Web services management approaches

by J. A. Farrell
H. Kreger

Web services are important to business-to-business and business-to-consumer application deployment and are poised to be a critical aspect of the Web architecture of a business. Their reliable operation is required for the smooth and profitable operation of the business, mandating that Web services be well managed. This management includes controlling the life cycle of the service and collecting information about existence, availability, and health. All these activities can be accomplished in a manner specific to no particular vendor so that a number of management applications, such as those from Tivoli, can manage Web services in the context of the business applications of which they are components, as well as in relation to the other resources in the enterprise.

Web services[1] are rapidly emerging as important building blocks for business integration. They are finding important applications in business-to-business, business-to-consumer, and enterprise application integration solutions. As such, Web services form a critical aspect of e-business architecture and, in that role, their reliable execution must be assured. Reliability must be a first-rank consideration for organizations deploying such solutions. The management of computer systems and applications is a discipline that is well developed and extensive.[2–4] It has evolved in recent years to encompass Web-based applications. The management of Web services continues this evolution.

Our purpose is to show how Web services developers can incorporate manageability into their applications, leveraging the existing application management discipline and the Web services environment. We first give a brief survey of the parts of application management that are relevant to our discussion. Then we show how Web services fit into the management environment and what unique issues emerge, allowing us to present a set of principles and patterns for the management of Web services and give examples of how these principles and patterns can be applied. Finally, we show how Web services management is supported by the IBM Web Services Toolkit.

Overview of application management

Application management encompasses the control and monitoring of an application throughout its life cycle. It spans a range of activities from installation and configuration to collecting metrics and tuning to ensure responsive execution. The architecture of most management approaches fits a manager-agent model. In this model the application, or managed service, is any computer program engaged in solving a problem or implementing a process. These programs can run on servers, clients, and even mobile or embedded devices. The manager-agent model is important to our discussion, so we describe it by breaking it down into its components.

The managed application has a set of responsibilities, including exposing appropriate data for use by a management system, responding to requests from the management agent, recognizing internal errors, and posting events to the management system. The application communicates with or contains a management agent or subagent.

The role of an agent is to communicate with the management system and the application. The agent usually runs in the same host or process as the application it is managing. It is responsible for sending events to the management system, relaying data and command requests from the management system to the application, gathering responses, and returning them to the requester.

The management system is the overall framework for application management. It is responsible for providing the infrastructure and user interface to manage applications. It communicates with management agents that support a given management protocol, such as Simple Network Management Protocol (SNMP)[5,6] or Java** Management Extensions (JMX**).[7] It can be a sophisticated enterprise manager, such as the Tivoli Management Environment, or a resource manager, such as a database management system. Or it may be more application-specific, like the WebSphere* administration application.

Application life cycle. Application management provides fundamental support for the life cycle of a program or solution. This life cycle can be summarized in the following steps:

1. Install or deploy
2. Start or make available
3. Execute
4. Update installed or deployed application
5. Stop or make unavailable
6. Uninstall or undeploy

Our focus is on steps 2, 3, and 5, since they encompass the application design considerations. Installation and deployment as well as subsequent application maintenance depend on the type of environment that hosts the Web service, such as a Web application server. They typically do not impose significant new requirements on the managed application.

In contrast, to start and stop an application and to monitor and control its execution require the management system to interact directly with the application or its execution environment. The application developer must supply commands and application programming interfaces (APIs) for operations that are invoked by the management system, including a means to start and stop the application. The developer must also define, maintain, and expose configuration and metric information through logs, events, commands, or APIs for the management system to monitor or understand. During the execution phase, the following information and services should be provided by the application and gathered and referenced by the management system.

Identification—Identification includes the static information that uniquely describes a manageable service. This information can include its instance name, the product name, product version, installation date, descriptive text, configuration file names, port, and uniform resource locator (URL).

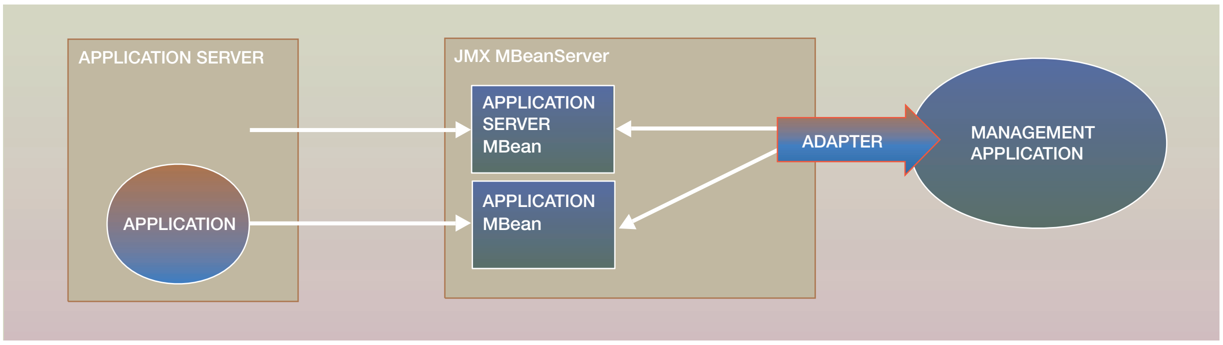
Availability—The availability of a service refers to its being accessible through the network, system, and application infrastructure. Availability also indicates whether the service is able to do its work in a valid, responsive way.

Metrics—Metrics are usually numeric information, provided by a manageable service, that can be used to indicate or calculate the health and performance of the service. Metrics are often polled and, once collected, are graphed and subjected to threshold analysis.

Operations—Operations can be management-oriented (start or make available, stop or make unavailable, obtain statistics), or they can be very service-specific and part of the business context. For example, DataBaseBackup( ) would be an operation for a database application. Operations may or may not change the current behavior of the application. But if they do, the change is rarely persistent over subsequent invocations of the application. An operation may turn certain trace points "on" during execution, but it will be reset to "off" when the application is started again.

Configuration—Configuration information includes parameters that control how an application operates. A configuration can be static (not changeable while the service is available) or dynamic (configuration can change while the service is available without service disruption). Configuration changes affect the current behavior of the service, and the changes are persistent over invocations of the service. A config-

Figure 1 An MBeanServer as a bridge between the application and management system



uration may exist for how the Web service interacts with its clients, or for its business logic.

