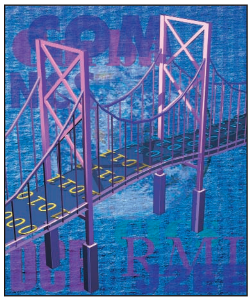
Web Services Interaction Models

Part 1: Current Practice



Steve Vinoski • *IONA Technologies* • *vinoski@ieee.org*

Each middleware approach has one or more interaction models associated with it that determine how applications built on top of the middleware interact with each other. Message-oriented middleware (MOM) applications interact rather simply, for example, by posting messages to and retrieving messages from queues. Object-oriented middleware applications such as those based on Corba or Enterprise Java Beans (EJB) interact by invoking methods on distributed objects. Because interaction models significantly influence the types of abstractions a middleware system makes available to applications, they figure prominently in determining the breadth and variety of application integration that the middleware supports.

As Web services evolve, they too will acquire standard interaction models; otherwise, their use will be limited to small-scale proprietary systems, rather than providing the standards-based "middleware for middleware" for uniting disparate islands of integration, as I outlined in my previous column.[1] At this point, however, the industry and standards bodies have yet to reach consensus on Web services interaction models. In this column, I explore some of the problems associated with a popular current approach to Web services interaction models.

Service-Oriented Architectures

Today, most Web services middleware is designed to let you wrap existing business logic and make it accessible as a Web service. Because they typically target systems implemented in Java classes, Java beans, or Corba objects, Web services toolkits help developers convert existing Java or Corba interface definitions into definitions written in the XML-based Web Services Description Language (WSDL),[2] a de facto standard the World Wide Web Consortium (W3C) is currently considering for standardization. These toolkits also provide components that enable distributed access to a Web service at runtime via SOAP, which is currently nearing completion as a W3C standard.[3] (Though the acronym originally stood for "simple object access protocol," the W3C decided in 2001 to just stick with "SOAP," which was a good move given that the protocol is neither simple nor object-oriented.) Some toolkits also supply an implementation of a Web services directory or registry service, typically based on Universal Description, Discovery, and Integration (UDDI, www.uddi.org), that enable Web services to register themselves so enable applications can find their WSDL definitions and interact with them.

In terms of interaction models, today's Web services middleware generally supports *service-oriented architectures* like the one in Figure 1 (next page). While much has been made of them, the concept is actually quite trivial: A service with a well-defined interface and data interchange characteristics advertises itself in a distributed directory service where applications can look to find the details for interacting with the service.

Service-oriented architectures are defined, in part, by a three-step interaction model, but that is only part of the story. Middleware systems such as Corba, EJB, the component object model (COM), distributed computing environment (DCE), and Web services each support their own specific interaction steps within the general interaction model, especially in step 3. For example, *session-oriented object systems* typically require the following interactions:[4]

1. A factory object advertises its object reference in the directory service.

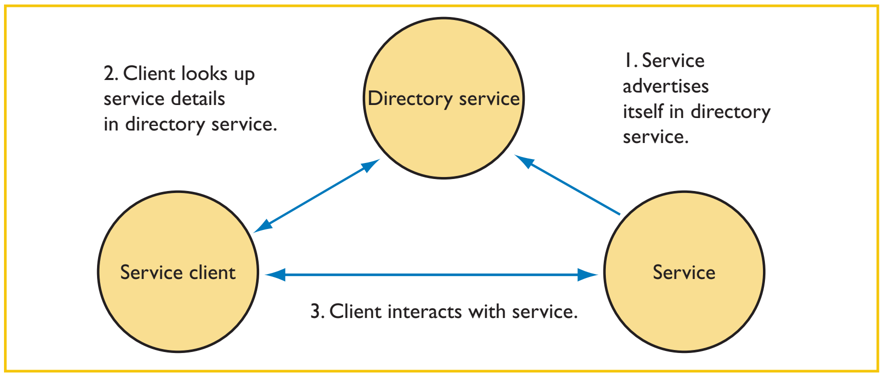

2. Client looks up service details in directory service.
Directory service
1. Service advertises itself in directory service.
Service client
Service
3. Client interacts with service.

Figure 1. Service-oriented architecture. Services advertise themselves in the registry, allowing clients to query the registry for service details and interact with the service using those details.

2. The client then must look up the reference to the factory object in the directory service.
3a. The client then asks the factory object to create a service object instance. The factory creates the service object and returns an object reference for it to the client.
3b. The client directly uses the service object created by the factory.
3c. The client destroys or releases the service object when finished with it, depending on the underlying middleware's object-lifetime semantics.

Similarly, a service-oriented architecture based on SOAP, WSDL, and UDDI requires the following interactions:

- A Web service advertises its WSDL definition into a UDDI registry.
- The client looks up the service's definition in the registry.
- The client uses information from the WSDL definition to send messages or requests directly to the service via SOAP.

Given the similarities between the interactions required for the Web service system and those for the object system, we might assume that if the middleware hosting the business logic supports service-oriented architectures, wrapping that business logic as a Web service would be trivial because Web services also supports service-oriented architectures. Unfortunately, we can't ignore the underlying middleware-specific interaction models, so it's not that simple.

Web Services Wrapper Interactions

Let's assume we want to wrap a session-oriented object as a Web service. Using our Web services toolkit, we first translate the interface specification for the session object into WSDL, and then create an entry in the UDDI registry to advertise the WSDL specification. Web services clients can then look up the WSDL in the registry and interact with the object as a Web service.

This example seems almost too good to be true, and in fact, it is. It fails to address several significant issues, including those involving object life cycle, object references, and fault handling.

