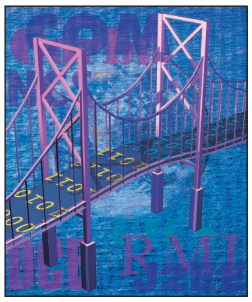
Integration with Web Services



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

Invariably, there's a difference between what we'd like our enterprise computing systems to be and what they really are. We like to envision them as orderly multitier arrangements comprising software buses, hubs, gateways, and adapters — all deployed at just the right places to maximize scale, load, application utility, and ultimately, business value. Unfortunately, we know that there's a wide gulf between this idealistic vision and reality. In practice, our enterprise computing systems typically are tangles of numerous technologies, protocols, and applications, often hastily hard-wired together with inflexible point-to-point connections.

Isn't middleware supposed to fix all this? After all, the whole point of middleware is to hide the diversity and complexity of the computing machinery underneath it. By adopting the abstractions that middleware provides, we're supposedly isolating our applications from the variety of ever-changing hardware platforms, operating systems, networks, protocols, and transports that make up our enterprise computing systems.

Unfortunately, middleware's success and proliferation has recreated — at a higher level — the very problem it was designed to address. Rather than having to deal with multiple different operating systems, today's distributed-application developers face multiple middleware approaches. Indeed, middleware does provide the promised abstractions, but different approaches provide different types of abstractions. For example, those found in message-queuing systems are quite different from those in distributed object systems. The differences between the various abstractions make it difficult for applications to access different middleware-based services simultaneously.

Middleware for Middleware

Working across multiple middleware systems has become especially debilitating over the past two or three years as the weak economic climate has forced numerous companies to restructure, consolidate, and downsize. They've had to reduce IT spending and get the most out of existing IT assets — in part, by integrating software systems they never imagined would need to work together. Without the ability for applications to simultaneously access multiple middleware systems, bridging such "middleware islands" is difficult and expensive.

To bridge multiple middleware systems, today's integration applications require abstractions for the underlying middleware itself. Because this is software, the answer naturally requires some kind of wrapping or additional level of indirection, but the trick is finding a suitable technology to provide it. In some cases, you might extend one of your enterprise middleware systems to wrap the others and provide the desired abstraction uniformity. But in many cases, this isn't possible — sometimes due to technical reasons, but just as often due to cost, lack of expertise, or even company politics. Another approach is to build your own custom middleware, but that can be expensive in terms of development, maintenance, and even lost opportunity should some new revolutionary technology come along.

For several reasons, we can best use Web services to provide this "middleware for middleware"[1] abstraction layer for modern integration applications. To begin with, they're relatively lightweight, avoiding the intrusive object models and single programming language requirements that other middleware systems force on us. Developers using "middleware dark matter"[2] languages, such as Perl and Python, generally find Web services easy to work with. Moreover, ongoing Web services standardization efforts free them from the proprietary stigma of enterprise application integration (EAI) systems.

Web Services Integration Framework

One project that's implementing the middleware-

for-middleware view of Web services is the Apache Web Services Invocation Framework (WSIF; http://ws.apache.org/wsif/). Over the past few years, most Web services development toolkits targeted the SOAP level, providing APIs that let applications create and receive SOAP messages. While this provides a certain useful level of abstraction, it's not enough.

SOAP APIs are not standardized, so moving from one to another requires changes in your application. More importantly, SOAP APIs don't hide underlying transport details from applications. This might not be a problem if all Web services used only HTTP, but they don't. Developers want to address Web services over a variety of protocols and transports, including proprietary messaging systems and standard distributed object protocols; some even want to use them as local programming-language objects. In fact, a group of engineers at Sun recently began working to define additional protocols in binary — not XML — to allow for "fast Web services."[3]

WSIF's intent is to supply a Java programming API that hides access details for multiprotocol Web services. Rather than focusing on SOAP, WSIF focuses on the Web Services Definition Language.[4] A WSDL definition has two parts:

- A *logical* part, called the port type or interface, defines the protocol-independent Web service types and input and output messages.
- A *physical* part defines the protocol-specific bindings you use to access the Web service. A binding for a given protocol specifies the details required to let applications exchange messages with a given service instance using that protocol.

Separating a service abstraction from its bindings is a key WSIF building block. Protocol-specific *providers* that handle the details of how messages are sent and received over a specific protocol can thus fulfill an application's invocations on the protocol-independent service abstraction. Under WSIF, SOAP is just one possible provider; others can access services implemented using Enterprise Java Beans, the Java Message Service, the J2EE Connector Architecture, and local Java objects. This approach lets services share interface definitions and still allow accessibility via multiple bindings, which is precisely what's needed to allow an application to have practical simultaneous access to multiple middleware systems. Or is it?

In addition to client applications with multimiddleware access, there's a need for server-oriented multimiddleware applications to serve as multimiddleware entry points to valuable back-end business services.

Middleware Switching

WSIF definitely is a step in the right direction, but it's not quite enough. One of the biggest drawbacks is that it's a Java-only solution. Given that a significant percentage of the world's middleware systems use C and C++, a Java-only solution is fairly limiting. Worse, it means that only those protocols readily and practically accessible in Java are available to your application unless you're willing to implement your own providers, which is probably contrary to the reasons you'd want to use WSIF in the first place.