Events—Events are messages from the Web service to a management system. They can flag a failure condition or warn of an impending application outage. They can also signal a life-cycle, status, configuration, or metric change. Reactions include running a recovery or tuning policy, notifying operators, logging, or filtering. Events may also indicate that the application is healthy; these are usually called "heartbeats." In this case, when the events are not received when expected, the management system should react.

Instrumentation options. The management data and capabilities listed above can be provided by external, execution environment, and internal manageability instrumentation. External instrumentation includes definition files describing the application and its management requirements, log files, and utilities. Definition files are used by the management system to guide its support of the managed service. Log files are monitored and analyzed by management systems and tools to detect and diagnose failures and trends. Utilities can provide support for the configuration and operation and are used to interact with the application directly from command lines or scripts. External instrumentation of an application does not directly affect the development of the application. These files and utilities can be shipped with the application, created during the install and update phases of the life cycle, or provided by a management application. The execution environment of the application, like an application server, can provide management information and operations on behalf of and without the involvement of the application. This information would include some execution statistics (i.e., number of invocations), start and stop operations, current availability status, life-cycle events, and events when the application is no longer executing within the run time. However, applications typically need to be internally instrumented to provide detailed metrics, configuration, status, and events related to the specific processing they perform.

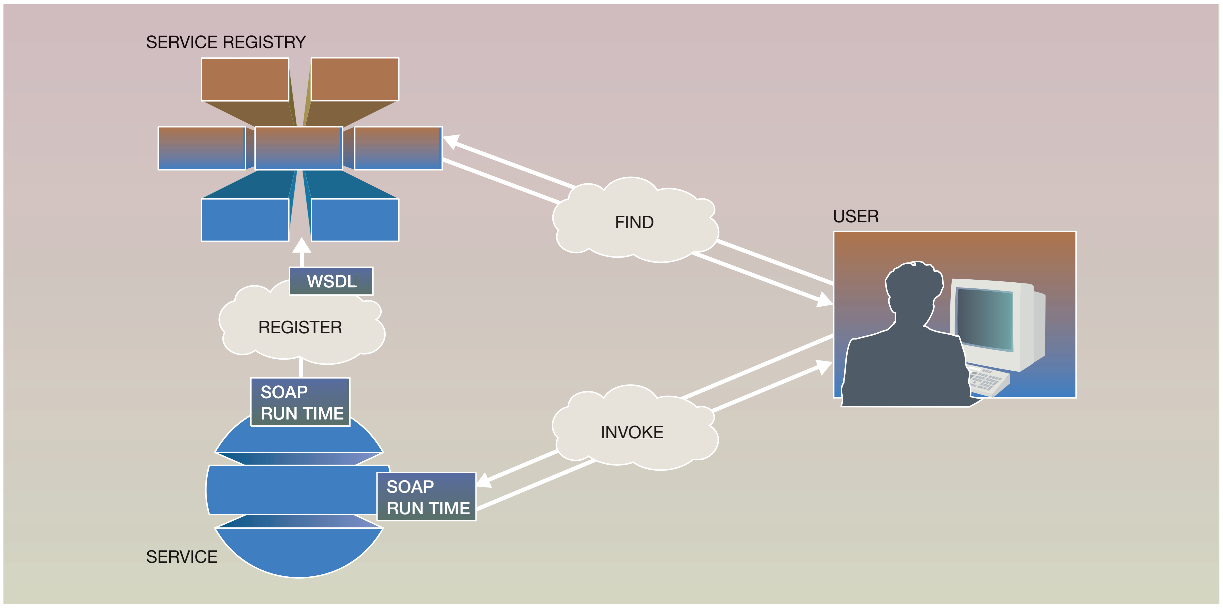
Internal instrumentation is usually provided from within the application itself. The application will publish events, status, configuration, and metric data specific to its business logic. Generally the application does this by creating and advertising a management object conforming to some management interface standard or to the requirement of a particular management system.

An example applicable to Java Web application servers is a JMX Management Bean (MBean). An MBean is an object that is accessible via standard JMX interfaces and provides a developer with the ability to expose application-specific management interfaces. MBeans run in an MBeanServer in the local application environment (the Java virtual machine or server running the application). The MBeanServer provides adapters to specific application managers and administration facilities and thus forms a bridge between the application and management system, as shown in Figure 1.

Overview of Web services

Web services represent an evolution of the Web to allow applications to interact over the Internet in an open and flexible way. Important in this approach

Figure 2 Web services conceptual architecture



is independence of the interactions from the platform, programming language, middleware, and implementation of the applications involved. Web services are self-contained, modular applications that are described, found, and called via a set of standards based on Extensible Markup Language (XML). These standards are being formalized chiefly by the World Wide Web Consortium (W3C).

The interface of a Web service is described in an XML format called the Web Services Description Language (WSDL).[8] A WSDL file contains descriptions of one or more interfaces and binding information for one or more services. A *service* is actually a collection of ports. A *port* is the combination of a *portType*, which describes the interface of the port, and a *binding*, which describes the mechanics of invoking the port.

