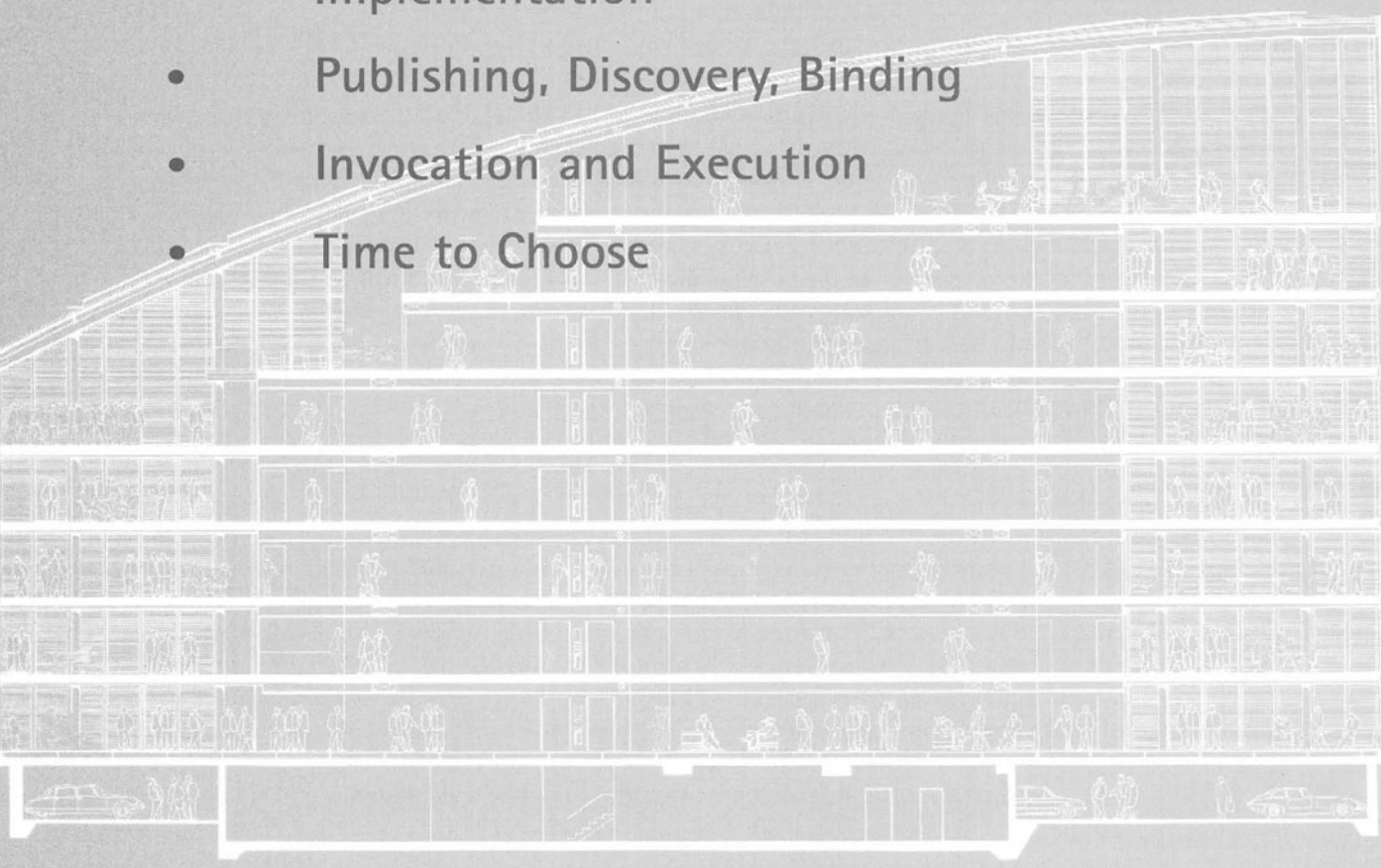
Authors: J. Jeffrey Hanson, with additional material by Chanoch Wiggers

- Description
- Implementation
- Publishing, Discovery, Binding
- Invocation and Execution
- Time to Choose



K. Apshankar et al., *Web Services Business Strategies and Architectures*

.NET and J2EE, a Comparison

Introduction

Web Services is the term being used to describe some of the technologies for solving the problems of integrating applications across the enterprise and between disparate companies over the Internet. Web Services are also being described as the concepts and technologies for delivering services and content to any Web-enabled client or device. A simplified attempt at defining Web Services generically might be as follows:

