Understanding Web Services

XML, WSDL, SOAP, and UDDI

Eric Newcomer

Independent Technology Guides
David Chappell, Series Editor

Preface

I first encountered XML as an integration technology in early 1998 during a visit to KPN Telecom in the Netherlands. The company was asking for proposals to help it develop an enterprise integration architecture based on the hub and spoke model, using XML as the canonical message format that would tie together the company's thousands of systems and hundreds of programming languages. My employer at the time, Compaq (Digital), did not win the project, but the controversial idea of using XML in a data-independent integration layer stuck with me. Now Web services are fulfilling that promise for everyone.

I joined IONA in the fall of 1999 and among other things soon began chairing the Object Management Group submitter's team drafting the XML Value specification, mapping XML to CORBA. In early 2000, I got involved in the new effort Microsoft was leading to define a distributed computing protocol for the Internet: SOAP. Previous attempts to promote the CORBA protocol had failed by then, and the W3C's own attempt, HTTP-NG, had also fallen flat. But the idea of serializing XML over HTTP seemed to hold promise for a solution.

IONA formally joined the SOAP effort in March 2000, before IBM joined and put the effort on the map. I worked with Andrew Layman, David Turner, John Montgomery, and others at Microsoft to bring IONA into the picture as a SOAP supporter and, in fact, as the first J2EE vendor to support SOAP. IONA demonstrated Web services interoperability at several Microsoft events during that year. The Microsoft presenter would introduce its SOAP Toolkit and demonstrate interoperability with a COM server. Then the IONA presenter was called on to describe how the same SOAP interface could interoperate with a Java server.

After that, I organized IONA's initial participation at W3C, supported the establishment of the XML Protocols Working Group, helped write the group charter, and began representing IONA at the XML Protocols Working Group, and more recently, at the Web Services Architecture Working Group. IONA has supported the submission of SOAP to W3C, WSDL, SOAP with Attachments, and XKMS. One thing led to another, and I eventually took on the responsibility of delivering IONA's implementation of Web services integration technologies.

In October 2000, I represented IONA at the UDDI kick-off meeting. It was then that I realized the potential for Web services technologies for application integration inside the firewall. Why not use SOAP, UDDI, and WSDL for internal projects? Then you could use the same approach for integration, regardless of whether it's inside the company or across the Internet.

David Vaskevitch presented at the UDDI conference, and this reminded me of the 1995 chapter in *The Future of Software* that I coauthored for Digital Equipment Corporation. David was author of the Microsoft chapter in that same book. In the Digital chapter, "The Key to the Highway," Peter Conklin and I compared the potential power of software standards to the impact of standards on the automobile. Standardized parts enabled mass production, which revolutionized the industry and society.

Today, software remains essentially a craft business, as automobiles were at the start of the twentieth century. Having widely adopted standards has remained elusive despite many attempts. We may be at the crossroads; Web services may finally do the trick.

I hope this book helps you understand what Web services are all about. If it serves as a decent introduction to the main ideas, concepts, and technologies, it will have done its job and find its place in the Web services community.

Acknowledgments

First of all, thanks to David Chappell for giving me the opportunity to contribute to his new series. David helped shape the organization, content, and overall approach of the book, which I greatly appreciated. The .NET information in this book is drawn primarily from David's book, which he was kind enough to share with me in advance of publication.

Second, I'd like to thank Steve Vinoski, who provided the most thorough and helpful review of the entire manuscript, commenting with equal emphasis on small details and big ideas. Qun (Joanna) Liang was tremendously helpful in providing and correcting examples in Chapter 2. Ben Bernhard and Daniel Kulp helped with examples for Chapters 3 and 4. Pyounguk Cho provided a helpful, last-minute review of Chapter 5.

Sean Baker, Vimal Kansal, Miloslav Nic, Jamie Osborne, Tom Sullivan, and Sanjiva Weerawarana reviewed the entire manuscript or large portions of it, offering many helpful comments and suggestions. Other people provided helpful reviews of specific portions of the manuscript on which they have expertise: Klaus-Dieter Naujok, ebXML; Igor Balabine, security; and Karsten Januszewski, UDDI.

I'd also like to thank the representatives of the vendors whose responses to the survey on Web services implementations are presented in Chapter 8 including: Christina Grenier and Terry Dwyer of BEA; Annrai O'Toole and Hugh Grant of Cape Clear; Joseph McGonnell and Mark Little of HP; Sanjiva Weerawarana and Heather Kreger of IBM; Rebecca Dias and Alex Roedling of IONA; Philip DesAutels and John Montgomery of Microsoft; Kuassi Mensah and Jeff Mischkinsky of Oracle; Peter Kacandes, Karen Shipe, and Peter Walker of Sun Microsystems; and Miloslav Nic and Ann Thomas-Manes of Systinet. Although they provided the original information and reviewed the text, any remaining errors are solely my responsibility.

Many thanks to the Addison-Wesley editorial and production staff, who made the preparation and finishing of the manuscript a truly professional, high-quality endeavor: Mary O'Brien, Alicia Carey, Marilyn Rash, Jacquelyn Doucette, and Evelyn Pyle.

Finally, I would really, really like to thank my wife, Jane, and kids, Erica and Alex—yes, really—for bearing with me and for understanding the time away.

Introduction

Web services are changing the way we think about distributed software systems, but there's a limit to what they can do. This book describes the core enabling technologies—WSDL, SOAP, and UDDI—and identifies where Web services begin and end and where existing technologies take over.

This book describes the concepts behind the basic Web services technologies, and it also includes chapters on ebXML, additional Web services technologies, and product implementations. The book is intended for IT professionals who are interested in understanding Web services, how they work, and what they are good for.

About Web Services

Web services provide a layer of abstraction above existing software systems, such as application servers, CORBA, .NET servers, messaging, and packaged applications. Web services work at a level of abstraction similar to the Internet and are capable of bridging any operating system, hardware platform, or programming language, just as the Web is.

Unlike existing distributed computing systems, Web services are adapted to the Web. The default network protocol is HTTP. Most existing distributed computing technologies include the communications protocol as part of their scope. With Web services, the communications protocol is already there, in the far-flung, worldwide Web.

New applications become possible when everything is Web service enabled. Once the world becomes Web service enabled, all kinds of new business paradigms, discussion groups, interactive forums, and publishing models will emerge to take advantage of this new capability.

Software and hardware vendors alike are rushing Web services products to market. The widespread adoption of the core standards represents a significant breakthrough in the industry. Applications can truly be built using a combination of components from multiple suppliers. Specialists are emerging to provide services in the areas of security, transaction coordination, bill processing, language translation, document transformation, registries and repositories, accounting, reporting, and specialized calculation. Applications being built anywhere, anytime, on any system can take advantage of prebuilt components, speeding time to market and reducing cost.

Meanwhile, ebXML, which chartered and maintains a separate course, continues to solve tough problems for corporate trading partners that are establishing automated supply chain purchasing and invoicing systems, large electronic document transfers, and business communities sharing common goals. The rightful heir to EDI, ebXML is providing an easier-to-use, lower-cost alternative to businesses automating their interactions with other businesses. With ebXML, a company's internal IT systems are connected to the IT systems of its trading partners, subcontractors, and business collaborators. The value inherent in these systems is therefore greatly increased, as they become essentially part of one large IT system, with essential information flowing freely across corporate boundaries rather than stuck within them.

Considerable overlap exists between the core Web services technologies and ebXML. Convergence between the two is based on their common adoption of SOAP as the transport and on the ability of the respective registries to share data. The ebXML specifications include many qualities-of-service requirements that are not yet included in Web services, such as message integrity and nonrepudiation, reliable messaging, business process flow, and protocol negotiation. Further convergence is possible as the core Web services technologies begin to adopt proposals in these additional technology areas.

Disagreement remains over the best approach to defining these additional technologies in the context of Web services. Once the core standards are adopted widely, the discussion moves up the stack to tackle quality-of-service issues. Security, transactions, process flow, and reliable messaging standards are needed, and some are further along than others.

The power of XML drives Web services technologies in general, whether it's the core standards, additional technologies, or ebXML. XML finally solves the problem of data independence for programming languages, middleware systems, and database management systems. Previously, data types and structures were specific to these types of software, and attempts at common definitions, such as CORBA IDL, gained limited acceptance. XML is well on its way to becoming as well established as its sibling, HTML.

The Web services technologies described in this book are all created using applications of XML in one way or another. XML is not one thing but rather a variety of technologies in itself, covering instance data as well as typing, structure, and semantic information associated with data. XML not only describes data independently but also contains useful information for mapping the data into and out of any software system or programming language.

Web services provide almost unlimited potential. Any program can be mapped to Web services, and Web services can be mapped to any program. Transformation of data to and from XML is essential, but XML is flexible enough to accommodate any data type and structure and even to

create new ones, if necessary. When all programs and software systems are finally Web service enabled, the world of distributed computing will be very different from what it is today.

About This Book

To provide a background and sufficient detail for practical understanding and use of these technologies, this book is organized into chapters on the main topics of interest.

Chapter 1, Introducing Web Services

This chapter highlights the most important aspects of Web services and what they can be used for, as well as contains a detailed overview of the entire book. Information is provided about the following:

- **XML** (Extensible Markup Language), the family of related specifications on which all Web services technologies are built
- **WSDL** (Web Services Description Language), providing the fundamental and most important abstraction of Web services, the interface exposed to other Web services and through which Web services are mapped to underlying programs and software systems
- **SOAP** (Simple Object Access Protocol), providing communications capability for Web services interfaces to talk to one another over the Internet and other networks
- **UDDI** (universal description, discovery, and integration), providing registry and repository services for storing and retrieving Web services interfaces
- **ebXML** (electronic business XML), an architecture and set of specifications designed to automate business process interaction among trading partners
- **Additional technologies,** going beyond the core Web services standards to meet requirements for security, reliable messaging, transaction processing, and business process flow so that more complex and critical business applications can use them
- **Vendor implementations,** providing a variety of implementations usually aligned with existing products but in some cases entirely new products for flexible and extensible Web services

Chapter 2, Describing Information: XML

The Extensible Markup Language (XML), like the Hypertext Markup Language, shares a common ancestry in the Standard Generalized Markup Language (SGML). One of the characteristics of SGML was the separation of format and content. Whether a document was produced for A4 or in letter format, for example, the format was described independently of the content of the document. The same document could therefore be output in multiple formats without changing the content. This principle of markup languages is applied to Web services through the separation of the document instance, which contains the data, and the schema, which describes the data structures and types, including semantic information useful for mapping the document to multiple programming languages and software systems.

XML represents a large number of specifications, many of which are more pertinent to document processing than to information processing. This chapter describes the XML specifications and technologies most important to Web services, which in general can be said to go "beyond markup" to provide facilities for structuring and serializing data. This chapter includes only those XML technologies relevant to Web services and explains how and what they are.

Chapter 3, Describing Web Services: WSDL

The Web Services Description Language (WSDL) provides the mechanism through which Web services definitions are exposed to the world and to which Web services implementers need to conform when sending SOAP messages. WSDL describes the data types and structures for the

Web services, explains how to map the data types and structures into the messages that are exchanged, and includes information that ties the messages to underlying implementations.

WSDL is defined so that its parts can be developed separately and combined to create a comprehensive WSDL file. The data types and structures can be shared among multiple messages, as can the definition of the services exposed within the interface. WSDL lists the interfaces and, within an interface, associates each service with an underlying implementation.

In order to achieve communication for Web services, WSDL maps them onto communication protocols and transports. Both parties in a Web services interaction share a common WSDL file. The sender uses the WSDL file to generate the message in the appropriate format and to use the appropriate communication protocol. The receiver uses the WSDL file to understand how to receive and parse the message and how to map it onto the underlying object or program.

Chapter 4, Accessing Web Services: SOAP

Once an interface is defined for them, Web services need a way to communicate with one another and to exchange messages. The Simple Object Access Protocol (SOAP) defines a common format for XML messages over HTTP and other transports. SOAP is designed to be a simple mechanism that can be extended to encompass additional features, functionalities, and technologies.

This chapter describes the parts of SOAP and the purpose of each. SOAP is a one-way asynchronous messaging technology that can be adapted and used in a variety of message-passing interaction styles: remote procedure call (RPC) oriented, document oriented, and publish and subscribe, among others.

If anything defines the minimum criterion for a Web service, it must be adherence to SOAP. SOAP messaging capability is fundamental to Web services. SOAP is defined at a very high level of abstraction and can be mapped to any number of underlying software systems, including application servers, .NET servers, middleware systems, database management systems, and packaged applications.

The chapter describes the required and optional parts of SOAP, explains how SOAP messages are processed, and discusses the role of intermediaries in SOAP message processing. Background information on the specification is provided, as are examples of the major SOAP parts. An explanation of SOAP with Attachments is also included.

Chapter 5, Finding Web Services: UDDI Registry

The initiative for universal description, discovery, and interoperability (UDDI) produces specifications and an active implementation of a repository for Web services descriptions. The UDDI registry can be searched using various categorization criteria to obtain contact information for businesses offering services of interest. UDDI provides a publicly accessible means to store and retrieve information about Web services interfaces and implementations.

This chapter describes the UDDI data formats and the SOAP APIs used to store and retrieve information from UDDI. This chapter also provides background on the UDDI organization that sponsors the physical registry and the process by which UDDI specifications and technologies are moving toward adoption.

The Web needs something like UDDI to provide a clearinghouse for Web services information so that publishers and consumers can find each other. Only then can the true value of Web services be realized: when Web services consumers can easily and quickly locate and begin accessing Web services implementations anywhere in the world.

Chapter 6, An Alternative Approach: ebXML

The electronic business XML (ebXML) initiative was started around the same time that the Web services community began to grow. For the first several months, ebXML was an entirely separate and parallel effort. Many of the goals of ebXML are common to Web services, and many of the technologies overlap in concept. In general, however, ebXML is focused more at the industrial or enterprise computing level, addressing as the top goal the issue of business process definition.

This chapter explains the background, goals, and origin of ebXML and covers the ebXML architecture in detail. Individual specifications are described and placed into their proper context within the overall architecture.

The ebXML architecture includes many of the same things as the core Web services technologies but goes beyond them in defining quality-of-service requirements for reliable messaging, security, and trading-partner negotiation. Beginning as a replacement for EDI, ebXML seeks to avoid some of the problems with EDI implementations and acceptance, especially in smaller businesses.

Chapter 7, Web Services Architecture: Additional Technologies

After the core Web services technologies are implemented and adopted, a whole range of additional technologies is needed to enable Web services to address complex and critical application requirements. Businesses will need to secure their Web services against unauthorized use, to guarantee that their SOAP messages arrive at their intended destinations and are processed reliably, and to define and execute automated business process flows according to a standard mechanism.

This chapter describes these and other technologies in the context of the vendor and industry initiatives in which they are likely to progress toward adoption. In some cases, competing proposals vie for adoption, and the leading candidates are discussed. The chapter also includes descriptions of two technologies on which many Web services concepts are based: RosettaNet and XML-RPC.

Chapter 8, Implementing Web Services

Web services specifications and technologies are not meaningful or particularly useful without implementations in software vendor products. This chapter summarizes the major architectural approaches to Web services implementation, describes the major development communities of .NET and J2EE, and presents information supplied by BEA Systems, Cape Clear, IBM, IONA, Microsoft, Hewlett-Packard, Oracle, Sun Microsystems, and Systinet on their current product offerings and views of the future.

Some vendors tend to view Web services implementations primarily within the context of their existing products, as additional clients or adapters into and out of the existing application servers, database management systems, and middleware systems. Other vendors seek to mine the value of the Web services layer itself, where multiple, disparate software system domains are put into relationship and integrated. Other vendors offer products in multiple categories, including some aimed purely at providing an implementation of Web services standards as well as some aimed at exposing Web services interfaces to existing products.

Although vendors tend to agree on the adoption and wide spread implementation of the core standards, very little, if any, agreement exists at the next level; that is, what should come next. Security, transactions, process flow, and reliable messaging are all part of various vendors' plans but in somewhat differing orders and levels of importance.

Chapter 1. Introducing Web Services

Like the effect of rail transportation on national economic systems, Web services are fundamentally changing the rules of Web commerce. They connect programs to each other across distant points on the global map, transporting large amounts of data more efficiently and cheaply than ever before. The result is faster, better, and more productive communication for businesses and consumers alike.

Web services are changing everything

The Web started out supporting human interactions with textual data and graphics. People use the Internet daily to look up stock quotes, buy consumer goods, and read the latest news. This level of interaction is fine for many purposes. But the essentially text-based Web does not support software interactions very well, especially transfers of large amounts of data. A more efficient method is needed that allows applications to interact directly with one another, automatically executing instructions that would otherwise have to be entered manually through a browser.

Individuals and companies doing business over the Web need a way to publish links to their applications and data, in much the same way that they publish links to their Web pages. Internet-based applications need to be able to find, access, and automatically interact with other Internet-based applications. Web services improve Internet use by enabling program-to-program communication. Through the widespread adoption of Web services, applications at various Internet locations can be directly integrated and interconnected as if they were part of a single, large IT (information technology) system.

The current Web does not support software-oriented interactions very well

The Basics of Web Services

Web services are Extensible Markup Language (XML) applications mapped to programs, objects, or databases or to comprehensive business functions. Using an XML document created in the form of a message, a program sends a request to a Web service across the network, and, optionall, receives a reply, also in the form of an XML document. Web services standards define the format of the message, specify the interface to which a message is sent, describe conventions for mapping the contents of the message into and out of the programs implementing the service, and define mechanisms to publish and to discover Web services interfaces.

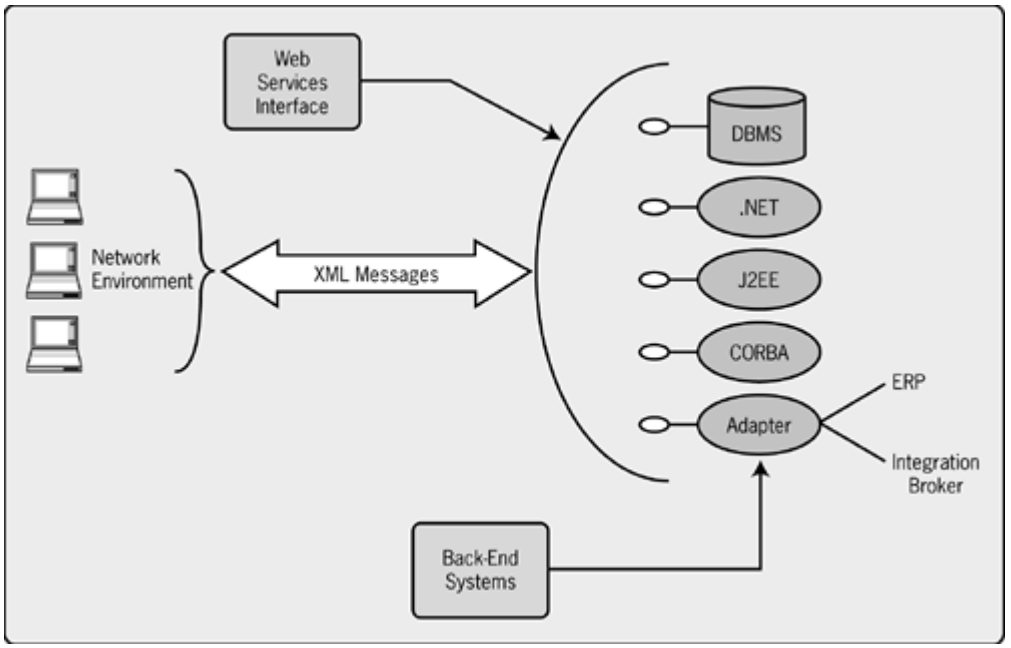
Web services transform XML documents into and out of IT systems

This technology can be used in many ways. Web services can run on desktop and handheld clients to access such Internet applications as reservations systems and order-tracking systems. Web services can also be used for business-to-business (B2B) integration, connecting applications run by various organizations in the same supply chain. Web services can also solve the broader problem of enterprise application integration (EAI), connecting multiple applications from a single organization to multiple other applications both inside and outside the firewall. In all these cases, the technologies of Web services provide the standard glue connecting diverse pieces of software.

Web services can be used in many applications

As illustrated in Figure 1-1, Web services *wrap*, presenting to the network a standard way of interfacing with back-end software systems, such as database management systems, .NET, J2EE (Java2 Platform, Enterprise Edition), or CORBA (common object request broker architecture), objects, adapters to enterprise resource planning (ERP) packages, integration brokers, and others. Web services interfaces receive a standard XML message from the networking environment, transform the XML data into a format understood by a particular back-end software system, and, optionally, return a reply message. The underlying software implementations of Web services can be created by using any programming language, operating system, or middleware system.

Figure 1-1. Web services interface with back-end systems.



Web services combine the execution characteristics of programmatic applications with the abstraction characteristics of the Internet. Today's Internet technologies succeed in part because they are defined at a sufficiently high level of abstraction to enable compatibility with any operating system, hardware, or software. The Web services-based Internet infrastructure exploits this abstraction level and includes semantic information associated with data. That is, Web services define not only the data but also how to process the data and map it into and out of underlying software applications.

Web services combine programming and Web concepts

A Simple Example: Searching for Information

Today, most services are invoked over the Web by inputting data into HyperText Markup Language (HTML) forms and sending the data to the service, embedded within a uniform resource locator (URL) string:

```
http://www.google.com/search?q=Skate+boots&btnG=Google+Search
```

This example illustrates how simple Web interactions, such as a search, a stock purchase, or a request for driving directions, are accessed over the Web by embedding parameters and keywords

in a URL. In this example, entering a simple search request for `Skate boots` into the Google search engine results in the URL shown. The `search` keyword represents the service being requested over the Web, whereas the `Skate+boots` keywords represent the search string entered into the HTML form displayed by the Google Web site. The Google search service then passes the request to a series of other search engines, which return lists of URLs to pages with text matching the search keywords `Skate+boots`. This inefficient way of searching the Web depends entirely on matching the given text strings to cataloged HTML pages.

XML provides a great many advantages for transmitting data across the Internet. Now the preceding request can be contained in an XML document instead:

```
<SOAP-ENV:Body>
  <s:SearchRequest
  xmlns:s="www.xmlbus.com/SearchService">
    <p1>Skate</p1>
    <p2>boots</p2>
    <p3>size 7.5</p3>
  </s:SearchRequest>
</SOAP-ENV:Body>
```

XML is a better way to send data

Sending the request within an XML document has many advantages, such as improved data typing and structure, greater flexibility, and extensibility. XML can represent structured and typed data—the `size` field can be typed as a decimal string or as a floating point, for example—and can contain a larger amount of information than is possible within a URL string.

Web services use XML documents

This example is shown in the form of a Simple Object Access Protocol (SOAP) message, a standard form of XML messaging and one of the major enabling technologies in the Web services foundation (see Chapter 4). In SOAP messages, the name of the service request and the input parameters take the form of XML elements. The example also illustrates the use of XML namespaces (`xmlns:`), another critical element of Web services (see Chapter 2). Because XML documents support data typing, complex structures, and the association of XML schemas, modern Web services technology provides significant advantages over existing URL and HTML capabilities for accessing software applications.

The Next Generation of the Web

The next generation of the Web will be based on software-oriented interactions

Web services are aimed at putting the vast global network of the Web, established for human interaction, to an entirely new purpose. Software-oriented interactions will automatically perform operations that previously required manual intervention, such as

- Searching for and buying goods and services at the best price
- Coordinating travel tickets and restaurant tables for a given date

- Streamlining business procurement, invoicing, and shipping operations

The next generation of the Web will use software-oriented services to interoperate directly with applications built using any combination of objects, programs, and databases.

But Web services are not only about interfaces to objects, programs, middleware, and databases for access over the Internet. By combining a series of Web services into a larger interaction, Web services also provide the means to perform new types of interactions.

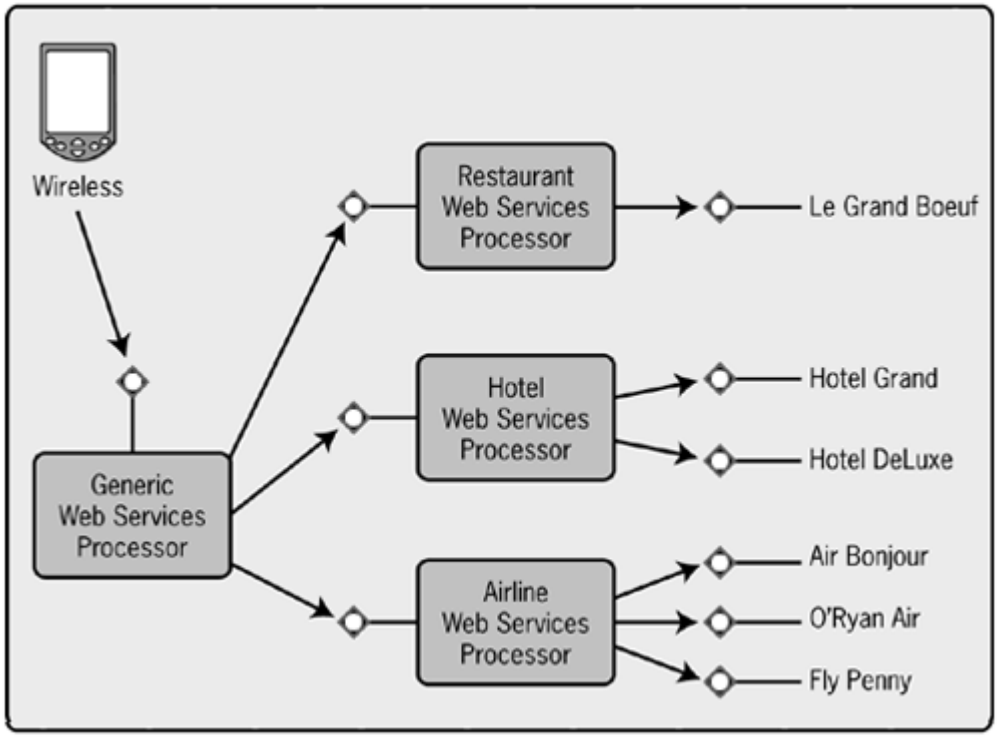
Suppose, for instance, that you live in San Francisco and wish to reserve a table at your favorite Paris restaurant and then make the necessary travel arrangements to be there at the agreed time. Today, you would have to call the restaurant directly to get the reservation, taking into account the 9-hour time difference and the language difference, and then call a travel agent to find a compatible flight and a hotel. But using Web services, you could schedule the dinner with your personal digital assistant (PDA) calendar and click on a button to automatically reserve a table at a convenient time. Once the reservation was made, the Web service could kick off other services that would book a cheap flight and reserve a room at a nearby four-star hotel.

Web services enable new types of interactions

Figure 1-2 shows how Web services can interact with a PDA connected to a wireless Web services processor to book a reservation at a favorite restaurant, using the restaurant's Web service.[1] The Web services processor accepts requests from the calendar function of the PDA and discovers Web services related to extended calendar functions, such as reserving a restaurant table. After successfully reserving a table, the Web services processor contacts Web services for hotel and flight reservations to complete the requested scheduling action.

[1] Throughout the book, the diamond-on-a-stick convention is used to represent a Web services interface.

Figure 1-2. Applications can use Web services to book a restaurant table and make hotel and flight reservations.

