Comparing Web Services to Other Technologies

Whenever business managers hear that a new widget is needed, they instinctively ask, "Why can't we just use the XYZ technology instead?" This is a perfectly reasonable question to ask if you are a cost-conscious manager. However, this type of thinking can often lead to the "Golden Hammer" approach. When this happens, all problems must be solved with the one magic bullet technology that is familiar to you. Usually this occurs with little thought put into the strengths and weaknesses of that technology and how well it fits the problem at hand. Good engineers always try to expand their tool chest of useful technologies. Web services are another solution to the problem, but they aren't the only solution.

The alternatives can generally be grouped into three categories:

- Stub/skeleton based remote procedure call architectures (CORBA, RMI, DCOM)
- HTTP-like transactional architectures (Servlets, JSP, ASP, PHP, CGI)
- Screen scrapers

As you read through this hour, you will see that Web services are actually an evolution of the first two methods and has many similarities to each of them—with few of the drawbacks.

Let's briefly look at each of these architectures and examine some of the solutions based on them.

Stub/Skeleton Based Architectures

Solutions in this category all work in generally the same fashion. They are meant to provide programmatic access to some form of remote service as though it was a local entity. The advantage to this sort of architecture is that it makes client program creation relatively simple because as far as the client program is concerned, the objects and their methods are local, just like every other object. The architecture itself takes care of the communications to get the data from client to server and vice versa, which frees up the client developer to worry about UI design, business logic, and everything else. Figure 4.1 illustrates the stub/skeleton architecture.

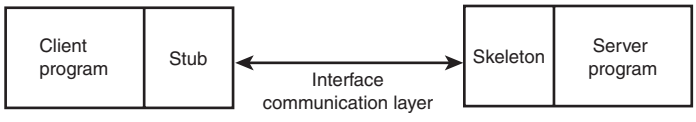

FIGURE 4.1
The stub/skeleton client-server architecture.

In order to make these architectures simple for the clients, developers are required to build two modules—a stub and a skeleton.

The stub is a block of code that sits on the client machine. It identifies which objects and methods are available on the server, and marshals (encodes) all calls from the client to the server into a format that the interface layer can understand. It also unmarshals (decodes) any data coming back from the server into something the client can understand.

The skeleton performs the same sort of operation as the stub, but in the opposite direction for the server side. It exposes what capabilities exist on the server that the client can call, receives the incoming requests from the clients, and returns any data back to the client.

All the solutions in this category are also denoted by the characteristic that when a new client connects with a server, it stays connected until the client is terminated. This has a tendency to reduce the number of clients that the server can handle because it must maintain those connections for long periods of time.

Finally, each of these architectures makes use of some form of directory that allows clients to look up the available services from each server. The differences are in the details because each architecture performs this process differently—hence, not allowing the solutions to be interchangeable.

The main differences between the following architectures are how the data is encoded and how the transport layer functions.

Some of the popular solutions that use this architecture are CORBA, RMI, and DCOM.

CORBA

NEW TERM *CORBA (Common Object Request Broker Architecture)* was designed in the early 1990s to provide a mechanism for building client/server applications in heterogeneous environments.

Some of the key features of CORBA are

- **Language Neutral**—CORBA was designed to work with any language. In order to bridge the gap between differing languages, an *Interface Definition Language (IDL)* is used to detail the structure of all objects that will be passed along the wire into a language-neutral format. Developers then take the IDL and run it through some form of code generator for the language used on each end of the transaction to get the corresponding language-specific stub or skeleton. By doing this, it's possible to write a Visual Basic client that talks to a Java server, using CORBA as the communication layer.

NEW TERM

- **Multiple Vendors**—Multiple vendors provide *Object Request Brokers (ORBs)*. This allows users to pick and choose between vendors for the capabilities and costs that are right for them. Some ORB vendors only support certain languages as well.

NEW TERM

Although CORBA would seem to be an excellent solution for heterogeneous client/server systems, it has instead proven itself to be a bit of a hassle initially in practice. Initial releases of the CORBA specification left many areas open to interpretation by the vendors. As a result, many vendor's ORBs refused to work with each other, limiting the ability to mix and match.

The cross-language features of CORBA have also proven to be a bit of a curse as well because it requires developers to learn IDL and specify all their interfaces and objects that are involved in the CORBA calls. There is also the performance penalty of converting an object from one language representation into the IDL representation, and then back into some other language on the other end. This time penalty can be deadly when used in a high-volume system.

CORBA requires the use of special ports on which the ORBs communicate and transfer the data. In many network environments, network administrators are reluctant to open ports to the outside world because these represent areas for possible intrusion and attack by hackers. This can sometimes limit a developer's ability to deploy systems based on CORBA. For systems communicating entirely within a secure intranet, this isn't an issue, but for those bridging internal systems to the Internet, this is quite a security risk.

