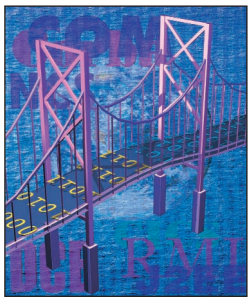

Putting the "Web" into Web Services

Web Services Interaction Models, Part 2

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

As I discussed in my previous column, each different style of middleware promotes one or more interaction models that determine how applications based on that middleware communicate and work with each other.[1] In the relatively simple publish-subscribe model, for example, a source publishes information, perhaps by posting it to a message queue, and subscribers interested in that information either retrieve it from the queue or receive it automatically from a broker. Complex interaction models, on the other hand, often arise in distributed object systems, in which stateful objects rely on clients to properly manipulate their state through sequences of method invocations.

Web services are still evolving, and so the industry is far from reaching consensus on what interaction models best suit them. Some vendors are attempting to push existing middleware system models, such as Corba and Enterprise JavaBeans (EJB), directly into Web services, but as I described last issue, this approach is fraught with problems. One of the worst is that it causes inappropriate details (about protocols, type systems, interaction models, and so on) to show through the Web services level from the underlying systems, destroying the service encapsulation and isolation that Web services are supposed to provide.

Still, it is difficult to say what the best interaction models would be for Web services — mainly because the World Wide Web Consortium (W3C) is still developing the architecture.[2] Architectural style profoundly influences interaction models because the elements of a given middleware's architecture ultimately determine what developers can use it for. Several high-level interaction models are possible given today's Web services architecture and capabilities, but there is one architectural style that is gaining popularity for use with Web services because of its affinity with the Web.

Remote Procedure Calls

"When all you have is a hammer, everything looks like a nail." This particular bit of wisdom can unfortunately be applied to numerous cases within our industry — now including RPC for Web services. Many people involved in Web services today have only the RPC hammer in their toolboxes, and they seem unable or unwilling to consider other possibilities.

It might seem obvious to apply RPC approaches to using Web services to integrate existing business logic implemented in RPC- or method-based middleware with another flavor of middleware. In practice, however, higher-level integration efforts typically follow different patterns than the lower levels they're integrating. RPC-based approaches usually result in a tight coupling between caller and service, which might be right at a low level but completely wrong for an upper-level integration.

A high-level integration might supply a service for handling purchase order documents, for example, making use of a credit card authorization service at a lower level. A developer would naturally use RPC to design and implement the authorization service because there is no choice after invoking the service but to wait for a reply before continuing with order processing. Given that purchases can sometimes take days or even weeks to process, using RPC to design and implement the overall purchasing service would be questionable, because RPCs do not support such long-running activities very well.

In general, Web services are best for high-level integration in which performance issues are not critical. Performance is almost always a factor in efforts that use proprietary messaging middleware, J2EE, or Corba, and implementers have put countless person-years of effort into meeting these demands and making such systems as fast as possible. Such systems are thus effective for tight integrations and

 1089-7801/02/$17.00

performance-critical applications — situations in which Web services simply cannot compete. In other words, there is no point reinventing these successful middleware wheels with Web services just because it's the latest technology fad. Rather, Web services are best for integrations that combine several high-performance applications, written using these already successful approaches, into new or improved coarse-grained business processes.

Web Services and Messaging

Messaging-based systems are generally more loosely coupled than RPC-based systems. Given the integration levels for which Web services are best suited, messaging fits well into the Web services picture.

Considering the example application again, we see that the overall purchasing system should be message-based: The buyer would initiate a purchase by sending a message containing a purchase order document. As the service processed the purchase order, it could send status messages back to the buyer for tracking purposes. Once it completed the order, the service could again notify the buyer with an "order completed" message containing shipping details. RPC would be inappropriate for the purchasing system because it would couple the buyer and service too tightly, both in terms of technology (you'd need the same RPC-oriented or method-oriented middleware on both ends) and in terms of interaction (if either side became unavailable, the other side could be unnecessarily blocked from further processing).