> **Web Services are URI-addressable resources that use existing Internet infrastructures and protocols to allow applications, services, and devices to discover them, connect to them, and execute their business logic using remote method calls.**

Sun is touting its Java Web Services Developer Pack (http://java.sun.com/webservices/webservicespack.html) as its Java 2 Platform, Enterprise Edition (J2EE) toolset for wrapping XML-based Web Services technologies such as SOAP, UDDI, ebXML, and WSDL with Java objects and interfaces. At the same time, Microsoft is working on its .NET platform. Among other things, .NET is designed to facilitate the development of interoperable Web Services. As usual, Microsoft is presenting a comprehensive set of development tools to accommodate this new technology. Among these is the SOAP 2.0 Toolkit (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/soap/htm/kit_intro_19bj.asp). This toolkit provides a broad range of SOAP support tools such as a component that maps Web Service operations to COM-object method calls. Similar facilities have been built into the .NET Framework from the start, and these can be unleashed using Visual Studio .NET.

Offerings from other sources include the Apache SOAP project and IBM's Web Services Toolkit.

The Apache SOAP project (http://xml.apache.org/soap/) is an open-source, Java-based implementation of the SOAP v1.1 (http://www.w3.org/TR/SOAP) and SOAP Messages with Attachments (http://www.w3.org/TR/SOAP-attachments) specifications in Java. Apache SOAP can be used as a client library to invoke Web Services or as a server-side tool to implement Web Services.

IBM's Web Services Toolkit (http://alphaworks.ibm.com/tech/webservicestoolkit) provides a run-time environment and examples to design, implement, and execute Web Services on any operating system that supports Java 1.3 or above. The toolkit provides a Web Services architectural blueprint, a private UDDI registry, some sample programs, some utility services, and tools for developing and deploying Web Services. The toolkit also includes a Web Services client API that can be used to access a UDDI registry.

Web Services Overview

The Web Services model as it stands attempts to allow potentially unrelated services to be dynamically combined, in a loosely coupled manner over a distributed network. Web Services encourages developers to evolve to a services-based model.

Web Services are currently concerned with four basic challenges:

1. Service Description.
2. Service Implementation.
3. Service Publishing, Discovery, and Binding.
4. Service Invocation and Execution.

Current technologies are solving these challenges by:

5. Describing Web Services using the Web Services Description Language (WSDL).
6. Using XML as the common language for Web Service communication. XML is ubiquitous throughout all aspects of Web Services. Web Services can be implemented in any programming language that can read and write XML, and can be deployed on any Web-accessible platform.

7. Publishing Web Services in a registry to be discovered later by interested parties accessing the registry. One type of registry provides a directory service for Web Service providers and their services. This registry provides information categorized by industry-type, product-type, service-location, service binding, etc. Taxonomies are used to enable searches on this information. One implementation of this registry is based on the Universal Description, Discovery and Integration specification (UDDI). Another type of registry also acts as a repository where Web Services entities such as a business-process schema are stored. A current example of this type of registry is the electronic business XML (ebXML) Registry and Repository.

8. Invoking Web Services over an existing Internet protocol such as HTTP or SMTP using the Simple Object Access Protocol (SOAP).

Service Description

In order for Web Services to proliferate it is important to be able to describe them in some structured way. The Web Services Description Language (WSDL) (http://www.w3c.org/TR/wsdl) addresses this need by defining an XML grammar for describing Web Services as collections of message-enabled endpoints or ports.

In WSDL, the abstract definition of endpoints and messages is separated from their concrete deployment or bindings. The concrete protocol and data format specifications for a particular endpoint type constitute a binding. An endpoint is defined by associating a web address with a binding, and a collection of endpoints defines a service. A WSDL document uses the following elements in the definition of Web Services:

- **Types** – data type definitions used to describe the messages to be exchanged.
- **Message** – a typed definition of the data to be exchanged. Messages and operations are bound in order to form an endpoint or port.
- **Operation** – a description of an action exposed by the service. Operations can support input and/or output messages.
- **Port Type** – a named set of abstract operations and related messages supported by one or more ports.
- **Binding** – a concrete protocol and message format specification for operations and messages defined by a particular port type.
- **Port** – a single endpoint defined as a combination of a binding and a network address. In a WSDL document, services are defined as collections of ports or endpoints. The abstract definition of a port is separated from its concrete protocol and data format specification.
- **Service** – a collection of related ports. Multiple port definitions, sharing the same port type within a service, provide semantically equivalent alternatives to service consumers. This allows service consumers to choose the port or ports to communicate with.