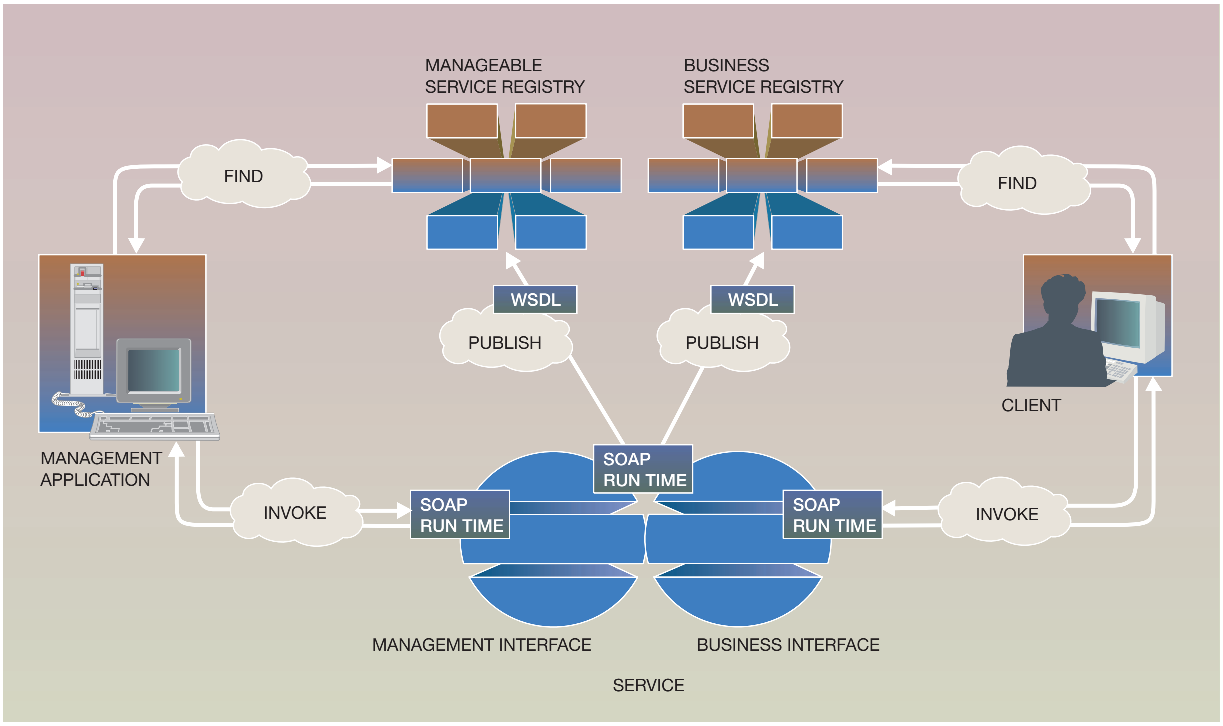
Figure 2 illustrates how these service descriptions can be published to a service registry where they can be discovered by potential Web service clients. The Universal Description, Discovery, and Integration (UDDI)[9] project, UDDI.org, is defining such a Web services registry. Once a service has been discovered, and a binding established based on information in the registry, the interaction between the calling application and the Web service can begin. This interaction is the most important aspect of Web services for our discussion.

The invocation of a service involves sending an XML message to the service and receiving an XML message in return. These XML interactions are governed by another open standard called the Simple Object Access Protocol (SOAP).[10] SOAP defines a message header that describes the message and indicates which operation in the interface of the service is being invoked. The header is an envelope that contains an XML message body in which the parameters are passed. SOAP supports both a remote procedure call and a general XML document passing paradigm. SOAP messages must be carried on a communications layer, which most often is the HyperText Transport Protocol (HTTP).

More extensive descriptions of Web services architecture, standards, and technology can be found at the IBM Web services development Web site.[11,12]

Many Web services are wrappers for existing applications so that these applications can be accessible on the Internet or an intranet. As such, many are very simple and can be generated automatically by tools. This condition implies a requirement that the addition of management to a Web service must not

Figure 3 An application with both its functional and management interfaces accessible as Web services



introduce complexity or undue developer work that would cause this simplicity to be lost.

Web services provide for a very dynamic, flexible, and reconfigurable execution environment. It is important that the management approach also support these attributes, that the general management architecture be correspondingly adapted, and that the application roles fit the Web services model. In the following sections, we describe how to address these requirements in both the management infrastructure and the Web service design.

Managing Web services

As we have seen, the Web services programming model is based on service definition through WSDL documents, discovery through registries of published services, and SOAP over HTTP communications between services. The relationships between the client, the service, the registry, and the SOAP run time are shown in Figure 2. The management of these services should not require the Web services to constrict this programming model; in fact, it should be based on the same set of principles.

A SOAP run time on the service provider's host will receive the Web service invocation and delegate the request to the Web service implementation class, so we can think of Web services as running within the scope of a SOAP run time. This positioning provides us a convenient place from which to do execution environment manageability instrumentation for the Web service. We cover the details of what the SOAP run time should track in the following sections.

Because internal manageability instrumentation is often exposed as an API or management object, it is natural to think of this interface as a management-oriented interface to the service. This means that it would have its own port in the WSDL document describing that interface, as shown in Figure 3. As a result, this WSDL can be published to a UDDI registry where a management application can discover and introspect the WSDL, and thus begin to manage the Web service. More detail is provided in the following sections.

Web services management principles

The approach of an organization to Web services management will be grounded in the generally applicable management models previously described. Web services-based applications will have some characteristics that make management a little more challenging, as follows:

- Web services are described with XML and are accessible using interoperable standard protocols and transports. Therefore, applications based on Web services will be able to use services that execute in many diverse environments—systems, languages, platforms, and enterprises. Thus, it is not practical to dictate the use of one particular management technology for all Web services. For example, JMX may work well for Web services implemented in the Java language, but it is not a reasonable alternative for services implemented in C++ or using Microsoft's .Net** platform.
- Web services-based applications will cross enterprise boundaries more now than applications that have gone before. As a result, it must be possible to interact with the management aspects of these applications across enterprise boundaries as well, preferably using the same communications pipes and technologies already agreed upon among the companies.
- Web services define a standard mechanism for publishing, finding, and interacting with other Web services. Management systems have traditionally also defined such a mechanism, both standards-based and proprietary, for publishing, finding, and interacting with manageable resources. Management systems that support Web services should interact with the Web services publication and discovery practice.
- Web services have a natural loose coupling that means a service should be able to "find" its management services—using the Web services paradigm—at run time, rather than having them statically bound or internalized during development. As a result, the Web services, as well as the management services, are more portable to different execution environments.

All of these facts drive the need to develop a management approach that stays within the Web services paradigm. It means defining the management interactions with WSDL, the management applications as Web services, the manageable services with WSDL, and discovery using service registries and WSDL.

Thus, when considering the nature and usage models of Web services and the challenges of managing them, several specific management principles emerge. We summarize them here, and then treat each in detail, illustrating each with the StockQuote service example.

