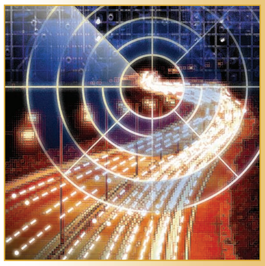


Web Services Are Not Distributed Objects

The hype surrounding Web services has generated many common misconceptions about the fundamentals of this emerging technology.

Werner Vogels
Cornell University

Web services are frequently described as the latest incarnation of distributed object technology. This misconception, perpetuated by people from both industry and academia, seriously limits broader acceptance of the true Web services architecture. Although the architects of many distributed and Internet systems have been vocal about the differences between Web services and distributed objects, dispelling the myth that they are closely related appears difficult.

Many believe that Web services is a distributed systems technology that relies on some form of distributed object technology. Unfortunately, this is not the only common misconception about Web services. In this article, I seek to clarify several widely held beliefs about the technology that are partially or completely wrong.

Fundamental Errors

At the International World Wide Web Conference in May 2003, a smart and gifted Internet architect I will call Peter asked me, "Don't you think Web services will fail like all the other wide-area distributed object technologies that people have tried to build?"

I was baffled. How could someone like Peter still view Web services as distributed object technology? Yet, he is not alone in his stubbornness: many developers, architects, managers, and academics still see Web services as the next episode in a saga that includes Corba, DCOM, and remote method invocation (RMI).

Web services are distributed systems technologies, but that is where the common ground ends. The only possible relation is that Web services are now sometimes deployed in areas where distributed object applications have failed in the past. Within the distributed technology world, it is probably more appropriate to associate Web services with messaging technologies because they share a common architectural view, although they address different application types.

Given that Web services are based on XML documents and document exchange, we could say their technological underpinning is document-oriented computing. However, exchanging documents is very different from requesting an object's instantiation, requesting a method's invocation on the basis of the specific object instance, receiving that invocation's result in a response, and releasing the object instance after several such exchanges.

I frequently encounter about a dozen other statements that fall into the same basic category. I hear people say, for example, that "Web services are just remote procedure calls for the Internet," or "You need HTTP to make Web services work." Before addressing several of the more common misconceptions, we should define a Web service in its purest form in order to begin with a clear model.

Minimalist Web Services Model

I believe much of the confusion surrounding Web services comes from press and vendor hype, which lacks the technical depth to make clear the foundational concepts. Of course, the political bickering among standards bodies such as the W3C (www.w3.org), OASIS (www.oasis-open.org), and the Web Services Interoperability Organization (www.ws-i.org) doesn't help clarify the simple, interoperable nature of Web services. One of the key architects in the W3C's Web Services Architecture working group stated quite bluntly that they did not have the luxury of describing Web services in a simple manner because none of the participating vendors could agree on a single definition.

To understand the misconceptions, we must first cut through the hype. If we get back to the core, we see that Web services comprise three components:

- The *service* is software that can process an XML document it receives through some combination of transport and application protocols. We don't care how this component is constructed, whether object-oriented techniques are used, or if it operates as a stand-alone process, as part of a Web or application server, or as a thin front end for a massive enterprise application. The only requirement is that the service be able to process certain well-defined XML documents.
- The *XML document* is the Web service's keystone because it contains all the application-specific information that a service consumer sends to the service for processing. The documents a Web service can process are described using an XML schema; two processes engaged in a Web services conversation must have access to the same description to ensure that they can validate and interpret the documents they exchange. This information is commonly described using the Web Services Description Language (WSDL).
- The *address*, also called a port reference, is a protocol binding combined with a network address that a requester can use to access the service. This reference basically identifies where the service can be found using a particular protocol (for example, TCP or HTTP).

In principle, these three components are enough to build a Web service; in practice, however, there is a fourth component: the *envelope*. It could be considered optional, but it provides an extremely useful framework for managing the message exchange. The envelope is a message-encapsulation protocol that ensures that the XML document to be processed is clearly separated from other information the two communicating processes might want to exchange. For example, an intermediary could use the envelope to add routing and security information to the message without modifying the XML document.