The following example taken from W3C's WSDL (http://www.w3.org/TR/wsdl) site shows a simple stock-quote WSDL document:

```
<?xml version="1.0"?>
<definitions name="StockQuote"
             targetNamespace="http://example.com/stockquote/definitions"
             xmlns:tns="http://example.com/stockquote/definitions"
             xmlns:xsd1="http://example.com/stockquote/schemas"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>

  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>

  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>
</definitions>
```

This, simply put, describes a service that provides Stock Quotes. It accepts a Trade Price Request and returns a Last Trade Price as output. This description of the service is quite devoid of implementation details and yet provides sufficient information to use the services. Web Services can, in this way, provide a system for hiding the implementation specifics of systems making interoperability possible.

J2EE

J2EE-enabled Web Services are described by WSDL documents. Third parties who want to transact business with a J2EE-enabled Web Service company can look up information about the company's Web Services in a registry. There, they will find links to URLs containing the WSDL information needed to format XML documents correctly for integrating with the Web Services exposed by the company.

.NET

As with a J2EE Web Service, a .NET Web Service supports the WSDL 1.1 specification and uses a WSDL document to describe itself. An XML namespace, however, is used within the WSDL document to uniquely identify the Web Service's endpoints.

.NET provides a client-side component that allows an application to invoke Web Service operations described by a WSDL document and a server-side component that maps Web Service operations to COM-object method calls as described by a WSDL and a Web Services Meta Language (WSML) (http://msdn.microsoft.com/library/en-us/soap/htm/soap_overview_72r0.asp) file. This file is needed for Microsoft's implementation of SOAP.

Service Implementation

Implementing Web Services currently means structuring data and operations inside of an XML document that complies with the SOAP specification. Once a Web Service component is implemented, a client sends a message to the component as an XML document and the component sends an XML document back to the client as the response.

J2EE

Existing Java classes and applications can be wrapped using the Java API for XML-based RPC (JAX-RPC) and exposed as Web Services. JAX-RPC uses XML to make remote procedure calls (RPC) and exposes an API for marshaling (packing parameters and return values to be distributed) and unmarshaling arguments and for transmitting and receiving procedure calls.

With J2EE, business services written as Enterprise JavaBeans are wrapped and exposed as Web Services. The resulting wrapper is a SOAP-enabled Web Service that conforms to a WSDL interface based on the original EJB's methods.

The J2EE Web Service architecture is a set of XML-based frameworks, which provide infrastructures that allow companies to integrate business-service logic that was previously exposed as proprietary interfaces. Currently, J2EE supports Web Services via the Java API for XML Processing (JAXP). This API allows developers to perform any Web Service operation by manually parsing XML documents. For example, you can use JAXP to perform parsing operations with SOAP, UDDI, WSDL, and ebXML.

J2EE also uses the Java API for XML Messaging (JAXM) to integrate one back-end system with other back-end systems via the following five document-centric message exchanges:

1. A synchronous update whose response is an acknowledgment that the update was received.
2. An asynchronous update whose response is an acknowledgment that the update was received.
3. A synchronous inquiry whose response is data requested in the original message.
4. An asynchronous inquiry whose response is data requested in the original message.
5. An exchange where the sender sends a message and does not expect a reply.

When a JAXM client sends a message, the message first goes to a JAXM provider, which then handles the actual transmission of the message to its destination. A JAXM client receives a message from a JAXM provider that has received the message on behalf of the client and then forwarded the message. A more likely scenario is that a JAXM client will receive an XML message from another platform, and because it is in XML it will understand it.

.NET

.NET applications are no longer directly executed in native machine code. All programs are compiled to an intermediate binary code called the Microsoft Intermediate Language (MSIL). This portable, binary code is then compiled to native code using a Just In Time compiler (JIT) at run-time and run in a virtual machine called the Common Language Runtime (CLR). This is similar to the way that Java works, except .NET encompasses several languages; each is translated to MSIL, which is executed in the CLR using the JIT – simple, really.

With the .NET platform, Microsoft provides several languages based on the Common Language Infrastructure (CLI), such as Managed C++, JScript.NET, VB.NET, and C#. On December 13, 2001, ECMA (formerly the European Computer Manufacturers Association) ratified specifications for C# and the CLI making them official industry standards.

