example, if done is too large while very few actions where hitting done-deadlines, the done-deadline period should be decreased. This could be done by having each guardian determine its done-deadline by hill-climbing using a heuristic that balances the tradeoff between the size of done and the amount of deadline extension that goes on. Another way to do this would be to have a single "done-deadline center" for a system. Each guardian would periodically send the center statistics concerning the size of its done and the amount of deadline extension that occurred locally. The center periodically distributes a done-deadline to all the guardians of the system. Since the center has global information about the system, it should be able to do a better job in setting done-deadlines than guardians can do individually.

Since the performance analysis seems to show that done could attain an ample size in large systems even when done deadlining is used, some method for reducing the amount of done transmitted should be developed. One such method would be for a guardian to remember what portion of its done it has previously transmitted to other guardians; the guardian would never transmit an identifier to any particular guardian more than once. Allchin [Allchin81] presents such a scheme in his thesis. Even in systems where done is not large, such a method could significantly reduce the

average size of the portion of done actually transmitted on messages. A guardian need not remember exactly what it has sent to every guardian it has ever communicated with for this scheme to be effective; just remembering what it has sent to the few guardians it communicates with most is sufficient.

Much work needs to be done in verifying the correctness of the algorithm in Chapter 4. Goree [Goree83] proved the abort-orphan detection portion of the algorithm correct, but no work has been completed concerning the correctness of the crash-orphan detection portion of the algorithm. Goree's proof is complex; proof techniques need to be developed that permit cleaner and simpler proofs.

Appendix A

Mathematical Derivations

A.1 Derivation of P[D = n]

1. The discussion in section 7.1.1 justifies equation 8-1:

$$P[D = n] = P[nP \leq L < (n+1)P] \tag{8-1}$$

2. Letting F_L denote the distribution function of L, equation 8-1 can be rewritten as equation 8-2:

$$P[D = n] = F_L((n+1)P) - F_L(nP) \tag{8-2}$$

3. Since L is exponentially distributed with mean $1/\lambda$, equation 8-2 can be rewritten as equation 8-3:

$$P[D = n] = 1 - e^{-\lambda[(n+1)P]} - \{1 - e^{-\lambda nP}\} \tag{8-3}$$

4. Simplifying equation 8-3 yields equation 8-4:

$$P[D = n] = F_L(P)\, e^{-\lambda nP} \tag{8-4}$$

A.2 Derivation of the Mean of D

1. $E[D] = \overline{D}$. Applying the definition of expectation yields equation 8-5:

$$E[D] = \sum_{n=0}^{\infty} nP[D=n] \tag{8-5}$$

2. Substituting using equation 8-4 gives equation 8-6:

$$E[D] = F_L(P) \sum_{n=0}^{\infty} ne^{-\lambda nP} \tag{8-6}$$

3. Eliminating the summation yields equation 8-7:

$$E[D] = F_L(P)\, \{e^{-\lambda P} / (1 - e^{-\lambda P})^2\} \tag{8-7}$$

4. Simplifying 8-7 leads to equation 8-8:

$$E[D] = 1 / (e^{\lambda P} - 1) \tag{8-8}$$

A.3 Derivation of the Mean of S

1. S denotes the amount of time a topaction identifier stays in done before being deleted. Let X be defined such that $S = P - X + \varepsilon$. We first derive the distribution of X. X denotes the amount of time that has passed since a topaction's last deadline when that topaction terminates. A topaction terminates after hitting some particular number of deadlines. Let X_i denote the amount of time that has passed since a topaction's last deadline when that topaction terminates, given that the topaction terminated sometime after its i^{th} deadline but before its $i^{th}+1$ deadline. Equation 8-9 gives the distribution of X_i.

$$F_{X_i}(t) = P[\, L < t - iP \mid iP \leq L < (i+1)P \,] \tag{8-9}$$

2. Due to the memoryless property of the exponential distribution, equation 8-9 can be rewritten as equation 8-10. Hence the distribution of X_i is independent of i, so $X = X_i$.

$$F_X(t) = P[\, L < t \mid L \leq P \,] \tag{8-10}$$

3. Applying the definition of conditional probability leads to equation 8-11:

$$F_X(t) = F_L(t) \,/\, F_L(P), \text{ where } 0 \leq t \leq P \tag{8-11}$$

4. Differentiating equation 8-11 yields the density function of X, given by equation 8-12:

$$f_X(t) = \lambda e^{-\lambda t} / (1 - e^{-\lambda P}), \text{ if } 0 \leq t \leq P \tag{8-12}$$

5. We now proceed to determine E[X]. Applying the definition of expectation yields equation 8-13:

$$E[X] = \int_0^P x\, f_X(x)\, dx \tag{8-13}$$

6. Simplifying equation 8-13 yields equation 8-14:

$$E[X] = \{\, F_L(P) - \lambda P e^{-\lambda P} \,\} / \lambda F_L(P) \tag{8-14}$$

7. Since $\overline{S} = E[S] = P - E[X] + \varepsilon$, $\overline{S}$ is readily obtained from equation 8-14

References

Allchin83 — Allchin, James, "An Architecture for Reliable Decentralized Systems," Ph.D. thesis, *Technical Report GIT-ICS-83/23*, Georgia Institute of Technology, Atlanta, Georgia, 1983.

Eswaren76 — Eswaren, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L., "The notion of consistency and predicate locks in a database system," *Comm. of the ACM 19*, 11 (November 1976), 624-633.

Goree83 — Goree, John Jr., "Internal Consistency of a Distributed Transaction System with Orphan Detection," Master's thesis, *Technical Report MIT/LCS/TR-286*, M.I.T. Laboratory for Computer Science, Cambridge, Massachusetts, January, 1983.

Gray78 — Gray, James, "Notes on Database Operating Systems," in Goos and Harmanis (editors), *Lecture Notes in Computer Science 60*, pages 393-481, Springer-Verlag, Berlin, 1978.

Kleinrock75 — Kleinrock, Leonard, *Queueing Systems, Volume I: Theory*, John Wiley and Sons, 1975.

Lampson81 — Lampson, Butler, "Applications and Protocols," in Butler Lampson, editor, *Distributed Systems: Architecture and Implementation, an Advanced Course*, Chapter 14, Springer-Verlag, 1981.

Liskov82 — Liskov, Barbara and Scheifler, Robert, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, January, 1982, 7-19. Also in *ACM Transactions on Programming Languages and Systems*, July 1983.

Liskov83 — Liskov, Herlihy, Johnson, Leavens, Scheifler, and Weihl, "Preliminary Argus Reference Manual," M.I.T. L.C.S. Programming Methodology Group Memo 39, October 1983.

Liskov84 — Liskov, B., "Overview of the Argus language and system," M.I.T. L.C.S. Programming Methodology Group Memo 40, Feburary 1984.

Marzullo83 — Marzullo, Keith, "Loosely-coupled distributed services: a distributed time service," Ph.D. thesis, Stanford University (1983).

Nelson81 Nelson, Bruce, "Remote Procedure Call," Ph.D. thesis, *Technical Report CSL-79-3*, Xerox Palo Alto Research Center, 1981.

Seitz83 Seitz, N. B., Wortendyke, D. R., and Spies, K. P., "User-Oriented Performance Measurements on the ARPAnet," IEEE Communications Magazine, 1983.

Wood80 Wood, W.G., "Recovery control of communicating processes in a distributed system," *Technical Report 158*, University of Newcastle upon Tyne, 1980.