The protocol used for almost all Web services is SOAP, which originally stood for "Simple Object Access Protocol." This naming was a mistake, however, because the protocol has nothing to do with accessing objects. Since the SOAP 1.2 specification's completion,[1] Internet architects have used the protocol without expanding the acronym. The SOAP message itself, also called the *soap envelope*, is XML and consists of two possible elements:

- The *soap header* holds all system information.
- The *soap body* contains the XML document that the Web service is to process.

Web service protocols and extensions use the header element to carry their protocol-specific information — a security signature, for example — while the body element remains untouched.

Whether you use your text editor to construct a SOAP message to send in an email or use an automatically generated proxy client from within your favorite programming language, the three core components are all it takes to make a Web services interaction work.

This minimalist model represents only the basic core of Web services technology. I do not include service-description techniques, service registration, or other technologies that are necessary to make Web services work in more complex settings. The simple framework presented here lets us examine several common misconceptions from a more principled view. For a more extensive look at Web services, see the Web Service Architecture draft from the W3C Web Services Architecture working group (www.w3.org/2002/ws/arch/).

Web Services Really Are Simple

At its core, Web services technology is quite simple; it is designed to move XML documents between service processes using standard Internet protocols. This simplicity helps Web services achieve the primary goal of interoperability.

The simplicity also means that we must add other technologies to build complex distributed applications. Over time, we will see that the issues vendors are now bickering about — reliability, transactions, asynchronous processing, and so on — will become reality in an interoperable manner. The process surrounding security extensions, for example, gives us reasonable hope that vendors can reach agreement on a set of interoperable primitives. For the other areas, such as asynchronous messaging, we can expect that either market forces will drive the selection of one particular proposal or that vendors will work out their differences in a standards forum. The goal of Web services is to provide an interoperable platform, so the market will not tolerate proprietary solutions.

On the other hand, the process surrounding reliable messaging has many distributed-system specialists scared that vendors' irrational behavior will produce major obstacles on the road to interoperability. The greatest threat to Web services' large-scale success will be vendor politics. In an attempt to preempt the reliable messaging specification's release by IBM, Microsoft, BEA, and Tibco,[2] a consortium led by Sun Microsystems and Oracle recently published a reliable messaging specification[3] that was little more than a cut-and-paste effort from the reliability section of ebXML. This specification was clearly released too early as a result of political pressures. The document does a disservice to the community; it is ambiguous in many places, incomplete in others, and riddled with errors throughout. Indeed, any company that implemented it would end up with an unreliable system.

Common Misconceptions

The distributed computing world of Web services centers around XML document design. Protocols and addresses are necessary only as glue — support technologies for getting the document to the right place in the right manner. In contrast to the simplicity of basic Web services, these XML documents can be extremely rich and complex. For example, a Web services system I worked on for the US Air Force publishes flight plans that can easily be a megabyte or more in size. Encoding these rich documents in XML ensures that they are extensible at predefined places without disrupting existing document consumers.

The lack of understanding that Web services primarily support the document-exchange contract is one of the root causes for many of the misconceptions about them. These document-centric principles can help dispel the most common misconceptions.

The greatest threat to Web services' large-scale success is vendor politics.

1. Web Services Are Just Like Distributed Objects

Given the strong similarities between Web services and distributed objects, it is understandable that many people believe they are the same. After all, both have some sort of description language, well-defined network interactions, and similar mechanisms for registering and discovering available components. Contributing to the misconception is that many tool vendors provide simple object-oriented techniques for implementing Web services, thus giving them the appearance of distributed objects. Several of these vendors have long histories with distributed object technologies, which gives them strong motivation to make Web services seem to be an evolutionary step in distributed object systems.

One thing to remember, however, is that Web services technology is currently very limited. For example, unlike well-established distributed object systems, which have numerous support tools, Web services toolkit vendors have only just begun to look at the reliability and transactional guarantees that distributed object systems have supported for years.