For existing Microsoft technologies like VB6, the Microsoft SOAP Toolkit offers components that construct, transmit, read, and process SOAP messages. The toolkit also includes an alternative to using the XML DOM API to process XML documents in SOAP messages called the SOAP Messaging Object (SMO) framework (http://msdn.microsoft.com/library/en-us/soap/htm/soap_adv_2wdv.asp). The toolkit provides a component that maps Web Service operations to COM-object method calls as described by the WSDL and WSML files of the service. Conversely, the toolkit provides a generator that will generate WSDL files from COM `typelib` descriptions.

Microsoft has merged its InterDev and Visual Studio products into a new development environment, Visual Studio .NET. Together with the .NET framework, practically all of the details of SOAP, XML, and Web Services discovery and binding are hidden from the user.

To create a Web Service with the .NET Framework outside of the Visual Studio.NET environment, you simply create a file with an ASMX extension. The ASP.NET run-time, part of the .NET Framework, recognizes this file as a Web Service and uses a built-in HTTP handler to process requests on it. The run-time then forwards the requests to a .NET class that's either included in the ASMX file or implemented separately. The following code shows a simple Web Service written in C#:

```
<%@ WebServices Language="C#" class="Hello" %>
using System.Web.Services;
class Hello
{
   [WebMethod]
   public string SayHello(string userName)
   {
      return "Hello " + userName;
   }
}
```

This example demonstrates the `WebServices` directive, the directive for referencing the `System.Web.Services` namespace, and a public method with the `WebMethod` attribute. The `WebMethod` attribute makes a method accessible from a Web Services client. To deploy a .NET Web Service, the implementation code is saved to a file with the ASMX extension, in an IIS virtual directory.

Service Publishing, Discovery, and Binding

Once a Web Service has been implemented, it must be published somewhere that allows interested parties to find it. Information about how a client would connect to a Web Service and interact with it must also be exposed somewhere accessible to them. This connection and interaction information is referred to as *binding* information.

Registries are currently the primary means to publish, discover, and bind Web Services. Registries contain the data structures and taxonomies used to describe Web Services and Web Service providers. A registry can be hosted either by private organizations or by neutral third parties. Currently two types of registries, UDDI and ebXML, are being addressed by the Web Services community.

At the time of this writing, IBM and Microsoft have announced the Web Services Inspection Language (WSIL) specification to allow applications to browse Web servers for XML Web Services. WSIL promises to complement UDDI by making it easier to discover available services on web sites not listed in the UDDI registries.

J2EE

Sun Microsystems is positioning its Java API for XML Registries (JAXR) as a single general-purpose API for interoperating with multiple registry types. The Java API for XML Registries (JAXR) provides a uniform and standard API for accessing disparate registries within the Java platform. A registry provider is an implementation of a Web Services registry conforming to a registry specification. A JAXR provider provides an implementation of the JAXR specification typically as a façade around an existing registry provider such as a UDDI or ebXML registry. A JAXR client uses the JAXR API to access a registry via a JAXR provider.

There are three types of JAXR providers:

- The JAXR Pluggable Provider.
- The Registry-specific JAXR Provider.
- The JAXR Bridge Provider.

The Pluggable Provider implements features of the JAXR specification, which are independent of any specific registry type. The Registry-specific JAXR Provider implements the JAXR specification in a registry-specific manner. The JAXR Bridge Provider is not specific to any particular registry. It serves as a bridge to a class of registries such as ebXML or UDDI.

Sun provides a freely downloadable (http://www.sun.com/software/xml/developers/regrep/) ebXML implementation based on the J2EE platform and implements the ebXML Registry Information Model 1.0 and the ebXML Registry Services Specification 1.0. This registry/repository implementation uses EJB technology and includes the following components: a registry information model, registry services, a security model, a data access API, Java objects binding classes, and a JSP tag library.

.NET

At first, Microsoft had the discovery of Web Services with DISCO in the form of a discovery (DISCO) file. A published DISCO file is an XML document that contains links to other resources that describe the Web Service. Since the widespread adoption of UDDI, however, Microsoft has supported it in order to maximize interoperability between solutions in what is, after all, a set of specifications for interoperability.

In addition to providing a .NET UDDI server, the UDDI SDK provides support for Visual Studio .NET and depends on the .NET framework. Products such as Microsoft Office XP offer support for service discovery through UDDI.