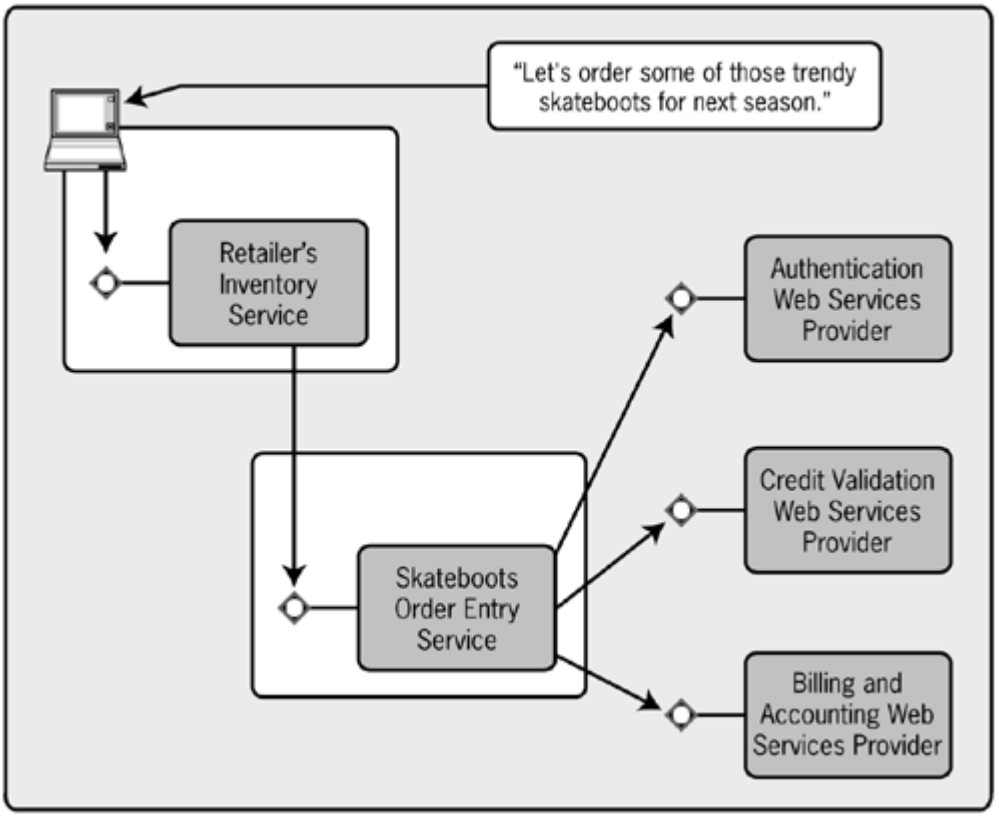


Web services are also very useful for discovering and interacting with Internet sites that provide online order entry systems, such as the one for the Skateboots Company's trendy skateboots— a boot with a retractable ice skate built in—like the ones that Batman and Robin used in the movie *Batman and Robin.* Sporting goods retailers interested in stocking the boots, this year's hot new item, can use Web services to place advance bulk orders in batch, to check the status of an order, or to place in-season restocking orders and be immediately notified of back orders, if the manufacturer is out of stock. Web services building blocks provide standard components of the application for the Skateboots Company, which isn't large enough to host its own entire application infrastructure. Web services hosting companies provide security services to ensure that Skateboots accepts orders only from approved retailers and to provide credit validation services for approving bulk advance orders. Still other companies help Skateboots by providing electronic funds collection and accounting services.

Web services discover and interact with one another

The entire Skateboots order entry system is exposed to the Internet as a Web service, but behind the top-level Web service are a number of other Web services working together to provide the necessary functionality. Figure 1-3 illustrates how Web services can change the way business applications are built and used. The retailer interested in stocking skateboots inputs a request to its local inventory management service, which is exposed to the shop computers as a Web service. The local inventory service then contacts the manufacturer's Web service over the Internet and sends the order for the correct number of skateboots, based on available shelf space and the most popular sizes.

Figure 1-3. The Skateboots order entry service comprises several other Web services.



The Skateboots Company's order entry system comprises multiple Web services, including a custom-built part that deals with the unique aspects of its product and several commodity parts that take care of standard functions, such as authenticating the user, credit authorization, and accounting and billing, all hosted by other companies that specialize in providing such services over the Internet.

Creating business applications using Web services entails putting into proper relationship a number of other Web services, which can be implemented by using any combination of programming language, operating system, or packaged software, inside or outside the firewall. (This is also the way in which Web services solve the difficult EAI problem.) In establishing the proper relationship, or flow, of related Web services, it also automates the corresponding business processes and procedures.

Through the widespread adoption of Web services, the Internet is becoming more efficient, especially for business interactions. In the next generation of the Web, Web services building blocks will enable automatic Internet interactions, combining direct access to software applications and business documents, bypassing familiar text-based Web pages to access software-based data directly. Furthermore, fundamental Web services building blocks are very likely to be hosted and published by a variety of companies focusing on a specific functional component, such as authentication, transactional coordination, or accounting and billing. This change to direct application-to-application interaction over the Web lies at the heart of Web services, what they mean, and how they work.

Web services create greater commercial efficiencies

Toward a Common Understanding

Web services technology exists at a sufficiently high level of abstraction to support multiple simultaneous definitions, which are sometimes contradictory. At the simplest level, Web services can be thought of as Internet-oriented, text-based integration adapters. Any data can be mapped into and out of ASCII text, and this type of mapping has long been the lowest common denominator for graphical display systems and database management systems. If all else fails, the saying goes, map the data to text. Text-based systems also are behind the success of the World Wide Web, on which the additional abstraction of Web services is based. Any computer or operating system is capable of supporting HTML and Web servers, and browsers, and when they download files, they don't care or even know what type of back-end systems they're interacting with.

The same is true for Web services, which often leads to a lot of confusion when developers of traditional, or established, computing environments try to understand Web services technology in reference to a single type of distributed software system, such as CORBA, J2EE, or .NET. Because Web services are much more abstract—more like adapters than they are like interfaces—it will be some time before the industry settles on truly common definitions and conventions for them.

Interacting with Web Services

The level of abstraction at which Web services operate encompasses such interaction styles as RPC (remote procedure call) emulation, asynchronous messaging, one-way messaging, broadcast,

and publish/subscribe. Most major database management systems, such as Oracle, SQL Server, and DB2, support XML parsing and transformation services, allowing direct interaction between Web services and database management systems. Middleware vendors typically also provide a mapping of Web services to their software systems, such as application servers and integration brokers. To the user, therefore, interactions with Web services can appear as batch or online interactions, supporting synchronous or asynchronous communications patterns, and as user interfaces written using Java programs, VB (Visual Basic) programs, office applications, browsers, or thick clients to database management systems, to name a few, and can map down to any type of underlying software system.

Web services support multiple messaging paradigms

Web services standards and technologies generally encompass two major types of application interaction patterns:

- Remote procedure call (online)
- Document oriented (batch)

Web services encompass RPC and document-oriented interactions

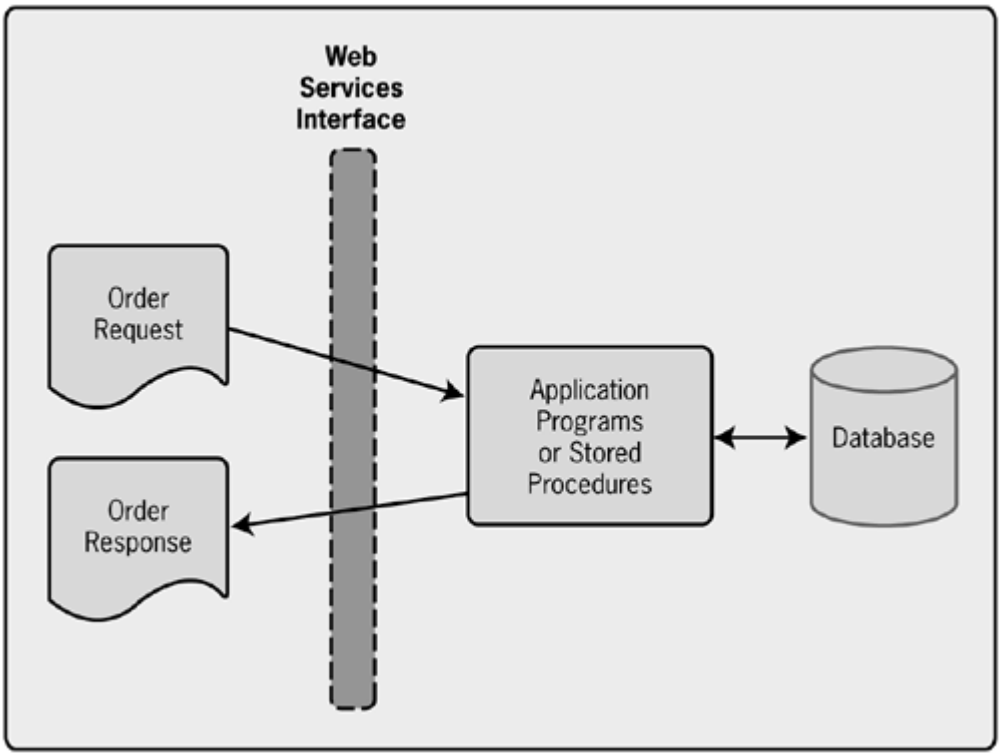
These two types of interactions are described in the following subsections.

RPC-Oriented Interactions

In RPC-oriented interactions, the Web services request takes the form of a method or a procedure call with associated input and output parameters. In contrast to the document-oriented interaction, the RPC-oriented interaction sends a document formatted specifically to be mapped to a single logical[2] program or database, as shown in Figure 1-4. Because the "real-time" or in-season order for skateboots depends on available inventory, for example, the program accesses the database to check the available supply of the ordered item. If everything is OK, the program returns an XML document to the distributor in the request/response format to indicate that the order has been accepted and will be shipped. If supply isn't available, the return message indicates a back order or rejects the order entirely. In contrast to the document-oriented interaction style, the request and the reply are modeled as synchronous messages. That is, the application sending the message waits for a response.

[2] A single logical program can, of course, comprise multiple subprograms.

Figure 1-4. This Web service supports an interactive order request/response.

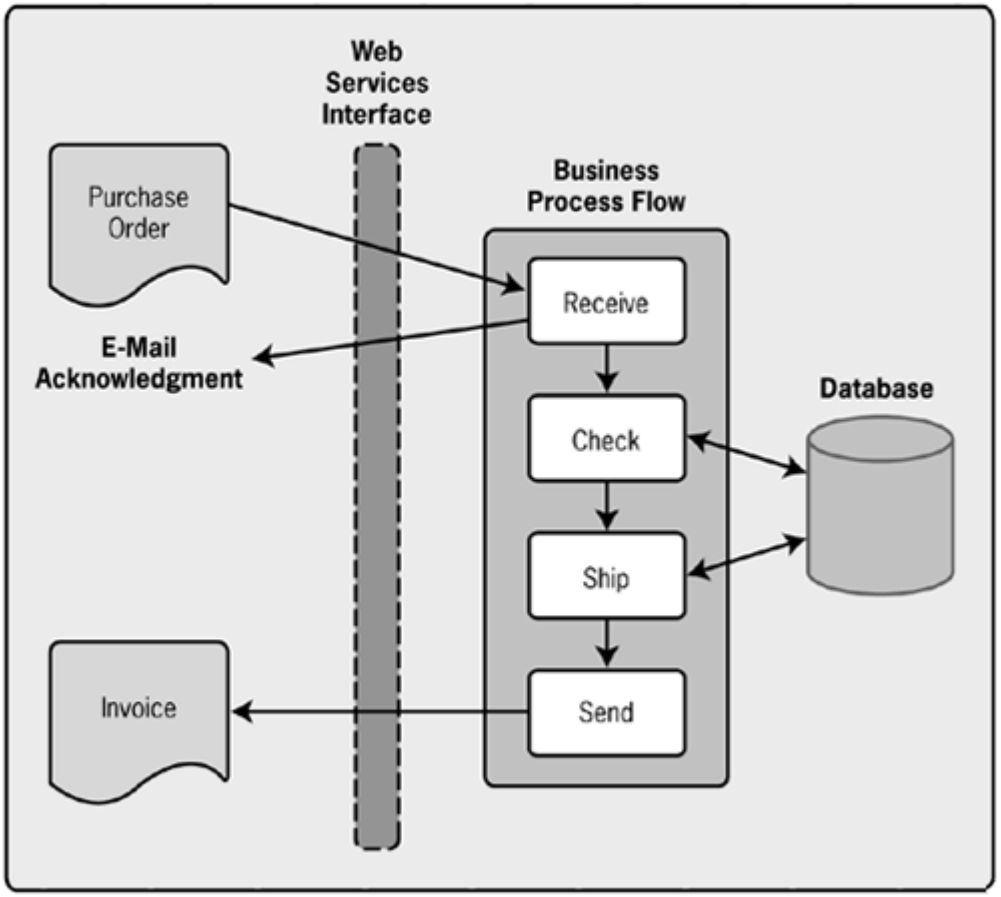


RPC-oriented interactions are good for brief data exchanges

Document-Oriented Interactions

In the document-oriented interaction style, the Web service request takes the form of a complete XML document that is intended to be processed whole. For example, a Web service that submits a complete purchase order, such as a preseason order for skateboots, would submit the entire bulk order to the manufacturer at once, as shown in Figure 1-5. This is like submitting a message to a queue for asynchronous processing. The manufacturer would typically send an e-mail or other form of acknowledgment to the retailer to indicate that the order was received and would be processed according to a predefined flow of execution. The flow might include such steps as checking the database for previous orders from the same retailer to ensure that it is not exceeding its credit limit or agreed capacity or scheduling a ship date for the order. In a real process flow, of course, many more steps are likely before the order is shipped and the invoice sent out, but the example shows only the final step: sending the XML invoice to the distributor for payment after the order has been shipped and received.

Figure 1-5. This Web service processes a complete purchase order.



The document-oriented style is good for bulk data exchanges

Document-oriented interactions often assume that the parties to a Web services conversation have agreed to share a common business document, such as a purchase order, a ship bill, or an invoice. These parties are often identified as trading partners, or collaborating partners. Trading partners also typically agree on a common process flow, or interaction pattern, for exchanging the shared document, such as requiring an acknowledgment on receipt of a purchase order, returning specific status information in reply to an order query, or sending an e-mail alert when an order has been shipped. During the execution of the business process, a complete document might be exchanged. If the document is already held in common, fragments of information required to fill in specific sections of the shared document, such as purchase price or promised delivery date, might be exchanged.

Trading-partner agreements determine required interactions

In the Skateboots Company example, preseason bulk orders are handled by using purchase orders submitted in batches according to predefined terms and conditions that help the manufacturer plan capacity. During the season, immediate restocking orders are handled by more interactive services that depend on filling orders from available inventory and that can immediately identify back orders. Thus Skateboots.com provides Web services supporting both major types of interaction.

The two styles map well to synchronous/asynchronous messaging paradigms

The Technology of Web Services

Programs that interact with one another over the Web must be able to find one another, discover information allowing them to interconnect, figure out what the expected interaction patterns are—a simple request/reply or more complicated process flow?—and negotiate such qualities of service as security, reliable messaging, and transactional composition. Some of these qualities of service are covered in existing technologies and proposed standards, but others are not. In general, the Web services community is working to meet all these requirements, but it's an evolutionary process, much like the Web itself has been. Web services infrastructure and standards are being designed and developed from the ground up to be extensible, such as XML and HTML before them, so that whatever is introduced in the short term can continue to be used as new standards and technologies emerge.

Standards define how Web services are described, discovered, and communicate with one another

The New Silver Bullet?

Web services are sometimes portrayed as "silver-bullet" solutions to contemporary computing problems, filling the role previously played by the original Web, relational databases, fourth-generation languages, and artificial intelligence. Unfortunately, Web services by themselves can't solve much. Web services are a new layer—another way of doing things—but are not a fundamental change that replaces the need for existing computing infrastructure. This new layer of technology performs a new function—a new way of working—but, most importantly, provides an integration mechanism defined at a higher level of abstraction.

Web services are important because they are capable of bridging technology domains, not because they replace any existing technology. You could say that newer languages, such as Visual Basic, C#, C/C++ and Java—replace older languages, such as COBOL and FORTRAN, although a lot of programs in those languages are still around, as are Web-services mappings for them. Web services, like Web servers, are complementary to, not in conflict with, existing applications, programs, and databases. Application development continues to require Java, VB, and C#. All that's new is a way of transforming data in and out of programs and applications, using standard XML data formats and protocols to reach a new level of interoperability and integration.

Developers may have to take Web services into account when designing and developing new programs and databases, but those programs and databases will still be required behind Web services wrappers. Web services are not executable things in and of themselves; they rely on executable programs written using programming languages and scripts. Web services define a powerful layer of abstraction that can be used to accomplish program-to-program interaction, using existing Web infrastructure, but they are nothing without a supporting infrastructure.

Web services require several related XML-based technologies to transport and to transform data into and out of programs and databases.

Web services require the use of several related XML-based technologies

- **XML (Extensible Markup Language),** the basic foundation on which Web services are built provides a language for defining data and how to process it. XML represents a family of related specifications published and maintained by the World Wide Web Consortium (W3C) and others.
- **WSDL (Web Services Description Language),** an XML-based technology, defines Web services interfaces, data and message types, interaction patterns, and protocol mappings.
- **SOAP (Simple Object Access Protocol),** a collection of XML-based technologies, defines an envelope for Web services communication—mappable to HTTP and other transports—and provides a serialization format for transmitting XML documents over a network and a convention for representing RPC interactions.
- **UDDI (Universal Description, Discovery, and Integration),** a Web services registry and discovery mechanism, is used for storing and categorizing business information and for retrieving pointers to Web services interfaces.

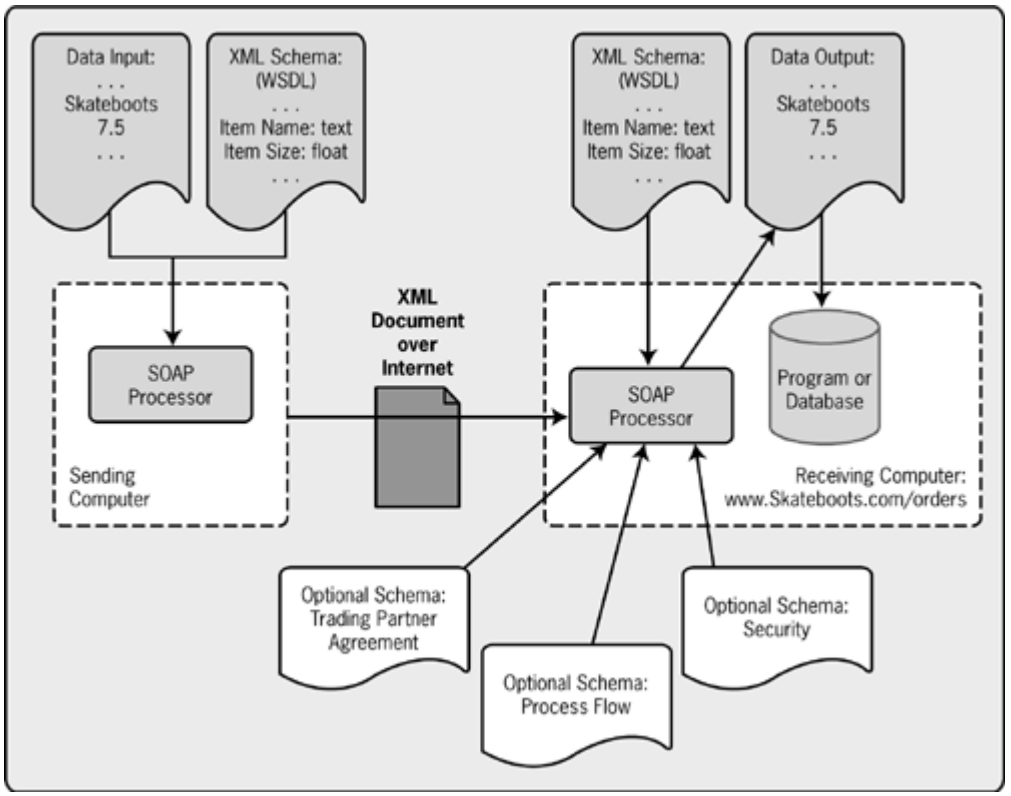
Usage Example

The basic Web services standards are used together. Once the WSDL is obtained from the UDDI or other location, a SOAP message is generated for transmission to the remote site.

Web services standards are typically used together

As shown in Figure 1-6, a program submitting a document to a Web service address uses an XML schema of a specific type, such as WSDL, to transform data from its input source—a structured file in this example—and to produce an XML document instance in the format consistent with what the target Web service expects, as described in the same WSDL file. The WSDL file is used to define both the input and the output data transformations.

Figure 1-6. Web services use XML documents and transform them into and out of programs.



The sending computer's SOAP processor transforms the data from its native format into the predefined XML schema data types contained in the WSDL file for text, floating point, and others, using mapping tables. The mapping tables associate native data types with corresponding XML schema data types. (Standard mappings are widely available for Java, Visual Basic, CORBA, and other commonly used type systems. Many XML mapping tools are available for defining custom or special mappings.) The receiving computer's SOAP processor performs the transformation in reverse, mapping from the XML schema data types to the corresponding native data types.

The URL, in widespread use on the Web, points to a TCP (Transmission Control Protocol) address containing a Web *resource.* Web services schemas are a form of Web resource, contained in files accessible over the Internet and exposed to the Web using the same mechanism as for downloading HTML files. The major difference between HTML file downloading and accessing Web services resources is that Web services use XML rather than HTML documents and rely on associated technologies, such as schemas, transformation, and validation, to support remote communication between applications. But the way in which Web services schemas are published and downloaded is the same: an HTTP operation on a given URL.

Web service description files are typically posted using URLs

When it receives a document, a Web service implementation must first parse the XML message and validate the data, perform any relevant quality-of-service checking, such as enforcing security policies or trading-partner agreements, and execute any business process flow associated with the document. The Web service at the fictional skateboots.com Web site is located in the `skateboots.com/order` folder, which is what the URL points to.[3]

[3] A more generic term, the Uniform Resource Indicator (URI), often appears in Web services specifications in place of the URL. A URI is a categorical syntax term that includes the Uniform Resource Locator (URL) and the Uniform Resource Name

(URN). A URN is a name that does not reference a physical resource. In practice, there is little difference between URI and URL.

Web services use XML schemas to validate messages

The Web services available at this Internet address are identified within a public WDSL file that can be downloaded to the sending computer and used to generate the message. The Skateboots Company also posted a listing in the public UDDI directory, pointing to the same WSDL file, for customers who might discover the company through the UDDI service. In general, anyone wishing to interact with the Web services that place or track orders for the Skateboots Company over the Web must find a way to obtain and to use that particular WSDL file to generate the message.

Programs at the skateboots.com address provide an HTTP listener associated with the Web services in order to recognize the XML messages sent in the defined format. The programs include XML parsers and transformers and map the data in the SOAP message into the formats required by the Skateboots Company order entry system.

These technologies are enough to build, deploy, and publish basic Web services. In fact, even basic SOAP is enough. Other technologies are continually being added to the expanding Web services framework as they emerge. These fundamental technologies are enough to support use of the Internet for basic business communication and to bridge disparate IT domains, however; and this form of Web interaction is being adopted very quickly.

Web services technologies are evolving from a basic framework

Over time, as standards for registry, discovery, and quality of service mature, the vision of an ad hoc, dynamic business Web will start to take hold, and Web services will begin to operate more like the current Web, allowing companies to find and to trade with one another purely by using Internet-style communications. In the meantime, the basic Web services technologies and standards covered in this book are sufficient for many solutions, such as integrating disparate software domains—J2EE and .NET, for example—connecting to packaged applications, such as SAP and PeopleSoft, and submitting documents to predefined business process flows.

XML: The Foundation

In the context of Web services, XML is used not only as the message format but also as the way in which the services are defined. Therefore, it is important to know a little bit about XML itself, especially within the context of how it is used to define and to implement Web services.

XML is used for multiple purposes

Reinventing the Wheel

Some people say that Web services are reinventing the wheel because they share many characteristics with other distributed computing architectures, such as CORBA or

DCOM. Web services do share considerable common ground with these and other distributed computing architectures and implementations, but there's also a good reason for inventing a new architecture. The Web is established, and to take advantage of this tremendous global network, the concepts of distributed computing need to be adapted. First, the Web is basically disconnected; that is, connections are transient and temporary. Distributed computing services, such as security and transactions, traditionally depend on a transport-level connection and have to be redesigned to provide equivalent functionality for the disconnected Web. Second, the Web assumes that parties can connect without prior knowledge of one another, by following URL links and observing a few basic rules. For Web services, this means that any client can access Web services published by anyone else, as long as the information about the service—the schema—is available and understandable and XML processors are capable of generating messages conforming to the schema.

Traditional distributed computing technologies assume a much more tightly coupled relationship between client and server and therefore cannot inherently take advantage of the existing World Wide Web. Because Web services adopt the publishing model of the Web, it's possible to wrap and to publish a specific end point, or business operation, using a Web services interface definition, without requiring a specific type of client for that end point. The paradigm shift that clients can develop and integrate later has many advantages in solving the problem of enterprise integration.

Purposes of XML

XML was developed to overcome limitations of HTML, especially to better support dynamic content creation and management. HTML is fine for defining and maintaining static content, but as the Web evolves toward a software-enabled platform, in which data has associated meaning, content needs to be generated and digested dynamically. Using XML, you can define any number of elements that associate meaning with data; that is, you describe the data and what to do with it by using one or more elements created for the purpose. For example:

```
<Company>
  <CompanyName region="US">
  Skateboots Manufacturing
  </CompanyName>
  <address>
    <line>
    200 High Street
    </line>
    <line>
    Springfield, MA 55555
    </line>
    <Country>
    USA
    </Country>
  </address>
  <phone>
  +1 781 555 5000
  </phone>
</Company>
```

XML allows any number of elements to be defined

In this example, XML allows you to define not only elements that describe the data but also structures that group related data. It's easy to imagine a search for elements that match certain criteria, such as `<Country>` and `<phone>` for a given company, or for all `<Company>` elements and to return a list of those entities identifying themselves as companies on the Web.

Furthermore, as mentioned earlier, XML allows associated schemas to validate the data separately and to describe other attributes and qualities of the data, something completely impossible using HTML.

Of course, significant problems result from the great flexibility of XML. Because XML allows you to define your own elements, it's very difficult to ensure that everyone uses the same elements in the same way to mean the same thing. That's where the need for mutually agreed on, consistent content models comes in.

XML schemas constrain flexibility

Two parties exchanging XML data can understand and interpret elements in the same way only if they share the same definitions of what they are. If two parties that share an XML document also share the same schema, they can be sure to understand the meaning of the same element tags in the same way. This is exactly how Web services work.

Technologies

XML is a family of technologies: a data markup language, various content models, a linking model, a namespace model, and various transformation mechanisms. The following are significant members of the XML family used as the basis of Web services:

- **XML v1.0:** The rules for defining elements, attributes, and tags enclosed within a document root element, providing an abstract data model and serialization format
- **XML schema:** XML documents that define the data types, content, structure, and allowed elements in an associated XML document; also used to describe semantic-processing instructions associated with document elements
- **XML namespaces:** The uniquely qualified names for XML document elements and applications

Several members of the XML family are used in Web services

The Future of the Web

The inventor of the World Wide Web, Tim Berners-Lee, has said that the next generation of the Web will be about data, not text; XML is to data what HTML is to text. The next generation of the Web is intended to address several shortcomings of the existing Web, notably the difficulty searching the Web for exact matches on text strings embedded in Web pages. Because the Web has been so successful, however, the future of the Web must be accomplished as an extension, or an evolution, of the current Web. It's impossible to replace the entire thing and start over! Solutions for application-to-application communication must be derived from existing Internet technologies.

If the future of the Web depends on its ability to support data communications as

effectively and easily as it supports text communications, Web services need to be able to refer dynamically to Web end points, or addresses (URLs), and to map data to and from XML transparently. These end points, or addresses, provide the services that process the XML data, in much the same way that browsers process HTML text. These addresses also can be included in any program capable of recognizing a URL and parsing XML. Thus it will be possible to communicate from your spreadsheet to a remote source of data or from your money management program to your bank account management application, make appointments with colleagues for meetings, and so on.

Microsoft and others are already developing these kinds of standard services accessible from any program, and a large part of Microsoft's .NET strategy is focused on development tools for creating and stitching together applications that use predefined Web services. But getting this to happen requires significant standardization, comparable to the effort involved in standardizing PC components, and might therefore not happen for several years.