An important notion at the core of distributed object technology is the object life cycle:

- Upon request, a factory instantiates the objects.
- The consumer who requested the instantiation performs various operations on the object instance.

- Sometime later, either the consumer releases the instance or the runtime system garbage-collects it.

A special case is the singleton object, which does not go through the instantiate–release cycle. In both the standard and specialized cases, however, the object is identified through a reference that can be passed between processes to provide a unique access mechanism for it. Objects frequently contain references to other objects, and distributed object technology comes with extensive reference-handling techniques to support correct object-lifetime management.

This notion of object reference is essential; without it, there is no distributed object system. This reference also gives the caller a mechanism for returning to a given object repeatedly and accessing the same state. Distributed object systems enable stateful distributed computing. A consumer accesses an object's state through a well-defined interface that is typically described in an interface definition language.

A TCP transport is not sufficient for end-to-end reliability of Web services.

Web services share none of the distributed object systems' characteristics. They include no notion of objects, object references, factories, or life cycles. Web services do not feature interfaces with methods, data-structure serialization, or reference garbage collection. They deal solely with XML documents and document encapsulation.

With a bit of a stretch, we could force an analogy between a Web service and a singleton object or a stateless JavaBean from the Java2 Enterprise Edition (J2EE) world, but we would need to make such an object very restrictive to make the comparison work. Web services cannot offer any of the stateful distributed computing facilities that most distributed object systems support as basic functionality.

Major differences between the two technologies are also obvious when we look at how information flows between client and server or producer and consumer. In the distributed object system, the information flow's richness is encapsulated in the interfaces an object supports; in a Web services system, it comes from the design of the XML documents that are passed around. Both technologies try to achieve the same goal — to provide the application designer with rich facilities for encapsulating application functionality — but they use very different techniques.

At the basic level, Web services do not define relationships between service invocations at the same or related services, but we can build only very limited distributed systems without identifying relationships between components in a computation. One of the first advanced Web service specifications released dealt with *coordination*[4] to enable multiple services and their consumers to establish contexts for their interactions. This context is not just a weak form of object references; it identifies an ongoing conversation rather than any states at the services.

Distributed object technology is very mature and robust, especially if you restrict its usage to the environment for which it was designed: the corporate intranet, which is often characterized by platform homogeneity and predictable latencies. Web services' strength is in Internet-style distributed computing, where interoperability and support for platform and network heterogeneity are essential. Over time, Web services will need to incorporate some of the basic distributed systems technologies that also underpin object systems, such as guaranteed, in-order, exactly-once message delivery. It is unlikely, however, that Web services can simply adapt the existing technology to achieve the same properties, given that they try to address interoperability on a scale at which other technologies have failed.

Web services and distributed object technologies can work together in two known approaches. First, we can wrap certain objects from an object system, such as J2EE, with a Web service. Of course, this approach has its limitations and will not work for just any object. (See Steve Vinoski's article on interaction models to learn more.[5]) Alternatively, we can use Web service protocols such as SOAP as the transport layer for the distributed object system. This is sometimes used to tunnel object-specific interactions over HTTP. It is, however, a poor choice because alternative solutions, such as the general inter-ORB protocol (GIOP), are better suited for that interaction pattern.

2. Web Services Are RPC for the Internet

RPC provides a network abstraction for remotely executing procedure calls in a programming language. It offers mechanisms for identifying a remote procedure, deciding which state must be provided to the procedure at invocation time, and what form to use to present the results to the caller at completion time. It also includes extensive mechanisms for handling errors at both the runtime and programming levels.

Basic Web services provide only a networking abstraction for transferring XML documents and for a remote service entity to process them. Web services use the notion of *actor* or *role* to identify the service that should consume the document, but they associate no predefined semantics with the XML document's content. We could implement an RPC-style interaction using pairs of SOAP messages and a transport such as HTTP, but this would require certain fixed rules for encoding the arguments in an XML document and rules for returning the results to the caller.