Service Invocation and Execution

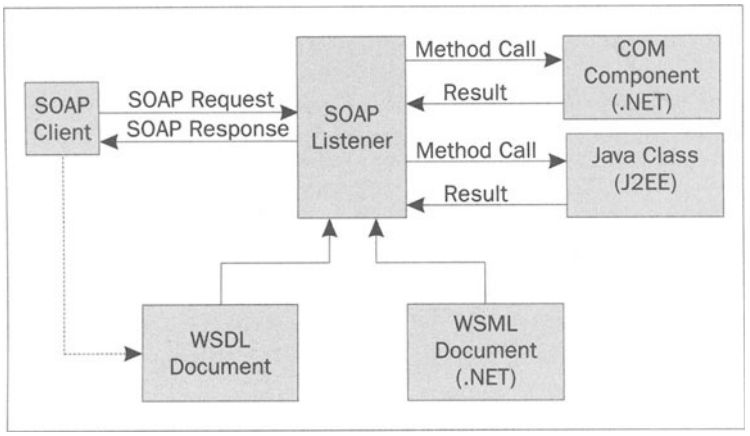
The Simple Object Access Protocol (SOAP) is a simple, lightweight XML-based protocol that defines a messaging framework for exchanging structured data and type information across the Web.

The SOAP specification consists of four main parts:

- A mandatory envelope for encapsulating data. The envelope contains an optional header element (`<SOAP-ENV:Header>`) and a mandatory body (`<SOAP-ENV:Body>`).
- Optional data encoding rules for representing application-defined data types, and a model for serializing non-syntactic data models (such as object).
- A request/response message exchange pattern.
- An optional binding between SOAP and HTTP.

SOAP can be used in combination with any transport protocol or mechanism that is able to transport the SOAP message.

Web Service recipients operate as SOAP listeners and can notify interested parties (other Web Services, applications, etc.) when a Web Service request is received. The SOAP listener validates a SOAP message against corresponding XML schemas as defined in a WSDL file. The SOAP listener then unmarshals the SOAP message, turning it into a format understandable by the Web Service implementation. Within the SOAP listener, message dispatchers can invoke the corresponding Web Service code implementation. Finally, business logic is performed to get the reply. The result of the business logic is transformed into a SOAP response and returned to the Web Service caller. This process is shown in the following diagram:



J2EE

J2EE uses the Java API for XML-based RPC (JAX-RPC) to send SOAP method calls to remote parties and receive the results. JAX-RPC enables Java technology developers to build Web Services incorporating XML-based RPC functionality according to the SOAP 1.1 specification.

Once a JAX-RPC service has been defined and implemented, the service is deployed on a server-side JAX-RPC run-time system. The deployment step depends on the type of component that has been used to implement the JAX-RPC service. For example, an EJB service that is implemented as a stateless session bean is deployed in an EJB container. A container-provided deployment tool provides support for the deployment of a JAX-RPC service.

During the deployment of a JAX-RPC service, the deployment tool configures one or more protocol bindings for this JAX-RPC service. A *binding* ties an abstract service definition to a specific XML-based protocol and transport. An example of a binding is SOAP 1.1 over HTTP.

A Web Service client uses a JAX-RPC service by invoking remote methods on a service port described by a WSDL document. A WSDL-to-Java compiler generates the client-side stub class, service definition interface, and additional classes for the service and its ports.

There are three different modes of interaction between clients and JAX-RPC services:

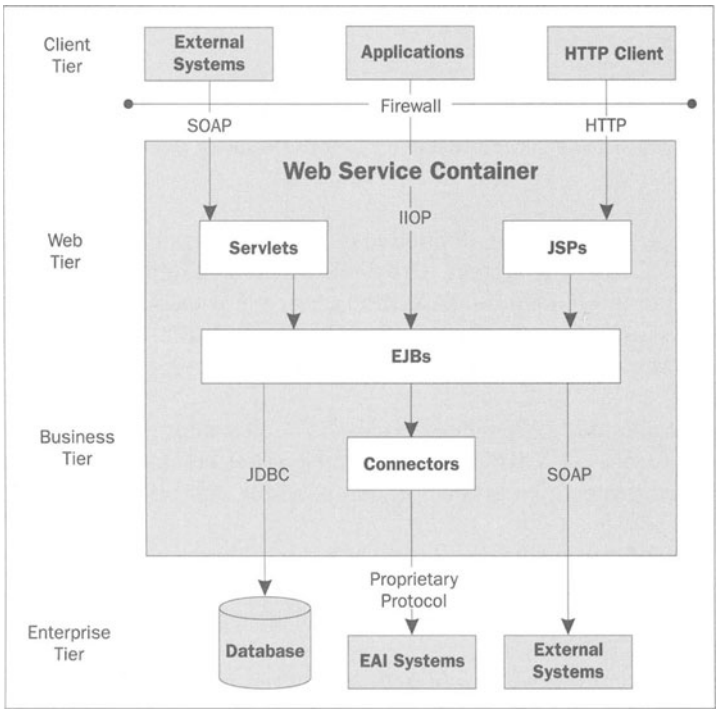
- **Synchronous Request-Response**: The client invokes a JAX-RPC procedure and blocks until it receives a return or an exception.
- **One-Way RPC**: The client invokes a JAX-RPC procedure but it does not block or wait until it receives a return.
- **Non-Blocking RPC Invocation**: The client invokes a JAX-RPC procedure and continues processing in the same thread then, later, blocks or polls for the return.