Finally, CORBA can be somewhat difficult to secure. In most cases, data transferred in CORBA-based systems is sent across the wire in clear text. This makes it rather easy for hackers to listen in on communications and steal data.

Newer versions of CORBA implementations have made great strides in overcoming many of these difficulties, but unfortunately these fixes have come too late. Most of the industry has already moved on to other solutions. CORBA is still a viable alternative in some cases though, and should not be overlooked in situations in which you need heterogeneous capabilities and you can control and secure the network properly.

As a result, compared to Web services, CORBA solutions

- Are nearly as capable for cross-platform and cross-language development.
- Are harder to understand because CORBA relies on IDL to translate data; Web services use XML, which is much more human readable. Most toolsets also create the WSDL for you.
- Can handle higher transaction loads because they keep a persistent connection between clients and servers at the expense of servicing fewer clients per server.
- Are a much more mature technology, and many of the initial interoperability issues between vendors have been worked out already. A lot more information is currently available on CORBA as well.

Java RMI

NEW TERM Similar to CORBA, *Remote Method Invocation (RMI)* is built on top of the stub/skeleton architecture. RMI is the Java-specific mechanism for performing client/server calls. It is actually very similar to CORBA in many respects. The biggest difference is that because RMI is usually used for Java-to-Java architectures, there is no need for the IDL. Developers are working with true Java objects at all times. This has changed over the years with the addition of RMI communicating over *IIOP* (the same protocol that CORBA uses), allowing RMI to talk directly to CORBA.

Again, similar to CORBA, RMI requires that a specific port be opened for communications between the client and server. As with CORBA, this can sometimes be difficult to get network administrators to open due to security concerns.

RMI can be somewhat easier than CORBA to secure against eavesdropping as long as the various objects that are being passed are written to include code to encrypt and decrypt their binary representations during marshaling/unmarshaling. This adds a bit of additional work for the developers, but pays off in piece of mind.

In cases in which you control both the client and server and can guarantee that both will be built with Java in a trusted network environment, RMI will usually perform faster than XML-based Web services because of the reduced work in getting the data into a wire-friendly format.

Compared to Web services, RMI is the better choice if both ends are Java based, but useless in non-Java guaranteed situations.

As a result, compared to Web services, RMI solutions

- Lock you into a purely Java solution on both the client and server
- Can be somewhat more difficult to deploy because of network port considerations
- Can handle higher transaction loads because RMI keeps a persistent connection between clients and servers at the expense of servicing fewer clients per server

DCOM

New Term *DCOM (Distributed Common Object Model)* is the Microsoft mechanism for performing remote calls. Objects are again converted into a wire-friendly format and converted back to language-specific representations at the endpoints of the communication.

Although DCOM can be built in several different languages (Visual C++, Visual Basic, C#, and so on), it only works on Microsoft platforms. As a result, if your business does not use Microsoft servers, DCOM doesn't help you. Both ends of the transaction (client and server) need to be Microsoft systems in order to use DCOM.

Although DCOM is supported by multiple languages, the strong ties to Microsoft mean that Web services still hold an edge in flexibility. Web services can be implemented with tools from many different vendors on various platforms.

Compared to Web services, DCOM solutions are nearly as capable for cross-language development, but usually lock you into a Microsoft-everywhere framework.

HTTP Transactional-based Architectures

The second category of Client/Server architectures is based on the familiar Web server paradigm. These systems operate by having some piece of code running on a server—

similar to (or extending) a Web server such as IIS or Apache. Clients communicate with the server code through the familiar HTTP or HTTPS; the server handles the request and performs the work, responding with whatever data was requested (typically with HTML or XML text as the response). After the transaction is complete, the connection between the client and the server is terminated. Figure 4.2 shows the communication process typically found in HTTP Transactional architectures.

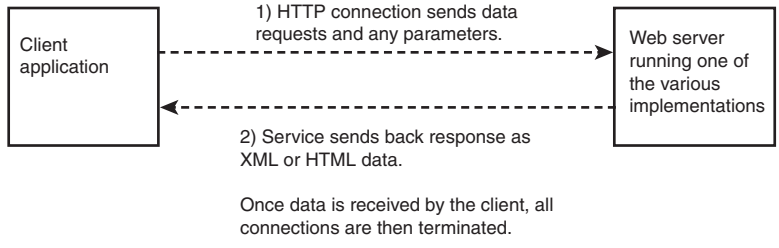

FIGURE 4.2
The HTTP Transaction process.