- The principle of *separate management interface* means to define the business interface separately from the administration or management interface.
- The principle of *data collection by the run-time infrastructure* means do not instrument an application to collect management data that should be collected by the execution environment, such as invocation counters, failure counters, execution timing, and life-cycle events.
- The *use an event collector* principle means send events for catastrophic events, metric data changes, configuration data changes, operation invocation, or business-specific life-cycle events to an event collector Web service.

The popular and simple StockQuote service example is used to illustrate these principles. Since they are illustrative, the examples will show fragments to be added to this WSDL for the port and portTypes. Bindings are not shown in the examples. One can assume a standard SOAP over HTTP binding will work. The StockQuote service example is shown in Figure 4.

Separate management interface. A Web service is fundamentally an interface accessible over a set of open standard discovery and invocation mechanisms. The Web services discovery and binding processes are driven by interface descriptions. A Web service interface is described as a port type in a WSDL file. When an organization searches a UDDI registry for a Web service, the target of the search is an interface described in WSDL. Management operations should be exposed through a separately described and published Web service interface, which allows a separation of concerns between business interactions and management interactions with the service. There are several reasons for this, as we now describe.

First, the search for a Web service usually involves looking for a standard or mutually agreed-to interface. The mixing of management operations with those that are formally part of the business interface will interfere with the search for the service based on interface contents by changing the signature of the interface.

Figure 4 StockQuote service example

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="StockQuoteInterfaceDefinitions"
                  targetNamespace="urn:StockQuoteInterface"
                  xmlns:tns="urn:StockQuoteInterface"
                  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

         <wsdl:message name="GetQuoteInput">
                      <part name="symbol" type="xsd:string" />
         </wsdl:message>
         <wsdl:message name="GetQuoteOutput">
                      <part name="value" type="xsd:float" />
         </wsdl:message>

         <wsdl:portType name="StockQuoteInterface">
                 <wsdl:operation name="GetQuote">
                          <wsdl:input message="tns:GetQuoteInput" />
                          <wsdl:output message="tns:GetQuoteOutput" />
                 </wsdl:operation>
         </wsdl:portType>

         <wsdl:binding name="StockQuoteBinding" type="tns:StockQuoteInterface">
                     <soap:binding style="rpc"
                                         transport="http://schemas.xmlsoap.org/soap/http" />
           <wsdl:operation name="GetQuote"/>
                 <soap:operation soapAction="urn:StockQuoteInterface#GetQuote" />
                 <wsdl:input>
                          <soap:body use="encoded" namespace="urn:StockQuoteService"
                             encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
                 </wsdl:input>
                 <wsdl:output>
                          <soap:body use="encoded" namespace="urn:StockQuoteService"
                             encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
                 </wsdl:output>
                 </wsdl:operation>
         </wsdl:binding>
         <wsdl:service name="StockQuoteService">
                  <wsdl:port name="StockQuoteServicePort"
                           binding="sqi:StockQuoteInterface">
                           soap:address location="urn"/>
                  </wsdl:port>
         </wsdl:service>

</wsdl:definitions>
```

Second, it will allow business partners to see and use management interfaces that are not relevant to their interactions with the Web service. Likewise, it will clutter management consoles with business operations when it should only display management operations.

Third, providing a separate service and port in the WSDL document for the management interface enables more targeted publishing of the business and management service ports in the WSDL. The service in the WSDL document for the management interface of a Web service does not have to be published in a public UDDI registry along with the business interface described in the business service port, since only management applications will be interested in them. If it is published in a public UDDI registry, the service in the WSDL must be categorized as "management" so that the management system can find it. The management interface will most likely be pub-

Figure 5 Port extensions

```
<wsdl:service name="StockQuoteService">
            <wsdl:port name="StockQuoteServicePort"
               binding="sqi:StockQuoteInterface"
                      soap:address location="urn"
            </wsdl:port>
            <wsdl:port name="StockQuoteManagementPort"
                     Binding="squi:StockQuoteManagementInterface"
                     Soap:address location="urn"
            </wsdl:port>
            <wsdl:port name="StockQuoteGenericManagementPort"
                     Binding="squi:GenericManagementInterface"
                     Soap:address location="urn"
            </wsdl:port>

</wsdl:service>
```

lished in a private UDDI registry that caters to management systems as the client, rather than business systems as the client. Here, management systems, administration utilities, or operator facilities can discover and locate manageable services. The management service can "introspect" the WSDL to find the management and administrative operations, available metrics, and events supported by the service. Of course, because the WSDL also contains the binding information for the management port, the management service will now be able to support and interface with the managed service. In this scenario, a private UDDI registry is one that is deployed by a company or organization to further the deployment of Web services to address its own internal application integration needs.

The manageable service can support a generic management interface that provides a simple access to identification, configuration, and metric data. Ideally, this port would be widely supported by management applications that support Web service management.

An example of a generic, overly simplified, management interface that could be implemented by any manageable Web service is shown below.

```
public interface ManageableService {
    // return an ID string for the service being
       managed
    public String getServiceID( );
    // return an array of metric names and a
       corresponding array of values
    public String[ ][ ] getMetrics( );
    // return an array of config property names
       and a corresponding array of values
    public String[ ][ ] getConfiguration( );
    // return the WSDL document that describes
       this interface
    public String getAdminInterface( );
    // return true to indicate that the service is able
       to respond to requests
    // return false if the service is experiencing
       out-of-range delays
    public Boolean isAvailable( );
}
```

There might also be a custom manageable service WSDL interface that contains the interactions for the specific administrative or management operations of a Web service. Examples of operations in this kind of WSDL would be "TraceOn( )," "AddNewUser( )," or "SelectNewVendor( )."

Using these concepts, we could add these management ports to the StockQuote service WSDL. Please note that these extensions are strictly illustrative. The service would now look like what is shown in Figure 5. The custom management interface might be defined to be as in Figure 6, and the generic management interface is defined to be as in Figure 7.

Data collection by the run-time infrastructure. Web services are invoked and send data via SOAP, currently being standardized by the World Wide Web Consortium's XML Protocol Working Group.[10] The SOAP processor and related Web services infrastruc-

Figure 6 Custom management interface