- **XML Information Set:** A consistent, abstract representation of the parts of an XML document
- **XPointer:** A pointer to a specific part of a document; **XPath,** expressions for searching XML documents; and **XLink,** for searching mulitple XML documents
- **Extensible Stylesheet Language Transformations (XSLT):** Transformation for XML documents into other XML document formats or for exporting into non-XML formats
- **DOM (Document Object Model) and SAX (Simple API for XML):** Programming libraries and models for parsing XML documents, either by creating an entire tree to be traversed or by reading and responding to XML elements one by one

These technologies and others are described in further detail in Chapter 2.

WSDL: Describing Web Services

The Web Services Description Language (WSDL) is an XML schema format that defines an extensible framework for describing Web services interfaces. WSDL was developed primarily by Microsoft and IBM and was submitted to W3C by 25 companies.[4] WSDL is at the heart of the Web services framework, providing a common way in which to represent the data types passed in messages, the operations to be performed on the messages, and the mapping of the messages onto network transports.

[4] To date, 25 is the highest number of companies to join any W3C submission. A submission is a specification proposed for adoption by W3C—see www.w3.org/Submission/2001/07.

WSDL is, like the rest of the Web services framework, designed for use with both procedure-oriented and document-oriented interactions. As with the rest of the XML technologies, WSDL is so extensible and has so many options that ensuring compatibility and interoperability across differing implementations may be difficult. If the sender and the receiver of a message can share and understand the same WSDL file the same way, however, interoperability can be ensured.

WSDL is the XML format that describes what a Web service consists of

WSDL is divided into three major elements:

- Data type definitions

- Abstract operations
- Service bindings

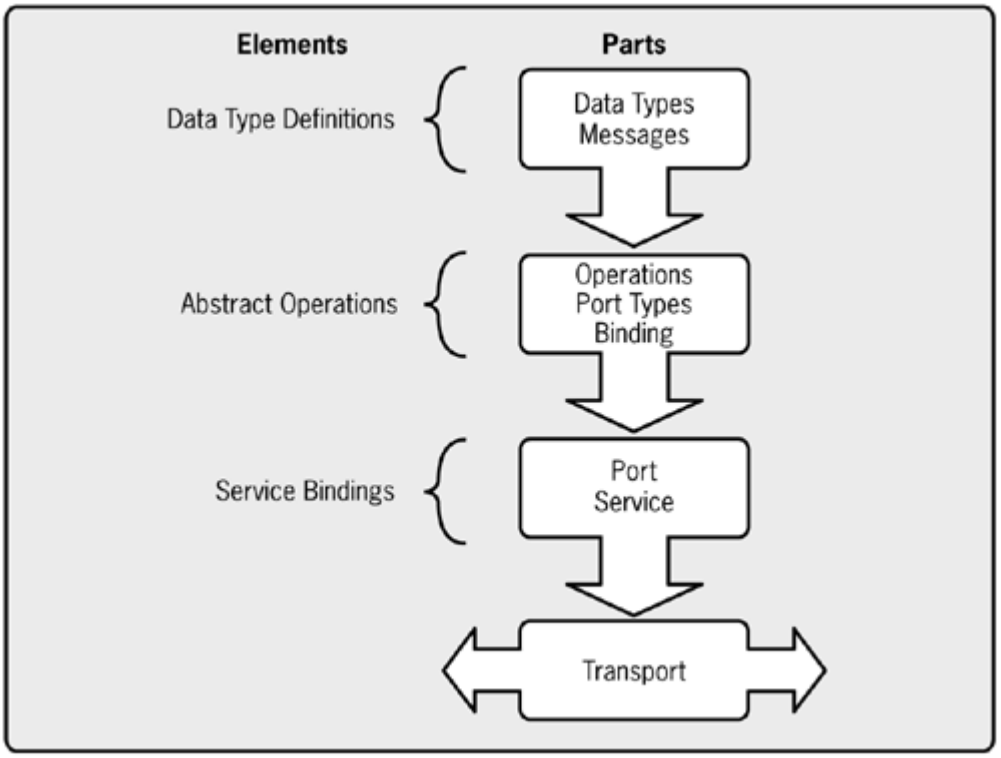
WSDL has three major elements, according to level of abstraction

Each major element can be specified in a separate XML document and imported in various combinations to create a final Web services description, or they can all be defined together in a single document. The data type definitions determine the structure and the content of the messages. Abstract operations determine the operations performed on the message content, and service bindings determine the network transport that will carry the message to its destination.

WSDL elements can be defined in separate documents

Figure 1-7 shows the elements of WSDL, layered according to their levels of abstraction, which are defined independently of the transport, specifically so that multiple transports can be used for the same service. For example, the same service might be accessible via SOAP over HTTP and SOAP over JMS. Similarly, data type definitions are placed in a separate section so that they can be used by multiple services. Major WSDL elements are broken into subparts.

Figure 1-7. WSDL consists of three major elements and seven parts.



The definition parts include data type definitions, messages, and abstract operations, which are similar to interface definitions in CORBA or DCOM. Messages can have multiple parts and can be defined for use with the procedure-oriented interaction style, the document-oriented interaction style, or both. Through the abstraction layers, the same messages can be defined and used for

multiple port types. Like the other parts of WSDL, messages also include extensibility components—for example, for including other message attributes.

WSDL interfaces are like CORBA or DCOM interfaces

WSDL data type definitions are based on XML schemas, but another, equivalent or similar type definition system can be substituted. For example, CORBA Interface Definition Language (IDL) data types could be used instead of XML schema data types. (If another type definition system is used, however, both parties to a Web services interaction must be able to understand it.)

Web service data types are based on XML schemas but are extensible to any other mechanism

The service bindings map the abstract messages and operations onto specific transports, such as SOAP. The binding extensibility components are used to include information specific to SOAP and other mappings. Abstract definitions can be mapped to a variety of physical transports. The WSDL specification includes examples of SOAP one-way mappings for SMTP (Simple Mail Transfer Protocol), SOAP RPC mappings for HTTP, SOAP mappings to HTTP `GET` and `POST`, and a mapping example for the MIME (multipurpose Internet messaging extensions) multipart binding for SOAP.

Abstract messages and operations are mapped to specific transports

XML namespaces are used to ensure the uniqueness of the XML element names used in each of the three major WSDL elements. Of course, when the WSDL elements are developed separately and imported into a single complete file, the namespaces used in the separate files must not overlap. Associated schemas are used to validate both the WSDL file and the messages and operations defined within the WSDL file.

Namespaces ensure WSDL element names' uniqueness

It's safe to say that WSDL is likely to include many extensions, changes, and additions as Web services mature. Like SOAP, WSDL is designed as an extensible XML framework that can easily be adapted to multiple data type mappings, message type definitions, operations, and transports. For example, IETF (Internet Engineering Task Force) working groups are proposing a new protocol standard—Blocks Extensible Exchange Protocol (BEEP)—to define a useful connection-oriented transport. (HTTP, by contrast, is inherently connectionless, making it difficult to resolve quality-of-service problems at the transport level.) Companies interested in using Web services for internal application or integration may choose to extend WSDL to map to more traditional protocols, such as DCOM or IIOP (Internet Inter-Orb Protocol).

SOAP: Accessing Web Services

So far, you have defined the data (XML) and expressed the abstraction of the service necessary to support the communication and processing of the message (WSDL). You now need to define the way in which the message will be sent from one computer to another and so be available for processing at the target computer.

The SOAP specification defines a messaging framework for exchanging formatted XML data across the Internet. The messaging framework is simple, easy to develop, and completely neutral with respect to operating system, programming language, or distributed computing platform. SOAP is intended to provide a minimum level of transport on top of which more complicated interactions and protocols can be built.

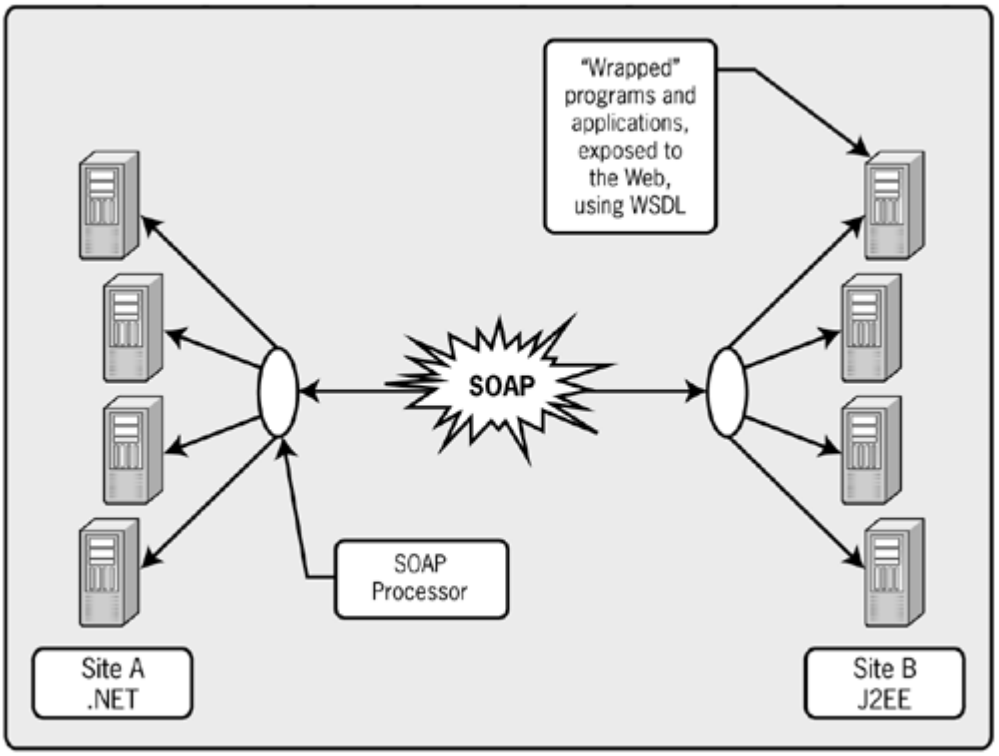
SOAP provides the communication mechanism to connect Web services

SOAP is fundamentally a one-way communication model that ensures that a coherent message is transferred from sender to receiver, potentially including intermediaries that can process part of or add to the message unit. The SOAP specification contains conventions for adapting its one-way messaging for the request/response paradigm popular in RPC-style communications and also defines how to transmit complete XML documents. SOAP defines an optional encoding rule for data types, but the end points in a SOAP communication can decide on their own encoding rules through private agreement. Communication often uses literal, or native XML, encoding.

SOAP is the XML way of defining what information gets sent and how

As shown in Figure 1-8, SOAP is designed to provide an independent, abstract communication protocol capable of bridging, or connecting, two or more businesses or two or more remote business sites. The connected systems can be built using any combination of hardware and software that supports Internet access to existing systems such as .NET and J2EE. The existing systems typically also represent multiple infrastructures and packaged software products. SOAP and the rest of the XML framework provide the means for any two or more business sites, marketplaces, or trading partners to agree on a common approach for exposing services to the Web.

Figure 1-8. SOAP messages connect remote sites.



SOAP has several main parts:

- **Envelope:** Defines the start and the end of the message
- **Header:** Contains any optional attributes of the message used in processing the message, either at an intermediary point or at the ultimate end point
- **Body:** Contains the XML data comprising the message being sent
- **Attachment:** Consists of one or more documents attached to the main message (SOAP with Attachments only)
- **RPC interaction:** Defines how to model RPC-style interactions with SOAP
- **Encoding:** Defines how to represent simple and complex data being transmitted in the message

SOAP messages contain an envelope, a header, and a body

Only the envelope and the body are required.

UDDI: Publishing and Discovering Web Services

After you have defined the data in the messages (XML), described the services that will receive and process the message (WSDL), and identified the means of sending and receiving the messages (SOAP), you need a way to publish the service that you offer and to find the services that others offer and that you may want to use. This is the function that UDDI (universal distribution, discovery, and interoperability) provides.

Inside the Enterprise

Many companies are exploring the potential advantages of using Web services both

inside and outside the enterprise. This is analagous to using browsers and Web servers inside the enterprise in internal networks. Existing internal Web infrastructure can be put to good use in support of Web services–style interactions. Although unlikely to replace existing distributed computing environments, such as COM and CORBA, Web services can be a valuable supplement to existing technologies. Sometimes, all you have is an HTTP or an SMTP connection. Because they represent a completely neutral format that can be used to achieve a new level of interoperability, Web services can also be used to bridge across COM, CORBA, EJB, and message queueing environments. Finally, because Web services use existing HTTP infrastructure, the impact on system administrators is minimal compared to introducing other distributed computing technologies into an IT department. Performance is certainly an issue compared to more traditional binary-oriented transports and protocols, but the potential benefits outweigh the costs for many applications, and performance issues tend to get solved over time, as they have been for the original Web.

The UDDI framework defines a data model in XML and SOAP application programming interfaces (APIs) for registering and discovering business information, including the Web services a business publishes. UDDI is produced by an independent consortium of vendors, founded by Microsoft, IBM, and Ariba, to develop an Internet standard for Web service description registration and discovery. Microsoft, IBM, Hewlett-Packard, and SAP are hosting the initial deployment of a public UDDI service, which is conceptually patterned after DNS, the Internet domain name service that translates Internet host names into TCP addresses. In reality, UDDI is much more like a replicated database service accessible over the Internet.

UDDI registers and publishes Web service definitions

UDDI is similar in concept to a Yellow Pages directory. Businesses register their contact information, including such details as phone and fax numbers, postal address, and Web site. Registration includes category information for searching, such as geographical location, industry type code, business type, and so on. Other businesses can search the information registered in UDDI to find suppliers for parts, catering services, or auctions and marketplaces. A business may also discover information about specific Web services in the registry, typically finding a URL for a WSDL file that points to a supplier's Web service.

UDDI is a directory of Web services

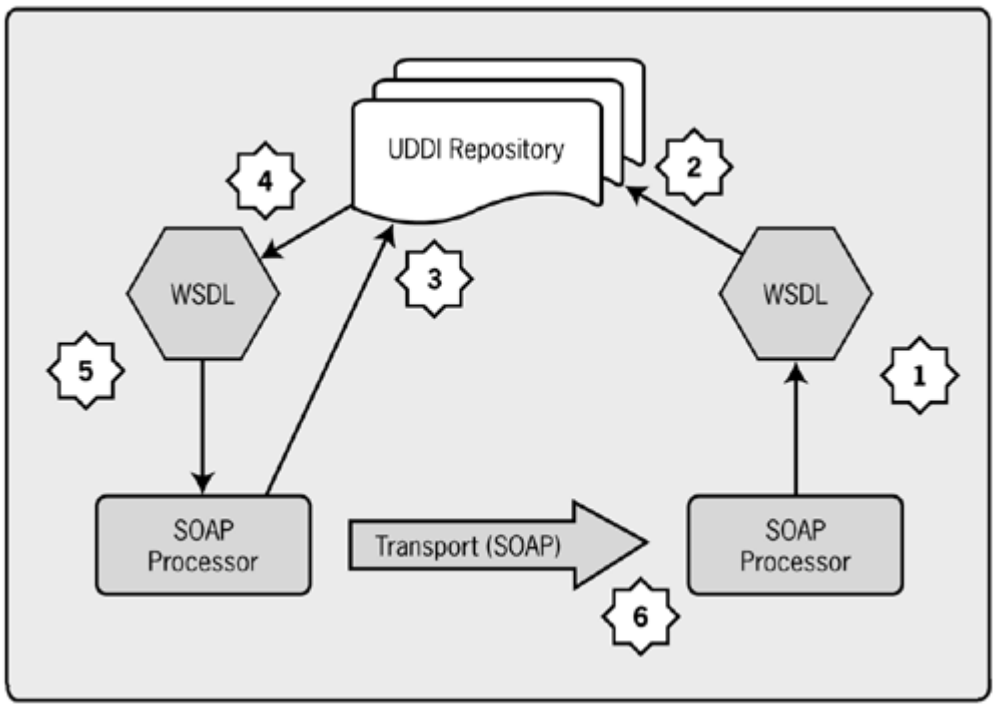
Businesses use SOAP to register themselves or others with UDDI; then the registry clients use the query APIs to search registered information to discover a trading partner. An initial query may return several matches from which a single entry is chosen. Once a business entry is chosen, a final API call is made to obtain the specific contact information for the business.

UDDI uses SOAP for registering and discovering information

Figure 1-9 shows how a business would register Web service information, along with other, more traditional contact information, with the UDDI registry. A business first generates a WSDL file to describe the Web services supported by its SOAP processor (1) and uses UDDI APIs to register

the information with the repository (2). After a business submits its data to the registry, along with other contact information, the registry entry contains a URL that points to the SOAP server site's WSDL or other XML schema file describing the Web service. Once another business's SOAP processor queries the registry (3) to obtain the WSDL or other schema (4), the client can generate the appropriate message (5) to send to the specified operation over the identified protocol (6). Of course, both client and server have to be able to agree on the same protocol—in this example, SOAP over HTTP—and share the same understanding, or semantic definition of the service, which in this example is represented via WSDL. With the widespread adoption of these fundamental standards, however, this common understanding of WSDL seems ensured.

Figure 1-9. The UDDI repository can be used to discover a Web service.



XML for Business Collaboration: ebXML

Several additional technologies, beyond what's provided in the basic Web services technologies, are required to support true business-to-business interaction over the Web. The Electronic Business XML (ebXML) consortium, for example, has defined a comprehensive set of specifications for an industrial-strength usage pattern for XML document exchange among trading partners. The ebXML messaging specification is based on SOAP with Attachments and does not use WSDL but does add several qualities of service, such as security, guaranteed messaging, and compliance with business process interaction patterns.

The ebXML spec provides more than basic Web services technologies

The ebXML initiative, the first phase of which ended in May 2001, was sponsored by an international group established by the United Nations Center for Trade Facilitation and Electronic Business (UN/CEFACT) and OASIS to research, develop, and promote global standards for the use of XML to facilitate the exchange of electronic business data.[5] The ebXML architecture begins with a business process and information model, maps the model to XML schemas, and

defines requirements for applications that process the documents and exchange them among trading partners.

[5] Although the original ebXML effort ended in May 2001, work continues on specific OASIS (The Organization for the Advancement of Structured Information Standards) and UN/CEFACT committees to enhance and further extend the original ebXML specifications.

The ebXML spec defines XML use for cooperative business processes

Web Services and EDI versus ebXML

Although electronic data interchange (EDI) has been around for more than two decades, it is very complex, has multiple interpretations, requires significant technical expertise to deploy, and is based on a tightly coupled, inflexible architecture. Although they can be deployed on public networks, EDI applications are most often used on expensive dedicated networks and require a lot of expertise to set up and run.

By contrast, ebXML and Web services hold the promise of realizing the original goals of EDI, making it simpler and easier to exchange electronic documents over the Internet. However, ebXML and Web services also will have to mature for several years before they encompass all EDI's current functionality and feature set.

Although the ebXML consortium has completed its initial work, OASIS, UN/CEFACT, and other organizations continue to promote the adoption of the architecture and specifications to a broader audience, hoping to establish a global e-business marketplace through the standardized exchange of XML documents and messages, regardless of geographic or political boundaries, and with the qualities of service that businesses expect. The ebXML architecture defines

- Business processes and their associated messages and content
- A registry and discovery mechanism for publishing business process sequences with related message exchanges
- Company profiles
- Trading-partner agreements
- A uniform message transport layer mapped to SOAP with multipart MIME attachments

The ebXML architecture extends basic Web services concepts

Similarly to the way in which UDDI facilitates a search for Web service definitions, the ebXML architecture allows businesses to find one another by using a registry, to define trading-partner agreements, and to exchange XML messages in support of business operations. The goal is to allow all these activities to be performed automatically, without human intervention, over the Internet. The ebXML architecture has many similarities to SOAP/WSDL/UDDI, and some level of convergence is already taking place with the adoption of SOAP in the ebXML transport specification.[6] RosettaNet also announced its adoption of the ebXML transport, as have many other vertical industry consortia.

[6] See www.ebxml.org for further information on ebXML's use of SOAP and other details of the ebXML initiative.

The ebXML registry allows businesses to find and to collaborate with one another

The ebXML architecture clearly centers on document-oriented interactions; as ebXML gains acceptance, it may come to define the paradigm for B2B-oriented Web service interactions. Companies that already have been exchanging information electronically, perhaps using EDI standards, will find many parallels in the goals of ebXML, although ebXML aims at addressing this type of requirement more broadly and for the Internet.

The ebXML specification focuses on document-oriented interactions

Comparison of ebXML and SOAP

Initially, it seemed that the ebXML group was competing with the group of companies sponsoring SOAP, WSDL, and UDDI. In fact, the ebXML specifications cover a lot of the same territory as SOAP, WSDL, and UDDI. You could view the SOAP effort at W3C as a "bottom-up" approach, starting with a definition of the way to map XML documents to HTTP messages, and look at the ebXML effort as a "top-down" approach, starting with a definition of the business process as a series of messages mapped to any transport.

The ebXML group was formed primarily to create business process standards, the areas in which the work of ebXML has the most promise. The areas of transport, services description, and registry seem more appropriate to efforts focused more purely on issues of infrastructure than of business process and document interaction. One of the major motivators for ebXML is to produce standards that serve the same or similar purpose as EDI, including support for the emerging industry-specific XML "vocabularies." It seems appropriate to consider the ebXML architecture as requirements on W3C and other XML-oriented initiatives as a way of ensuring that Web services will be ready for real business use, rather than as a competitive effort to define core infrastructure services.

Web Services versus Other Technologies

Web services are not as much like traditional distributed computing technologies such as CORBA, DCOM, and EJB, as they are like Web servers, HTML, and HTTP, on which they are based. Web services are fundamentally one-way, asynchronous messages mapped onto executable software programs. Web services define a data format independent of programming language, operating system, network transport, and data storage mechanism; therefore data has to be mapped into and out of the independent format. Data typing and structure are abstracted from underlying implementations of services.

Web services differ from traditional distributed computing technologies

Web services are often compared to remote procedure call invocations or software components. However, Web services are more appropriately compared to enterprise application integration adapters. Web services define a canonical message format, as EAI software systems, such as MQSeries, TIBCO, NEON, Vitria, and IONA's Orbix E2A, do and define the way in which the

message is directed to a service interface through which the data is mapped or transformed onto an underlying application. In other words, the intelligence for understanding how to map a message into a software program is not contained within the interface itself, as it is in CORBA, J2EE, and DCOM, all of which are based on RPC concepts, which tightly couple the service name to the program being invoked. Rather, that intelligence is contained within the XML processor, which consumes the message and follows associated instructions on how to parse the message and map the data into whatever program implements the Web service.

Web services are more like adapters

In addition, Web services do not require or assume the existence of the same software system on both ends of a communication path. EAI adapters similarly accept a canonical message format and map the information in the message to an enterprise resource planning (ERP) or other type of enterprise application. Web services are defined at a similar level of abstraction, which allows the same message type to be mapped to multiple applications, including, but certainly not limited to, RPC-based components.

Web services map to any software

Unlike RPC-oriented middleware, such as CORBA and DCOM, Web services use unidirectional, asynchronous messaging, which is more naturally mapped to a message queuing system, such as MQSeries or JMS, than to CORBA or DCOM; although, of course, Web services are also often mapped to CORBA-, J2EE-, and DCOM-based products. Web services support a request/response paradigm typical of synchronous, RPC-style communications through emulation; that is, the XML processor rather than the protocol correlates requests with replies. The HTTP mapping of SOAP, for example, does not support protocol-level request/reply correlation.[7] The Web services emulation of an RPC is easily mapped to such traditional RPC-based systems as CORBA, EJB, and DCOM, although qualities of service (e.g., security, transactions, and exception handling), are likely to be very different from those available in traditional distributed computing technologies, which are often tied closely to the transport layer, and are specific to each technology.

[7] Various proposals address this issue, including HTTPR (reliable HTTP) and BEEP, a new session-oriented protocol from IETF; see Chapter 7 for further information.

Web services are fundamentally one-way, asynchronous messaging systems

Because interactions with Web services are accomplished through the programs and databases to which the Web services are mapped, the user experience is likely to be very different from a typical browser-based experience: Web services are more like traditional applications than like browsers, although, of course, browsers may be used. (As mentioned previously, Web services by themselves are not executable but instead have to be mapped to a program, an object, a middleware system, or a database management system.)

Interacting with Web services is like interacting with traditional applications

Additional Technologies

Core Web services technologies, such as SOAP, WSDL, and UDDI, are useful for bridging disparate technology domains and submitting documents to business process flows. However, to become useful for more types of applications and to fulfill the complete vision of Web services as enabling the use of application building blocks over the Internet, Web services technologies have to be extended to encompass additional features, functions, and qualities of service.

The ongoing work of evolving Web services toward a more useful technology substrate is very similar to the evolution of the common object request broker architecture, undertaken by the Object Management Group (OMG) during the 1990s. The OMG work defined a comprehensive software architecture that guided an open, collaborative effort that produced a rich set of specifications for transactions, asynchronous messaging, security, failover, fault tolerance, and so on. The same type of effort is being initiated at W3C for Web services, and a similar architecture is evolving.

In the world of Web services, the major industry software vendors have already agreed on the core standards, which is the true test of standardization. Microsoft, IBM, Sun Microsystems, BEA Systems, Oracle, IONA, and others have agreed on implementing SOAP, WSDL, and UDDI, although some difference of opinion remains on the role of the ebXML registry. However, other than for the fundamental standards, proposals often compete, such as the difference of opinion between Microsoft and IBM on business process flow definition, that is, XLANG versus WSFL (Web Services Flow Language), and competing proposals for handling security context.

Additional technol-ogies may or may not become part of the standard

Additional technologies are focused primarily in the following key areas:

- Security
- Process flow
- Transactions
- Messaging

Some of the most important additional technologies for Web services involve security technologies.

Security is important to ensure the confidentiality and integrity of Web services data. No one other than the intended recipient of the data should be allowed to examine or to tamper with message contents. Security also is necessary to control access to Web services, especially when multiple Web services are used together, so that only those for whom they are intended use them.

Security is most important

Proposed standards exist for authentication and authorization (SAML, or Security Authorization Markup Language) and for public key management for encryption (XKMS, or XML Key Management Specification). Of course, fundamental to all Internet security is Secure Socket Layer (SSL) and, for HTTP-based protocols, HTTPS (secure HTTP) for basic encryption-level security.

In addition to HTTPS, firewalls, SAML, XKMS, the use of digital signatures, and XML encryption, Microsoft has proposed WS-License for credential management and WS-Security for propagating security credentials associated with Web service interactions.

Process flow is critical to automating business process interactions over the Web and inside an enterprise. Process flow is also often called *orchestration* because it defines the relationship among a series of interactions necessary to accomplish a given purpose, such as completing a purchase order, processing a travel reservation, or executing a manufacturing plan. A flow is modeled as a sequence of steps defined for a given business process. The series of steps creates an aggregation of functions for which a Web service interface can be defined.

Flows automate business process execution

In the world of automated business operations, transactions have long played the part of enforcer, ensuring that the execution platforms produced consistent results from a series of related operations on data, despite software or hardware failures. These traditional protocols and techniques are not directly applicable to the Web, however, as they are designed for a tightly coupled environment in which it's possible to hold database locks pending notification of the transaction result and in which a connection-oriented protocol is available to detect communication failures automatically. The Business Transaction Protocol (BTP) proposal from OASIS is designed to resolve this problem for Web services by defining a loosely coupled protocol that ensures that the results of multiple Web service interactions are correctly propagated and shared.

Transactions are being redefined for the Web

Messaging protocols execute the communication patterns defined for Web service interactions, such as asynchronous one-way, request/response, broadcast, and conversational, or peer-to-peer. Additional Web services technologies also may depend on the messaging layer for certain qualities of service, such as reliable or guaranteed delivery, propagation of security and transaction contexts, and correctly routing messages along a defined path that includes one or more intermediaries. IBM has proposed reliable HTTP (HTTPR) to address requirements in this area.