The Java API for XML Messaging or JAXM provides an API for packaging and transporting of message-based business transactions using on-the-wire protocols defined by emerging standards.

Implementations of JAXR (Java API for XML Registries) providers may use JAXM for communication between JAXR providers and registry providers that export an XML Messaging-based interface.

J2EE uses the Java Architecture for XML Binding (JAXB) to map elements in XML documents exchanged with third parties to Java classes, so that a business system can process them. JAXB compiles an XML schema into one or more Java classes. The generated classes handle the details of XML parsing and formatting. Similarly, the generated classes ensure that the constraints expressed in the schema are enforced in the resulting methods and Java data types.

The following diagram illustrates the J2EE Web Services programming model:



.NET

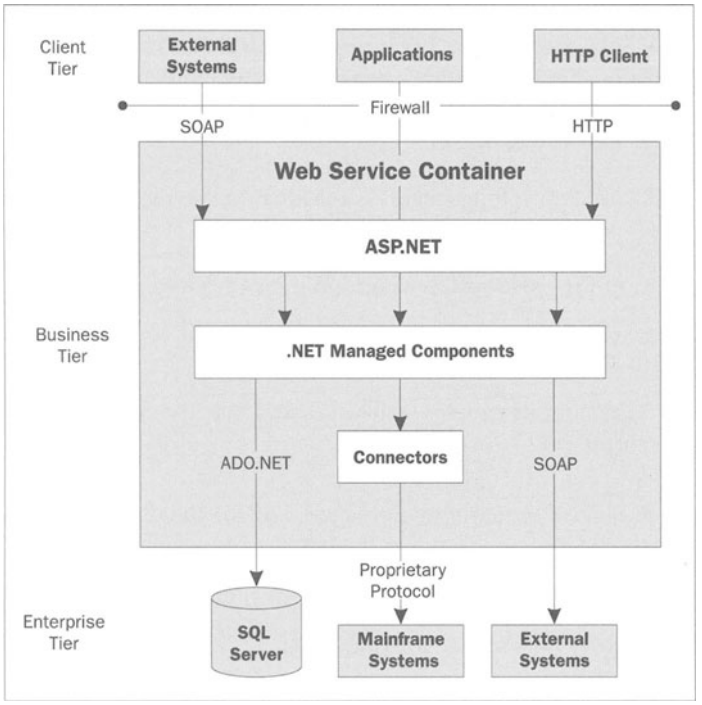
In Microsoft's .NET framework, interested parties can gain access to a Web Service by implementing a Web Service listener. In order to implement a Web Service listener, a system needs to understand SOAP messages, generate SOAP responses, provide a WSDL contract for the Web Service, and advertise the Service via UDDI. If the Web Service is hosted within IIS as an ASMX file, there is no need to implement a SOAP Listener, since IIS does that job for you. Also .NET has built-in classes that understand SOAP messages.

Microsoft Developers creating SOAP-based Web Service listeners and consumers currently have three choices:

1. Construct a Web Service listener manually, using MSXML, ASP, ISAPI, etc.
2. Use the Microsoft Soap Toolkit version 2 to build a Web Service listener that connects to a business facade, implemented using COM.
3. Use the built-in .NET SOAP message classes.

The Microsoft SOAP Toolkit 2.0 offers a client-side component that lets an application invoke Web Service operations described by a WSDL document.

The following diagram illustrates the .NET Web Services programming model:



Time To Choose

Having looked from a high level at how J2EE and .NET handle Web Services, we are faced with a choice – which version do we implement? From a purely technical standpoint, each method has advantages and disadvantages: .NET code often runs faster than J2EE systems, but only works on Microsoft operating systems; it is often easier to turn existing J2EE code into a Web Service due to the object-oriented nature of Java, whereas .NET code will have to be written from scratch, particularly where the original code was Visual Basic; the list continues.