Because the connection between the clients and the server stays in place only for the duration of the transaction, these types of systems typically have a larger client-to-server ratio. The downside of these systems is that because the connection is not persistent, in situations in which clients make a large number of discreet transactions with the server, a large percentage of time is wasted creating and terminating connections. In such situations, the stub/skeleton architectures tend to be better. Some systems utilize the HTTP 1.1 keep-alive mechanism that gets around this issue by maintaining the connection between the client and the server, but at the expense of limiting the number of clients the server can handle.

Another strike against these types of solutions is the lack of a directory service such as UDDI. Clients must know of the service in advance, must know what data the service expects, and must know how the service expects to receive that data. Whereas Web services utilize WSDL to describe how to interface between the client and service, with the HTTP-based architectures, a good deal of cooperation must take place between the developers of the service and the client-side developers in order to build a functioning system. This limits how effective these solutions can be in an open service-type environment in which you might want to allow anyone on the Internet to use your service.

Error handling can also be tedious on these types of systems because there is no established formal mechanism for indicating and handling problems. Where Web services have a true fault mechanism, and the stub/skeleton systems have notions of formal exception objects, neither exists in the HTTP transaction-based architectures.

Type safety is another concern. Again, it's the lack of formalized communications that comes into play. Without such agreed upon specifications, it becomes more difficult to agree on data representations (how many digits represent an integer versus a long or a string, and so on). Also, there's no way to stop somebody from sending the wrong type of data and causing problems.

One advantage that these solutions have over their stub/skeleton counterparts is that these solutions all communicate over the familiar HTTP ports. Most networks leave the common HTTP and HTTPS ports (80 and 81, respectively) open for communications, so usually no additional work is required to get a solution based on these technologies deployed on the network.

HTTP-based solutions also send all their data across as plain text, which can be a security issue. It is rather simple, however, to use HTTPS (SSL encryption over HTTP) to secure your data. This usually incurs a slight performance penalty and should only be used in situations with sensitive data.

Let's look more closely at the more popular solutions based on this architecture.

CGI

Back in the early 1990s when the World Wide Web and HTML were new, very little interactive capability was built into the HTML and Web server specifications. Users could really only connect to a server and retrieve prebuilt, static documents with the occasional picture. There was no way to request specific data or get information tailored to the user.

NEW TERM In order to overcome this shortfall, HTML input forms and *common gateway interface (CGI)* was created. CGI is a mechanism by which when data is sent to the server, the Web server can invoke a program (the CGI) and pass all the data that was sent along with the request to the program. The CGI program processes the data and builds up a response page that it then sends back to the user.

CGI was a great solution for the time. (In fact, it was the only solution!) Users could submit requests and get dynamically generated data returned to them. For instance, a user could go to a sports Web site, select his favorite baseball team, and get the statistics for all the players on the team for any day in the season. This added a whole new level of capabilities to the Web.

The CGI solution does have problems though. Developers quickly found that for popular sites using CGI, the Web servers needed to be rather large. Every request to the server caused a new instance of the CGI program to be instantiated, run, and then terminated.

All this starting and stopping put a drain on system resources. Some solutions were created to attempt to fix this issue over time, but the problem still exists. As a result, other solutions have taken the place of CGI in the mainstream.

CGI can be written in many different languages (C, Perl, Python, Shell scripts, and so on). This does allow the server-side programmer some flexibility. CGI is a mechanism supported by pretty much every Web server as well. CGI programs usually aren't cross platform capable because of language and platform differences, but they usually can be ported fairly easily from server platform to platform. For instance, a CGI program written in C for a UNIX platform will probably need at least a little modification before it can be compiled to run on a Windows platform. In some cases, the language that the CGI is written in might not even be built in to the OS and would need to be fetched as an add on (such as Perl, which is common on UNIX, but not on Windows without fetching a copy of ActivePerl and installing it). Even then, it is sometimes necessary to modify the code to tailor it to the new system. Even with these limitations, because CGIs rely on only HTTP for input and HTML/XML for output, client-side applications can be written in pretty much any language.

Web services still tend to be better than CGIs, however. CGIs really can only accept string-type data or binary attachments sent to them, not true objects. This means that a client would need to convert any data into some sort of CGI-specific data representation as a string, post the data, and then parse any data returned to it into something meaningful again. All this must be done with no mechanism to enforce the encoding mechanism or structure. Web services do all that for you. There are no real indexes of CGI services out there either. You simply need to know the URL to point your browser or client application to in order to call the CGI-based service. Finally, without knowing what the CGI is expecting to have posted to it, it becomes very difficult to write a client.

In short, compared to Web services, CGIs are

- Harder to find because of no directory service
- Harder to write clients for without a well-documented service-specific API to rely on
- Harder to interact with programmatically because there's no accepted data interchange format
- All over the place on nearly any Web server or platform

Servlets/JSP

The Java-based solution to the CGI world is the servlet.