We can view Web services as just XML document processors, but this paradigm doesn't easily integrate with current application-development techniques. Tool vendors will do their best to provide the infrastructure to let developers apply traditional procedure calls to simple Web services. For example, Microsoft's Web Service Enhancements 2.0 toolkit provides a set of object types we can use to implement an RPC-style interaction, in which the programming infrastructure tries to interpret the document for the programmer. The toolkit also includes a similar set of types that give the programmer simple but powerful support for receiving raw XML documents.

Internet-wide RPC has failed in the past, and Web services are not going to be much help in solving the issues. The Web services infrastructure provides no magic that can suddenly overcome what excellent protocol architects were unable to achieve with established RPC systems or GIOP. Although Web services might solve some of the interoperability issues, many challenges remain. Synchronous interactions over wide-area networks are not scalable, for example, and large-scale versioning of procedure interfaces is extremely difficult.

3. Web Services Need HTTP

Web services are "transport agnostic," meaning that we can access them over any type of transport or application protocol. We can use SOAP to transport messages over HTTP, but we can also use it to send messages over plain TCP or UDP. With some bindings, the messages can flow over simple mail-transfer protocol (SMTP) by encapsulating the SOAP messages in email, or over a traditional messaging infrastructure such as MQ-Series or Java messaging service (JMS). A common scenario for the Web services architecture describes a message flowing over multiple transport protocols before reaching its destination.

For example, a parts-ordering application might deliver a SOAP request to an enterprise gateway using HTTP. The gateway could then use a load-balancing mechanism to pick a node in a server farm to process the request and use a persistent TCP connection to forward the incoming document. In another case, a purchase order encapsulated in a SOAP message might be delivered using an email message over an SMTP transport. The receiving server could take the SOAP content, encapsulate it in a JMS message, and then insert it into the order-processing workflow system, which might be based on traditional message queuing. The service that actually consumes the SOAP request might not be identified until the message has visited a few intermediate processors that determine whether the request is entitled to "gold" priority treatment after some auditing has occurred. The requesting process (remember that Web services are for computer-to-computer conversations with no humans involved) will eventually receive an email message confirming or rejecting the order.

Although the Web services architecture was developed with this transport independence in mind, most Web services run over HTTP. One reason for this is that most of the early Web services toolkits used the existing infrastructure offered by the major Web servers: Apache, IBM WebSphere, and Microsoft IIS. Leaving the request parsing and message dispatching to the Web server lets developers abstract away all the tedious work with Web services. Web server add-ons such as Axis (http://xml.apache.org/axis) or ASP.NET (www.asp.net), which implement Web service runtimes, automatically generate the service's WSDL and provide simple service exercise tools, thus creating a great environment for prototyping and learning Web services. For example, these simple tools let developers use Web-based interfaces to inspect and exercise services.

HTTP is also popular for implementing Web services for a second, more strategic, reason: in contrast to the dot-com boom period, most enterprise software projects now require short-term returns on investment. This forces most production Web service projects to focus on improving access to corporate data and services for partners and customers, without requiring much new infrastructure. The logical first step is to use the Web servers that are already functioning as front ends to, say, the enterprise's J2EE infrastructure. We can view this

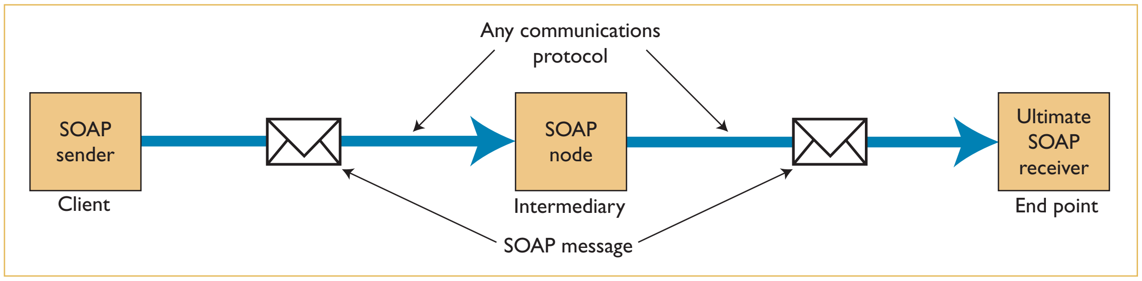


Figure 1. SOAP's transport independence. Web service documents encapsulated in a SOAP message can be delivered directly to a destination over a single transport or via a collection of intermediaries over a variety of transports.