Mechanisms for reliable messaging are needed

IBM and Microsoft have collaborated on the WS-Inspection proposal for discovering information about Web services available at a particular message target. Microsoft has also proposed WS-Referral and WS-Routing to define a specific message path for a Web service, including any number of intermediaries, and how to route messages forward and backward along the specified route.

The Blocks Extensible Exchange Protocol (BEEP) from IETF defines a connection-oriented Internet protocol. A SOAP mapping for BEEP has been defined, and in this case, SOAP messages inherit the additional qualities of service from BEEP for maintaining session context at the sender and the receiver nodes. The context can be used to relate multiple messages into a larger unit of transfer and to relate multiple messages as coming from the same source or intended for the same target. Security and transaction context can also be associated with a connection.

BEEP provides a connection-oriented protocol

Other relevant standards and technologies include many of those defined by the following organizations:

- OASIS, hosting ongoing ebXML and other related XML proposals, such as BTP and SAML
- RosettaNet, influencer of Web services concepts, developed by a group of electronics vendors for B2B business process flow interaction over the Internet
- UserLand, developer of XML-RPC, a precursor of SOAP
- OAGI (Open Applications Group, Inc.), defining canonical XML document formats for business and industry

Many other technologies and standards are relevant to Web services

The work of these and other groups often focuses on promoting the adoption of XML for specific business purposes, such as building on the base standards to define document formats and protocols for the electronics, financial, health care, and other industries. Because Web services are based on XML, the work of almost any standards body or consortium promoting the use of XML-related technologies for Internet business is relevant. Some of the other work, such as BTP and SAML, emerges as candidate technology for adoption by W3C within its Web services architecture activity.

The Long Road Ahead

Additional technologies, such as security, transactions, and reliable messaging, currently found in existing distributed computing environments, have to be defined again for Web services because of the fundamental shift involved in the infrastructure—XML and HTTP—on which they now need to be built. The World Wide Web Consortium will undertake the effort to define Web services architecture, just as OMG defined architecture for CORBA, although this is likely to be a very difficult and daunting task. The W3C is not set up to resolve major differences of opinion among its members, especially when those differences are motivated by commercial interests. This is the downfall of many standards efforts, in fact.

Vendor Approaches to Web Services

Software vendors, both large and small, are providing Web services implementations as product add-ons or as entirely new products.

Web services do not fundamentally change existing software systems, although they can change how software systems are put together. The differences in implementation usually follow the differences in philosophy, or approach, of the vendors: Are Web services a fundamental enabling technology? Or are they simply entry and exit points to and from existing software systems? In other words, vendors vary in their approach to Web services, depending on the extent to which they view Web services as impacting existing software system architectures. For example, do Web services invalidate J2EE, or are they complementary? The answers to these and other similar questions can be discovered in a vendor's approach.

Web services do not change the underlying software systems

The five basic approaches to Web services are to map them

- Into and out of a database management system

- Into and out of an application server
- Into and out of an integration broker
- Between technology domains
- To architectural software building blocks or functional components

In other words, Web services implementers fundamentally distinguish between Web services technologies and underlying software implementations. Web services, therefore, are either incidental aspects of existing software systems or a required part of the infrastructure.

Can an application server, object request broker, or database management system successfully continue to exist without support for Web services? Or can Web services exist on their own?

Thus the question is, where does the value lie? With application servers, database management systems, and integration brokers, leaving Web services to be merely a means of mapping data into and out of existing software systems? Or does the value lie with the Web services themselves, as fundamental to a new category of software systems?

Value is either in the underlying software or in the Web services layer

Vendor implementations tend to be divided among these varying views of the value of Web services. Not surprisingly, Microsoft has its own view, whereas Sun Microsystems, IBM, BEA Systems, Oracle, and others are taking an alternative view. To some extent, this divergence of vendor views, or initiatives, represents a continuation of the Visual Basic/Java developer battle, but Microsoft is taking a very bold and aggressive stance on Web services, even breaking current Visual Basic applications to ensure that the future version of VB will support Web services as fundamental enabling technology. The Java community is taking a less radical view, extending Java APIs for Web services rather than requiring a rewrite to incorporate them.

Vendor views vary, often along Java/Visual Basic lines

Industry business consortia, such as ebXML and OASIS, as well as integration broker products from such vendors as IBM, Microsoft, IONA, and WebMethods, tend to focus on the business process, or document-oriented type of applications for Web services. Other vendors' products, such as the Web services toolkits shipped with BEA's WebLogic and IONA's J2EE Edition, tend to focus on the RPC style of interaction. The same XML-based technologies and standards can generally be used for either, but initiatives and products tend to focus on one or the other because the paradigms are so different. In general, application servers tend to support the RPC style of interaction, whereas integration brokers tend to support the asynchronous document-oriented style of interaction.

Products tend to focus on either the RPC or the asynchronous style

What Are Web Services Good For?

The answer to this question may well vary by vendor, depending on the particular approach to Web services implementation. Web services are generally not replacements for any existing technologies but rather are complementary, another tool in the toolbox,

as it were. Web services represent loosely coupled interactions, which are better suited to integrating disparate software domains and bridging incompatible technologies, rather than heavy-duty, high-performance applications. Web services are also very good for submitting documents to long-running business process flows, which seem in any case to be a good way to start with interactions over the Internet.

Integration broker vendors, such as WebMethods, Vitria, SeeBeyond, Software AG, and Mercator, typically view Web services as an extension of classic enterprise and business-to-business integration technologies and have built adapters for Web services as they would build adapters for any other technology with which their products have to integrate. Other vendors, such as IONA, take a more neutral and encompassing view of Web services as both enabling technology for extending existing application server, CORBA, and COM middleware and as fundamental to the next generation of enterprise and business integration standards. IONA's Orbix E2A product line provides not only Web services adapters for asynchronous, document-oriented processing and RPC-oriented Web services interfaces for CORBA and J2EE-compliant objects but also fundamental Web services building blocks. The IONA business process engine, XML conversion and transformation engine, packaged application adapters, and business protocol framework all export Web service interfaces. The IONA products support a consistent approach to application integration, using Web services technologies inside and outside the firewall.

Finally, a number of vendors view Web services as an interesting and potentially profitable technology in their own right and have developed "pure-play" Web services products. These products, based entirely on Web services technology, typically require use with other technologies and products. For example, Cape Clear markets a Web services product aimed at bridging J2EE and .NET. Shinka markets a product that presumes that Web services are a fundamental design center and that programs will be developed to map into them, rather than vice versa, which is what most other vendors appear to believe.

Some vendors focus purely on Web services

Summary

Web services are quickly becoming significant technology in the evolution of the Web and distributed computing. Web services leverage the data independence of XML to solve enterprise integration problems, both inside and outside the firewall. Web service interfaces are shells, or wrappers, that map to any type of software program, middleware system, database management system, or packaged application.

New types of applications are being created by using standard Web services building blocks, thus creating greater economies of scale in automating business and consumer interactions with the Web and with each other. Web services technologies are rapidly changing, and a long list of additional features and functionality is required to complete the vision. The basic Web services standards—SOAP, WSDL, and UDDI—are immediately useful for many applications, such as publishing interfaces to automated business processes, bridging disparate software domains, and connecting wireless clients to Web functions.

The ebXML initiative offers an alternative view of an XML-enabled distributed computing infrastructure, specifically aimed at connecting business process interactions among Internet trading partners. ebXML represents a form of industrial-strength Web services, although ebXML does not include WSDL or UDDI. Many vendors view Web services and ebXML as significant aspects to be added to their existing products; other vendors view Web services as sufficient technology on which to base entire products.

Chapter 2. Describing Information: XML

The Extensible Markup Language (XML) is the foundation on which Web services are built. XML provides the description, storage, and transmission format for data exchanged via Web services. XML also is used to create the Web services technologies that exchange the data.

Web services are built on XML

XML is similar to the Hypertext Markup Language (HTML), having elements, attributes, and values. Well-formed XML documents can be displayed in browsers, although this aspect of XML is not relevant to Web services. HTML contains a finite set of elements and attributes, but XML allows any number of them to be defined.

XML elements and attributes independently define type and structure information for the data they carry, including the capability to model data and structure specific to a given software domain. (A software domain is a programming language, a middleware system, a packaged application, or a database management system.) XML-aware programs and tools parse, map, and transform generic XML data types into and out of software domain–specific types. Transforming a generic XML representation of data into an application, or a software domain–specific representation of data, is an essential aspect of Web services.

XML defines data type and structure

The XML syntax used in Web services technologies specifies how data is generically represented, defines how and with what qualities of service the data is transmitted, and details how the services are published and discovered. Web services implementations decode these various bits of XML to interact with the various applications and software domains underneath the services.

XML also defines transmission formats and qualities of service

Two broad categories of XML usage in Web services are (1) a data storage representation and format and (2) a specification of the software that manipulates the data. Although it originated as a text markup language for document processing, XML has become widely used for data formatting and manipulation. The XML community is therefore somewhat divided between those interested in text formatting and those interested in data formatting, with the Web services community belonging firmly to the latter category.[1]

[1] For a good description of XML's use as a data-formatting language, see *Essential XML* by Don Box, Aaron Skonnard, and John Lam.

Much of the XML community is focused on data-formatting applications

A Simple Example

For use in Web services, data can be either created in XML or converted to XML from one or more "native" or existing formats, such as ASCII or the Java type system. For a simple example, imagine that the Skateboots Company's business analysts identified the following basic data type information required for a customer record in an ASCII file:

```
Customer Number   Integer
Customer Name     Character
Customer Address  Character
Customer Phone    Numeric
Postal Code       Character
Credit limit      Decimal
Credit rating     Integer
```

Web services data can be converted into XML

(Obviously, a lot more information would be necessary for a real business.)

After collecting the basic requirements from the customer service department as shown in the example, the analysts formatted the data in XML, as follows:

```
<Customer>
  <CustomerNumber>12345</CustomerNumber>
  <CustomerName>Joe's Boots</CustomerName>
  <CustomerAddress>500 High Street</CustomerAddress>
  <CustomerPhone>555 123 4567</CustomerPhone>
  <PostalCode>12345</PostalCode>
  <CreditLimit>1000000</CreditLimit>
  <CreditRating>5</CreditRating>
</Customer>
```

The analysts created a `Customer` element containing the data items and created XML element tags descriptive of the enclosed data items. However, the data is not typed.

After representing the customer data as XML, the Skateboots Company analysts created the XML schema to validate the customer information to ensure that it would have the correct structure and data types. When receiving customer data from a file, a program, or a database, the data can be converted into XML, using the information contained in the schema. When XML documents are sent to the company from distributors, the schema can validate that the correct customer data and types have been included in the document. For example:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer" type="CustomerType"/>
  <xsd:complexType name="CustomerType">
    <xsd:sequence>
      <xsd:element name="CustomerNumber"
            type="xsd:integer"/>
      <xsd:element name="CustomerName"
            type="xsd:string"/>
      <xsd:element name="CustomerAddress"
            type="xsd:string"/>
      <xsd:element name="CustomerPhone"
```

```
            type="xsd:string"/>
      <xsd:element name="PostalCode"
            type="PostalCodeType"/>
      <xsd:element name="CreditLimit"
            type="xsd:decimal"/>
      <xsd:element name="CreditRating"
            type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="PostalCodeType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{ 5}  | \d{ 5} -\d{ 4} } "/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Define the data and then the associated schema

The example illustrates a schema that defines validation, data typing, and document structure for the customer record. The top declaration of `schema` references the specific XML schema edition in use—in this case, the May 2001 edition, referenced via the namespace www.w3.org/2001/XMLSchema. This mechanism is used consistently by W3C to identify applicable versions of its recommendations, or standards. Referencing this namespace means that the schema in the example conforms to the May 2001 version of the XML schema specification and that the XML program handling the schema has to understand the May 2001 schema version or generate an error.

Schemas validate the data

The sample schema contains both simple and complex types and illustrates the use of a sequence to establish the order of elements. That is, in any document conforming to this schema, the `CustomerName` element must precede the `CustomerNumber` element, and so on. The `CustomerNumber` must be an integer, the `CustomerName` must be a string, and so on.

Schemas feature simple and complex data types

The `restriction` keyword requires the postal code to be input as a string. The `PostalCodeType` is an example of a data type defined within the schema, a feature designed to support the definition of special data types when none of the standard defined types will do the job. The SOAP specification defines its own array types, for example.

Original document markup efforts, such as Runoff, TeX, and LaTeX, focused on embedding commands within the text to format the text into paragraphs, bulleted lists, and headings and to specify the place on the page for page numbers, running headers and footers, and so on. Later, especially with the advent of proportional-spacing display and print technology, the commands

were designed to be generic enough so that the rendering into type, page layouts, and so on could be changed dynamically without recoding the original document. In other words, a company could decide to switch from a serif typeface, such as Times Roman, to a sans-serif typeface, such as Helvetica, without changing the document markup tags. Similarly, a document could be formatted for both the U.S. standard letter size and the European standard A4 size without recoding the file.[2] In Web services terms, this means that the schema can be changed independently of the instance document and the data reformatted according to the types and structure in the changed schema.

[2] For a good overview of XML in the context of markup languages, see Elizabeth Castro's *XML for the World Wide Web*.

Schemas can be changed independently of the data

Instance and Schema

Because software domains traditionally have defined their own data types and structures, application programmers wishing to interoperate across software domains have had to format data according to the requirements of each specific software domain and to write programs to convert data from one domain format to the other. With XML, however, programmers can use a rich library of XML tools to transform the data from native formats to XML and back again. XML can be used as an independent data type and structure mechanism, eliminating the many-to-many problem of mapping all domain data types and structures. Programmers have to define the mappings only from the particular domain to XML and back again.

Schemas help transform data into and out of XML format

XML independently stores data values within descriptive element tags in a text-based *instance* of a document. (Everything is a document in XML.) For example:

```
<CustomerNumber>12345</CustomerNumber>
```

XML provides data independence

The `<CustomerNumber>` is the descriptive element tag, and `12345` is the data value contained within the element.

XML elements are enclosed in angle brackets (< >) and have a start and an end. The end is marked by a slash (/). Elements that enclose only attributes can be self-ending, for example:

```
<CustomerNumber Format="any five digits" />
```

Elements can have one or more attributes associated with the element name, using a name/value pair for each attribute. For example:

```
<CompanyName CountryOfOrigin="Ireland"
      PubliclyTraded ="Yes">
 IONA
```

```
</CompanyName>
```

XML elements can have one or more attributes

One or more XML schemas, which too are instances of XML documents, separately define the types, structure, and semantic meaning to be applied to the instance data contained within the element tags. For example:

```
<xsd:element name="CustomerNumber"
      type="xsd:integer"/>
```

The `xsd` prefix identifies the element as an XML schema element with the name `CustomerNumber` and the type `xsd:integer`, which is a schema simple type. Simple types can be any of the predefined XML schema types—such as string, integer, double, float, date, and time—or XML specific types—`Entity`, `NMToken`, and `ID`.

XML processors validate instance documents by matching element names declared in the schema with elements in the instance document. Elements that are not declared in the schema may be rejected as invalid by the XML processor. As in HTML, however, XML processors often will parse and accept what they can understand and validate, without necessarily rejecting documents that contain undeclared elements, because a single instance document might have multiple schemas associated with it for different purposes.

XML processors input the data and Schema

Type and structure are also associated with the instance document by matching the element names in the instance document to the element names in the associated schema and applying the type and structure information declared in the schema, if any. The simple data type in this example might map to an `int` in Java or C++ or to a `99999 COMP` in COBOL, depending on the particular software domain in use with the Web service.

Matching element names with their definitions associates type and structure

Simple types can also be defined for a specific purpose in a document. For example:

```
<xsd:simpleType name="PostalCodeType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{ 5} -\d{ 4} }"/>
  </xsd:restriction>
</xsd:simpleType>
```

The simple type in this example is `PostalCodeType` and is restricted to the XML schema type of `xsd:string` with a pattern requiring the nine-digit ZIP code format. (This would not, of course, work for postal codes outside the United States.)

XML schema `ComplexTypes` model element content; that is, they determine the permissible set of elements in a document. `ComplexTypes` can inherit from `SimpleTypes`, and repeating components can be defined in `Groups`. The `CustomerType` element is an example of a complex type that restricts the children of the `Customer` element to the defined list. Schemas are also often called "content models" for this reason.

Complex types model element content

Data Type and Programming Language

Traditionally, application typing and data structuring are contained within a programming language definition. The type information references locations in binary memory allocated by the operating system on behalf of the program when it executes. A programming language views data through its own specific set of data types, which are basically abstractions of the binary code used to physically store and manipulate bits and bytes. Programming languages provide data types consistent with their design centers. COBOL, widely used in traditional business applications, is rich in text data types and processing capabilities. C/C++, widely used for operating system utilities, is rich in low-level operations, such as memory allocation and array pointers.

XML breaks the association of data type and programming language

XML represents a major shift in the way applications view data, especially common data shared across multiple types of programs and applications. XML stores all data as text, like the markup language it basically is. Programs that access XML map the data into and out of the text representation using the associated type information. Because the associated type information is stored independently, multiples are allowed and can be changed independently of the data.

XML *processors* therefore map application data into and out of XML, much as browsers map HTML into a display, except that the target of the mapping for Web services is not a graphical user interface but rather a data file or a program. The XML processors match the element names in the schema file to the element names in the data file to apply the type and structure information. XML processors also have to understand special schema elements that pertain to interpreting the data, such as how to serialize the data for transmission over HTTP or map a SOAP message to an object method. For example, to specify that the SOAP encoding style is to be used for data serialization, the following namespace[3] is included in a Web service document:

[3] This is the SOAP v1.1 encoding namespace. See Chapter 4 for information on the SOAP v1.2 namespaces.

```
encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
```

An XML instance can be compared to the physical disk storage of data in a database management system, and the XML schema can be compared to the SQL data description language schema. Schemas in XML, however, have a broader application than those in SQL, also being used to contain instructions for how to manipulate the data and how to format it for storage.

XML schemas are comparable to SQL schemas

It is worth mentioning again in this context that XML has no fixed or finite syntax: Schema elements can be defined to fit any purpose, not only to format data. The SOAP specification, for example, creates XML schema elements to define how an XML processor produces and consumes instances of data conforming to the SOAP schemas. XML processors then have to be created or modified to understand the significance of these special elements, but that's what the world of XML and Web services is all about.

Markup Languages Rule

It's interesting that the future of the Web is evolving from a markup language rather than from a programming language. In part, this stems from the fact that no single operating system, programming language, middleware system, or database system is ever going to be able to handle everything. The same abstractions that drove the popularity of programming languages and software systems by making binary data easy to work with in specific syntax structures also constrains them in solving the problem of generic interoperability. The common set, or intersection, of data types among COBOL, C/C++, Java, Visual Basic, C#, and SQL, for example, is by definition less rich and powerful than the set of data types supported within any given one of the languages.

The XML content model was developed so that a document and its appearance could be separated. Now it's being used to separate data and its relationship to software programs. Separating the definition of data and its relationship to software programs solves a big problem in the application integration space. Traditional software systems have tended to adopt their own, unique mechanisms for data formatting, communication, and storage, as if no other programming languages existed, or as if the particular software system had managed to solve the perennial problem of data abstraction once and for all.

XML takes the mapping one step further, so that programs have to map only their own specific data formats into and out of XML. Mapping everything into and out of text is very ineffecient in terms of both internal hardware storage space and processing speed. However, as we saw with the adoption of the text-based Web, performance is often less of an issue than raw capability. And in this case, because XML provides a solution to a significant, previously unresolved problem of application data integration, performance is truly a secondary issue.

More on XML Schemas and DTDs

XML schemas were developed to resolve some of the limitations and problems with document type definitions (DTDs). For example, DTDs can't describe data types, including complex or custom-defined data types, and DTD definitions are all global; that is, element names can't be duplicated, even within complex structures, as they can be in schemas. DTDs are not used for defining Web services, so they are not covered in detail. DTDs were developed to express a content model for XML documents, defining valid elements, attributes, and some ordering constraints. Validation of content in document-oriented interactions may still often be done using document type definitions, especially when existing XML documents are transmitted via Web services.

Schemas were developed to improve on and to replace DTDs

The following examples illustrate the use of schemas and DTDs in relation to a common instance of a simple purchase order document. The document instance is:

```
<PurchaseOrder OrderDate="2001-10-15">
  <Contact Country="US">
    <Name>Joe's Sports, Inc.</Name>
    <Street>5555 High Street</Street>
    <City>Springfield</City>
    <State>CA</State>
    <Zip>95555</Zip>
  </Contact>
  <LineItem>
    <SupplierName>Skateboots</SupplierName>
    <ProductName>Special Red Color</ProductName>
    <Quantity>10</Quantity>
    <UnitPrice>150.00</UnitPrice>
  </LineItem>
  <LineItem>
    <SupplierName>Skateboots</SupplierName>
    <ProductName>Cool Blue Boots</ProductName>
    <Quantity>5</Quantity>
    <UnitPrice>195.00</UnitPrice>
  </LineItem>
</PurchaseOrder>
```

Both schemas and DTDs can be used to validate instance documents

The purchase order shown in the example contains the `PurchaseOrder` root element—well-formed XML documents contain a single root element within which the rest of the elements are enclosed—and several child elements of `PurchaseOrder`, including `Contact` and `LineItem`. XML documents are represented using a hierarchical structure, in which elements are nested, as `SupplierName` is nested within `LineItem`.

The schema definition is:

```
<xsd:schema xmlns:xsd="http:
     //www.w3.org/2001/XMLSchema"
          xmlns:po="Skateboots.com/order"
          elementFormDefault="qualified"
          targetNamespace="Skateboots.com/order">
  <xsd:element name="PurchaseOrder">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Contact"
              type="po:ContactType"/>
        <xsd:element ref="po:LineItem"
              maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="OrderDate"
            type="xsd:date"/>
```

```
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="ContactType">
    <xsd:sequence>
      <xsd:element name="Name" type="xsd:string"/>
      <xsd:element name="Street" type="xsd:string"/>
      <xsd:element name="City" type="xsd:string"/>
      <xsd:element name="State" type="xsd:string"/>
      <xsd:element name="Zip"
            type="xsd:positiveInteger"/>
    </xsd:sequence>
    <xsd:attribute fixed="US" name="Country"
           type="xsd:NMTOKEN"/>
  </xsd:complexType>
  <xsd:element name="LineItem">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="SupplierName"
              type="xsd:string"/>
        <xsd:element name="ProductName"
              type="xsd:string"/>
        <xsd:element name="Quantity"
              type="xsd:positiveInteger"/>
        <xsd:element name="UnitPrice"
              type="xsd:decimal"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

The example contains the schema that validates the purchase order shown in the previous example, ensuring the proper data types for the elements and that the line-item element has the proper structure. Such a schema ensures the document's conformance to the agreed-on format. Without such agreement, it would be possible to receive documents that contained information that wasn't recognized or was ignored and therefore could not be processed correctly.

Complex types, such as in this example, define the names of elements permitted to appear in the associated document and use attributes to define additional constraint information for the elements.

The schema references, within the `xsd:schema` root element, the namespace for the May 2001 version of the XML schema specifications:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:po="Skateboots.com/order"
          elementFormDefault="qualified"
          targetNamespace="Skateboots.com/order">
```

The root element also contains an attribute declaring the namespace for the purchase order document itself—`xmlns:po="Skateboots.com/order"`—and specifies that the default format for elements is to be namespace qualified.

Qualified element names are also called QNames

The entire `PurchaseOrder` element is structured as a complex type enclosing sequences for `ContactType` and `LineItem`. These element names are declared, referenced, and namespace qualified, using the `po`: prefix, within the top-level element, `PurchaseOrder`.

The specific simple types for other elements are declared with-in their specific sequences, such as `xsd:string` for `Name` and `xsd:positiveInteger` for `Zip`. The fixed attribute of the `ContactType` element limits, or restricts, the customer information to a single country type, United States.

The corresponding DTD is:

```
<!ELEMENT   PurchaseOrder (Contact, LineItem+)>
<!ATTLIST   PurchaseOrder
            OrderDate CDATA #IMPLIED>
<!ELEMENT   Contact (Name, Street, City, State, Zip)>
<!ATTLIST   Contact
            Country CDATA #FIXED "US">
<!ELEMENT   Name (#PCDATA)>
<!ELEMENT   Street (#PCDATA)>
<!ELEMENT   City (#PCDATA)>
<!ELEMENT   State (#PCDATA)>
<!ELEMENT   Zip (#PCDATA)>
<!ELEMENT   LineItem (SupplierName, ProductName,
            Quantity, UnitPrice)>
<!ELEMENT   SupplierName (#PCDATA)>
<!ELEMENT   ProductName (#PCDATA)>
<!ELEMENT   Quantity (#PCDATA)>
<!ELEMENT   UnitPrice (#PCDATA)>
```

The example contains a DTD for the same purchase order instance document shown previously. The DTD is a bit easier to define for use in validation than the schema, but schemas provide better flexibility for defining semantics. The DTD is simpler to understand and use, especially for basic document validation. The declaration of valid elements for the associated instance document is very straightforward, using the `!ELEMENT` tag in the DTD.

DTDs are easier for validation but limited for semantics

Also, basic structural information, such as the sequence of elements contained within another element, is very straightforward to define. For example:

```
<!ELEMENT LineItem (SupplierName, ProductName,
          Quantity, UnitPrice)>
```

Alternatives to Schemas

XML schemas are relatively new and somewhat unusual among W3C specifications in that the W3C initiated the work from scratch. Many other W3C initiatives, such as XML, SOAP, and WSDL, were started by small groups of individuals or companies seeking to develop a solution to a particular problem. Later, the members of the group

submitted the material to the W3C for possible adoption.

Schemas, on the other hand have taken a relatively long time to develop and are fairly difficult to understand, at least in their entirety. Many W3C members, including some of the original authors of XML, such as James Clark, feel that the result is much too complex, is difficult to learn and use, and did not meet the original goals for simplicity.

Several alternatives to schemas have been proposed, and DTDs remain in widespread use for validation, despite the fact that schemas are intended to replace them. For applications other than document validation, however, schemas are the language of choice, and that's why Web services technologies are defined using schemas.

This element declaration indicates that `SupplierName`, `ProductName`, `Quantity`, and `UnitPrice` are required sub-elements of `LineItem`. However, no data type information is available for document type definitions, namespaces are not used, and very limited structure information is available, making schemas required for most Web services applications.[4] As mentioned previously, the one place a DTD is likely to be used is for simple document validation, when a Web service interaction sends a complete document such as this purchase order.

[4] Schemas are intended to replace DTDs and do, in fact, subsume all DTD functionality.

Processing XML Documents

The two most popular programming APIs for parsing XML are the document object model (DOM) from W3C[5] and the Simple API for XML (SAX), which was collaboratively developed and is maintained by the members of the XML-DEV mailing list.[6]

[5] See http://www.w3.org/TR/DOM-Level-2-Core/.

[6] See http://www.saxproject.org/.

DOM and SAX APIs allow parsing XML documents

The major difference between SAX and DOM is that the DOM API provides a generic object model to represent the structure of documents and a standard set of interfaces for traversing and manipulating them. Although vendors are free to use any data structures to support the standard DOM interfaces, most popular DOM implementations work only in main memory. The SAX API, on the other hand, works by firing callback events into the application as the document is parsed, element by element.

SAX reads through once; DOM supports multiple traversals

The SAX approach uses less memory and is more efficient for messaging and transformation, whereas the DOM API allows multiple passes through the document, using it more like an in-memory database, or a repository that can be searched multiple times.

In other words, if you're parsing the document only to do one thing with it, such as map it to an existing software program or database, SAX is probably more efficient. But if you're using the

document as a continuing source of data or as something with which to interact several times, DOM makes more sense.

SAX and DOM implementations are freely available for Microsoft and Java programming environments and are easily obtainable for other traditional or more specialized programming environments. Basic XML handling is therefore becoming a commodity.

