Web Services: A New Paradigm for Distributed Computing

by Jason Bloomberg
Director of E-Services
Ashton Services

With the downturn in today's Internet technology marketplace, Web Services shine like a beacon of hope. Fueled by Microsoft's .Net initiative, Sun's SunOne architecture, and IBM's WebSphere platform, Web Services promise a new level of compatibility across multiple technology platforms. Many companies who still have R&D budgets are pouring time and money into Web Services, and bringing forth an alphabet soup of new technologies: WSDL, SOAP, and UDDI, to name a few.

The ideas behind the Web Services movement are reasonably straightforward. Start with XML, the movement's backbone. This straightforward language provides a "least common denominator" method for different systems to talk to each other. Next, use XML to create a set of industry standard protocols for finding companies, describing and exchanging information, and handling transactions between companies. Add water, and voila! You have a brave new world of e-commerce.

Or do you? Many are skeptical about the Web Services movement. Maybe what you really have, they say, is nothing more than a new spin on some old technologies. Web Services look an awful lot like the distributed component-based technologies CORBA and DCOM.

So, how significant is the Web Services movement, really? Are these new technologies the key to e-commerce compatibility nirvana? Or are they nothing more than another incremental development in the never-ending saga of technology companies looking for new stuff to sell?

More Than Old Wine in a New Bottle
Just as object-oriented (OO) programming, when it first appeared, looked a lot like procedural programming, programming with Web Services looks a lot like programming the OO way. Early OO programs were little more than procedural programs shoehorned into classes. Eventually, programmers learned to take advantage of the power of OO programming, and a new way of thinking about coding became established. The same thing is happening with Web Services.

The Web Services programs that people are writing today are straightforward applications that leverage the technology's encapsulation and message transmission capabilities -- precisely those capabilities that Web Services inherited from the OO world. What's missing is a broad understanding of the new potential for Web Services, which goes beyond existing models. There's no way to grasp this big picture simply by learning the nuts and bolts of Web Services technologies. What's needed is a top-down architectural vision for Web Services.

**Limitations of Remote Procedure Call (RPC) Architectures**

Until quite recently, we have always thought of computer programs as discrete chunks of code that reside on one computer. Even OO programming began in a single-computer environment. This isolated computer mindset has been around so long that it pervades all thinking about software.

Then along came networks. Only then did people realize that computer programs do not necessarily have to reside on individual computers. Early communication protocols -- such as the Network File System for Unix and Microsoft's Distributed Computing Environment -- focused on the network layer. These protocols, in turn, led to the development of wire protocols for distributed computing, most notably Microsoft's DCOM and the OMG's Internet Inter-ORB Protocol (IIOP) that underlies CORBA.

Remote Procedure Call (RPC) architectures like DCOM and CORBA/IIOP enabled programs to be broken up into pieces, with different pieces running on different computers. OO techniques were particularly suited to this distributed environment, for a couple of important reasons. First, objects maintained their own discrete identities, and could easily be located in different places. Second, the nuts and bolts of communication between objects could be encapsulated into its own set of classes; programmers working in a distributed environment didn't need to worry about how this communication worked.

Programmers, however, continued to work within an isolated computer mindset. Both DCOM and CORBA took basically the same approach: Write your programs so that the remote computer appears to be part of your own computer. If you need code that lives on someone else's computer, you can make a simple request, and the hidden machinery of the distributed platform will marshal the remote code and ship it to you via the preferred wire protocol. Of course, the process of marshaling executable code and shipping it over the Internet opened up a Pandora's box of security concerns. And DCOM and CORBA handle key attributes
In addition, both of these technologies had other serious limitations. DCOM is a Microsoft-only architecture, and CORBA, although intended to provide cross-platform interoperability, in reality is too complex and semantically ambiguous to provide real interoperability without considerable manual integration work.

**Web Services: A Better RPC Architecture**

It was in this environment that Web Services came to life, and it offers major advantages over these prior technologies. Web Services are defined as encapsulated, loosely coupled, contracted objects offered via standard protocols. Let's break this definition down to make it more understandable:

- **Encapsulated** means that the implementation of each Web service is invisible from outside the Web service. Its functionality is known only by the interface it exposes. Encapsulation is a fundamental requirement of OO architectures.

- **Loosely coupled** means that changing a Web service does not require a change in the program that invokes the Web service. In addition, Web Services provide dynamic interface descriptions, allowing for Service invocation at runtime. DCOM and CORBA are both tightly coupled, which limits their flexibility.

- **Contracted** means that the Web Service’s behavior, as well as how to bind to it, and its input and output parameters, are publicly available.