```
<wsdl:message name="ChangeAvailabilityInput">
            <part name="action" type="xsd:string" />
</wsdl:message>
<wsdl:message name="ChangeAvailabilityOutput">
            <part name="newState" type="xsd:string" />
</wsdl:message>
<wsdl:message name="setTraceInput">
            <part name="action" type="xsd:string" />
</wsdl:message>
<wsdl:message name="changeStockExchange">
            <part name="exchangeName" type="xsd:string" />
</wsdl:message>

<wsdl:portType name="StockQuoteManagementInterface">
        <wsdl:operation name="setTrace">
                <wsdl:input message="tns:setTraceInput" />
        </wsdl:operation>
        <wsdl:operation name="changeStockExchange">
                <wsdl:input message="tns:changeStockExchangeInput" />
        </wsdl:operation>
        <wsdl:operation name="changeAvailability">
                <wsdl:input message="tns:ChangeAvailabilityInput" />
                <wsdl:output message="tns:ChangeAvailabilityOutput" />
        </wsdl:operation>
</wsdl:portType>
```

ture form a control point that allows automatic instrumentation for certain classes of management information.

Figure 8 shows how the Web services infrastructure can implement automatic instrumentation using the Java Management Extensions (JMX). The SOAP processor finds or instantiates an MBeanServer when it is initialized. It then creates or locates an existing MBean for itself and each Web service it manages. An MBeanServer fills the role of the agent in the manager-agent model. Whenever the SOAP processor is called, it collects execution statistics about the invocation and response of the Web service. It should also provide the interface called by the management system to collect the data for and control a managed service and the SOAP processor itself. This instrumentation should not be duplicated in the application or an application-provided agent. Information that the infrastructure can collect for a Web service includes:

- Identification information that includes Service ID (identifier) and URL
- Current availability for the URL and port of the Web service, as well as an indication of whether or not a request on the service itself completes in an acceptable time period
- Metrics for:
  –Number of requests
  –Number of responses
  –Number of failure responses
  –Number of invocations of each method
  –Average response time for invocation of each method
  –Total elapsed execution time
- Operations to enable and disable the ability to invoke the Web service
- Events to signal life-cycle changes and request failures

The execution environment can also collect statistics to be used as a basis for usage monitoring and billing applications. Some execution environments will be able to provide basic life-cycle operations (i.e., enable, disable) and events (i.e., life cycle and service not available).

This principle is in keeping with the often very simple nature of a Web service. Many Web services are unsophisticated wrappers around existing applica-

Figure 7 Generic management interface

```
<wsdl:message name="GetServiceIdOutput">
                <part name="value" type="xsd:float" />
</wsdl:message>

<wsdl:message name="GetMetricsOutput">
                 <part name="name" type="xsd:string" />
                 <part name="type" type="xsd:string" />
                 <part name="value" type="xsd:string" />
</wsdl:message>

<wsdl:message name="GetConfigurationOutput">
                <part name="name" type="xsd:string" />
                <part name="type" type="xsd:string" />
                <part name="value" type="xsd:string" />
</wsdl:message>

<wsdl:message name="GetAdminInterfaceOutput">
                <part name="AdminInterfaceWSDLurn" type="xsd:string" />
</wsdl:message>

<wsdl:message name="GetAvailabilityOutput">
                <part name="value" type="xsd:integer" />
</wsdl:message>

<wsdl:portType name="GenericManagementInterface">
          <wsdl:operation name="GetServiceId">
                    <wsdl:output message="tns:GetServiceIdOutput" />
          </wsdl:operation>
          <wsdl:operation name="GetMetrics">
                    <wsdl:output message="tns:GetMetricsOutput" />
          </wsdl:operation>
          <wsdl:operation name="GetConfiguration">
                    <wsdl:output message="tns:GetConfigurationOutput" />
          </wsdl:operation>
          <wsdl:operation name="GetAdminInterface">
                    <wsdl:output message="tns:GetAdminInterfaceOutput" />
          </wsdl:operation>
          <wsdl:operation name="IsAvailable">
                    <wsdl:output message="tns:IsAvailableOutput" />
          </wsdl:operation>
</wsdl:portType>
```

tions. As such, they are easy to create. Many, in fact, can be automatically generated by tools such as the IBM WebSphere Studio Application Developer. By relying on the Web services execution environment to provide this basic level of instrumentation automatically, this principle promotes management while keeping the Web services development tasks as simple as possible.

For the StockQuote service example, there would be a number of invocations and average response time metrics kept for the getQuote method, but not the management port methods. In fact, if we adhere to this principle, we would remove the changeAvailability operation in the StockQuoteManagementInterface, since this would be addressed by the enable and disable operations provided by the execution environment for all Web services.

Use an event collector. A Web service may need to signal the occurrence of significant events to the management system. These include catastrophic events, metric data changes, configuration data changes, operation invocation, or business-specific life-cycle

Figure 8 Management infrastructure for Web services

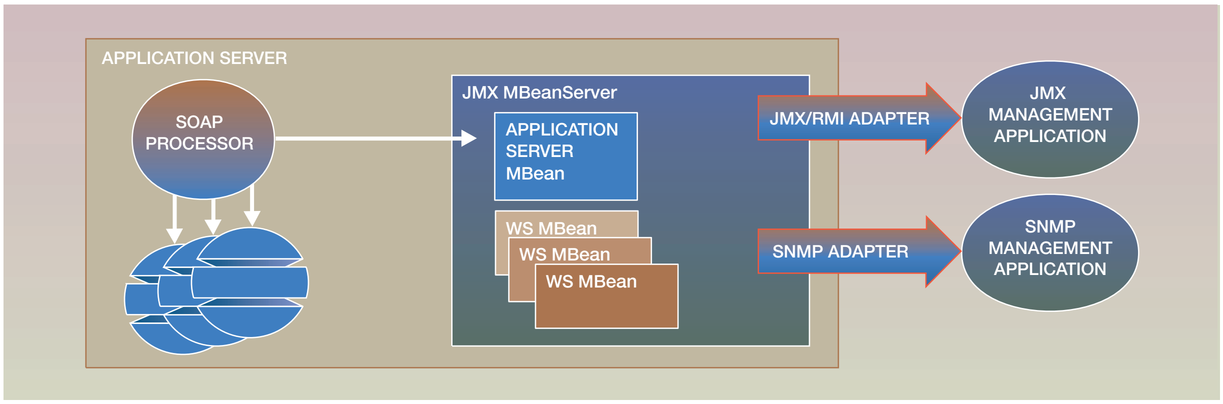
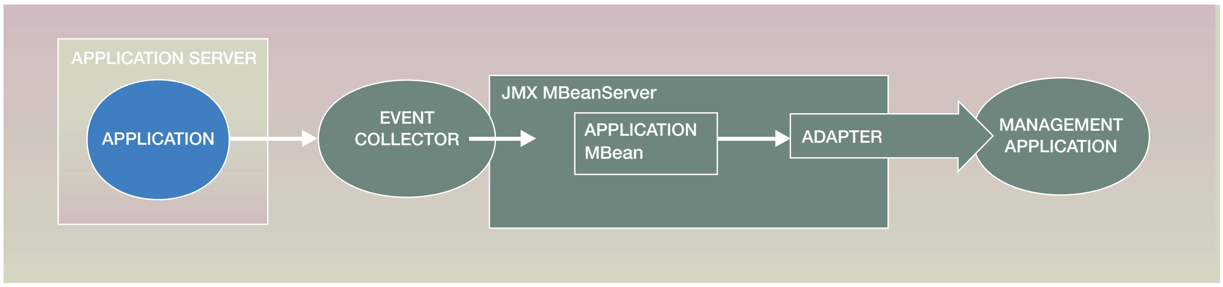


Figure 9 Event collector management intermediary



events beyond those automatically collected by the infrastructure. A convenient approach to implement this interaction is to create an event collector Web service. This reusable service can encapsulate the interactions with the management system while allowing the managed Web service to signal its events through a portable management system-independent mechanism.

Figure 9 shows the use of an event collector. A set of Web services makes use of a common event collector service that exposes a simple interface for receiving events. Calls on the event collector interface cause the events to be forwarded to the management systems that have registered for them. In this example, the event collector uses an MBeanServer to create or locate an existing JMX MBean on behalf of each Web service with which it interacts (theoretically, the same one used by the SOAP processor). When events are received from a manageable service, the event collector uses the MBeanServer to forward the event to management adapters.

The event collector interface can be quite general and simple. Minimally, it must include a method that allows it to receive the management event. For example:

```
public interface EventCollector {
  public void deliverEvent (String id, String source,
    String severity, String text);
}
```

The parameters identify the Web service, specify the cause of the event, indicate the severity of the condition, and provide additional information in the form of unformatted text. When the Web service initializes, it can dynamically discover a Web service that implements this interface, as described by a published WSDL document. Alternatively, the Web ser-

Figure 10 Messages and ports in WSDL for event collector service

```
<wsdl:message name="Event">
            <part name="id" type="xsd:string" />
            <part name="source" type="xsd:string" />
            <part name="severity" type="xsd:string" />
            <part name="text" type="xsd:string" />
</wsdl:message>

<wsdl:message name="requestEventInput">
            <part name="idpattern" type="xsd:string" />
            <part name="sourcepattern" type="xsd:string" />
            <part name="severityrange" type="xsd:string" />
            <part name="textpattern" type="xsd:string" />
            <part name="receiverWSDLPort" type="xsd:string"/>
</wsdl:message>

<wsdl:portType name="EventCollectorInterface">
        <wsdl:operation name=" deliverEvent">
                <wsdl:input message="tns:Event" />
        </wsdl:operation>
</wsd:portType>

<wsdl:portType name=EventRequestorInterface">
        <wsdl:operation name="requestEvents">
                <wsdl:input message="tns:RequestEventsInput" />
                <wsdl:output message="tns:RequestEventsOutput" />
        </wsdl:operation>
        <wsdl:operation name="deliverEvent">
                <wsdl:output message="tns:Event" />
        </wsdl:operation>

</wsdl:portType>
```

vices developer can statically bind the Web service to the event collector. The event types are application-defined and are not constrained by the event collector. If an event collector supported delivering events to interested management services via Web services, it could also provide a WSDL-described port that other Web services could use to ask the event collector to invoke their "receive event" operation based on certain conditions or contents of the event.

The WSDL for the event collector service would contain the messages and ports given in Figure 10.

For our StockQuote service example, it would emit events whenever the stock exchange from which it obtains quotes takes longer than three seconds to reply or has more than three "too busy to service you" responses in a row. A management application could listen for these events and change to a different stock exchange. The WSDL for emitting those two events would look like Figure 11. Note that this WSDL matches the expected input for the event collector Web service.

Figure 11 WSDL for delayed reply or busy condition

```
<wsdl:message name="Event">
        <part name="id" type="xsd:string" />
        <part name="source" type="xsd:string" />
        <part name="severity" type="xsd:string" />
        <part name="text" type="xsd:string" />
</wsdl:message>