Object Life Cycle

We implemented the Web service using a session-oriented object from an underlying middleware system. In the object system, a client application first interacts with the factory object to create the session object, which the client must later release or destroy. The objects' life cycles are controlled explicitly through client interactions, but we have not accounted for handling them in the Web services system.

One way to address this problem might be to expose the factory object as a Web service. The Web services system's interaction model would then precisely parallel that of the underlying object system, which should let Web service clients control life cycle issues exactly like their object system counterparts.

Unfortunately, exposing the factory object as a Web service means that every object in the underlying object system must be exposed just to make the Web services system work. Not only does the Web services interaction model parallel that of the underlying system, but the design and topology of the Web services system also precisely parallels the underlying system. In other words, there is a one-to-one correspondence between Web services and the underlying objects.

Also, exposing all the underlying objects as Web services makes it difficult to hide the details of the objects' communication protocols from the Web service clients. In a Corba system, for example, invoking a request on an object using the standard Internet inter-ORB protocol (IIOP) might result in a *location forward* result — meaning that the target object for the request actually resides at a different location — that returns the object's actual location. The object request broker (ORB) middleware under the client application normally handles location forwarding transparently, completely shielding the client application from the details of the forwarding protocol. Because we cannot assume that all clients in a Web services system are built over an ORB, however, direct exposure of Corba objects as Web services requires intermediaries or gateways to take the place of the ORB for shielding those clients from protocol details such as location forwarding. Intermediaries and gateways are generally undesirable because of their potential as performance bottlenecks, single points of failure, and administrative burdens.

Object References

Exposing the factory object as a Web service appears to solve the life cycle problem, but it introduces another: When a client invokes the factory in the object system, the factory returns a reference for a newly created object. Representing this interaction at the Web services level requires the equivalent of a Web services object reference.

Web services do not have object references, but they do have universal resource identifiers (URIs). We could thus represent every object exposed as a Web service using a URI. Unfortunately, this means that somewhere — most likely in the intermediaries or gateways mentioned above — the system must keep track of the mappings between the Web services' URIs and their corresponding references in the underlying object system. (Although some object systems support URI forms for object references, those URIs still require middleware specific to the object system to interpret them, and we can't assume that all Web services clients will be based on such middleware.) Creating and maintaining these mappings can be difficult and error prone, especially given that object references can be embedded deeply into other data structures. Intermediaries and gateways are therefore required to comb through all data passed between the Web services system and the underlying object system to find all URIs and convert them to object references.

Another issue that URIs do not resolve with respect to object references is that in the session-oriented object system approach, the objects typically maintain state on behalf of the client that creates them. For reasons of scalability, performance, and correctness, exposing stateful objects directly as Web services is a questionable practice. Creating a stateless object that wraps the entire session-oriented interaction model and exposing that object as a Web service would almost certainly be a better approach, although it would still suffer from some of the same problems.

Faults and Exceptions

Distributed object systems typically use exceptions to signal error conditions. A client attempting an operation on a target object might get an exception because of an error in business logic processing or an error in the distributed system — from a temporary network breakdown, for example. In the latter case, exceptions often inform the client to retry the operation.

Like object references, exceptions must also be converted as they pass into the Web services system. Application-specific exceptions are usually easy to convert, but converting exceptions that are specific to the underlying object system can be difficult because they might not make any sense in the Web services system.

Failure to Abstract

This example shows that what initially seemed to be a simple exposure of a distributed object as a Web service required us to expose the whole distributed object system as a Web service system. The resulting interaction models at the Web services level necessarily became identical to those of the underlying distributed object system. Most fundamentally, we failed to properly abstract the object system when exposing it as a Web service, instead letting details of the object system's interaction model — implementation details, in this case — become visible in the Web services layer.

Unfortunately, this example is not contrived. Some Web services products on the market today promote developing Web services systems in ways that will result in the types of problems I've described here. While building systems this way is fine for gaining experience with Web services technologies, production systems require different approaches to achieve the desired degree of integration, and thus, a reasonable return on investment. For reasons I'll cover next issue, successful Web services integration systems typically operate at a level of abstraction that involves business process flow and business documents, rather than at the lower level of function-oriented components and type systems related to programming languages.

Conclusion

Our initial goal was to expose the services of a single object, but by failing to properly abstract the service offered by the underlying objects, we ended up creating a Web services copy of the underlying session-oriented object system. The same types of problems will arise when attempting to directly expose Java classes, EJBs, or any underlying function-oriented middleware as Web services.

Other problems that will arise include semantic mismatches, data type mappings between systems, service granularity issues, and state management. Unfortunately, rediscovering all these problems seems inevitable whenever a higher-level approach comes along. When Corba was first introduced in 1991, for example, many wanted to know how to expose their C++ objects as Corba objects. The answer was simple: "Don't do that." Attempting to masquerade software assets from a lower level of abstraction directly at a higher level can often cause significant mismatch and exposure problems, as I have described here.

I will continue this discussion next issue by investigating some promising alternatives to Web services interaction models. As always, if you have comments on this or any of my columns, please e-mail me.

References

1. S. Vinoski, "Where is Middleware?" *IEEE Internet Computing*, March/April 2002, vol. 6, no. 2, pp. 83-85.
2. Web Services Description Language (WSDL) 1.1. W3C Note, 15 March 2001; available at www.w3.org/TR/wsdl.
3. XML Protocol working group homepage, www.w3.org/2000/xp/Group/
4. M. Henning and S. Vinoski, *Advanced Corba Programming with C++*, Addison Wesley Longman, Reading, Mass., 1999.

Steve Vinoski is vice president of platform technologies and chief architect for IONA Technologies. He is coauthor of *Advanced CORBA Programming with C++* (Addison Wesley Longman, 1999). Vinoski currently serves as IONA's representative to the W3C's Web Services Architecture working group.