- **Standard protocols** are open and freely available. Web Services are built upon XML and HTTP, which are both standard protocols.

In addition, Web Services can interact via the familiar synchronous request/response pattern of RPC architectures, as well as the asynchronous messages familiar in messaging architectures like IBM's MQ. However, this article focuses on Web Services as an RPC architecture.

<table>
<thead>
<tr>
<th>RPC Architecture</th>
<th>Payload Parameter Value Format</th>
<th>Endpoint Naming</th>
<th>Wire Protocol</th>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA</td>
<td>Common Data Representation (CDR)</td>
<td>Interoperable Object Reference (IOR)</td>
<td>IIOP (Binary)</td>
<td>Interface Definition Language (IDL)</td>
</tr>
<tr>
<td>DCOM</td>
<td>Network Data Representation (NDR)</td>
<td>OBJREF</td>
<td>DCOM (Binary)</td>
<td>Inherited from COM</td>
</tr>
</tbody>
</table>
Table 1: RPC Architectures Compared

Table 1 compares important features of CORBA, DCOM, Java's Remote Method Invocation (RMI), and Web Services. Because XML is a text-based, standard language, Web Services, as you can see, solve many of the problems inherent in CORBA, DCOM, and RMI:

- DCOM's format for payload parameter values is the Network Data Representation (DR), while CORBA's corresponding format is the incompatible Common Data Representation (CDR) format. Web Services use XML as the payload parameter value format, providing a simple, open language for tagging payload parameter values.

- Web Services provide a better method for RPC protocols to name communication endpoints. DCOM handles endpoint naming with OBJREF representations, while CORBA uses an Interoperable Object Reference (IOR). OBJREFs and IORs do not correlate, and translations between the two are problematic. Web Services and RMI, in contrast, use URLs, which are universally understood and accepted.

- Web Services use the Simple Object Access Protocol (SOAP) as their wire protocol. SOAP is built on top of XML, which makes it text-based and human readable. As a result, it is firewall-friendly and easy to debug, as opposed to IIOP, DCOM, and JRMP, which are binary protocols. On the downside, the binary protocols are faster than SOAP messages.

- Once RPC architectures have defined their wire protocols, they must provide a way to define their interfaces. DCOM relies on COM's standard memory layout, which it specifies on the binary level, allowing access with multiple languages at the expense of robustness and ease of debugging. CORBA provides its Interface Definition Language (IDL), which, along with its APIs, allow objects to interact with Object Request Brokers (ORBs). RMI provides for Java-only interfaces. Web Services, on the other hand, provide the Web Services Description Language (WSDL), which is an XML grammar for specifying the properties of a Web service, including what it does, where it resides, and how to invoke it. Like SOAP, WSDL is text-based, giving it the same firewall-friendly and easy-to-debug advantages that SOAP enjoys.

- A programmer can use WSDL to create service proxies for remote Web Services. These proxies can be thought of as client stubs (analogous to stubs in CORBA or RMI or proxies in DCOM), which are the local representation of a remote program that characterizes
all RPC architectures. Therefore, a Web Services architecture can be thought of as an RPC architecture, following in the footsteps of DCOM, CORBA, and RMI, but with substantial improvements in interoperability and ease of development.

A New Software Paradigm

In my opinion, these advantages alone are enough to justify the investment in tools and training necessary to leverage Web Services. In fact, there's a good chance that many of the Web Services applications developed over the next few years will be nothing more than new and improved ways of handling distributed computing, because of that persistent, single computer mindset we talked about earlier.

In truth, however, Web Services also offer a whole new way of thinking about software -- a new paradigm, if you will -- that goes far beyond the current approach to distributed computing. The key to this new paradigm is the technology used to contract Web Services' functionality: It is this contracting that puts the "services" into Web Services.

The technology that provides publicly available descriptions of Web Services' behavior, binding, and communication protocols is the Universal Description, Discovery, and Integration (UDDI) specification. UDDI provides an XML-based, platform-independent way for Web Services to describe themselves, discover other Web Services, and integrate with those services. UDDI provides both an API for programmers and a system of business registries through which companies can list their Web Services. The API is split into a publication API, for putting Web Services into registries, and an inquiry API, for discovering, binding to, and integrating with Web Services in a registry.