rather successful approach as the first move toward deeper integration of Web services in the enterprise.

Some suggest that the main reason for tunneling Web service messages through HTTP is to bypass firewalls. Although there is sufficient evidence that port-80 traffic is heavily overloaded with various non-Web-server communications, this has triggered the emergence in firewall devices of more extensive content-based filtering techniques for HTTP traffic. The firewall-protected systems have not become less secure because of the tunneling of application traffic through HTTP, whether as SOAP-encapsulated Web service documents, peer-to-peer relay messages, or Corba inter-ORB messages. Firewall-filter techniques did, however, have to extend beyond the realm of simple port-based protection.

4. Web Services Need Web Servers

Some have discussed whether to drop the "Web" from Web services, as it really leads to greater confusion instead of a clearer worldview. This is already obvious in such terms as service-oriented architectures, service-oriented integration, or services bus. None of these enterprise concepts uses the term "Web" because they are not relying on any Web technologies, such as HTTP or Web servers.

Quite a few toolkits now let you develop and integrate Web services without Web server infrastructures. Examples include Simon Fell's PocketSoap, Systinet's Web Applications and Services Platform (WASP), IBM's Emerging Technologies Toolkit, and Microsoft's Web Services Enhancements (WSE). Enterprise integration systems, such as IONA's Artix and DocSOAP, also facilitate Web-server-independent Web service development.

As I mentioned, most early Web services have exploited Web servers' application-server functionality. Now that developers have made the initial business case and need a wider choice of transports, most will move away from implementing systems inside Web servers.

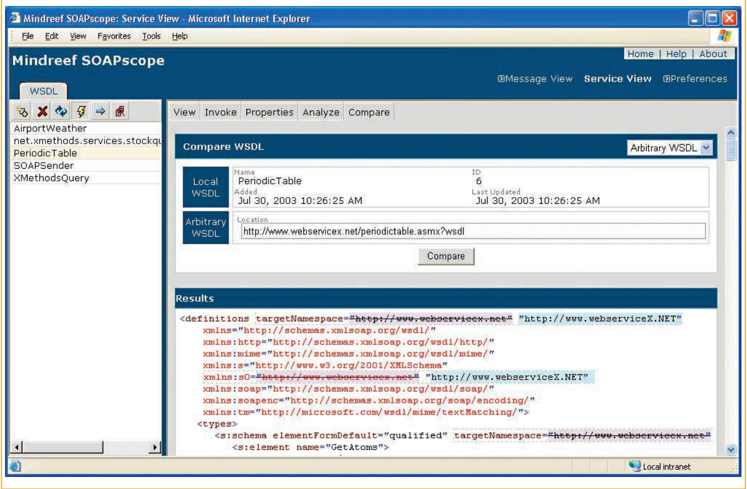
Several forums such, as the W3C Web Services Architecture working group, have witnessed a high-profile debate over the past few months about applying representational state transfer (REST)[6] principles to Web services architectures. (REST principles underpin many of the scalable Web-resource naming and access mechanisms.) This debate about Web principles is valuable, particularly with respect to resource identification and operation visibility, but it is quickly becoming irrelevant to the bigger picture as transport independence surpasses the Web's importance to Web services. The REST principles are relevant for HTTP bindings, and for the Web server's parsing of resource names, but they are useless in the context of TCP or message-queue bindings where the HTTP verbs do not apply.