In addition to client applications with multimiddleware access, there's a need for server-oriented multimiddleware applications to serve as multimiddleware entry points to valuable back-end business services. For example, an article in the *Boston Globe* discussed how the e-commerce push in recent years was not as successful as hoped because many companies made their internal brick-and-mortar and Internet sales groups compete against each other.[5] One reason for this was that many companies couldn't envision the new Internet sales divisions using and accessing the same dreary old (often mainframe-based) business systems that the traditional sales channels used. The article also pointed to Sears as a success story in this area because it "ripped open" its store-oriented computing systems to accommodate the Internet sales channel, letting traditional and new sales groups use the same enterprise computing systems. Often, companies are afraid to do what Sears did because they can't figure out how to address the multimiddleware problem of leaving existing services' protocols and message formats intact while letting new clients access the same systems via other protocols and message formats.

The typical EAI approach to integration is to use adapters to convert all traffic to canonical formats and protocols. To allow access to a given service, you simply write an adapter that converts between the EAI system's canonical protocol and format and those of the service. This avoids the N^2 protocol/format conversion problem and makes building and maintaining adapters easier. However, this approach penalizes the runtime performance of every transaction passing through the users' computing systems by requiring conversions whether they're needed or not. Furthermore, canonical formats and protocols usually change over time, so they don't really solve the N^2 conversion problem in the long run.

To ensure high performance, a multimiddleware router must allow

for service abstraction while avoiding conversions whenever possible. As in WSIF, WSDL can provide both the service abstraction definitions and the binding definitions for such a router. When a caller sends a message in a given format through the router to an abstracted service that happens to expect that same format, there's no point in having the router first convert the incoming message to a canonical format, and then convert the outgoing message back to the original format. Similarly, when the sender and receiver use the same protocol, the router wastes cycles by performing unnecessary conversions in the middle. Instead, the router must recognize that the sender and receiver are speaking the same protocol, format, or both, and stay out of the way wherever possible.

A multimiddleware router must support message routing and communication pattern bridging at varying levels of service. For example, it should direct messages by simply connecting an incoming port to an outgoing port, by performing content-based routing based on message-header attributes, or by routing single incoming messages to multiple destinations. It also must bridge different communications models used by the underlying middleware. For example, it might have to transparently bridge a synchronous Corba client into a pseudo-synchronous messaging system that uses one queue for requests and another queue for replies.

Carried to its logical conclusion, such a router would appear to be a magical universal translator, capable of converting any incoming protocol/format pair into any other, all at the highest possible efficiency level. Needless to say, this is impossible because not all protocols, formats, and service semantics can be seamlessly translated into others. For example, bridging a synchronous client that expects a single response to a broadcast message expecting multiple voting responses makes little sense. The hard part for users is deciding how best to abstract their existing services into WSDL definitions suitable for use with such a router, and to ensure that their middleware supplier supports a router with the necessary bindings. In any case, the middleware supplier should handle the difficult parts, rather than taking the easy way out with hard-wired canonical formats and passing the resulting inefficiencies off to their users.

Of course, no matter how efficient routers might be, they automatically introduce overhead by creating a network hop. The router works well when the client application and service application already exist, such that the router can use the WSDL contract created for the service to route messages between them. This avoids the need to rebuild and redeploy either application. However, for new applications, it's better to build in a multimiddleware router. Of course, that is exactly what WSIF does for outgoing messages from clients, but it's also needed for incoming messages for services.

Equipping new applications with multimiddleware switching capabilities requires linking a multimiddleware router into the application underneath a WSIF-like API. The API abstracts away the underlying middleware and protects the application from the details of the multimiddleware capabilities beneath it, even when such capabilities dynamically load into a running process. Multiprotocol and multiformat applications are nothing new — we've built them before, using a variety of technologies and approaches. (This has been the main focus of my work for the past decade, for example.) With the advent and popularity of WSDL, combined with the growing need for multimiddleware integration, such systems are poised to become the norm rather than the exception.

Conclusion

When applied correctly, Web services can effectively to solve the multimiddleware problem. Much work remains, but WSIF is garnering attention, and IONA's Artix (see www.iona.com/products/middlewareint.htm) already supports the routing and switching capabilities described above. As Werner Vogels' article on p. 59 in this issue shows, many misconceptions unfortunately remain about Web services.[6] The technical press has overhyped them to the point at which they cannot possibly deliver on all the promises made about them, and standards bodies continue to fight over just what they are and who gets to define them.

Despite all this, Web services are proving in practice to be the key to providing interconnections in our multiple middleware enterprise computing systems. In some circles, such as the Corba technical community, Web services are nevertheless still viewed with disdain, being seen as "Corba done wrong" — a technically poor reinvention of a wheel already working correctly. Personally, however, I view Web services as "EAI done right."

References

1. S. Vinoski, "Where is Middleware?" *IEEE Internet Computing*, vol. 6, no. 2, 2002, pp. 83–85.
2. S. Vinoski, "Middleware 'Dark Matter'," *IEEE Internet Computing*, vol. 6, no. 5, 2002, pp. 92–95.
3. P. Sandoz et al., "Fast Web Services," technical report; http://developer.java.sun.com/developer/technicalArticles/WebServices/fastWS/.
4. *Web Services Description Language (WSDL) Version 1.2*, W3C working draft, 11 June, 2003; www.w3.org/TR/wsdl12/.
5. R. Weisman, "Online, Off Target: Retailers Must Integrate Sales," *Boston Globe*, 14 Sept. 2003, p. C2.
6. W. Vogels, "Web Services Are Not Distributed Objects," *IEEE Internet Computing*, vol. 7, no. 6, 2003, pp. 59-66.

Steve Vinoski is chief engineer of product innovation for IONA Technologies. He's been involved in middleware for 15 years. Vinoski is the coauthor of *Advanced Corba Programming with C++* (Addison Wesley Longman, 1999), and he has helped develop middleware standards for the OMG and W3C. Contact him at vinoski@ieee.org.