On the surface, UDDI looks like little more than an automated Yellow Pages, which would make you wonder whether enough companies will want to participate in UDDI to make it useful. However, a global eCommerce Yellow Pages is only one vision for UDDI -- and in many ways, it misses the point of the initiative.
In reality, UDDI provides a spectrum of options for discovering and binding with remote Web Services, as shown in Figure 1. At its simplest level, UDDI allows programs to bind to Web Services at design time, essentially hardwiring the connection to a Web service. UDDI also allows for dynamic binding to a static service, which enables a program to discover the specifics of a Web service’s interface at runtime. Finally, it is also possible for a program to both discover a Web service as well as bind to it at runtime, essentially enabling a Just In Time (JIT) interface to a set of Web Services. It is this JIT integration to remote services that has never been feasible with existing RPC architectures.

Available Web Services may or may not be registered in a global public registry. The UDDI specification also allows companies to build their own specialty registries, which opens up the possibility of entirely new "Web Services broker" business models. UDDI registries might either belong to private e-marketplaces or be internal to large enterprises. As a result, enterprise software packages, including ERP, CRM, and enterprise portal applications, can be rearchitected into a loose collection of Web Services, tied together with a UDDI-compliant registry located on the enterprise's intranet.

**Architecture for Web Services**

How, then, must architects shift their thinking to move from object-oriented architectures to the new "Service Oriented Architecture" (SOA)
enabled by Web Services technologies? A useful approach is to look at the SOA from multiple views. The 4+1 View Model of Software Architecture, introduced by Philippe Kruchten and detailed in the Rational Unified Process® (RUP®), provides a broad and useful technique for thinking about the SOA.

Figure 2: The 4+1 View Model of Architecture

As shown in Figure 2, the 4+1 View Model of Architecture suggests a five-view approach, with the four main views taking the perspective of key stakeholders in the development project. On top of the four views is a fifth view, the **Use-Case View**, which overlaps the other views and plays a special role with regard to the architecture.

The **Implementation View**, also called the **Development or Component View**, describes the organization of the software code, and addresses issues of software management. In the SOA, the Implementation View focuses on the internal workings of individual Web Services, and on the Web Services technology stack (HTTP, XML, SOAP, WSDL, and UDDI). Since Web Services are encapsulated, managing their construction is very similar to managing the construction of traditional components.

The **Logical View**, also described as the **Design View**, starts with the end user's functional requirements of the system, that is, what the system is supposed to do, and uses them to create a top-down abstraction of the design model for the system. Much of the work of an architect approaching the Logical View of the SOA will involve user context issues of Web Services, including user privacy and security, service contracts, billing, and management, as well as personalization issues. Such user context capabilities are currently being built into Microsoft's Hailstorm environment as well as Sun's Open Net Environment (ONE). As Hailstorm and SunONE become more established, SOA architects will need to concern themselves with these environments. The Logical View will help them with these concerns.

The **Deployment View**, or **Physical View**, maps the software to the underlying platforms, including the hardware. This view is where software engineering meets system engineering. Deployment view concerns for the
SOA architect will include discussions of n-tier, application server based platforms versus peer-to-peer (P2P) platforms (popularized by the Napster music service), as well as whether to use the request-response model of RPCs or go with a messaging model. Web Services should ideally be agnostic with respect to these decisions, but in reality, such platform decisions will continue to be important. In addition, building Web Services on P2P platforms opens up new worlds of bleeding-edge possibilities sure to interest many die-hard techies.

The bulk of the work in an SOA, however, will take place in the Process View. The Process View addresses runtime issues, including processes, concurrency, and scalability. As the work with Web Services moves into JIT integration, the Process View will become increasingly important. In fact, the service aspect of Web Services is fundamentally a set of processes.

In moving from OO architectures to the SOA, most of the real architecture work will occur in the shift from the Implementation View to the Process View. This shift is the new paradigm for distributed computing. In object-oriented architectures, structuring class hierarchies is the focal point of an architect’s work. In the SOA, however, structuring the process flows among interacting Web Services in a JIT environment will be at the core of the architect’s life.

The Fifth View

I saved our discussion of the fifth view -- the Use-Case View -- for the conclusion to make a point. The Use-Case View is intended to drive the discovery and design of the architecture, and then to validate the different views. The existence of use cases, however, presupposes that there are business users who know what they want a software system to do. The truth is, today’s business users don’t know what Web Services might be capable of, so they don’t know what to ask for. Web Services are just too new.

Today, people express the functionality of Web Services in the context of existing solutions, because that's what they know. It is up to the early adopters to understand and discover how Web Services and the SOA enable a new way of thinking about component technology. Only then will the new capabilities of this paradigm filter down to the business users. So what will the use cases of tomorrow look like? Only time will tell, but they will most likely be framed in the context of global, business-to-business e-commerce.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!