XML parsers are commodities

Examples of DOM interfaces are `Node`, `Element`, and `Document`. Everything in the document object model is considered a node, but some nodes are also called document elements. For example, the root element is called the root element node *and* a document element. Many more operations and interfaces are available in the complete list of DOM interfaces. The following text contains a few simple examples:

DOM APIs assume a hierarchical document model

```
Node interface examples:
  +getParentNode()
  +getChildNodes()
Element interface examples:
  +getAttributeNS()
  +setAttributeNS()
Document interface examples:
  +createElement()
  +createAttribute()
```

Sample SAX operations shown in the following example are from the `ContentHandler`, `XMLReader`, and `Attributes` interfaces:

```
ContentHandler interface examples:
  +startDocument()
  +startElement()
XMLReader interface examples:
  +getProperty()
  +setProperty()
Attributes interface examples:
  +getValue()
  +getType()
```

SAX APIs read through the document and process events

Both SAX and DOM interfaces can be mixed in a single program. Together, they provide XML processors with the capability to process documents linearly with greater efficiency and as a hierarchical information resource for multiple passes.

Namespaces

Once you have multiple XML documents, you need a way to scope element names within each respective document. Namespaces provide that mechanism and are also used for other purposes.

Namespaces scope, or qualify, element names

XML namespaces create unique prefixes for elements in separate documents or applications that are used together. Namespaces are also sometimes used as unique keywords that indicate specific processing semantics to be interpreted when the documents are processed.

Namespaces are used primarily to avoid problems that might be caused if the same element name appears in multiple related documents or document fragments. This is a significant issue for Web services, as they often involve handling multiple related documents at the same time. For example, a basic Web services interaction has at least four related documents:

- The instance document message carrying the data
- The SOAP envelope schema defining the message format
- The WSDL instance document describing the interface
- The WSDL schema validating the interface definition

Namespaces avoid naming clashes when multiple documents are processed

Depending on the use of other optional technologies, there can easily be many more documents, each distinguished using its own namespace.

Some namespaces will be given to you, but others you will have to make up. Namespaces are usually modeled as uniform resource indicators (URIs)[7] in the familiar format of the Web. For example:

[7] URIs include uniform resource locators (URLs), which point to files and other Web resources, and uniform resource names (URNs), which are simply names and don't point to anything. In practice, however, URIs can be considered equivalent to URLs.

```
xmlns:myns="http://www.xmlbus.com/namespaces/WSDL"
```

Web services technologies typically include namespaces

This example assigns the prefix `myns`—shorthand for "my namespace"—to a namespace derived from the xmlbus.com URI. By using a namespace prefix, each element name consists of not only the local name defined within the document itself but also the namespace URI. The full element name includes the namespace URI, whatever it is, when namespaces are used. Prefixes are used to shorten the namespace URI when it needs to be referenced within the document.

URIs are recommended but not required for use in namespaces. URIs are likely to be unique because they typically include host names that are registered with DNS. However, XML parsers

do not validate the namespace URIs and do not check uniqueness. You can make up and use any string of characters that you want. For external use, that is, over the Internet, it makes sense to stick with Internet conventions, such as the unique URI-based name.[8] However, if you are merely testing or using Web services in a local area network or other controlled environment, you can certainly be more free and flexible with what you use for a namespace string.

[8] Universally unique identifiers (UUIDs) are other good candidates for ensuring namespace uniqueness.

URIs are recommended for namespaces

Namespaces also can be used to indicate a requirement for a certain behavior to the XML processor. When the SOAP encoding namespace appears in a SOAP message, for example, the SOAP processor knows that it has to use the encoding mechanism described in the SOAP specification for serializing and deserializing the message.

Namespaces can also indicate semantics

A namespace URI is normally simply an identifier intended to be unique for the purpose of qualifying the element names within the XML document in which the namespace is used. However, if something at the end point is referenced by the namespace URI, it is possible to retrieve it and to use it for validating the document, which may also be an XML schema, as URIs point to Web resources. The idea of placing a schema at the end point referenced by a URI is that the schema can contain additional information about the element names within a namespace—for example, to help validate them or to define their data types.

Namespace URIs can reference schemas

Namespace Reference Ambiguity

Having namespace URIs reference files in which schemas are stored leads to ambiguity: Is something there or not when you're processing the XML file? If there is, how do you reconcile what's there with what's defined in the current XML document and/or other associated schemas and DTDs? XML processors dealing with Web services technologies need to be able to handle either situation successfully.

Namespaces are often used at the highest-level element possible so that their scope is as broad as possible. However, namespaces can be used on an element at any level. For example:

```
<sktb:boot xmlns:sktb=http://www.xmlbus.com/skateboots/>
```

Namespaces can be used at any element depth

In this example, `boot` is the XML element name, and `xmlns` is the namespace construct identifying the prefix `sktb` as a namespace for the element name; thus the namespace URI becomes part of the element name. The namespace URI is `http://www.xmlbus.com/skateboots`, but it could just as easily have been `1234:abcdefg` or anything else. The namespace qualifies all element names within the element for which it's defined, meaning that the best use of them is typically at the highest level, or overall enclosing element. Namespace-aware tools deal correctly with namespaces, but older DOM-oriented tools propagate namespaces simply as attributes of the elements.

Some Web services use the namespace construct as shorthand to reference the location of methods to be executed. For example:

```
<p:GetOrderStatus
xmlns:p="http://www.xmlbus.com/Skateboots/OrderEntry">
  <OrderID>12345</OrderID>
</p:GetOrderStatus>
```

Namespaces can indicate the location of executable programs

In this way, you can associate a URI with a Java object, which in this example is the `OrderEntry` object. The input parameter is included within the enclosing element (`<OrderID>`). The service name is included in the namespace—and therefore also unique to the namespace—via its prefix reference in the element name <p>.

Transformation

Because XML document instances can have multiple associated schemas, it's important to be able to map and then transform a document instance from one schema format to another. Many vendors offer such mapping tools, usually based on the Extensible Stylesheet Language: Transformations (XSLT) specification.

Transformation among multiple document formats is often required

Mapping data for compatible schema instances allows document instances to be transformed from one application-specific format to another. Of course, this means that a common understanding of the same contents of a document is possible, even when it's formatted or structured differently.[9]

[9] Some efforts are under way at W3C to address this issue formally: the XML Information Set and Canonical XML. The XML Information Set specification defines a consistent list of XML document parts, and Canonical XML defines a way to reduce an XML document to its essence so that two documents with identical contents but different structures and elements can be compared.

XSLT is typically used to transform one XML format to another

The problem of converting native data to XML is typically handled by using proprietary extensions to transformation engines, such as IONA's XML converter. These tools work very much like XML transformation but allow the input document to be in a native data format, such as ASCII or SQL. The output document is still XML. In fact, XSLT can be used with native data formats as long as someone writes a custom parser for that native data format and makes it appear to the XSLT engine as if it were in XML format. Transformation among multiple standard message formats for Web services is as important as conversion between EBCDIC and ASCII is for file transfer operations.

Mapping tools convert native data to XML

XSLT

XML unites the worlds of documents and data, and transformation technology is needed for both worlds: to put data into documents and vice versa and to transform one document format to another. XSLT emerged partly as a refinement on how to separate text from its presentation, such as displaying one document on both a PC browser and a Wireless Application Protocol (WAP—see www.wapforum.org) phone. In addition, XSLT was designed to support mapping data between XML formats.

XSLT is a transformation technology

XSLT is part of XSL (Extensible Stylesheet Language), which was developed to transform XML documents into presentation formats for screen, paper, or spoken word. XSLT has broader application for Web services, especially in transforming one form of XML to another.

Extensible style sheets are the foundation of transformation

As with many of the markup-language technologies described in this book, the content of a document needs to be separated from its format. Therefore XSL was split into XSLT and XSL-FO (Formatting Objects). Luckily, Web services are not concerned with formatting XML for text or voice, and we can focus on XSLT. Many applications of XSLT use Cascading Style Sheets (CSS) to specify how to display an XML document, although this popular application of transformation technologies is not directly relevant to Web services.

Transformation originated in the separation of content and format

XSLT requires a parser. Either the DOM or the SAX parser can be used, although a SAX parser is typically used to create its DOM structure, which is a node tree structure much like the result of a DOM parsing operation. XSLT defines a common set of transformation rules for mapping the node structure into something else. XSLT applies an XSLT *style sheet* to an XML document in order to produce the output document as a transformation of the input document. This is the same

way XML is transformed to HTML; another document is simply a different kind of target for the transformation.

XSLT works with both DOM and SAX

XSLT implementations are typically called XLST processors. Microsoft provides one, and Saxon is an open source Java implementation by Michael Kay.[10] XSLT evolved from Document Style Semantics and Specification Language (DSSSL), a technology related to Standard Generalized Markup Language (SGML). This parallels the evolution of SGML into HTML and XML, in which separation of content from presentation was an important goal.

[10] See http://saxon.sourceforge.net/ for further information on this open source project. A good book on XSLT is Michael Kay's *XSLT Programmer's Reference*.

XSLT processors are readily available

Once you achieve this kind of separation, you have to have a way to map content to multiple presentation formats. Thus the focus on style sheets and declarative rather than procedural languages was found more appropriate to the transformation of structured data into multiple formats.

Schemas or DTDs can be used to convert from one document instance format to the other, although Web services technologies use schemas. If you need to convert a document instance from one schema format to another, you can use XSLT to do the transformation.

Transformation maps between schema definitions

XSLT style sheets define rules for transforming an input document into an output document and can be used to convert any type of XML document to any other. XSLT is also capable of calling out to a program and passing data from the document to the program. Finally, XSLT can generate an output document in formats other than XML, such as ASCII text or SQL schema.

XPath

While XSLT was being developed at W3C, a requirement for an expression language to select parts of a document for transformation was identified. At the same time, the XPointer effort was developing an expression language for linking multiple XML documents. Rather than develop separate expression languages, W3C decided to combine the two; the result was XPath (see http://www.w3.org/TR/xptr).

XPath provides search expressions to select document parts

XPath is often used with XSLT to define search expressions that locate document elements to be used in the transformation. But XPath can also be used independently to link multiple documents, using specific expressions and search paths.

XPath defines basic search expressions, calculations, and string manipulations

XPath, as a sublanguage within an XSLT style sheet, is used for numeric calculations or string manipulations or for evaluating Boolean expressions. XPath defines the document path used to select elements for transformation. See the A Simple Example (Revisited) section for a complete XSLT style sheet.

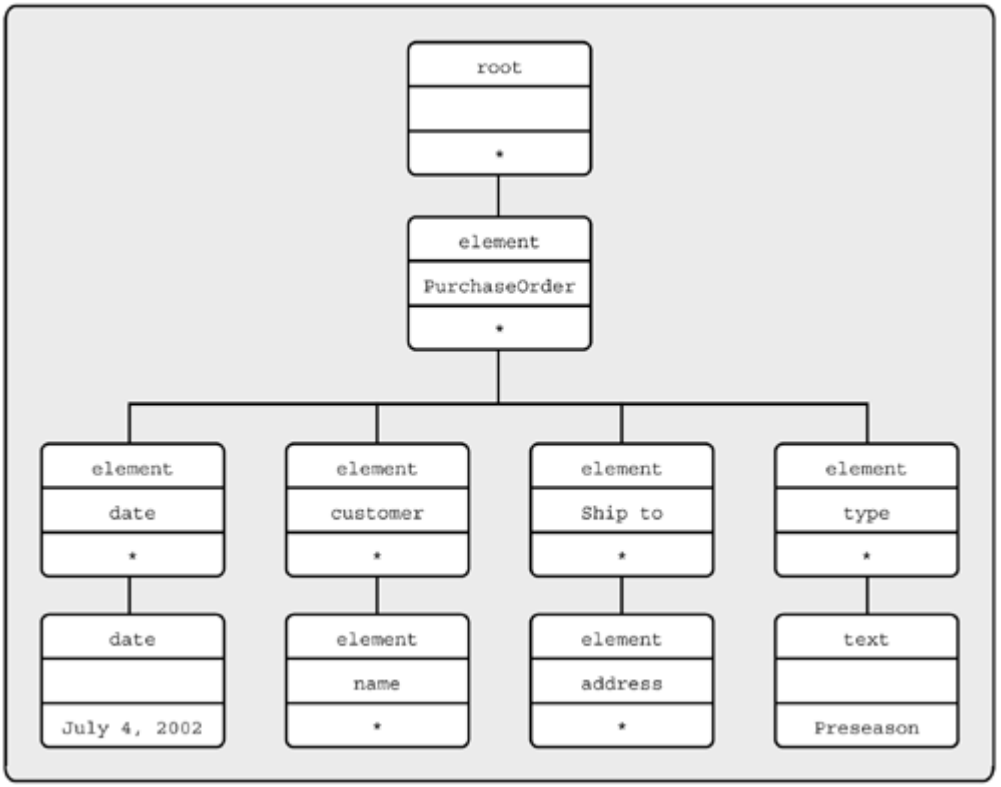
Document Structure

The hierarchical structure of XML is one of its key characteristics. One of the major requirements of Web services is to recognize the hierarchical nature of XML and map existing data into XML structures and to transform data among multiple XML formats. Conversion from native data to XML—another major requirement of Web services—is beyond the scope of XSLT and is typically handled by using proprietary products, such as IONA's converter/mapper tool. An XML document is always required as input to an XSLT operation.

XML documents are structured as hierarchies

Figure 2-1 illustrates the hierarchical tree structure of an XML document. Each document has a root and a top-level element that encloses the rest of the document. Elements within the top-level element in the diagram—`PurchaseOrder`, in this case—can have subelements, as depicted with an the asterisk (*). Elements that do not have subelements are illustrated using the data type of the element and the data of the element. The element name is put into a leaf node under which is placed the element type and value, if any.

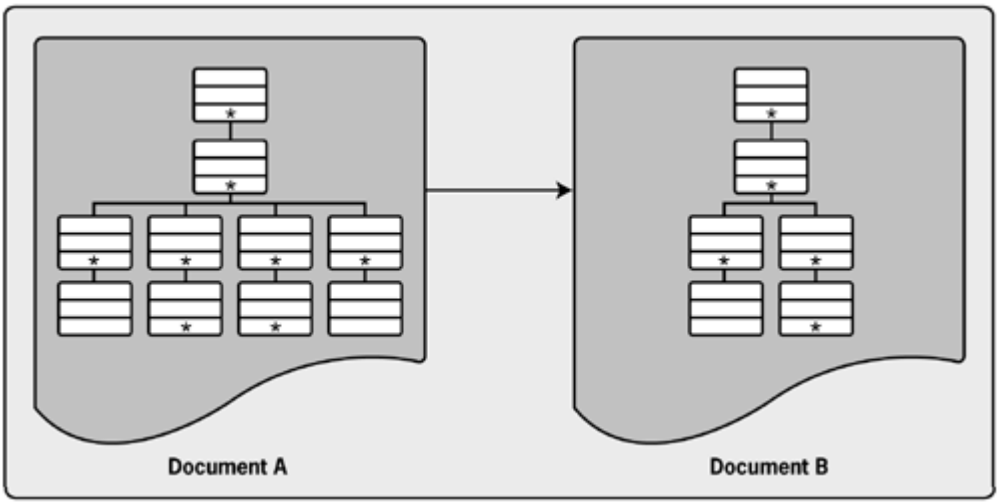
Figure 2-1. An XML document has a tree structure.



Parsers read through the XML document and create a tree structure like this in memory for XML programs to access. XSLT reads through the document structure and includes processing instructions, attributes, expressions, and so on, to produce a new document with a new structure—for example, transforming the date element into a Java or SQL date data type, if such a mapping were defined.

As shown in Figure 2-2, the structure of the target document can be different from that of the original document. In this case, for example, imagine that the XSLT rules define that only the customer data is to be extracted and moved to the new document. Transformations therefore represent an important technology for combining multiple Web services into a larger application, since Web services exchange XML documents of varying structure and content.

Figure 2-2. XML documents can be transformed or mapped onto different structures.



Target documents can be structured differently

If two or more documents are merged, perhaps to aggregate multiple orders from the same customer, or multiple elements are to be included from two or more services that contain the same element names, namespaces are used to distinguish between them. Namespaces play an important role in XSLT because elements in the input and output documents might be the same but have different meanings or pertain to different XML applications. Namespaces allow the XSLT processor to mix tags from different XML applications in the same document unambiguously and to produce correct results.

Multiple documents use namespaces to avoid naming clashes

As companies increasingly use XML for internal data representation, as more and more ERP systems export XML formats, and as more and more XML vocabularies are defined by standards bodies, the requirement for transformation will increase. Also, the ability to extract information from XML documents is likely to be very popular, such as extracting the relevant lines from a purchase order or fitting together a Web services interaction containing partial data into a larger document context. All this can be accomplished via transformation technology, of which XSLT is the leading implementation.

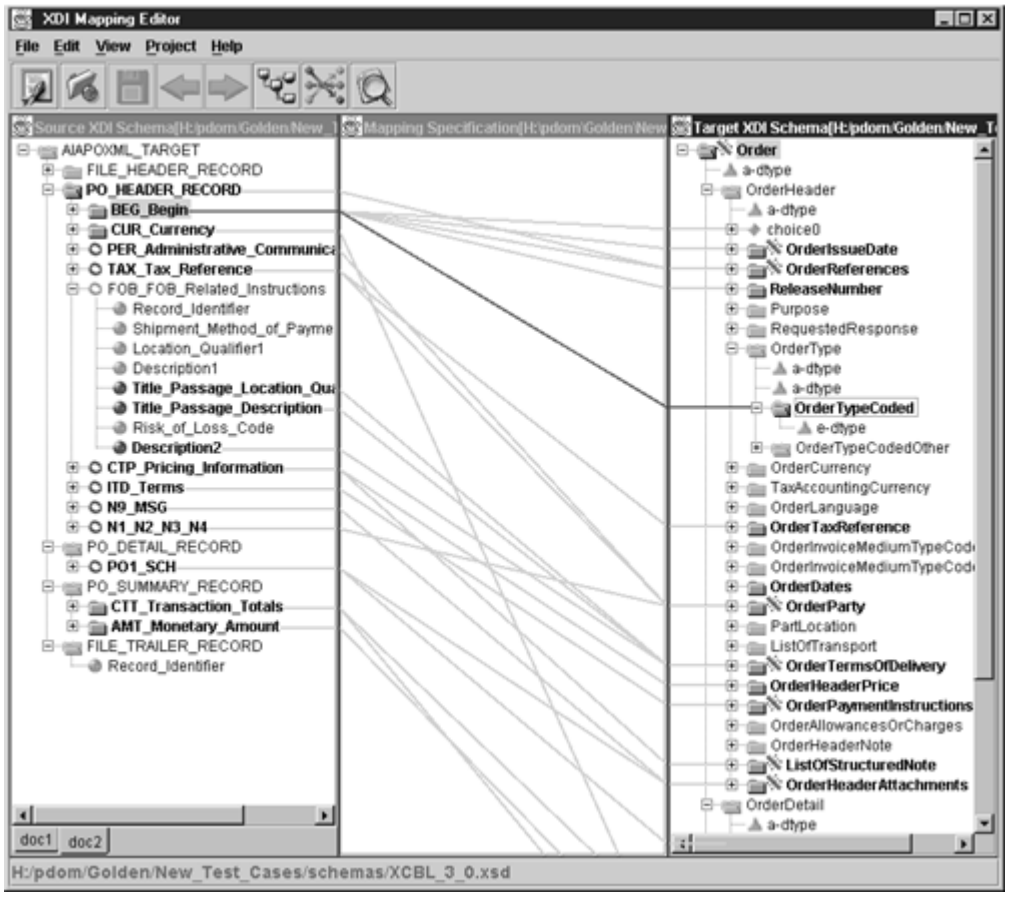
Transformation requirements are likely to increase

Mapping Tools

Many tools are available for editing XML documents, converting data to and from XML, transforming documents from one XML format to another XML format, and storing, retrieving, and searching XML documents.[11] Figure 2-3 shows a complex purchase order document being transformed, or mapped, from its original format to a new format.

[11] For a good starting point, check www.xml.org.

Figure 2-3. IONA's XML transformation tool can map a complex purchase order.



Many vendors offer mapping tools for conversion and transformation

The object of this sample mapping exercise is to relate data items from a company's generic purchase order schema to a specific, externally accepted schema, such as Commerce One's. The result of the mapping exercise is a generated XSLT style sheet that can transform an instance document conforming to the original purchase order into an instance document conforming to the Commerce One purchase order format. The transformation from one schema format to another is therefore accomplished in stages: The mapping between the two schemas is identified, a style sheet is generated, and, finally, the new instance document is produced as a result of executing the transformation.

A Simple Example (Revisited)

The following example contains an XSLT style sheet that transforms the XML document from the Skateboots Company simple customer example back to its original ASCII text:

Transforming from XML to ASCII

```
  <?xml version="1.0" ?>
-                                                <xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
```

```
  <xsl:output method="text" indent="yes" />
- <xsl:template match="*">
  <xsl:apply-templates />
  </xsl:template>
- <xsl:template match="CustomerNumber">
  CustomerNumber:
  <xsl:value-of select="." />
  </xsl:template>
- <xsl:template match="CustomerName">
  CustomerName:
  <xsl:value-of select="." />
  </xsl:template>
- <xsl:template match="CustomerAddress">
  CustomerAddress:
  <xsl:value-of select="." />
  </xsl:template>
- <xsl:template match="CustomerPhone">
  CustomerPhone:
  <xsl:value-of select="." />
  </xsl:template>
- <xsl:template match="PostalCode">
  PostalCode:
  <xsl:value-of select="." />
  </xsl:template>
- <xsl:template match="CreditLimit">
  CreditLimit:
  <xsl:value-of select="." />
  </xsl:template>
- <xsl:template match="CreditRating">
  CreditRating:
  <xsl:value-of select="." />
  </xsl:template>
  </xsl:stylesheet>
```

The simple XSLT style sheet shown in the example is used as the XSLT processor reads through the input XML document shown, as in the A Simple Example section. The `xsl:template match="*"` instruction asks the XSLT processor to match all nodes or elements in the document. The `xsl:apply-templates` instruction causes the element children to be processed: in this case, all child elements of the document root. When an element match occurs between the input document and the style sheet declarations, the following XPath expression selects the text value from the element contents: `xsl:value-of select="."`. The period in the expression is the XPath operator, which indicates that the text value of the element is to be extracted. At the top of the style sheet, the output method was defined to be `text`.

Style sheets are input to the transformation process

The output in the following example is the ASCII output from using XSLT to apply the preceding transformation style sheet to the XML example in the A Simple Example section:

Transformation output can be XML, ASCII, or another format

```
CustomerNumber:   12345
CustomerName:     Joe's Boots
CustomerAddress:  500 High Street
CustomerPhone:    555 123 4567
PostalCode:       12345
CreditLimit:      1000000
CreditRating:     5
```

Transformation technology is also capable of calling out to such programs as Java beans and classes.

XML Specifications and Information

XML in general is represented by a set of specifications published by the World Wide Web Consortium (W3C). Many applications—utilities, document formats, and tools—incorporate one or more of these specifications. Two aspects of XML are of primary importance to Web services: the XML data representation format, or instance of the data, and the associated schemas that define semantics for the data. However, many other XML-related technologies are of interest and significance to Web services and are incorporated into Web services technologies and the XML programs and tools that implement them.

XML is used for a wide variety of applications

XML Specifications Related to Web Services

The following XML specifications are related to Web services:

- Extensible Markup Language 1.0 (http://www.w3.org/TR/REC-xml): The basic specification, first published in February 1988; second edition, October 2000
- XML Base (http://www.w3.org/TR/xmlbase/): An XML v1.0 addendum that describes how to derive the base URI for resolving URI references in XML elements and attributes, such as references to associated schemas, document fragments to be included, and so on
- XML Names (http://www.w3.org/TR/REC-xml-names/): An XML v1.0 addendum that describes how to uniquely qualify element names when processing multiple related documents (usually referred to as namespaces)
- DOM (http://www.w3.org/TR/DOM-Level-2-Core/): Document object model—in several parts—the definition of platform- and language-neutral interfaces that allow programs and scripts to access and to update document structure and contents
- XML Schema (http://www.w3.org/TR/xmlschema-1/, http://www.w3.org/TR/xmlschema-2/: Parts 1 and 2—data types and structures using XML
- XML Path Language (http://www.w3.org/TR/xpath): Expressions for finding, evaluating, and extracting information from XML documents
- XML Pointer Language (http://www.w3.org/TR/WD-xptr): Based on XPath, identifies document fragments for locating internal structures and external parsed entities
- XML Linking Language (http://www.w3.org/TR/xlink/): Hyperlinks that point from one XML document to another, including fragments
- XSL Transformation (http://www.w3.org/TR/xslt): A technology that transforms the structure and contents of one XML document into another

- Canonical XML (http://www.w3.org/TR/xml-c14n): A method of generating a physical canonical representation of a document such that its contents can be compared to another document of a different structure
- XML Information Set (http://www.w3.org/TR/xml-infoset/): A consistent set of definitions for other specifications to use when referring to information in a well-formed XML document
- XML Inclusions (http://www.w3.org/TR/xinclude/): A definition of merging multiple XML Information Sets into a single composite Information Set

Some of these specifications were developed with direct reference to others. For example, the XML Base and XML namespaces specifications were added directly as addenda to the XML v1.0 specification and thereby included by reference within the basic standard. Other specifications, such as XML Pointer Language, were developed such that they may or may not be used with the basic standard and therefore may or may not be supported by a particular XML tool. Most of these specifications are undergoing change, and working drafts of the new versions are often available.

Many XML specifications are related

General Information

Many sources for XML information exist. The specifications listed here are of particular relevance to Web services and are excellent sources for further detail on the fundamental standards and technologies outlined here. The generic XML home page maintained by the W3C is an excellent starting point (see http://www.w3.org/XML/). That page provides links to all the major specs, working drafts, notes, and other technical publications of the W3C pertaining to XML. The page shows the latest status and updates on the XML specifications and related technologies. For example, new versions of XML and XSLT are well under way.

It's easy to find more information on XML

XML Is Daunting

Wading through the numerous XML specifications and applications is a daunting task: There are so many of them and the relationships among them Byzantine. Do you need a validating or a nonvalidating XML processor? Are namespaces required? What about transformation: Do you use SAX or DOM or both?

Imagine the difficulties of formalizing and regularizing completely unstructured, free-form information. It's one thing to deal with it in the context of information to be displayed in a browser or printed out but quite another when it's intended for software integration. It's likely that XML processors, like the XML community itself, will begin to split between the markup and data-oriented applications. It's too much to understand, otherwise: There has to be a way to narrow down the scope. For some people, that scope is defined by the XML Information Set specification, which lists the XML parts produced by parsing a given document. Even this is not enough, however, as the Information Set recognizes processing instructions and DTDs, which are not used in Web services. Nonetheless, an authoritative way is needed to represent the essential parts of an XML document intended for use in software-based applications.

A good source of summary information on XML and W3C's work in progress is a document called *XML in 10 Points.* It's available at http://www.w3.org/XML/1999/XML-in-10-points.

The W3C provides a summary document

There's even a report from a task force that studied the difference between URLs and URIs. They mean essentially the same thing, although there's a difference between a locator and an indicator, or name, when the name is just a name and doesn't refer to anything. However, the same form is possible for both uses. (See http://www.w3.org/TR/uri-clarification/.)

Clarification on URIs and URLs

Summary

XML is used in a broad variety of applications. Its main applications in Web services are in data formatting, serialization, and transformation. Web services communicate by exchanging formatted instances of XML documents that carry data. Web services are described using XML schemas that define data types, structure, and semantics. A wide variety of XML specifications is used in Web services technology definitions, including schemas, namespaces, extensible style sheets, and others.