Because they are document-oriented, messaging systems fit well with Web services, which manipulate XML documents.[3] Messaging-based systems essentially let data travel from one service to another, allowing each to process the data as necessary without tightly coupling the services. The fact that each service generally expects data input in a particular format to be able to produce the desired output suggests that upstream services must produce the exact data expected downstream. This is not necessary in practice, however, because data can be transformed between services into the appropriate format — especially with XML data, which is easy to transform via tools such as XML Style Language Transformations (XSLT).

Given that numerous messaging solutions already exist, it is fair to ask whether Web services just reinvent the messaging wheel. Fortunately, there are real differences between existing systems and Web services. One significant example is that Web services are being defined through open standards. Existing messaging systems tend to be proprietary, requiring that sender and receiver both install the same middleware from the same vendor to properly communicate. This approach inhibits scalability and ignores the reality that enterprise networks are inherently heterogeneous, especially at the middleware level. Basing Web services on open standards greatly reduces such issues by allowing for interoperability between different suppliers' implementations.

Using Web services for higher-level integration works naturally with messaging. Not only does a high-level Web service integrate and encapsulate lower-level, more tightly coupled technologies into coarse-grained business processes, but it does so while providing loosely coupled access to the overall encapsulated service. Such a Web service might take in an XML document from a message, for example, and extract data fields from the document to feed into the underlying services it encapsulates. Alternatively, it might produce a new XML document by invoking the lower-level services, using their outputs to populate fields in the document. Web services like these serve as orchestrators or aggregators because they combine multiple lower-level services into single higher-level offerings at new levels of abstraction. In the process, they eliminate the impedance mismatch between business and technology layers, and between loosely and tightly coupled layers.

Interface Complexity

Messaging systems typically provide simple fixed interfaces based on queuing or hub-and-spoke abstractions, whereas RPC-oriented systems (such as distributed object systems) usually exhibit highly complex and varied interfaces. This complexity is not accidental: Reflecting their evolution from statically typed programming language systems, RPC-based systems explicitly promote the development of specific and separate interfaces for each object or service. Interfaces developed for such systems tend to be fine-grained, in part to capture and publish the interfaces' associated semantics, and in part to enhance performance and flexibility.

Without Rest, the name "Web services" is a misnomer, as tunneling Soap through HTTP is hardly Web-friendly.

An interface establishes the "protocol" used to communicate with the object or service that implements it. In this context, I do not mean network protocol, but rather the "conversation" or sequence of method invocations the caller must perform to properly interact with the object or service. Each different interface effectively establishes a protocol with its own nuances and semantics.

As a system scales, it becomes increasingly difficult to keep track of each interface's semantics. As I described last time, that gives rise to the need for the discovery portion of the service-oriented architectures. Unfortunately, because discovery services cannot convey semantics, they can't teach a random application what it needs to know — the protocol it must understand — to properly communicate and interact with a service that it

initially knows nothing about.

As the number of different interfaces in a system rises, the number of possible interface protocol interactions increases by the square of the number of interfaces deployed: $O(n^2)$. Interface versioning that occurs as a side effect of system evolution also contributes to the overall complexity. Many (including me) believe that this complexity factor has been the primary inhibitor for expanding interface-oriented systems such as Corba and EJB to an Internet scale. If Web services systems are to scale beyond the limits of traditional middleware, they need to avoid both the interface interaction complexity problems of RPC-oriented systems, and the proprietary nature of messaging-based systems.

Web Services at Rest

Fortunately, we already know how to avoid the interface complexity problem — using the same solution that allows you to interact with any Web site by providing a browser with the HTTP URI, even though you've never visited the site before. It's essentially what makes the World Wide Web work.

Though it is often mistaken for a transport protocol, HTTP is really an application protocol.[4] Those seeking to exploit HTTP's ubiquity — to transport Soap over it, for instance — tend to focus on its ability to transport other protocols, an act commonly called "tunneling." In reality, the fact that HTTP is an application protocol means that it is much more than just a transporter of bytes.