<wsdl:operation name="deliverEvent">
        <wsdl:output message="tns:Event" />
</wsdl:operation>
```

Web services management patterns

The developer of a manageable application is faced with many choices concerning how to interact with the management system and how to fit into a manager-agent model. The management principles discussed above provide a framework for approaching the problem, but many decisions still remain. We have found that a set of patterns can provide sufficient guidance to simplify the problem. The developer chooses the pattern that applies to his or her application behavior and requirements.

These management patterns are architectural patterns that define components that make up the solution and the relationship between the application and the management system. They show the placement of function and the flow of information.

Event generator. This pattern is applicable to Web services whose processing contains events and metrics of interest that are not already collected by the Web services execution environment. Further, the service does not require run-time operations or dynamic configuration control. If these conditions are met, this pattern offers a very simple approach to instrumentation. By using an event collector, as described earlier, the Web service can be completely isolated from the details of the management system under which it operates. The service contains instrumentation and sends management data via events to a management system. Events may represent failures, life-cycle changes, state changes, metric data, or configuration data changes. The receiver of this information is responsible for putting the information in context, since the information in the event may be minimal. The Web service does not publish a management object and does not support being invoked for operations or reconfiguration by the management system. It does not have to implement any facilities to listen for and respond to management requests nor implement the generic manageable service port. The Web service does not have a custom management port. The WSDL interface for this pattern is a set of notification messages representing the events.

Noninterruptible. Here again, the service contains instrumentation and sends events and publishes management information to a management presence. Unlike the event generator pattern, the application makes available to the management system a standard object that contains the identity and metric information as well as details of the current configuration. Notifications of changes to the management object can be sent via events specified for that purpose or by resending the updated management object to the management system. This pattern is useful over the event generator pattern when there are great amounts of potentially complex management data that may be frequently updated. When the management is not "real time," sending the entire management object on a periodic basis rather than every time a value is updated tends to be more efficient. It also allows a set of changes to be sent so that they are consistent with one another, rather than sending a set of events, some of which may not arrive in order or at all, leaving the management system's version of the management object in an invalid or inconsistent state. This pattern should also be used when a particular management system requires the data in a particular format. The receiver of this information receives data already in context of the management object it understands. The Web service does not support being invoked for operations or reconfiguration by the management system. It does not have to implement any facilities to listen for and respond to management requests nor implement the generic manageable service port. The Web service does not have a custom management port. The WSDL interface for this pattern contains notification messages for events and a notification message for updating the management object.

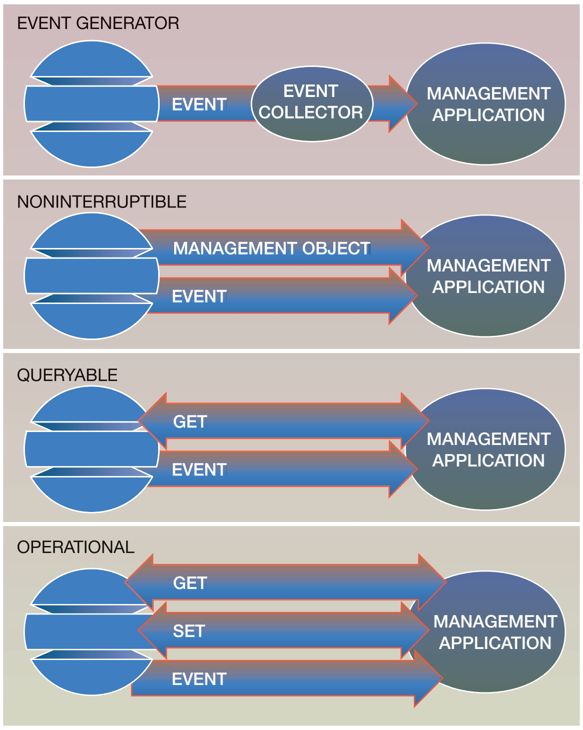
The management object could be any of several types. For Web services being monitored by a JMX management environment, the object could be an MBean. Some management systems also support the Common Information Model (CIM)[13] so that the service could alternatively create a CIM object to fill this role. It could also create a custom object, but such an approach may require some kind of bridge code to make the object understandable to the management system. MBeans or CIM objects are more workable choices.

Queryable. The queryable pattern is related to the event generator pattern in that the Web service does not require configuration or operational control, but in this case, the service may send events and also implements a management interface that can be called by the management system. The management system can retrieve configuration and metric data directly from the managed services. This is essentially read-only management. The managed Web service supports being invoked by the management system to obtain current metric and configuration data, but not being controlled or altered. The management data may be represented as complex data types and objects. Management systems would typically retrieve the management data when they need the data or poll for entire sets of related management data rather than one or two metrics. This approach works well for applications with a high rate of change of metric values or large number of complex metrics such that sending events to signal changes represents too much overhead. The Web service may support a management object and supports being invoked for query operations by the management system. It implements facilities to listen for and respond to management requests and may implement the generic manageable service port. The Web service may have a custom management port. The WSDL interface for this pattern contains a set of notification messages for events and request-response messages for data requests.

Operational. As in the queryable pattern, the management system can retrieve configuration and metric data and obtain events from managed service. However, in this pattern the management presence can set configuration data, thus changing the operation of the application. It can also invoke operations on the managed Web service. Correspondingly, the Web service must implement methods that can be called by the management system and must have a structure designed to allow for configuration changes. The Web service may support a management object and supports being invoked for operations or reconfiguration by the management system. It implements facilities to listen for and respond to management requests and may implement the generic manageable service port. The Web service may also have a custom management port. This pattern employs a WSDL interface to express a set of queries, operations, and notifications.

Figure 12 summarizes the essential characteristics of these four patterns. These patterns give increasing amounts of information and control over the Web service to the management application. Any of the patterns may send events directly to the management application or through an event collector. Both the event generator and noninterruptible patterns use send-only communications and do not implement any way for the management application to initiate communications with the Web service. Events are generally more generic and easily translated to be understood by any management application; therefore, the event generator is more portable than the other patterns. The noninterruptible pattern sends a specific management object and is much more dependent on a management application understanding that object. The queryable and operational patterns will usually use a management object as well. These two patterns must support two-way communications. They must both implement a way for the management application to initiate communications with the Web service. Only the operational pattern allows the management application to control the Web service.

Figure 12 Summary and comparison of Web services management patterns



In Figure 12, arrows pointing toward the management system indicate that events are being sent asynchronously by the agent. Bi-directional arrows between the Web service and the management application indicate that the management system requests information or operations according to its data collection scheme.