The main advantage XML offers Web services is data independence so that data types and structures are not tied to the underlying implementations of the services. Previously, data types and structures for distributed computing were defined within individual programming languages, database description languages, and middleware interface description languages. To take advantage of the data independence, applications must convert data into XML and transform data out of XML into its native format. Although XML began as a document markup language, a large portion of the XML community is now focusing on XML as a technology for data description and serialization.

Chapter 3. Describing Web Services: WSDL

Web services expose a software-oriented view of a business or consumer function with which applications may interact over the network. To successfully enable such an interaction using a Web service, it must be described and advertised to its potential consumers. Furthermore, users must be able to find out about how to interact with the service: what data it expects to receive, whether it delivers any results, and what communication protocol or transport it supports.

In a somewhat primitive form, Web services have been around for a long time. It's possible to send a search string or a stock quote request by appending information to URLs. However, this format is of limited usefulness and scope for many reasons, not the least of which is the size limit and the usability of a URL for passing data. It's also difficult to program and to understand, and no consistent, standard approach has been defined for describing it.

Web services have been around for a long time in primitive form

The standard form of Web services puts the request and the reply into XML documents, that is, into the resources that are referenced by the URLs rather than within parameters of the URLs themselves. The emerging standards for Web services also facilitate a more universal approach, so that every Web site is not deciding how to use extended URLs individually. The following example illustrates the most primitive type of Web service:

```
http://internal.iona.com:8080/iona/phonelist.jsp?search=vinoski
```

Here, the input data is carried as a parameter to the URL address of the Web page implementing the service—in this case, an employee search function that returns an e-mail name and a telephone number.

WSDL Basics

The Web Services Description Language (WSDL) specification was created to describe and publish the formats and protocols of a Web service in a standard way. Web service interface standards are needed to ensure that you don't have to create special interactions with each server on the Web, as you would today, using the extended URL approach from a browser.

WSDL establishes a common format for describing and publishing Web service information

WSDL elements contain a description of the data, typically using one or more XML schemas, to be passed to the Web service so that both the sender and the receiver understand the data being exchanged. The WSDL elements also contain a description of the operations to be performed on that data, so that the receiver of a message knows how to process it, and a binding to a protocol or transport, so that the sender knows how to send it. Typically, WSDL is used with SOAP, and the WSDL specification includes a SOAP binding.

WSDL elements describe data and operations on it

WSDL was developed by Microsoft, Ariba, and IBM, and v1.1 of the specification was submitted to the W3C, which accepted WSDL as a *note* and published it on the W3C Web site.[1] Twenty-two other companies joined the submission, comprising at that time the largest number of W3C members ever to support a joint submission. WSDL therefore already enjoys broad-based support, and many companies offer implementations of WSDL in their Web services products. In fact, with such near unanimity within the vendor community, it could be said that the WSDL specification provides the de facto definition of a Web service description. However, it is very likely that a W3C working group will nonetheless make significant improvements and changes.

[1] See http://schemas.xmlsoap.org/wsdl/for the schema and http://www.w3.org/TR/wsdl for the specification.

WSDL was developed collaboratively by IBM, Microsoft, and Ariba

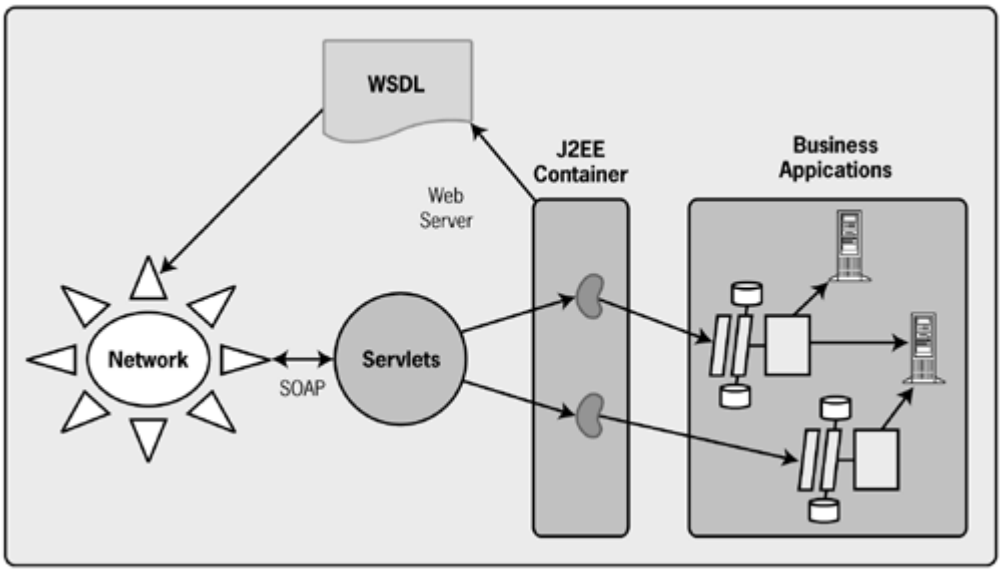
Both parties that participate in a Web Services "conversation" or interaction must have access to the same WSDL to be able to understand each other.[2] In other words, both the sender and the receiver of a message involved in a Web service interaction must have access to the same XML schema. The sender needs to know how to format the output message correctly, and the receiver needs to understand how to interpret the input message correctly. As long as both parties to the interaction have the same WSDL file, the implementations behind the Web services can be anything. This is the magic of WSDL: It provides a common format to encode and to decode messages to and from virtually any back-end application, such as CORBA, COM, EJB, JMS, MQ Series, ERP systems, and so on.

[2] Alternatively, a privately agreed on, shared XML schema file will do the same trick, but WSDL has already solved the problem, so why not use it?

Both parties to a Web service interaction need copies of the same WSDL file

As shown in Figure 3-1, Web services typically are implemented using programming languages designed for interaction with the Web, such as Java servlets or Application Server Pages (ASPs) that call a back-end program or object. These Web service implementations are also typically based on WSDL or represented using WSDL. That is, either new services can be generated from WSDL, or existing services can be described using WSDL. It's likely that both approaches will catch on, as designing and exposing Web services is bound to be an iterative process. Neither approach by itself is going to provide the best solution in all cases.

Figure 3-1. A business can use WSDL to advertise its Web services.



Web services are implemented using Web-oriented languages

Figure 3-1 shows an EJB described using a WSDL file and exported using a servlet over the network. The servlet listens to the network through either a Web server or a custom HTTP listener. In the example, the servlet accepts a SOAP message and forwards it to the beans that are represented by the WSDL file. The beans in turn implement, or connect to, the specific business applications being exposed to the network, providing the information that is carried in the SOAP body as input, if any, and returning any results back through the beans to the servlet. If results need to be sent back across the network, data is returned to the beans through output arguments and back to the servlet from which the reply goes back to the servlet. The servlet that provides the implementation of the WSDL file packages the data into a SOAP message reply to send back across the network.

In this way, simple extensions to existing Internet infrastructure can implement Web services for interaction via browsers or directly within an application, such as the one in Figure 3-1. As well as an EJB, the application behind the servlet could be implemented using .NET, JMS, CORBA, COBOL, or any number of proprietary integration solutions. Furthermore, Web services can represent B2B document-exchange interactions.

Web services definitions can be mapped to any language, object model, or messaging system

Exposing Objects and Beans Directly

It may not always make sense to expose a bean or an object directly as a Web service; more often, it seems likely that a bean or other program will be written specifically for the purpose of exposing a Web service. In other words, it's unlikely that existing programs will be a good fit or be built according to a design that fits Web service requirements, and special wrapper programs may need to be written to expose the level of granularity appropriate to a Web service.

WSDL Elements

WSDL breaks down Web services into three specific, identifiable *elements* that can be combined or reused once defined. Mapping from existing applications means mapping to these elements, which identify the contents and data types of the messages, the operations performed for the messages, and the specific protocol bindings, or transports, for exchanging the messages with the operations over the network. Within these elements are further subelements, or parts:

- **Data types**: the data types—in the form of XML schemas or possibly some other mechanism—to be used in the messages
- **Message**: an abstract definition of the data, in the form of a message presented either as an entire document or as arguments to be mapped to a method invocation
- **Operation**: the abstract definition of the operation for a message, such as naming a method, message queue, or business process, that will accept and process the message
- **Port type**: an abstract set of operations mapped to one or more end points, defining the collection of operations for a binding; the collection of operations, because it is abstract, can be mapped to multiple transports through various bindings
- **Binding**: the concrete protocol and data formats for the operations and messages defined for a particular port type
- **Port**: a combination of a binding and a network address, providing the target address of the service communication
- **Service**: a collection of related end points encompassing the service definitions in the file; the services map the binding to the port and include any extensibility definitions

WSDL files can be divided into up to three separate, reusable elements

Fortunately, these parts of WSDL usually are generated using tools that transform the existing back-end application metadata into XML schema information, which is then merged into the Web Services Desciption Language file. WSDL is typically generated by Web services products and tools, such as IONA's XMLBus Edition.[3]

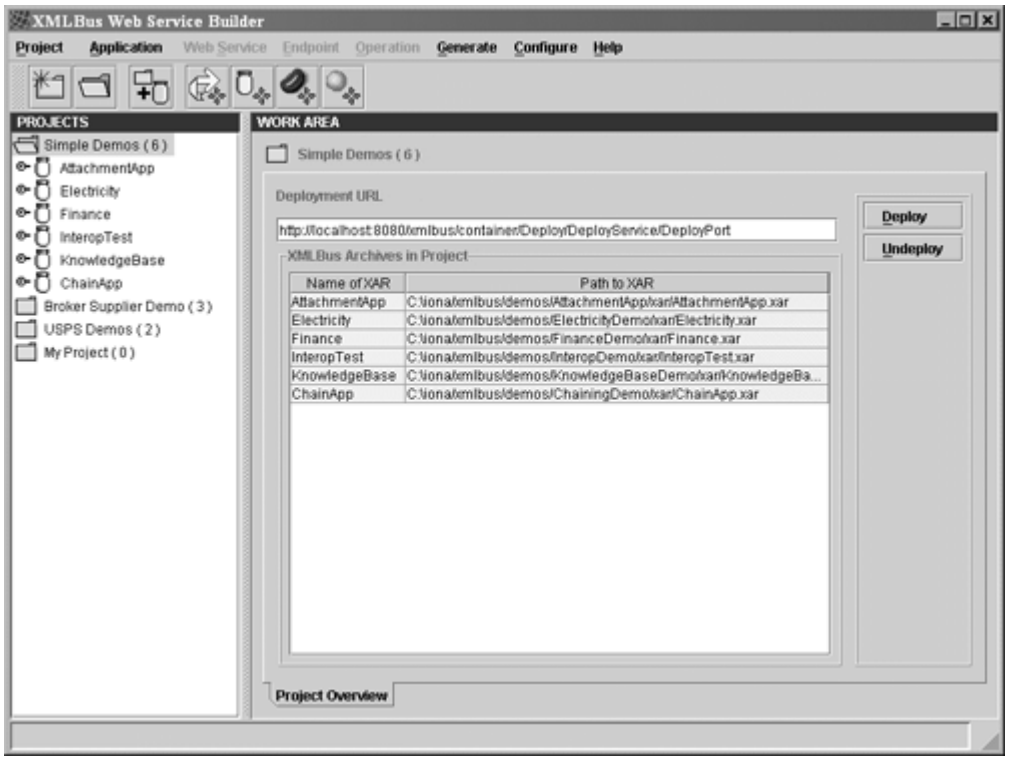
[3] See www.xmlbus.com.

WSDL parts usually are generated automatically using Web services–aware tools

IONA's XMLBus Edition provides several options for Web service generation. Figure 3-2 shows the Edition's main menu, which is displaying the list of sample demos included with the product.[4] These demos are prebuilt services, previously generated from Java classes and EJBs. The Edition also builds new Web services from Java classes, JavaBeans, and CORBA objects.

[4] An XMLBus Web services archive (XAR) file represents the XMLBus Web services container, which is deployable standalone or on an application server.

Figure 3-2. The XMLBus Edition's main menu lists the product's demos.

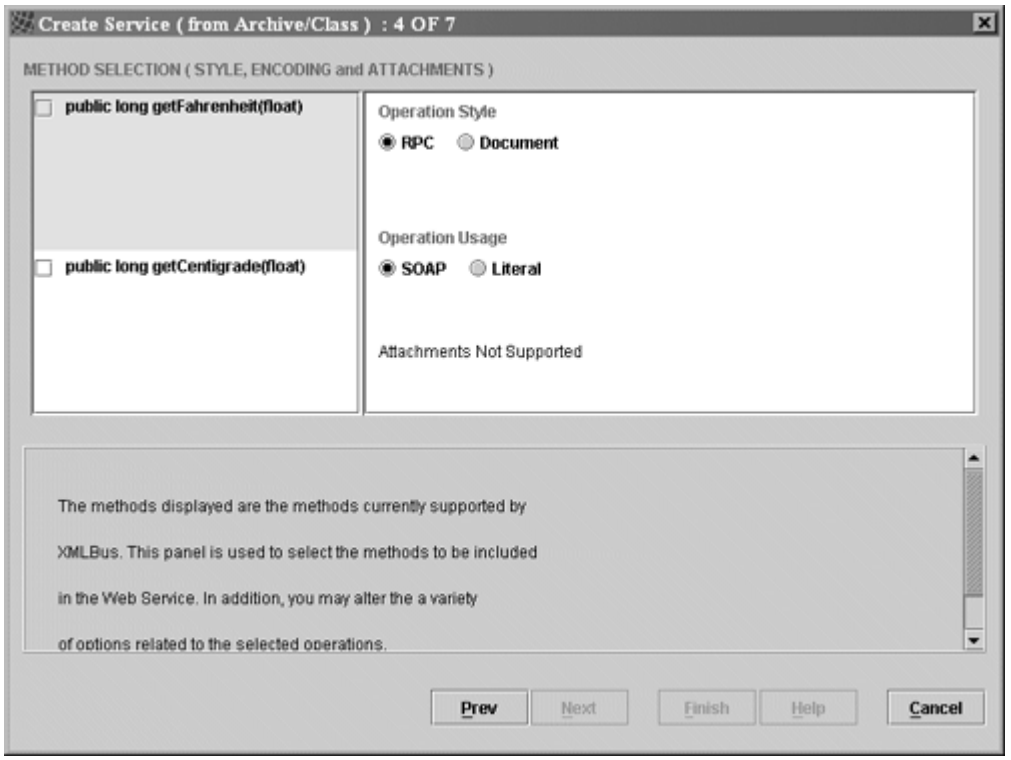


When building a new Web service, the Edition gathers necessary input and automatically generates the corresponding WSDL file.

The IONA XMLBus product builds a Web service using input forms such as the one in Figure 3-3, which asks the user to select the specific Java class methods for which a Web service is to be generated. The form also asks the user to select the document-oriented or RPC-oriented interaction style and to select either SOAP encoding[5] or literal encoding—XML/text—for the service. Attachments are supported for the document-oriented style. Another form allows the user to input a specific namespace for the service instead of having it generated by the product. Finally, the user can generate either a proxy client for Java 2 Platform, Micro Edition (J2ME) devices, such as PDAs or mobile phones, or a regular Java proxy. Most Web service implementers, such as IBM and Microsoft, offer similar tools for automatically generating WSDL schemas, proxies, and namespaces.

[5] See Data Type Mapping in Chapter 4.

Figure 3-3. Forms are used to select methods and interaction style to build a Web service.



The Extensible WSDL Framework

Like SOAP and the other XML integration framework technologies, WSDL is an extensible framework. The bindings for SOAP, for example, extend WSDL, as do the bindings for HTTP `GET` and `POST` and MIME. In this way, WSDL separates the abstract definition of end points and messages from their concrete network deployments, or data format bindings, which permits reuse of the abstract definitions.

WSDL is completely extensible to multiple formats and protocols

The three major elements of WSDL that can be defined separately are

- Types
- Operations
- Binding

As noted previously, WSDL has seven parts, but they are contained within these three main elements, which can be developed as separate documents and then combined or reused to form complete WSDL files. The major elements are divided according to their level of abstraction in the stack representing a Web service.

The data type declarations are the most abstract element of a service, defining the data items—the XML documents—to be exchanged by a Web service. The data types can be defined once, like include files, and referenced within any of the operations. The operations on the data types represent the next level of abstraction, defining the way data is passed back and forth. The binding, the final level of abstraction, defines the transport used to pass the message.

Defining Message Data Types

At its most abstract level, a Web service consists of an XML document sent to and/or received from a remote software program. The first job in defining a Web service, therefore, is to identify the data requirements for the software program implementing the Web service functionality.

Web service processing includes an XML mapping phase

A Web service needs to define its inputs and outputs and how they are mapped into and out of services. WSDL *types* take care of this. Types are XML documents, or document parts. WSDL allows the types to be defined in separate elements so that the types are reusable with multiple Web services.

WSDL uses basic XML schema types by default

Data types address the problem of how to identify the data types and formats you intend to use with your Web services. Type information is shared between sender and receiver. The recipients of messages therefore need access to the information you used to encode your data and must understand how to decode the data.

Types are typically defined using XML schemas; like other parts of WSDL, however, the types portion is completely extensible, and other type systems can be used instead. For example, if you know that the target of a particular instance of a Web service is a CORBA object, you can use the CORBA type system instead of the XML schema type system. You could also simply introduce another standard self-describing encoding, such as Abstract Syntax Notation One (ASN.1). This may be useful for local area network applications of WSDL, for which optimizing or bypassing the mapping stage—into and out of XML schema types—would be helpful.

Another option is to hard code the type definitions within the Web Services Description language file. Types can be defined within the WSDL element or inside other files referenced within that element.

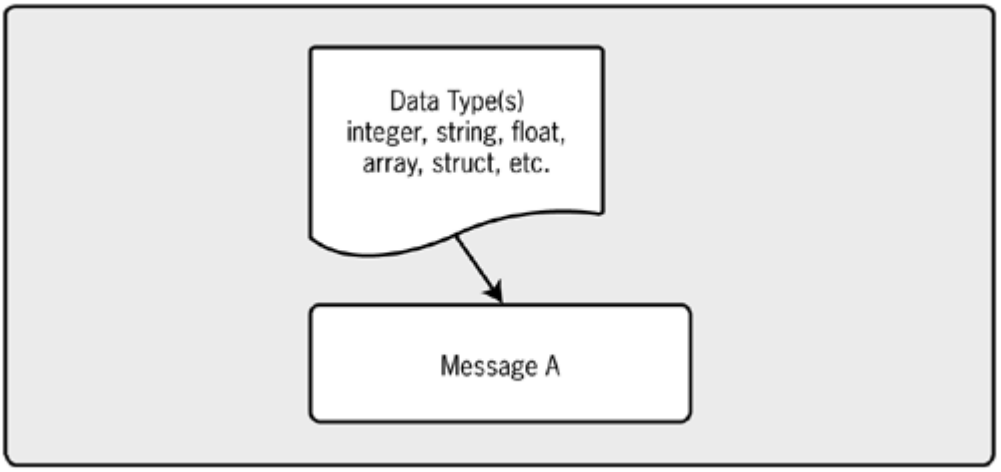
Because XML is inherently flexible, transformable, and extensible, the XML data defined for a Web service does not have to exactly match the data required by the program behind it; in fact, it probably should not. As long as the data can be manipulated in the mapping stage, it can be defined using a level of abstraction or format different from the original application. Because it's not executable by itself, a Web service includes a mapping stage, during which the XML data is mapped to the software program, as well as an execution stage, during which the program itself is run.)

A mapping stage is required to transform the XML data and the XML schema representation of a service into the software program that is executing it. The requirements for the message data and operations part of WSDL therefore are that they express enough of the data and semantics of the software program to allow a bridge to be constructed to it over the network using the capability of the transformation and mapping phase at each end.

WSDL is a loose representation of an object or a database system

As shown in Figure 3-4, one or more individual data types are mapped into messages. The same message can be mapped into multiple operations. Types are typically any of the XML schema-supported data types, such as integer, string, Boolean, or date, and can include complex types, such as structures and arrays, including those defined for SOAP. The data types are therefore either simple schema types or schemas that define complex types. As in the other areas of WSDL, types are not restricted to XML schemas, because no one expects a single type of system to be capable of describing all possible message formats for the Web.

Figure 3-4. Data types are mapped to messages.



The following example illustrates the type and message definitions for a Skateboots.com purchase order service that returns the total value of one or more purchase orders. The XML schema data types used in the WSDL file are mapped to messages using the schema element names.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PurchaseOrderService"
    targetNamespace="PurchaseOrderService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="PurchaseOrderService"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="PurchaseOrderService-xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <types>
        <schema targetNamespace="PurchaseOrderService-xsd"
             xmlns="http://www.w3.org/2001/XMLSchema"
                 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            <complexType name="PurchaseOrder">
                <all>
                    <element name="CompanyName" type="xsd:string"/>
                    <element name="Items" type="xsd1:ArrayOfItem"/>
                    <element name="Address" type="xsd1:Address"/>
                </all>
            </complexType>
            <complexType name="Item">
                <all>
                    <element name="Price" type="xsd:float"/>
                    <element name="PartID" type="xsd:string"/>
                    <element name="Description" type="xsd:string"/>
                    <element name="Quantity" type="xsd:int"/>
                </all>
            </complexType>
            <complexType name="ArrayOfItem">
                <complexContent>
                    <restriction base="SOAP-ENC:Array">
```

```
                    <attribute ref="SOAP-ENC:arrayType"
                          wsdl:arrayType="xsd1:Item[]"/>
                </restriction>
            </complexContent>
        </complexType>
        <complexType name="Address">
            <all>
                <element name="State" type="xsd:string"/>
                <element name="PostalCode" type="xsd:string"/>
                <element name="City" type="xsd:string"/>
                <element name="Line2" type="xsd:string"/>
                <element name="Country" type="xsd:string"/>
                <element name="Line1" type="xsd:string"/>
            </all>
        </complexType>
        <complexType name="ArrayOfPurchaseOrder">
            <complexContent>
                <restriction base="SOAP-ENC:Array">
                    <attribute ref="SOAP-ENC:arrayType"
                        wsdl:arrayType="xsd1:PurchaseOrder[]"/>
                </restriction>
            </complexContent>
        </complexType>
    </schema>
</types>
<message name="postPurchaseOrderRequest">
    <part name="order" type="xsd1:PurchaseOrder"/>
</message>
<message name="postPurchaseOrderResult">
    <part name="return" type="xsd:float"/>
</message>
<message name="postPurchaseOrdersRequest">
    <part name="orders" type="xsd1:ArrayOfPurchaseOrder"/>
</message>
<message name="postPurchaseOrdersResult">
    <part name="return" type="xsd:float"/>
</message>
```

When using SOAP, a message is carried as the payload of the SOAP request or response. That is, the WSDL message definition does not include any information that is mapped to the SOAP envelope, headers, or fault code. In other words, you can say that WSDL targets a layer of abstraction entirely above that of SOAP.

Information in WSDL does not map to SOAP headers

Defining Operations on Messages

The next level of abstraction, operations, addresses the requirement of a Web service to identify the type of operations being performed on behalf of a given message or set of messages. The operation is defined so that the Web service knows how to interpret the data and what, if any, data is to be returned on the reply.

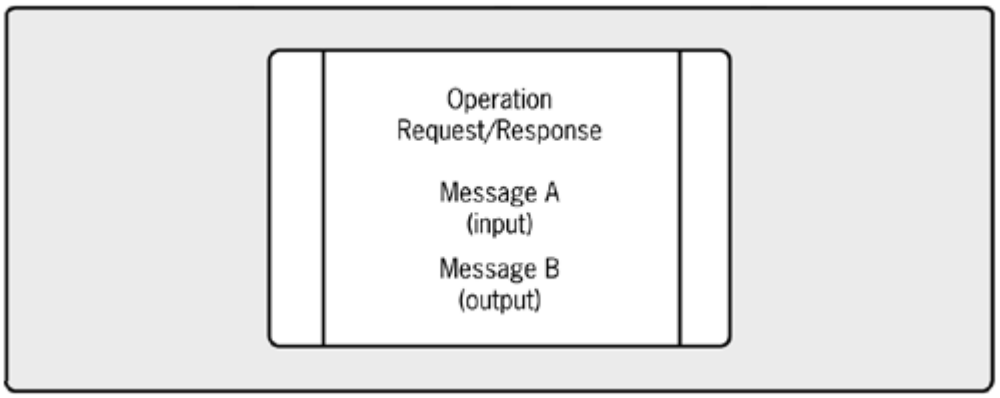
Once you have the data, define the operations

Operations are defined in correspondence to common message patterns, such as one-way and request/response. WSDL does not include specific definitions for other operations, but more

complicated interactions can be constructed by combining these basic types. For example, something like the cooperating partner profile specification from ebXML could be used to define a sequence of one-way and request/response operations in support of a complex message pattern interaction.

As shown in Figure 3-5, operations can group messages for input and output to match the pattern of the request/response message.

Figure 3-5. Operations group message types to match the message pattern.



WSDL has four types of operations:

- **One-way:** Similar to fire-and-forget, but more simply it means that the message is sent without a requirement to return a reply.
- **Request/response:** Similar to an RPC-style interaction; the sender sends a message, and the receiver sends a corresponding reply. (Some protocols may not guarantee that a response is returned for every request.)
- **Solicit response (no definition yet):** A simple request for a response with no input data. It's a request to get a message and does not involve sending a message, in the sense of a WSDL message consisting of one or more defined types. It's the reverse of the one-way operation.
- **Notification (no definition yet):** This type of operation defines multiple receivers for a message, similar to a broadcast, and often involves a subscription mechanism, as in publish/subscribe, to set it up.

Operations match request/response and other message patterns

Operations allow sequences of messages to be correlated into specific patterns without having to introduce a more complex flow specification. Operations are not the same as methods on objects, although certainly the input and output parameters defined for operations will normally map to method input and output arguments when the services are implemented using an object-oriented technology such as .NET, EJB, or CORBA.

Operations correlate messages into specific patterns but not flows

Operations may include optional fault messages, although their content is outside the scope of the specification. The following example illustrates a request/response operation definition for the Skateboots.com service.