5. Web Services Are Reliable Because They Use TCP

TCP guarantees reliable, in-order message delivery; it seems that Web services that use TCP should be able to achieve the same guarantees. However, the reliability guarantee is only partially true for TCP programming. In a few scenarios, the protocol cannot completely deliver a message to a remote peer, and if the local participant closes the connection, it won't receive an error notification.

More importantly, as Figure 1 shows, document and message routing for Web services provide for the use of intermediaries. Under a network, node, or component failure, several scenarios are possible in which the document arrives successfully at the first station but never reaches its final destination, and thus, never gets processed by the service.

Web services, and distributed systems in general, require end-to-end reliability. In the coming year, we will see whether any of the established techniques for achieving such reliability can also be applied to Web services, or if new technologies are needed. In general, we achieve reliability through


Mindreef SOAPscope: Service View - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Mindreef SOAPscope
Home | Help | About
Message View Service View Preferences
WSDL
AirportWeather
net.xmethods.services.stockqu
PeriodicTable
SOAPSender
XMethodsQuery
View Invoke Properties Analyze Compare
Compare WSDL
Arbitrary WSDL
Compare
Results
Local intranet

Local WSDL
Name: PeriodicTable
ID: 6
Added: Jul 30, 2003 10:26:25 AM
Last Updated: Jul 30, 2003 10:26:25 AM

Arbitrary WSDL
Location: http://www.webservicex.net/periodictable.asmx?wsdl

```
<definitions targetNamespace=~~"http://www.webservicex.net"~~ "http://www.webserviceX.NET"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:s0=~~"http://www.webservicex.net"~~ "http://www.webserviceX.NET"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/">
  <types>
    <s:schema elementFormDefault="qualified" targetNamespace=~~"http://www.webservicex.net"~~
      <s:element name="GetAtoms">
```

Figure 2. Document-versioning debugging. Dedicated Web service monitors such as SOAPscope (shown) can detect changes in document descriptions, thus helping to solve the major version headaches that Web service developers encounter.

message retransmission, but we must also weed out duplicate messages in case the original was not really lost. Estimating timeouts and handling other potential sources of message loss in a heterogeneous network such as the Internet is not trivial.

When building reliable distributed systems, we frequently want some information about the service request's processing state to flow back so that the document producer can take local actions. Feedback about the document's arrival, the service's consumption of the document, and the request processing's completion is essential in activities such as selecting retransmission strategies or performing local garbage collection.

In addition to reliability, we also want to ensure that the service consumes the messages in the order they were sent, if the sender so desires. This puts more stress on the reliability system because it might have to delay message processing until the producer retransmits the lost message.

These guarantees have been around for years, working in all sorts of distributed systems, including distributed object systems and multiparty fault-tolerant systems. Until these technologies are added, however, Web services should be considered unreliable — whether they use TCP or not.

6. Debugging Web Services Is Impossible

As Web services enable Internet-scale distributed computing, in which the conversing parties frequently belong to different organizations, Web service developers and deployers confront a new set of problems that traditional debugging and monitoring tools cannot handle. Web services' federated nature introduces most of these new challenges because the developer does not "own" both ends of the wire. Two of the most prominent challenges are cross-vendor interoperability and WSDL versioning.

Although traditional tools offer little help with these problems, new diagnostic tools such as SOAPscope (www.mindreef.com) are emerging to address Web services' development and deployment issues. SOAPscope is unique in that it focuses on "watching" the wire — logging the traffic and providing a suite of functions to detect and resolve federation-related and other potential problems.

With the wide variety of toolkits for developing Web service clients and servers, it is increasingly common to find different ones operating at either end of a SOAP interaction. Given that each toolkit vendor is likely to interpret the specification somewhat differently, interoperability problems are becoming more common. When a client encounters an obscure error from a server, the developer must attempt to diagnose the problem without access to the code running at the server. The only available solution is to focus on the SOAP traffic on the wire and the WSDL contract between services.