It is worth noting that there is a project underway to create an open-source implementation of the .NET Framework, so non-Microsoft operating systems will soon be able to support .NET. See http://www.go-mono.com/ for more details. As standardization continues, the availability of .NET on non-Microsoft platforms will become easier. Although much existing Microsoft code is not object oriented, .NET is entirely object oriented, including VB.NET. Writing wrappers for VB components is no harder than doing the same for a Java component – the difficulty arises when the VB code provides functionality that isn't exposed as a component.

The key advantage, perhaps, of using the .NET approach to Web Services is that it has been designed for that purpose, whereas J2EE is being retrofitted by the addition of a number of APIs. Despite this, J2EE is an inherently modular platform, so the problem here is one of maturity of technology; .NET has only just reached final release.

One advantage of using J2EE as a base for your system is that you have a much wider choice of vendor for your pre-built software (application servers mostly), including many open source projects. In many ways, open-source J2EE application servers are closer to the standards laid down for Java, because they don't add proprietary extensions to overcome problems. As time creeps on, open-source application servers are becoming more popular, and more competitive than the more expensive vendor-driven options.

Microsoft offers several compelling business reasons for developing Web Services using its .NET architecture:

- Easy migration for existing COM and Windows-based systems.
- .NET's abstraction away from the hardware offers increased security for applications developed to the Common Language Runtime.
- Developer tools are, as usual, world-class and greatly ease the learning curve and ease of development.

J2EE, as well, offers several enticing business reasons for developing Web Services using its technologies:

- Easy migration for existing Java shops.
- Proven security at the code-execution level.
- Support from many different industry leaders.

Ultimately, unless you are starting your system from the bottom up, your choice of Web Services implementation is more than likely going to be influenced by your present system. If you have a team of skilled Java programmers, with a J2EE business system, realistically you'll want to continue with J2EE. Similarly with .NET, there is no sense in wasting your investments in Microsoft products (both time and money) by switching to J2EE – you're going to want to keep on with what your team knows best. As Integrated Development Environments (IDE) for Java become more powerful, the ease of development with Visual Studio (.NET) may no longer be the deciding factor it may once have been.

From an adoption point of view, while Microsoft has provided systems for wrapping pre-.NET code so that it can work in the .NET framework and vice versa, the nature of the MS languages before this change means that many will probably be unsuitable for this purpose. The reverse – the ability to wrap .NET services so that they can work in a COM environment – means instant interoperability with legacy systems while allowing a migration to the .NET framework.

Existing Microsoft shops are unlikely to move to J2EE, due to the current investment in Microsoft technologies (skills base, hardware, software, and business relationships). In addition, the perceived cost of Microsoft-based solutions is lower, although there is room for clarification on the total cost of ownership. A primary problem in J2EE systems appears to be over-engineering at the cost of development time, performance, and the hardware needed to run these systems. On the other hand, Microsoft systems are PC-based, and these systems are inherently less scalable and powerful, and the cost for scaling is high.

It may be worth noting that Microsoft has a conflict of interest in the area of Web Services. Since Microsoft wish to be the providers of Web Services in addition to providing the tools and specifications for implementing Web Services, it appears to have a first option advantage in providing services such as single sign-on authentication and authorization, and referral services that will make competing with them strictly for the biggest companies if at all.

J2EE is a mature, proven platform with architecture and operating system-independence – this independence so resembles the principle of Web Services that it is no wonder it is a natural fit. The ability to move from Intel-based machines to more powerful servers makes applications written on this platform very scalable with an excellent Cost of Change curve. Assuming that the business case for the application is sound, the system should work equally well on a 486 PC with Windows (intranet application/small Internet user base) or an IBM server farm with thousands of users (Internet enabled – customer and client system), and will scale in cost as well as in performance according to the business needs at hand.

Perhaps one way of expressing the difference between the two is that the Microsoft camp advocates buying more computers whereas Sun/IBM would support buying bigger computers.

In addition, Java has proven to be a language that makes it easy to architect and implement maintainable systems. The fact that patterns are core to Java and the Java platform makes it easy to communicate architectural semantics. The specification system that is in place for Java means that industry leaders are responsible for making sure that the available APIs for application development are relevant to the needs of existing and future systems.

Existing J2EE software houses will no doubt appreciate these benefits, in addition to the flexibility that well-architected Java systems offer and the existing base of supporting packages and open-source initiatives that can be a starting point for the creation of servers. In addition, there is a wealth of information available on the subject.