In this section, whenever we say servlets, we actually mean servlets and JSP. In reality, although they look very different when written by the developer, JSP code is actually compiled into a servlet by the Java servlet container. As such, a servlet and a JSP are synonymous.

Java servlets provide all the same capabilities as their CGI heritage, but without the resource penalty. Instead of running as a program, servlets are inherently multithreaded. Each request invokes a new lightweight thread instance instead of the heavyweight process instances used by CGIs.

Servlets can respond to more than just HTTP requests. It is possible to build servlets to accept nearly any protocol. In fact, many of the Java-based Web services toolkits work by placing servlet wrappers around the service code that the developer writes and executing the servlet in a servlet container.

Because servlets are written in the Java language, they gain the "write once, run anywhere" capabilities of Java. Java servlet containers are available for nearly every imaginable platform, thus allowing your servlet to be deployable anywhere. You have a multitude of vendors to choose from as well.

Unfortunately, even with the additional capabilities, servlets find themselves bound by the same limitations that hold back CGIs. There is no directory indicating what CGIs are available, no interface specification explaining how to communicate with them, and all data must be written into a format the servlet can understand.

The points about servlets that stand out in relation to Web services are as follows:

- Servlets can only be written in Java.
- Servlets are harder to write clients for without a well-documented service-specific API to rely on.
- Servlets are harder to interact with programmatically because there's no accepted data interchange format.
- Servlets can be found on many Web server platforms.

ASP and PHP

NEW TERM The remaining two technologies typically used for HTTP-based services are ASP and PHP. These two systems are fundamentally similar in scope and as such will be discussed together. *Active Server Pages (ASP)* is the solution championed by Microsoft and is based on the Visual Basic language. PHP is another solution, which is

minal or a Web browser), push data into the interface, and then read (or scrape) the returned data off the interface and convert it into something the client application needs.

This sort of solution actually has proven popular in some mainframe-centric shops that have legacy applications in which nobody truly understands the business rules explaining how the applications work, but they do understand how to use the applications. In these cases, it is actually easier to write a scraper to act as sort of a virtual middleman than it is to attempt to tear apart the legacy code and add a true Web service interface to it.

Screen scrapers aren't pretty. They work, but they rely on the format of the screens that they're scraping not to change. If the server-side application changes, the scraper must also be changed to handle the new format or workflow process. This can be a tedious process at best, a nightmare at worst. Still, if you have no other choice, screen scraping does work.

Summary

In this hour, we have discussed some of the alternative solutions available for building client/server systems. We've examined the strengths and weaknesses of each solution and pointed out where those solutions could be better than Web services.

We've also discussed areas where Web services have an advantage over each of these solutions. In nearly all cases, Web services do a better job of providing cross-platform, cross-language service while maintaining ease of development, more flexibility, and more vendor choices for the developer.

With the information presented here, you should be able to narrow down your selection of solution architectures and pick the one best suited to your needs.

Q&A

Q If I already have a system based on one of the technologies discussed in this hour, can I continue to leverage it while using Web services?

A Absolutely! The beauty of Web services is their capability to provide open interfaces to existing systems. Simply build a Web services wrapper interface layer around your existing system. Have the wrapper make calls to your existing system, and then take the results and send them back via SOAP.

Q Web services looks a lot like CORBA and RMI. If this is true, why the big deal?

A As you've seen this hour, yes, Web services do look very much like these other systems. That's not surprising because many of the people who created the specifications for Web services are the same engineers who grew up building systems with CORBA and RMI. They designed Web services to take the best from those technologies while at the same time fixing some of their problems. Some people even like to refer to Web services as "CORBA over XML" because the architectures are so similar.

Q Does knowing one of these existing technologies help in my understanding of Web services?

A Definitely! Again, Web services are an evolution, not a revolution. If you know CORBA or RMI, and have at least a little knowledge of HTTP, Web services should seem very straightforward to you.

Workshop

This section is designed to help you anticipate possible questions, review what you've learned, and begin learning how to put your knowledge into practice.

Quiz

1. What are the three types of typical solutions other than Web services for building client/server solutions?
2. What features denote most stub/skeleton solutions?
3. What features of HTTP-based solutions limit their usefulness as compared to Web services?

Quiz Answers

1. Stub/skeleton based, HTTP-based, and screen scrapers.
2. Tight coupling between the client and the server throughout the client lifetime, some sort of directory used to find the service, stubs and skeletons to provide the interface between the client and server.
3. Lack of any real indexing service, mainly based around text-based data, no data formatting standards or interface specifications.

Activity

1. If you are unfamiliar with any of the technologies we've discussed in this hour, it is recommended that you read up on them. Knowing more about other competing technologies is always a good thing. Sams Publishing has a large line of books covering all these topics.