Management in the IBM Web Services Toolkit

The IBM Web Services Toolkit is a technology package available on the IBM alphaWorks* Web site.[14]

It includes the core SOAP and WSDL facilities needed to deploy Web services, as well as support for creating applications that use Web services, security facilities, a UDDI registry, and other infrastructure technology. It also provides a JMX-based instrumentation feature that provides the automatic execution environment data collection capability called out in our earlier management principles discussion. In the toolkit package, cited above, this feature is described in the SOAP technology preview documentation. The Web Services Toolkit automatically collects data of two types: server-wide data and service-specific data. Data collected at these two levels include the following.

For the overall SOAP server:

- Total number of Web services deployed
- Total number of calls to all services combined

For each deployed Web service:

- Total number of successful invocations
- Number of invocations per method
- Number of failed invocations (the request was received by the server but resulted in an exception)
- Average response time for successful requests

The toolkit supports SOAP through the Apache Axis SOAP processor. It uses a message interception facility to support a generic management proxy. This proxy can be used to support any management approach required by a given implementation. In this case, the toolkit uses this proxy to create a JMX agent within which it automatically creates MBeans for deployed Web services. SOAP requests and responses are monitored, and the appropriate MBean is updated with incremental data. This is accomplished without examining the bodies of the SOAP messages. Since the Web Services Toolkit does not require any particular management system, it provides an extension to the Apache SOAP administration user interface that allows a user to view the collected data. This use is a prototype of how execution environment data collection could function. At this time, the management support in the execution environment does not provide support for operations or emit any notifications. The Web Services Toolkit does not provide the event collector, Web services management application, or management-oriented Web services registry that were discussed in this paper.

Concluding remarks

Web services can be managed using commonly used management systems and approaches. Choices for instrumentation and management system interaction have significant implications on the complexity of the Web service itself. Because simplicity of development is a central goal of the Web service model, we have proposed several principles and patterns that facilitate the inclusion of the appropriate level of management support, while minimizing the impact on the application. The principles of separating the management interface from the business interface, pushing core metric collection down to the Web services infrastructure and using intermediate Web services that act as event collectors, promote this two-pronged goal. The management patterns we described follow from these principles and allow the Web service developer to balance management requirements and complexity implications.

The principles and patterns we proposed can be implemented entirely at the application level, with the exception of one. Pushing metrics collection down to the infrastructure requires that such support be included by an infrastructure provider. We gave the example of the IBM Web Services Toolkit that implements execution environment data collection and provides transparent interaction with a JMX-enabled management system.

Cited references

1. K. Gottschalk, S. Graham, H. Kreger, and J. Snell, "Introduction to Web Services Architecture," *IBM Systems Journal* **41**, No. 2, 170–177 (this issue, 2002).
2. *Java Management Extensions Instrumentation and Agent Specification v1.0, Final Release, April 2000*, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303; available through http://java.sun.com/products/JavaManagement.
3. M. T. Rose and K. McCloghrie, *How to Manage Your Network Using SNMP: The Network Management Practicum*, Prentice Hall Inc., Englewood Cliffs, NJ (1995).
4. M. T. Rose and K. McCloghrie, *Structure and Identification of Management Information for TCP/IP-Based Internets*, STD 16, RFC 1155, Internet Engineering Task Force (May 1990), http://www.ietf.org/rfc/rfc1155.txt?number=1155.
5. J. Case, M. Fedor, M. Schoffstall, and J. Davin, *Simple Network Management Protocol*, STD 15, RFC 1157, Internet Engineering Task Force (May 1990), available through http://www.ietf.org/rfc/rfc1157.txt?number=1157.
6. M. Rose, *SNMP Multiplexing Protocol and MIB*, RFC 1227, Internet Engineering Task Force (May 1991), available through http://www.ietf.org/rfc/rfc1227.txt?number=1227.

7. H. Kreger, "Java Management Extensions for Application Management," *IBM Systems Journal* **40**, No. 1, 104–129 (2001).
8. Web Services Description Language (WSDL) 1.1 W3C Note, World Wide Web Consortium (March 2001), http://www.w3.org/TR/wsdl.
9. Universal Description, Discovery, and Integration, UDDI.org Consortium, http://www.uddi.org.
10. SOAP Version 1.2 Working Draft, World Wide Web Consortium (July 2001), http://www.w3.org/TR/2001/WD-soap12-20010709/.
11. H. Kreger, K. Gottschalk, and S. Graham, "Web Services Conceptual Architecture," http://www.ibm.com/webservices.
12. D. Ferguson, "Web Services Technical Architecture and Product Roadmap," http://www.ibm.com/webservices.
13. W. Bumpus, J. W. Sweitzer, P. Thompson, A. R. Westerinen, and R. C. Williams, *Common Information Model Implementing the Object Model for Enterprise Management*, Wiley Computer Publishing, John Wiley & Sons, Inc., New York (2000).
14. IBM alphaWorks, IBM Corporation, http://www.alphaworks.ibm.com.

Accepted for publication November 9, 2001.

Joel A. Farrell *IBM Software Group, One Charles Park, Cambridge, Massachusetts 02142 (electronic mail: joelf@us.ibm.com).* Mr. Farrell is a Senior Technical Staff Member in the Emerging Technologies group of the IBM Software Group. He joined IBM in 1981 in Endicott, New York, and has worked on large-scale operating systems, parallel and distributed computing, and formal methods in software development. Recently he has been involved in Internet technologies and is a member of IBM's core XML technology team. He is currently a member of the IBM Web Services architecture team. Mr. Farrell received his B.S. degree in computer science from Kansas State University in 1980 and his M.S. degree in 1985 from Syracuse University, also in computer science.

Heather Kreger *IBM Software Group, P.O. Box 12195, 3039 Cornwallis Road, Research Triangle Park, North Carolina 27709 (electronic mail: kreger@us.ibm.com).* Ms. Kreger is a senior architect in the Emerging Technologies area of the IBM Software Group. She represented IBM as a member of the Java Management eXtensions (JSR0009) Expert Group. Her years in lead positions in network management, combined with her experience on the IBM Web server and WebSphere Application Server products, gives her unique insight into the problems and solutions for managing applications and e-business. She has contributed to the specification, reference implementation, and compatibility test suites for the JMX Reference Implementation and authored "Java Management Extensions for application management" in Vol. 40, No. 1, 2001, of the *IBM Systems Journal*. She was also involved in management issues with other standards bodies, including: The Open Group Management Program, the DMTF (Distributed Management Task Force) Application Management Work Group Chair, and WBEM (Web-Based Enterprise Management) JSR (Java Specification Request) Expert Group. Ms. Kreger is currently the lead architect for Web Services in Emerging Technologies, and recently authored the document, "Web Services Conceptual Architecture." She chairs the IBM Web services architecture team, holds an associate position on the IBM AIM Architecture Board, and is the Specification Editor for JSR109: *Implementing Web Services in the Enterprise* being led by IBM.