```
<portType name="PurchaseOrderPortType">
    <operation name="postPurchaseOrder">
        <input message="tns:postPurchaseOrderRequest"
            name="postPurchaseOrder"/>
        <output message="tns:postPurchaseOrderResult"
            name="postPurchaseOrderResult"/>
    </operation>
    <operation name="postPurchaseOrders">
        <input message="tns:postPurchaseOrdersRequest"
            name="postPurchaseOrders"/>
        <output message="tns:postPurchaseOrdersResult"
            name="postPurchaseOrdersResult"/>
    </operation>
</portType>
```

Input and output messages are defined for the request and response operations, using the message definitions created in the previously shown elements of the WSDL file.

Request/response operations do not require use of the RPC attribute in the SOAP binding (see Extensions for Binding to SOAP later in this chapter), although it's probably a good idea. It's good programming practice to preserve within the SOAP binding, for example, the signature of an object to which the Web service is mapped. The `parameterOrde`r attribute lists message part names, and the order of the messages must match those of the RPC signature.

Although not required, it's good to map operations to SOAP correspondingly

There is no syntax to define a return value, but if a part name appears in both the input and the output messages it's an in/out parameter; if it's on input only, it's an input parameter. This convention makes it easier to map WSDL operations onto RPC bindings—for example, the RPC binding for SOAP.

SOAP also has a document binding, and Web services interactions will likely include both. For example, a purchase order is shared between the buyer and the seller of goods. Both the buyer and the seller may first exchange a copy of the same document. They may exchange information dynamically over the Web to fill in the fields on the form, such as available inventory, ship date, quantity discount, and so on. This will allow a more dynamic, real-time negotiation between buyer and seller to set the terms and conditions of a sale, based on a shared document.

Operations put input/output messages in correspondence, although it varies by transport what type of guaranteed correlation is available; SOAP does not require correspondence, although HTTP `GET` and `POST` do. Today, separate services have to be defined if you want to advertise both a document-oriented and a procedure-oriented service, but it seems likely that these may be combined in a future version of WSDL.

Mapping Messages to Protocols

After the data types and the operation types are defined, they have to be mapped onto a specific transport protocol, and an end point, or address to which the data is sent and at which it's possible to find and invoke the particular operation, must be identified. This operation is required because the same data types and operations can be mapped onto multiple transports, such as SOAP, BEEP, IIOP, JMS, and others, and a Web service can live at potentially multiple end-point addresses. This part of the WSDL solves the problem of how a transport expects to understand the data being passed—for example, it may be serialized according to the SOAP specification—and where to find the service—at an Internet IP address, intranet, LAN, and so on.

Now map messages to transports and end points

First, operations are grouped into *port types,* as shown in the previous example. Each port type is then mapped to one or more specific ports representing the various transports over which a service might be available; for example, a port type might be mapped to specific ports for HTTP `GET` and `POST`, SOAP, or MIME. Transport binding extensions are then mapped to each specific port in order to define the information necessary to offer a given service over a specific transport.

The transport binding extensions underneath the data types, operation types, and port types identify the receiver of the data and the operation to be performed. So for any given Web service, it's both necessary and possible to define the data, the operation on the data, and the place to which the data is sent and how.

Port Types

A port type is a logical grouping of operations, similar to type libraries in .NET, classes in Java, or an object's IDL (Interface Definition Language) in CORBA. If an operation is analogous to a method on an object and if messages are analogous to input and output arguments, a port type is analogous to an object definition that potentially contains multiple methods. But these analogies are not exact, because WSDL is extensible and is intended to provide a level of abstraction greater than what's provided by any of these object-oriented systems.

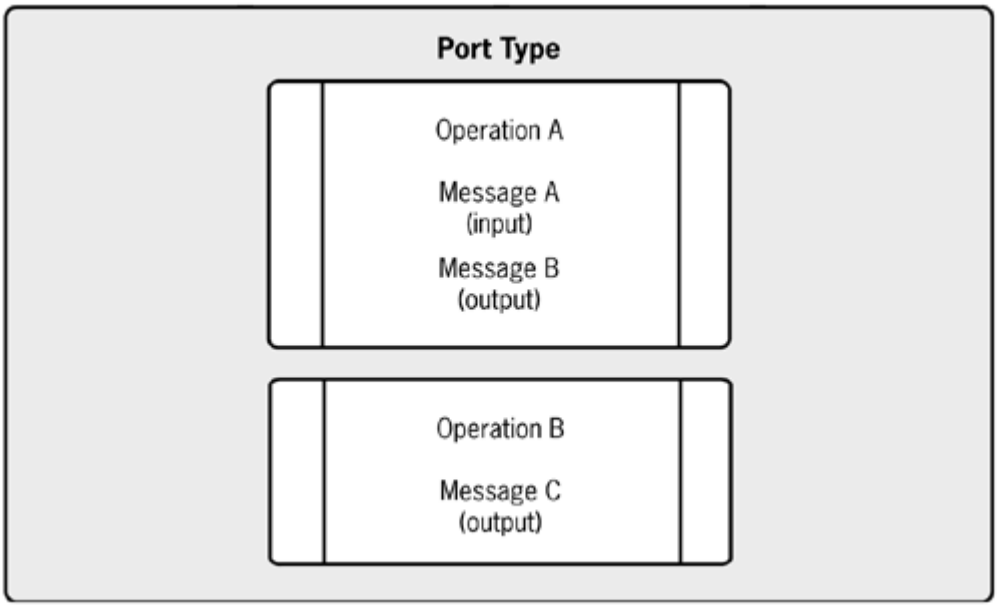
Port types identify to whom the message is sent and how

In other words, WSDL is an independent data abstraction that can be used for much more than simply mapping into and out of .NET, EJB, or CORBA objects. But when working with these object-oriented systems, it helps to understand the parts of WSDL that correspond to parts of these systems.

The request/response style uses different message definitions for the input and output messages; as with document passing, the one-way style uses a single type as a complete document. Both types of interactions can be defined within a given port type.

As shown in Figure 3-6, a port type represents a collection of operations, in the same way that an object represents a collection of methods. For Web services, the mapping between a port type and an object type or a class is quite natural. But because Web services are defined at a high level of abstraction, mappings can also be made to documents and procedure-oriented technologies.

Figure 3-6. A port type represents a collection of operations.



A port type is a collection of operations

Ports

A port is used to expose a set of operations, or port types, over a given transport. The port types can be grouped for one or more bindings, which is how the services are connected over the network. A port identifies one or more transport bindings for a given port type. To continue the comparison with object-oriented systems, a port is analogous to the transport. For example, a common transport for CORBA is IIOP; for EJB, it's RMI (remote method invocation) or RMI/IIOP; and for COM, it's DCOM, which is based on Distributed Computing Environment (DCE) RPC.

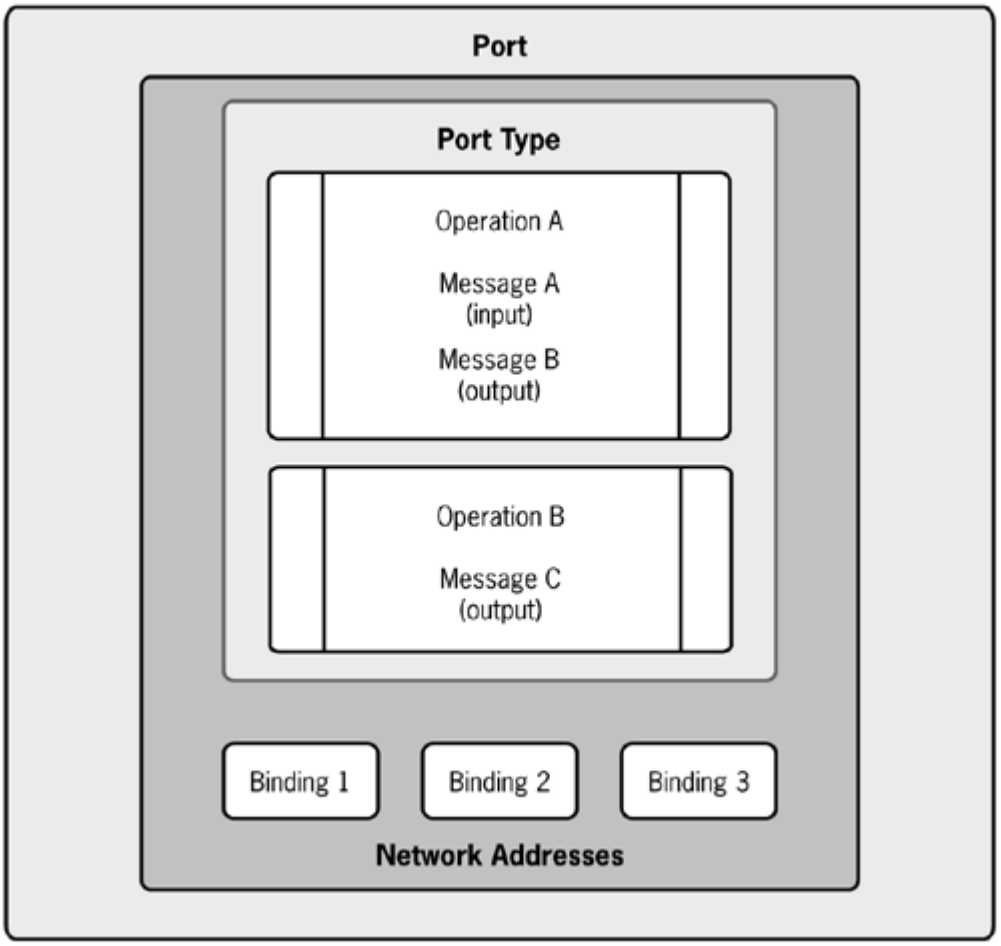
Ports are transport specific

These systems do not provide a truly equivalent definition mechanism with which to define a particular transport, although EJB and CORBA systems certainly can be and have been mapped to a variety of transports. But the mapping is not considered part of the object definition.

With WSDL, however, it's necessary to advertise within the service definition which transport bindings are available for a given collection of operations. The sending or discovering system at a remote network end point will typically access a published WSDL file via HTTP as a typical document `GET` operation but may then wish to negotiate with the receiver or publisher of the Web service to interact on a different transport that provides additional functionality.

Figure 3-7 illustrates the concept of a port in WSDL, which puts together the operations, the binding, and a URL defining a specific IP address at which the Web services implementation can be found. The following example illustrates the SOAP binding for the operations in the Skateboots.com purchase order.

Figure 3-7. In WSDL, a port combines operations, binding, and a network address.




```
<binding name="PurchaseOrderBinding" type="tns:PurchaseOrderPortType">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http/"/>
    <operation name="postPurchaseOrder">
        <soap:operation
            soapAction="PurchaseOrderService/postPurchaseOrder"
                style="rpc"/>
        <input name="postPurchaseOrder">
            <soap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/
                encoding/"namespace="PurchaseOrderService"
                use="encoded"/>
        </input>
        <output name="postPurchaseOrderResult">
            <soap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/
                encoding/"namespace="PurchaseOrderService"
                use="encoded"/>
            </output>
    </operation>
    <operation name="postPurchaseOrders">
        <soap:operation
            soapAction="PurchaseOrderService/postPurchaseOrders"
                style="rpc"/>
        <input name="postPurchaseOrders">
            <soap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/
                encoding/"namespace="PurchaseOrderService"
                use="encoded"/>
        </input>
        <output name="postPurchaseOrdersResult">
            <soap:body
```

```
                encodingStyle="http://schemas.xmlsoap.org/soap/
                encoding/"namespace="PurchaseOrderService"
                use="encoded"/>
        </output>
    </operation>
</binding>
```

Note the use of the SOAP Action, the SOAP encoding, and the RPC interaction style.

The transport binding can be done per operation. Ports by definition include the transport binding or bindings, which in the case of the SOAP binding includes a declaration of whether the interaction is request/response (RPC) or document passing (DOCUMENT). This is the SOAP binding style. Several other extensions to WSDL are defined specifically for use with SOAP, such as a way to define the SOAP Action[7] and input and output messages.

[7] WSDL will need to be modified for consistency with SOAP v1.2 once it's completed—for example, removing its use of the SOAP Action header, since v1.2 removes the SOAP Action feature.

Transport bindings are done for operations

```
<soap:operation
    soapAction=http://www.iona.com/GetLastTradePrice"/>

    <input>
    <soap:body use="literal"
        namespace="http://www.iona.com/stockquotes.xsd"/>

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>

    <output>
    <soap:body use="literal"
        namespace="http://www.iona.com/stockquotes.xsd/>

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>

</operation>
```

The SOAP encoding optionally can be used with WSDL, as shown in the example, and both options are explicitly supported. (See Extensions for Binding to SOAP later in this chapter for further information on the SOAP binding extensions.)

Bindings can be defined for other transports, such as SMTP, and extensions included specifically for them. The separation of the transport binding extensions from the definition of the port type allows one Web service to be available over multiple transports without having to redefine the entire WSDL file. Again, because it is designed to be completely extensible, WSDL allows other binding extensions to be used, such as for example for IIOP, .NET, JMS, MQ Series, and so on.

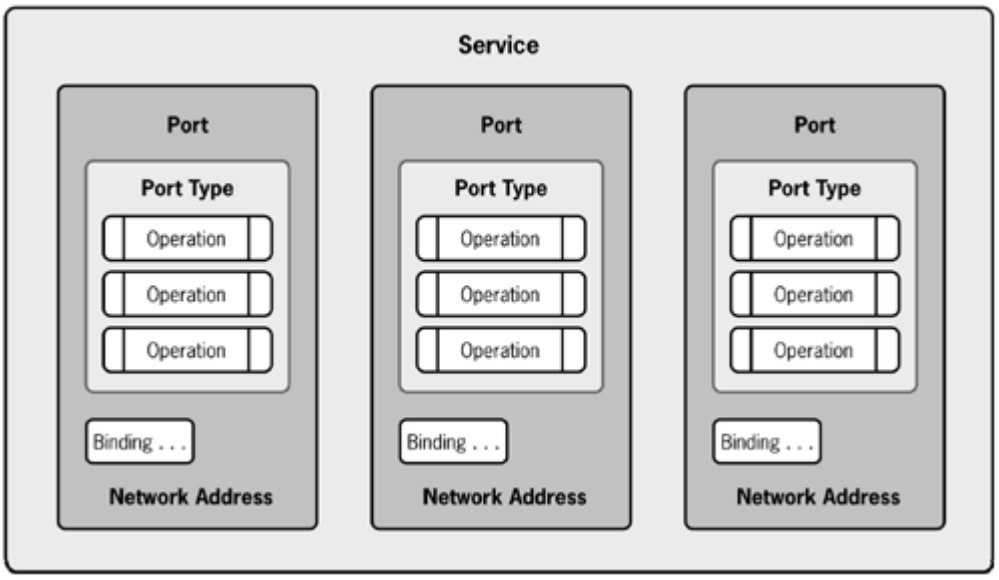
Bindings can be defined for multiple transports

Services

As shown in Figure 3-8, the service part of WSDL encloses one or more port types, similar to the way in which an object class can contain multiple objects. The following example illustrates the service binding for the Skateboots.com purchase order totaling service:

```
<service name="PurchaseOrderService">
    <port binding="tns:PurchaseOrderBinding"
        name="PurchaseOrderPort">
        <soap:address
            location="http://localhost:9000/
            xmlbus/container/PurchaseOrder/
            PurchaseOrderService/
            PurchaseOrderPort"/>
    </port>
</service>
```

Figure 3-8. A service is a collection of port types.



Services group operations in the same way that objects or classes group methods

Note the definition of the service address (in this case a locally hosted address).

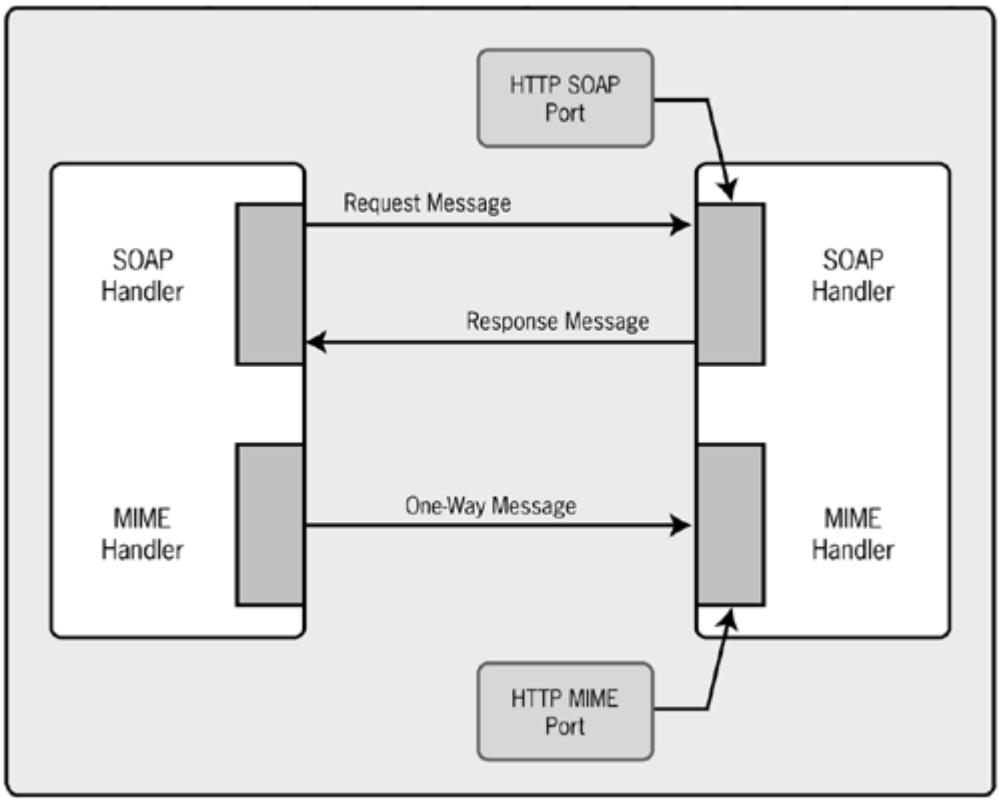
The service allows a given end point in a remote application to choose to expose multiple categories of operations for various kinds of interactions. For example, one category might contain a set of document-oriented interactions to asynchronously exchange and complete a purchase order for a future shipment. Another category might contain a set of RPC-oriented interactions to synchronously interact on an order for immediate shipment. In the former case, access to real-time inventory data is not required; but in the latter case, it is.

Putting It All Together

Once all the parts and elements of WSDL are defined, the WSDL file is complete and can be placed in a directory accessible to the Web via a URL. (WSDL files are typically generated, so don't worry too much about their individual complexity; the main point is to understand how WSDL solves particular problems.)

Figure 3-9 illustrates the two types of operations, or usage patterns, for Web services defined in the WSDL specification: request/response and one-way messages. For the request/response operation, an input message is sent to the receiving SOAP handler as a part of an HTTP request, and an output message is returned via HTTP response. For the one-way operation, an input message is sent to a MIME handler on a different port identified by the same service.

Figure 3-9. WSDL defines messages, operations, ports, and transports for SOAP and MIME.



Importing WSDL Elements

The *import* element allows WSDL elements that were separated into independent documents to be imported, as needed, to create a complete document. Different namespaces are used to qualify names in the three elements.

WSDL elements can be imported

As mentioned previously, types, abstract operations, and bindings can be developed independently and combined later to create the complete WSDL file used to describe a particular Web service instance. Thus WSDL allows different data types to be combined with different operations and different bindings. Namespaces are defined for each element, and the tricky part is to ensure that the namespaces don't overlap.

Usefulness of Multiple Transports

Multiple ports means multiple transports for the same service. This flexibility can be handy for the Internet, where some end points might understand MIME but not SOAP, and for an intranet, where some end points will not understand any of the Web protocols or even any standard protocol.

Part of the power of Web services for use inside the enterprise derives from the ability of the services to map easily to multiple protocols and transports through the separation of the binding extensions from the abstract information about a service. WSDL layering clearly separates the abstract definition of a service from its physical, or network, realization and allows for extensions to be defined for any network protocol capable of carrying XML data, or in other words, just about anything. The advantage for a business is that existing transports can be used; Web services do not assume or require that a specific software component has to be installed on each end point.

Multiple schemas may be associated with a particular namespace, and it is up to a processor of XML to determine which one to use in a particular processing context. The WSDL specification provides the processing context via the `<import>` mechanism, which is based on the XML schema's grammar for the similar concept.

WSDL-Related Namespaces

WSDL includes a namespace specifically for use within the current document. In addition to application-defined name spaces, several namespaces important to WSDL are defined in the specification:

- **http://schemas.xmlsoap.org/wsdl/**— used for the WSDL framework
- **http://schemas.xmlsoap.org/wsdl/soap**— used for the SOAP envelope definition in the WSDL bindings for SOAP
- **http://schemas.xmlsoap.org/wsdl/http**— used for HTTP `GET` and `POST` binding for WSDL
- **http://schemas.xmlsoap.org/wsdl/mime**— used for the WSDL MIME binding
- **http://schemas.xmlsoap.org/soap/encoding**— used for the SOAP v1.1 encoding scheme
- **http://schemas.xmlsoap.org/soap/envelope**— used for the SOAP v1.1 envelope
- **http://www.w3.org/2001/XMLSchema**— used for the XML schema namespace
- `tns`— the namespace prefix used by convention to refer to the current document

WSDL makes extensive use of XML namespaces

Extensions for Binding to SOAP

Because SOAP is the most popular transport for WSDL, it's important to cover its binding in further detail. The SOAP specification contains predefined rules for physically representing such data types as Booleans, integers, and arrays. Binding to SOAP therefore requires the abstract data types, messages, and operations to be bound to concrete physical representations on the wire. The WSDL `<binding>` element associates a port type with a port to define the concrete aspects of operations. For example:

```
  <binding name='InSeasonOrderBinding'
      type='wsdlns:InSeasonOrderPort' >
....
  </binding>
```

SOAP is the most important binding for WSDL

The named binding attribute is also required for the WSDL `<port>` element. Inside the `<binding>` element is a WSDL SOAP extension element called `<soap:binding>`, which specifies the physical transport protocol and the style of request: RPC or document. For example:

```
<soap:binding style='rpc'
    transport='http://schemas.xmlsoap.org/soap/
    http' />
```

This example defines the RPC style of interaction for a SOAP mapping to HTTP. For each operation within the service, the value of the `SOAPAction` HTTP header has to be included. `SOAPAction` is an HTTP header that the requesting system sends when it invokes the service. The SOAP processor on the receiving system can use the header to determine the ultimate destination of the service. For example:

```
<binding name='InSeasonOrderBinding'
    type='wsdlns:InSeasonOrderPort' >
    <soap:binding style='rpc'
        transport='http://schemas.xmlsoap.org/
        soap/http' />
    <operation name='OrderEntry' >
      <soap:operation
          soapAction='http://www.skateboots.com/
          order/OrderManagement.OrderEntry' />
              ....
    </operation>
</binding>
```

The SOAP Action HTTP header is required

The `<operation>` element contains the same name as the operation defined earlier. A `<soap:operation>` with the `soapAction` attribute is included within this `<operation>` to show that the ultimate destination of the message is the `OrderEntry` service in the `/order` folder.

Next, the encoding mechanism for the input and the output messages must be specified, as shown here.

```
<binding name='InSeasonOrderBinding' type='wsdlns:InSeasonOrderPort' >
    <soap:binding style='rpc'
        transport='http://schemas.xmlsoap.org/soap/http' />
    <operation name='OrderEntry' >
        <soap:operation
            soapAction='http://www.skateboots.com/action/
            OrderManagement.OrderEntry' />
        <input>
          <soap:body use='encoded'
              namespace='http://www.skateboots.com/typesIn/'
              encodingStyle='http://schemas.xmlsoap.org/soap/
              encoding/' />
        </input>
        <output>
          <soap:body use='encoded'
              namespace='http://www.skateboots.com/typesOut/'
              encodingStyle='http://schemas.xmlsoap.org/soap/
```

```
            encoding/' />
        </output>
    </operation>
</binding>
```

Both the <input> and the <output> elements use a <soap:body> element to specify data type encoding rules. The URI references the optional SOAP encoding style, meaning that it will be used for the serialization, or data type encoding, of the input and output messages.

Summary

The Web Services Definition Language provides a complex, full-function mechanism for defining interfaces to Web services. Interfaces can be defined as a collection of Web service operations supported at a given end point. WSDL, a specific type of XML schema, defines a language for expressing Web services interfaces in a way that commonly available XML software can understand and use. Designed for use with SOAP as the messaging transport, WSDL includes an attribute to specify whether a given interface supports the document-oriented or the RPC-oriented interaction style.

WSDL is difficult to read and to understand, but Web service toolkits typically generate and consume WSDL files automatically. Interfaces from established distributed computing technologies, such as Java classes, JavaBeans, CORBA objects, Visual Basic classes, and C# classes, translate easily into WSDL, although they might not be defined at the level of granularity appropriate for Web services.

WSDL contains a description of the data types and structures used in Web services messages, as well as information required for mapping the Web service definition onto an underlying execution environment. The three main parts of WSDL—message types, operations, and bindings—can be defined in separate documents and combined at execution time. By default, message types use XML schemas for data typing and structuring. Operations typically map to method or program names implementing the Web service. Bindings describe the protocols and transports used to send the data to the operation.

Chapter 4. Accessing Web Services: SOAP

The Simple Object Access Protocol (SOAP) is perhaps the most significant of all Web services technologies. True, Web services wouldn't exist without a way to abstractly represent the data and publish the interface definitions, but SOAP accomplishes arguably the most important aspect of Web services: getting the data from one place to another over the network.

With the Web emerging as the world's premier network and XML emerging as the world's premier data representation format, it makes sense that Web services data transport requires a combination of the two. This is exactly what SOAP provides. SOAP allows the sender and the receiver of XML documents over the Web to support a common data transfer protocol for effective networked communication.

SOAP provides data transport for Web services

In relation to the Web, SOAP is a kind of extension to the HyperText Transport Protocol (HTTP) that supports XML messaging. Instead of using HTTP to request an HTML page to be downloaded and displayed in a browser, SOAP sends an XML message via HTTP request and receives a reply, if any, via HTTP response. To handle the XML message correctly, the HTTP listener, such as Apache or Microsoft Internet Information Server (IIS), needs to provide a SOAP *processor*. In other words, an HTTP listener receiving a SOAP message must include an XML processing capability.

SOAP can extend HTTP for XML messaging

More specifically, an HTTP listener receiving a SOAP message must be able to validate and to understand the particular XML document format defined in the SOAP specification.[1] The SOAP specification allows the SOAP messaging protocol to be mapped to other transports, although the mapping to HTTP is the only mapping defined in the specification.

[1] This chapter is primarily based on the SOAP v1.1 specification; v1.2 is the W3C version, due for release in mid-2002. Whenever possible, notes are included to highlight differences between v1.1 and v1.2. For further information on the current status of this and other W3C specifications, see the W3C technical publications Web site: http://www.w3.org/TR.

Despite its name, SOAP does not include an object model. SOAP defines a one-way XML messaging protocol on top of which additional applications can be built, including the request/response interaction style familiar to object- and procedure-oriented processing, asynchronous messaging and event notification familiar to message-oriented middleware systems, unacknowledged messages, and forwarding via SOAP intermediaries.

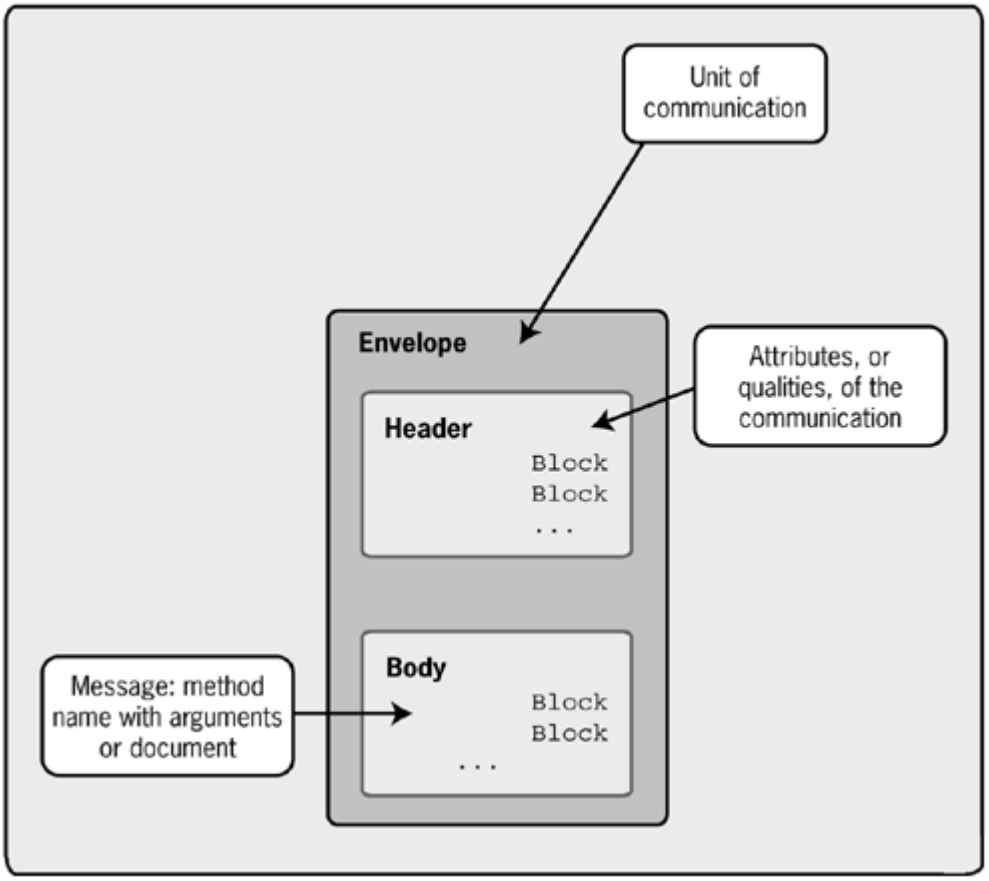
SOAP processors run on SOAP nodes

SOAP interactions are modeled as occurring between SOAP *nodes*, which can be SOAP message senders, receivers, or both. A special case of a SOAP node can play the role of an *intermediary* between sender and receiver for the purpose of handling special headers. A SOAP node supports one or more SOAP processors and is responsible for handling the SOAP *blocks* when a message is received.

A SOAP intermediary is a special case of a SOAP node

Figure 4-1 illustrates the three major blocks, or parts of SOAP messages: the envelope, the header, and the body. The envelope is required and marks the start and the end of the SOAP message. The header is optional, and can contain one or more header blocks carrying the attributes of the message or defining the qualities of service for the message. Headers are intended to carry contexts or any application-defined information associated with the message, such as security tokens, transaction identifiers, and message correlation mechanisms. The body is required and contains one or more body blocks comprising the message itself.

Figure 4-1. SOAP consists of three parts.



SOAP has three major parts: envelope, header, and body

SOAP transports an XML document across the Web and, potentially, across other types of networks, to reach a Web service implementation. A SOAP message is, in fact, an XML document created in a specific format. A conforming SOAP implementation understands how to interpret such a document and how to map the data to an underlying software implementation of the service. SOAP does not define the service itself, however, but rather just enough about the message so that a SOAP processor can recognize. SOAP defines what it means to recognize the message as a SOAP message, but the Web service implementation has to know how to interpret the data contained in the message as a target for the underlying software implementation of the service.

SOAP bridges Web service implementations

Using SOAP and WSDL, an enterprise can choose to expose business data using services, or interfaces, that exchange complete documents or that map the data onto objects using method names and input and output arguments. Complete documents also can be exchanged using SOAP with Attachments (see SOAP Multipart MIME Attachments later in this chapter), as ebXML and RosettaNet have defined. Remote procedure call (RPC) mappings must be done within the body of the SOAP message, using complex data types, the SOAP encoding, and the RPC convention

defined in the SOAP specification (see RPC Convention later in this chapter). One of the trickier aspects of understanding SOAP is that it is defined at a sufficient level of abstraction to encompass both document- and RPC-oriented interactions.

SOAP can exchange complete documents or call a remote procedure

A Simple Example

The following example shows a simple application of SOAP, a one-way broadcast message to a list of comunication mechanisms. The header block contains the list of devices to which the message is to be sent. The body block contains the actual notification message to be delivered: a reminder to join a conference call in progress.

SOAP can be used for broadcasting a message