The primary difficulty with J2EE is Sun's apparent conflicting tendencies as far as Web Services are concerned. While strongly pushing a proprietary platform for Web Services in the form of the iPlanet server, the work of creating usable specifications for development appears to be lagging behind Microsoft in effort. In this way, Sun's strategy of both offering products for the creation and hosting of Web Services, and providing Web Services strongly resembles that of Microsoft, only in this case it appears to be interfering with their ability to provide the systems for Web Services creation and deployment.

The upside to this is that there are several other proponents of Web Services, such as IBM, BEA, and others, that are strongly supporting the creation of Web Services on the Java platform, with the additional support of open-source initiatives such as the Apache Software Foundation. As well as this, several companies are offering products that automatically expose Java applications as services through a process known as introspection. This means that developers are able to create software in the usual way, and make it available for use by in-process non-distributed systems and distributed systems using technologies such as RMI that resemble Web Services but that involve Java-only protocols or heavyweight protocols such as CORBA as is currently the case. Additionally, they can be made available as Web Services using XML as the communication and data encoding mechanism. This makes Web Services a glue or façade to the existing Java infrastructure.

In any case, Java programmers are not used to the Point and Click support for software development that Microsoft developers receive and so the lack of these systems will not hold them back.

Conclusion

Web Services promise to revolutionize not only the way we develop software systems, but how we do business. Some aspects of Web Services development, such as security and transaction handling, are yet to be completely solved, and they must be in order to make the Web Services dream a reality.

WDSL, SOAP, UDDI, XML, Microsoft's .NET, and Sun's J2EE provide the technologies and tools needed to get Web Services off to a running start. Are these technologies and tools enough to make Web Services a reality and fulfill all of their promises? Time will tell.

For the sake of simplicity, we have assumed the world of software development in the business world in neatly divided into two camps – Microsoft vs. Java. In any case, this view is sufficiently accurate to validate the analysis of the competing standards. It is apparent that Microsoft-based development will, on the whole, continue to be based on Microsoft technologies, while Java-based development will continue to use Java technologies after the introduction of the Web Services model. No surprises there.

A final nod must be made to those entities for which the above considerations do not apply. In companies where systems are contracted, there is more flexibility in choosing between the two since there is less risk in terms of current investment in one technology or the other. In absolute terms, the two platforms are suitable for somewhat different needs. Microsoft-based solutions are generally more suitable for smaller companies (up to SME), which need simpler lighter systems with less need for scalability. This is both in terms of the type of hardware supported, and in the level of flexibility available to modify low-level aspects of the platform for specific needs. The .NET platform is ideal for the creation of new systems due to the rapid development offered through Visual Studio .NET and as it naturally benefits from the last few years of distributed application research by virtue of being new.

It is, however, not especially suitable for integration, even with existing Microsoft-based solutions. Legacy applications are usually run on legacy systems, and it is therefore a considerable advantage for J2EE that the Java platform is hardware and OS independent. Integration on .NET is more a matter of controlled migration.

J2EE is also architected for enterprise systems and so is currently more scalable. The fact that a Microsoft solution is PC-based means that scalability comes through clustering and with this comes the increased complexity and cost of data management, synchronization (especially relating to transactions), and session management. While J2EE is often also clustered, this has been factored in at its inception, so it is generally less necessary to cluster J2EE servers, and when it is used each machine can handle more load so fewer machines can serve the same load as a cluster of Microsoft servers, thus simplifying the clustering process.

Finally, J2EE systems are more secure. This comes partly through Microsoft's focus on ease of use over security, and partly through their dependence on vast amounts of existing and ancient code. The recent drive for reducing bugs and improving security by overhauling the code base may help, which will go some way to revising the existing record of Microsoft with security problems.

The choice between the two systems should be based on business needs. This will certainly include considerations such as the prevalence of the systems with one or the other of the platforms. Companies with existing relationships with Microsoft and Microsoft-based software houses will no doubt find that it is more convenient and less costly to stay with this technology and it may be that the inconvenience of a completely new platform (that .NET represents) will be outweighed by the benefits of the countless improvements in software engineering that .NET also represents in addition to the close customer relationships that Microsoft keeps.

On the other hand, the flexibility that J2EE offers means that a company's systems will be more readily sensitive to meet changing business needs. Current investment in J2EE products (which may be as much as $15,000/processor) may also prohibit changing systems, even if there is a perceived need for it since many companies find that J2EE already meets most of their needs.

The choice will of course ultimately be dependent on the prevalent conditions in the company. The considerations above, however, should give you a number of pointers as to how each of these conditions should be weighted in making your decision.