HTTP provides application-level semantics via its "verbs:" `GET`, `POST`, `PUT`, and `DELETE`. These verbs form a generic application interface that can be applied in practice to a broad range of distributed applications despite the fact that it was originally designed for hypermedia systems. Because of HTTP's application-level semantics, tunneling other protocols over or through it is an abuse of the protocol if the tunneling does not respect those semantics. For example, failure to conform to HTTP application semantics can break intermediaries that serve as proxies or caches, as they usually perform their functions based on the standard semantics associated with the HTTP verbs in the messages flowing through them.

In his doctoral dissertation, Roy Fielding coined the term "representational state transfer" (Rest) to describe the Web's architectural style.[5] Rest uses standardized interfaces to promote stateless interactions by transferring representations of resources, rather than operating directly on those resources. There is far more to Rest than I can describe here, so please refer to Roy's dissertation and the RestWiki site (internet.conveyor.com/RESTwiki/moin.cgi/FrontPage) for more details, and see Paul Prescod's convincing case study that applies Rest principles to improve the Google search engine Web service.[6]

As it relates to this column, however, the key point about Rest is that its interaction models — the very ones that govern traditional Web interactions — are wholly suitable for Web services as well. Web services are identifiable via URIs, and regardless of the wide variety of abstractions they might collectively represent, they can all be implemented using the same generic interface that HTTP's verbs provide. Without Rest, the name "Web services" is a misnomer, as tunneling Soap through HTTP is hardly Web-friendly. ("Internet services" might be a more accurate, though less marketing-friendly, name for such a tunneling system.)

Given that one of W3C's goals is to create standards that are consistent with the existing Web architecture, it is not surprising that Rest is garnering attention in the Web services standardization efforts. Until recently, the Soap specification required all messages sent via the Soap-over-HTTP binding to be tunneled through HTTP `POST`, regardless of whether those messages obeyed `POST` semantics. As of this writing, however, the W3C XML protocol working group is looking at supporting HTTP `GET` in Soap 1.2 to make Soap fit better with the Web architecture. The relationship between Rest and Web services is also under discussion within W3C's technical architecture group (TAG) and Web services architecture working group. Rest is clearly influencing the Web services architecture, and it stands to play an important role in determining the interaction models the industry will eventually adopt for Web services.

Add Rest to Your Toolbox

Because I am a longtime RPC advocate, associates are often surprised when I promote Rest. From what I've learned about it to date, I know that it neatly solves some of the complexity and scaling issues faced by RPC-based distributed systems. I suggest those with backgrounds similar to mine take the time to seriously study Rest and learn how it works. The fact that the Web — the largest and most successful distributed system in existence — is built on Rest principles should be enough to convince even the most stalwart advocates of other interaction models that Rest holds significant promise for putting the "Web" back into Web services, and aligning their interaction models with existing Web architecture.

References

1. S. Vinoski, "Web Services Interaction Models — Part 1: Current Practice," *IEEE Internet Computing*, vol. 6, no. 3, May/June 2002, pp. 89-91.
2. D. Austin, A. Barbir, and S. Garg, "Web Services Architecture Requirements," W3C working draft, Apr. 2002.
3. E. Newcomer, *Understanding Web services: XML, WSDL, SOAP, and UDDI*, Addison-Wesley, Boston, Mass., May 2002.
4. R. Fielding et al., "Hypertext Transfer Protocol — HTTP/1.1," Internet Engineering Task Force RFC 2616, June 1999; available at www.ietf.org/rfc/rfc2616.txt.
5. R.T. Fielding, *Architectural Styles and the Design of Network-Based Software Architectures*, doctoral dissertation, Dept. of Computer Science, Univ. of California, Irvine, 2000.
6. P. Prescod, "Google's Gaffe," O'Reilly & Assoc., Apr. 2002; available at www.xml.com/pub/a/2002/04/24/google.html.

Steve Vinoski is vice president of platform technologies and chief architect for IONA Technologies. He currently serves as IONA's representative to the W3C Web Services Architecture working group.