New diagnostic tools offer several features that can help developers understand and fix interoperability problems. SOAPscope, for example, has `resend` and `invoke` features that let the user test "what if" scenarios against a server to isolate problem requests. It also has "viewing" capabilities that help clarify SOAP messages by visualizing them at higher abstraction levels than raw XML. To maximize interoperability, SOAPscope also has a `WSDL analysis` feature that can detect and help resolve potential interoperability problems prior to service deployment.

Another increasingly common challenge arises from changes to and versioning of Web services. A small change to the XML document specification in the WSDL contract at the server can easily break existing clients. Clients might not even be aware that the specification has changed, let alone know how to accommodate that change. A Web service client might start receiving fault messages from the server that indicate a problem, but these are seldom useful for resolving the issue. Tools such as SOAPscope, shown in Figure 2, can inspect the XML document specification in the current WSDL and compare it to the specification used to create the client. Such debugging and deployment tools that use historical data and real-time views of the Web service interaction provide extremely powerful new tools for Web service developers.

Conclusions

Web services technology is still evolving, even at the most basic levels, but many vendors, trade magazines, and venture capitalists have already tagged the technology as the trigger for a new wave of applications, enabled by federated interoperability. This early exposure has generated many incomplete and incorrect publications, toolkit releases that have little or no architectural vision, and fights among different standardization bodies seeking the right to control the underpinnings of Web services. The waters are further muddied by the fact that many of the vendors who jumped on board to promote Web services also have vested interests in Web and applications servers or distributed object technologies.

Web services are going to play an important role in the future of distributed computing, significantly impacting application and system development. However, developers and researchers in both industry and academia must work to clear up common misconceptions about the technology, or else we will end up with architectures that have severely limited functionality and performance.

Educating system architects and developers about Web services in a manner that is independent of specific vendors' solutions is an important first step, and academia's involvement in the process is essential. We must cultivate broad awareness that, although important, object-oriented and remote-procedure-style technologies provide just a few options for building applications and integrating them with Web services.

Web services technology will have a dramatic enabling effect on worldwide interoperable distributed computing once everyone recognizes that Web services are about interoperable document-centric computing, not distributed objects.

Acknowledgments

This article improved significantly because of comments and criticisms from several people: Dave Seidel, Don Box, Doug Lea, Eric Newcomer, Felipe Cabrera, Frank McCabe, Jim Gray, Ken Birman, Mark Baker, Mark Ericson, Mike Champion, and Ted Herman. In addition, Steve Vinoski provided valuable editorial assistance.

References

1. N. Mitra, "SOAP Version 1.2 Part 0: Primer," World Wide Web Consortium recommendation, June 2003; www.w3.org/TR/soap12-part0/.
2. C. Ferris and D. Langworthy, "Web Services Reliable Messaging Protocol (WS-Reliable Messaging)," joint specification by BEA, IBM, Microsoft and Tibco, Mar. 2003; http://www-s106.ibm.com/developerworks/webservices/library/ws-rm/.
3. C. Evans et al., "Web Services Reliability (WS-Reliability), ver. 1.0," joint specification by Fujitsu, NEC, Oracle, Sonic Software, and Sun Microsystems, Jan. 2003; http://developers.sun.com/sw/platform/technologies/ws-reliability.html.
4. F. Cabrera et al., "Web Services Coordination (WS-Coordination)," joint specification by BEA, IBM, and Microsoft, Aug. 2002; http://www-106.ibm.com/developerworks/library/ws-coor/.
5. S. Vinoski, "Web Services Interaction Models, Part 1: Current Practice," *IEEE Internet Computing*, vol. 6, no 3, 2002, pp. 89-91.
6. R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, doctoral dissertation, Univ. of California, Irvine, 2000; www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Werner Vogels is a researcher in the computer science department at Cornell University. He is the principal investigator in several projects that focus on the scalability and robustness of mission-critical enterprise computing systems. Vogels received a PhD in computer science from Vrije Universiteit, Amsterdam. Contact him at vogels@cs.cornell.edu.