```
<env:Envelope xmlns:env="http://www.w3.org/2001/12/
     soap-envelope">
 <env:Header>
  <n:broadcastService xmlns:n="http://
       www.xmlbus.com/broadcastServices">
   <n:list>PDA, Cell, Email, VoiceMail, IM</n:list>
  </n:broadcastService>
 </env:Header>
 <env:Body>
  <m:Function
   xmlns:m="http://www.xmlbus.com/broadcastServices/
        send">
   <m:message>
    Eric, you are late for the concall again!
   </m:message>
  </m:Function>
 </env:Body>
</env:Envelope>
```

The example illustrates a simple SOAP message containing the envelope, an optional header block that defines attributes of the message, and a body block that contains the message itself. In this example, the meeting reminder message will be broadcast to the device list by the `send` service located at the address defined by the URL: *www.xmlbus.com/broadcastServices*. Separate namespaces (`broadcastService` and `Function`) are used to qualify the element and attributes for each part of the message. The envelope references the most recent v1.2 envelope namespace, www.w3.org/2001/12/soap-envelope.

No encoding is specified, so no encoding namespace appears in the example, which means that the default XML types are used (that is, text). The header and the body both use application-defined namespaces to qualify the respective elements and attributes.

This simple one-way message can easily be bound to HTTP using HTTP `POST`, as in the following example:

```
Request header:
POST /broadcastService HTTP/1.1
Host: www.xmlbus.com
```

```
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
<?xml version='1.0' ?>
...SOAP request document

Response header (if any):
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
<?xml version='1.0' ?>
...SOAP response document
```

Outgoing messages typically are bound to HTTP `POST`

As shown in the example, the proscribed way to use SOAP with HTTP is to send a request document (such as in the previous example) using HTTP `POST`. The HTTP header indicates that the document is being sent to the `broadcastService` service at the *http://www.xmlbus.com/broadcastService* address. When the software program implementing the `broadcastService` receives the message, it will decode the body block and execute the `send` request. If a response were defined for this message (such as a confirmation that the message was sent), the response would be contained within the HTTP response—`HTTP/1.1 200 OK`.

Incoming messages typically are bound to HTTP response

The `200` code is an HTTP code indicating success. If some kind of failure occurred in processing the request, the HTTP code on the response would be a failure code, such as `HTTP/1.1 500 Internal Server Error`. In case of a failure to process a SOAP request, the body of the response includes a SOAP *fault* message detailing the problem (see SOAP Faults later in this chapter).

The SOAP Specification

The SOAP specification was developed and published on the Web in late 1999. SOAP is an extension of the XML-RPC specification, originally defined by Dave Winer of UserLand. Implementations of XML-RPC predate SOAP and continue to be used, but SOAP has gained a wider acceptance because of its additional features and functionality.

SOAP was developed by UserLand, DevelopMentor, and Microsoft

IONA and then IBM joined the effort in mid-2000 and helped produce the v1.1 specification, which was submitted to the W3C by 11 vendors. The v1.1 specification became the foundation of the XML Protocols Working Group at W3C, which is moving the v1.2 specification toward *recommendation* status.[2] (A recommendation status is the highest offered by W3C and

recommends adoption of a specification as a standard. W3C does not call its specifications standards, however, because it does not have the official standing of a standards organization, such as the International Organization for Standardization.)

[2] To date, the December 2001 SOAP v1.2 specification draft is scheduled to progress to recommendation, or final status, in May 2002. However, it is not unusual for these target dates to slip a bit while last-minute controversies are resolved.

Submitting SOAP to the W3C Started the Web Services Movement

SOAP can be considered the start of the Web services movement. Submitting the SOAP v1.1 specification to W3C and starting the XML Protocols Working Group were watershed events. Since then, the software industry has entered a consolidation phase and has begun to broadly adopt SOAP, WSDL, and UDDI.

More than 80 independent implementations of the SOAP v1.1 specification have been developed, testifying to its simplicity, popularity, and the appropriateness of the solution it offers to the problem of transporting data across the Web. IBM provides an open source Java version as part of its AlphaWorks program, and implementations are also available in .NET, C, C++, Visual Basic, Perl, and Python, among others.

A wide variety of SOAP implementations exists

With all the varied implementations and interpretations of the SOAP specification, one of the big areas to be worked on is getting the various implementations to interoperate. Resolving issues uncovered in vendor interoperability testing will be a major part of the ongoing work on SOAP at W3C to ensure that SOAP reaches the interoperability levels achieved using HTTP and HTML.

The authors of the original SOAP specification intended to keep the specification small and simple to ensure that a basic level of functionality would be supported in all SOAP implementations and that higher-level functionality could be built on a common basic protocol. As the specification matures, it will likely define more and more of these things while preserving compatibility at the most basic level—as HTTP and HTML do today, for example. But, implementations of SOAP are not necessarily compatible in how they address these higher-level issues, such as how to ensure that a request has a corresponding reply or what it means to coordinate business transactions across two or more Web services. Addressing these higher-level issues represents another major step for W3C.

The original intent was to keep SOAP simple

To provide a basic, common XML messaging protocol for the Web, the SOAP specification defines

- The start and the end of an *envelope* that encloses the XML document to be transported
- How to represent any optional *headers* for additional information, including pointers to additional schemas associated with the document, such as security, transaction coordination, and so on

- How to serialize data type encodings optionally over the wire, with specific support for RPC-style interactions
- A binding to HTTP to ensure that it carries the document correctly to its destination and that the destination directs the message to an appropriate SOAP processor

SOAP defines the elements and the rules of XML messaging

The main syntax parts are defined as independent *blocks* so that one or all may be implemented and processed by separate *handlers* associated with each. The envelope elements and the encoding rules are defined using different namespaces to ensure that element and attribute names within them don't clash. The SOAP specification defines a layered approach to the protocol so that it can be used in combination with other transports, such as the OMG Internet Inter-Orb Protocol (IIOP), Java Messaging System (JMS), and the IETF Blocks Environment Extensible Protocol (BEEP). However, only the HTTP binding is defined in the current document.

SOAP syntax blocks are associated with handlers for processing

SOAP Envelope

The envelope, the outermost element in a SOAP message, comprises the XML document root element. The SOAP envelope is specified using the `ENV` namespace prefix and the `Envelope` element. The envelope changes when SOAP versions change. A v1.1-compliant SOAP processor will generate a fault when receiving a message containing the v1.2 envelope namespace. A v1.2-compliant SOAP processor generates a `VersionMismatch` fault if it receives a message that does not include the v1.2 envelope namespace. The following example illustrates the v1.2 draft envelope namespace and the original v1.1 envelope namespace:

```
V1.2:   <SOAP-ENV:Envelope
            xmlns:SOAP-ENV="http://www.w3.org/2001/
            12/soap-envelope"
V1.1:   SOAP-ENV="http://schemas.xmlsoap.org/soap/
            envelope/"
```

The envelope is the top-level XML element in a SOAP message

Looseness of the Specification

One problem with the SOAP specification is that it contains a lot of rules that may or may not be enforced. Thus it is very likely that two conforming SOAP implementations will not implement the same collection of optional features and thus be incompatible. There's always a balance between required rules in a specification and broad acceptance.

A lot of the discussion about SOAP at the W3C concerns where to draw the line. Some members push for requiring more features in the base protocol, whereas other members believe that it's better to keep the base protocol as simple as possible. Loose rules help encourage implementation because it's easier to produce a minimally conformant

version. Similarly, specification rules that are too tight might discourage implementation. Striking the right balance is critical.

The Web Services Interoperability Organization was founded for precisely this reason—to help ensure interoperability by agreeing on a common interpretation of the SOAP, WSDL, and UDDI specifications.

The optional SOAP encoding is also specified using a namespace name and the optional `encodingStyle` element, which could also point to an encoding style other than the SOAP one. The following example illustrates the v1.2 draft encoding namespace and the original v1.1 encoding namespace:

```
V1.2:   SOAP-ENC:encodingStyle="http://www.w3.org/
            2001/12/soap-encoding"
V1.1:   SOAP-ENC="http://schemas.xmlsoap.org/soap/
            encoding/"
```

Namespaces identify the envelope version and specify encoding style

The encoding namespace references simple types and enumerations from XML schema but uniquely defines complex structures, such as arrays and structs. The encoding namespace, like namespaces in general, can be referenced at any level of the document, including the envelope, or top-level element. The encoding namespace can be set to null to turn off, or disallow, encoding for a specific part of the message.

The SOAP envelope indicates the start and the end of the message so that the receiver knows when an entire message has been received. The SOAP envelope solves the problem of knowing when you're done receiving a message and are ready to process it. The SOAP envelope is therefore basically a packaging mechanism.

Both the publisher and the consumer must agree on the encoding rules—typically by downloading and using the same XML schema that defines them. The envelope and the encoding rules are defined using different XML namespaces, however, which allows multiple encoding rules to be applied to the same message. The following brief example on SOAP envelope syntax is based on the SOAP v1.2 specification draft.[3]

[3] Note the absence of the `SOAPAction` field in the HTTP header. This SOAP v1.1 feature was made optional in the v1.2 draft specification.

The envelope can specify encoding rules for the entire message

```
POST /OrderEntry HTTP/1.1
Host: www.xmlbus.com
Content-Type: application/soap; charset="utf-8"
Content-Length: nnnn
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://www.w3.org/2001/12/
  soap-envelope SOAP-ENV:encodingStyle="http://
  www.w3.org/2001/12/soap-encoding /">
```

```
  . . .
</SOAP-ENV:Envelope>
```

This example illustrates the use of a SOAP message within an HTTP `POST` operation, which sends the message to the server. It shows the namespaces for the envelope schema definition and for the schema definition of the encoding rules. The `OrderEntry` reference in the HTTP header is the name of the program to be invoked at the xmlbus.com Web site.

The HTTP binding specifies the location of the service

SOAP Header

The header, an optional part of a SOAP message, can occur multiple times, if it occurs at all. The semantics, or meaning, of each header is typically defined within an associated XML schema. The name of the top-level element in the header block can be used to map the block to the semantics defined in the associated schema. The header element is a child element of the envelope.

Headers are optional and can be multiple

Headers are the main mechanism by which SOAP can be extended to include additional features and functionality, such as security, transactions, and other quality-of-service attributes associated with the message. The header is encoded as the first immediate child element of the SOAP envelope. When more than one header is defined, all immediate child elements of the SOAP header are interpreted as SOAP header blocks.

Headers are intended to add new features and functionality

The SOAP header contains header entries defined in a namespace. The information in the headers is used along the message path somewhere and is not always intended for use at the ultimate SOAP processor destination, although, of course, it may be. The headers may have attributes, such as an encoding-style URI or an actor URI, for intermediaries that process things, such as user name/password, ID for reliable messaging, or a digital signature for security checking.

Header entries are defined in a namespace

Standard Headers Needed

The header mechanism in SOAP, imprecisely conceived and defined, was intended for use by specific applications built atop the base protocol, providing a place in which to define SOAP extensions. Of course, for such extensions to be generally meaningful, they need to be standardized. This is part of the work that needs to be done to complete Web services: to define standard SOAP headers for security, transactions, process flow, routing, message correlation, guaranteed message delivery, and so on. However, until

one or more other W3C working groups start to define standards for them, SOAP headers are likely to remain confusing and ill-defined. There is no compelling reason yet to use them. Application-specific information can just as easily be carried in the SOAP body.

Headers include an attribute called `mustUnderstand`, which allows senders and receivers to negotiate agreement on support of a given header or set of headers. This attribute is the same as the mandatory attribute in the HTTP Extension Framework, to which the SOAP specification also provides a mapping. Thus the `mustUnderstand` attribute, like many other aspects of SOAP, is a carryover of a concept defined originally in HTTP. The intent of this concept is to allow applications built using the older versions of the specification, or the applications not up to the same revision level as new applications, to fail gracefully when they receive a message that they do not understand. For example:

```
<SOAP-ENV:Header>
   <t:Transaction
      xmlns:t="www.xmlbus.com"
      SOAP-ENV:mustUnderstand="1">
          5
   </t:Transaction>
</SOAP-ENV:Header>
```

A sender can require the receiver to understand a header

The example shows a possible header for transaction support, which is not yet defined, that uses the `mustUnderstand` attribute to require the receiver to support transactions if the sender wants to use them. If the receiver of the SOAP message doesn't support transactions, the receipt of such a message will generate a SOAP fault.

Headers can require SOAP processors to understand them or to reject the message

When a header block is tagged with a SOAP `mustUnderstand` attribute with a value of 1 or `True`, the targeted SOAP node must process the SOAP block according to the requirements of the header or not process the SOAP message at all and return an error code.

Headers are for adding features to a SOAP message in a decentralized manner without prior agreement between the communicating parties. But SOAP does not define any headers, and it's not clear which headers will be used, although SOAP does define some attributes for use with the headers in anticipation of such use.

Headers allow decentralized addition of features to SOAP

In designing the headers, the specification authors anticipated that a wide variety of header functions would be developed over time. Each SOAP node may include the software necessary to implement one or more such extensions. A SOAP header block is understood by a receiving

SOAP node if the software at that SOAP node has been written to fully implement the semantics conveyed by the fully qualified name of the outermost element of that block, that is, by the semantics defined in the associated schema for that element name.

SOAP Body

The SOAP body contains the application-defined XML data being exchanged in the SOAP message. The body must be contained within the envelope and must follow any headers that might be defined for the message. The body is defined as a child element of the envelope, and the semantics for the body are defined in the associated SOAP schema.

The body contains information that must be sent to the ultimate recipient

The body contains mandatory information intended for the ultimate receiver of the message. For example:

- **Request**
- `<SOAP-ENV:Envelope`
- `. . .`
- `<SOAP-ENV:Body>`
- `<m:GetOrderStatus`
- `xmlns:m="www.xmlbus.com/OrderEntry">`
- `<orderno>12345</orderno>`
- `</m:GetOrderStatus>`
- `</SOAP-ENV:Body>`
- `</SOAP-ENV:Envelope>`
- **Response**
- `<SOAP-ENV:Envelope`
- `. . .`
- `<SOAP-ENV:Body>`
- `<m:GetOrderStatusResponse`
- `xmlns:m="www.xmlbus.com/OrderEntry">`
- `<status>shipped June 18</status>`
- `</m:GetOrderStatusResponse>`
- `</SOAP-ENV:Body>`
- `</SOAP-ENV:Envelope>`

The preceding body portion of two typical RPC-oriented SOAP messages shows a request and a response for an order status service. The request sends the order number, and the response message returns the most recent status. A `GetOrderStatus` request is sent to the `OrderEntry` service. The request takes an integer—the order number—and returns a string in the SOAP response. The `www.xmlbus.com/OrderEntry` XML namespace is used to qualify application-specific element names.

Normally, the application also defines a schema to contain semantics associated with the request and response elements. The `OrderEntry` service might be implemented using an EJB running in an application server; if so, the SOAP processor would be responsible for mapping the body information as parameters into and out of the EJB implementation of the `OrderStatus`

service. The SOAP processor could also be mapping the body information to a .NET object, a CORBA object, a COBOL program, and so on.

The SOAP processor maps the request and response information to an EJB or other program

SOAP Faults

When an error occurs during processing, the response to a SOAP message is a SOAP fault element in the body of the message, and the fault is returned to the sender of the SOAP message. For the HTTP binding, a successful response is linked to the 200 to 299 range of status codes; the SOAP fault is linked to the 500 to 599 range of status codes. The SOAP fault mechanism returns specific information about the error, including a predefined code, a description, the address of the SOAP processor that generated the fault, and application-generated details about the error, if it occurred while processing the body block.

A fault is returned when a SOAP message can't be processed

A SOAP response message can carry a single fault block in the body of the message. Contained within the SOAP fault body block are the following subelements:

- <`faultcode`>: a software-generated XML qualified fault name—qualified as either `Sender` or `Receiver`, depending on whether the message was found to be incorrect on receipt or an error occurred during processing of the message
- <`faultstring`>: associated with the fault code, an explanatory text that describes the problem, similar to the Reason-Phrase defined in HTTP
- <`faultactor`>: a URI identifying the address of the SOAP processor that generated the fault
- <`detail`>: application-specific information about the fault, required when the message could not be successfully processed

A message can carry only one fault block

Error information generated when processing header blocks must be carried in the header blocks themselves on a response message. If the <`detail`> element is not filled in, the fault was not related to processing the contents of the body block. This signifies whether the body was processed at all. Fault codes are defined in the SOAP envelope schema. For example:

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<env:Envelope xmlns:env="http://schemas.xmlsoap.org/
     soap/envelope">
```

```
  <env:Header>
    <V:Upgrade xmlns:V="http://www.w3.org/2001/12/
        soap-upgrade">
      <envelope qname="ns1:Envelope"
       xmlns:ns1="http://www.w3.org/2001/12/
           soap-envelope"
      </envelope>
    </V:Upgrade>
  </env:Header>
  <env:Body>
    <env:Fault>
      <faultcode>env:VersionMismatch</faultcode>
       <faultstring>Version mismatch</faultstring>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

The presence of the detail element indicates that an error occurred during body processing

As shown in the example, a SOAP processor generates a fault response when unable to understand a previous version of the specification. For the HTTP mapping, the fault corresponds to the HTTP 500 error, and the body block of the response contains the <fault>, <faultcode>, and <faultstring> elements.

Because the fault was generated by the ultimate recipient of the message, the <faultactor> element is optional and is not included. A fault generated by an intermediary is required to contain a <faultactor> element, however. And because the fault was generated as a result of a version mismatch, not while processing a a body block, the <detail> element is not included. The <detail> element is included only when the fault occurs during the processing of a body block.

Note the v1.1 namespace used for the envelope, which indicates that the fault was generated by a v1.1-compliant SOAP processor that received a v1.2 SOAP message it couldn't understand.

For another example:

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
<env:Envelope xmlns:env="http://www.w3.org/2001/12/
    soap-envelope" >
  <env:Body>
    <env:Fault>
      <faultcode>env:Server</faultcode>
      <faultstring>Server Error</faultstring>
      <detail>
        <e:myfaultdetails xmlns:e="http://
            www.xmlbus.com/faults" >
          <message>
```

```
            The database was unavailable
          </message>
          <errorcode>1001</errorcode>
        </e:myfaultdetails>
      </detail>
    </env:Fault>
 </env:Body>
</env:Envelope>
```

The example illustrates a SOAP fault block generated as the result of a failure during the processing of a body block. Again, the HTTP 500 error is returned along with the fault response. This time, however, the `<detail>` element is provided to give additional information about the error that caused the fault block to be returned, in this case to signal that processing failed because the database was unavailable.

RPC Convention

The SOAP specification includes an optional definition of how to encapsulate remote procedure call (RPC) functionality using XML. For the RPC-oriented interaction pattern, this is the heart of SOAP.

Requirements for the RPC mapping include providing a URI for the target SOAP node, a procedure or method name, an optional procedure or method signature, and the parameters to the procedure or the method. Optional header data can be included to pass security information, transaction context, or other attribute information that may be associated with the procedure call.

A major feature is mapping SOAP messages to remote procedure calls

SOAP relies on the underlying transport mechanism, such as HTTP, to carry the URI address of the message destination. Life-cycle management for the objects accessed via Web services are outside the SOAP spec, but it is possible to send cookies via the SOAP headers to implement such additional features at the application level.

An RPC invocation is modeled as a single `struct` containing an `accessor` for each in or in/out parameter. The `struct` is named and typed identically to the target procedure or method name. Each in or in/out parameter is viewed as an `accessor`, with a name corresponding to the type of the parameter, in the same order as in the target procedure or method. (See Data Type Mapping later in this chapter for further information on SOAP data types.)

RPC invocations are modeled as `structs`

The request `struct` name is identical to the method name so that the mapping occurs correctly. The response `struct` name is defined using a naming convention that makes it easier for the SOAP processor to map the response from a program. For the RPC mapping, the use of headers roughly corresponds to the use of attributes in the runtime environment for the target procedure or method, such as what may be defined for an EJB container or .NET package attributes.

A response message indicates that the request message was successfully processed. A fault message indicates that a failure occurred either before or during message processing. A SOAP processor cannot return both a response message and a fault message.

The result can be a fault message or a response message, but not both

The following example illustrates a request/response SOAP message definition. The example contains a SOAP message in the form of a request to a purchase order service that totals the purchase order and returns the total value of the order. The customer can then decide whether to confirm the order.

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
     xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsd1="http://www.xmlbus.com/PurchaseOrderService-
xsd"
     xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <ns1:postPurchaseOrder
     xmlns:ns1="http://www.xmlbus.com/PurchaseOrderService">
      <order xsi:type="xsd1:PurchaseOrder">
        <CompanyName
xsi:type="xsd:string">IONA</CompanyName>
        <Items xsi:type="SOAP-ENC:Array"
            SOAP-ENC:arrayType="xsd1:Item[1]">
          <item xsi:type="xsd1:Item">
           <Price xsi:type="xsd:float">129.99</Price>
           <PartID xsi:type="xsd:string">1234</PartID>
           <Description xsi:type="xsd:string">
            SkateBoots
           </Description>
           <Quantity xsi:type="xsd:int">40</Quantity>
          </item>
        </Items>
        <Address xsi:type="xsd1:Address">
         <State xsi:type="xsd:string">MA</State>
         <PostalCode
xsi:type="xsd:string">02451</PostalCode>
         <City xsi:type="xsd:string">Waltham</City>
         <Line2 xsi:type="xsd:string"></Line2>
         <Country xsi:type="xsd:string">USA</Country>
         <Line1        xsi:type="xsd:string">200        West
Street</Line1>
        </Address>
      </order>
```

The example includes a reference to the associated XML schema for the application-defined parts of the message (`PurchaseOrderService-xsd`) and the `post` request (`postPurchaseOrder`), which also uses an XML namespace to qualify the element names in the request. The two major categories of data types are shown, including a simple, schema-standard string type for the `CompanyName`, and a custom- defined SOAP array type for the `items`. Note that the custom-defined array is included within the `PurchaseOrderService-xsd` schema.

Importance of Data Typing

The RPC mapping is an example why data typing is so important, especially for structs. Structs and complicated data types are used for RPC method name and argument mapping. Without them, it would be much more difficult to define the specific way in which data in the SOAP body could be mapped into and out of procedures or objects using remote calls. As it is, SOAP implementations generally decode the information in the `structs` and create remote procedure call signatures for Enterprise JavaBeans, .NET objects, CORBA objects, COBOL programs, and so on.

Without the definition of arrays, accessors, and `structs` in SOAP, each implementation would have to design and build this type of mapping individually, in a proprietary fashion. So although the array and `struct` definitions in the SOAP data types may appear complex and unnecessary, they are helpful in the key area of mapping SOAP messages to remote procedure and object invocations.

The following example illustrates the Java programming language class definition for the implementation behind the purchase order example.

A Java or C# class implements the executable service

```
public class PurchaseOrderService {
    public float postPurchaseOrder(PurchaseOrder order) {
        Item items[] = order.getItems();
        float total = 0.0f;
        for (int x = 0; x < items.length; x++) {
            total                                          +=
items[x].getPrice()*items[x].getQuantity();
        }
        return total;
    }
```

This very simple example, which could just as easily have been a C# or VB.NET class shows executable code providing a Web service implementation to which a SOAP message can be sent.

Other programs are, of course, required to accept and to generate the SOAP messages themselves. The associated XML schema contains semantic information, or a kind of "programming instruction" that the programs can key off of to understand how to process the service.

As shown in the examples, the SOAP RPC mapping convention references a method name in the HTTP `POST` request. Because SOAP defines a one-way messaging model, a request doesn't require SOAP response; the response can be empty. The response can also be a fault (see SOAP Faults later in this chapter).

Schemas contain semantic information to instruct the underlying service implementation

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsd1="http://www.xmlbus.com/PurchaseOrderService-xsd"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Body>
<ns1:postPurchaseOrderResponse
 xmlns:ns1="http://www.xmlbus.com/PurchaseOrderService">
<return xsi:type="xsd:float">5199.6</return>
</ns1:postPurchaseOrderResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This example illustrates the response to the SOAP request shown in the previous example.[4] The same schema is used (`PurchaseOrderService-xsd`), and the method name, `postPurchaseOrderResponse`, uses the SOAP convention of naming the result in correspondence to the request. The return uses a simple standard float data type.

[4] Note that both examples use the v1.1 namespaces.

Is the RPC Convention Part of the Protocol or Not?

The RPC representation, or convention, for SOAP is the subject of some contention in the SOAP community and is optional in the current specification draft. The protocol does not guarantee that a response message will be related to a request message; it has to be done programmatically by the application, possibly using correlation IDs passed in header blocks. SOAP is a one-way messaging system, although request/response patterns can be achieved, and WSDL supports them as well.

Some specific additional bindings—for example, to IIOP in a LAN environment—to protocols beyond those in the specification can be used to make this more explicit or required. Some qualities of service, or attributes of the message, are intended to be inherited from the underlying transport. Part of the SOAP community anticipates a solution through mapping SOAP to BEEP, but the time and effort required to launch a new Internet protocol leave this potential solution in some doubt.

Others in the SOAP community propose adding spec definitions that require the response correlation to be part of SOAP itself: for example, by defining a required correlation ID header, which may be a more practical resolution. The natural mapping of HTTP `POST` to the request message and HTTP reply to the response message ensures at least that the response will be returned using the same HTTP connection as the request but is not guaranteed if the HTTP connection is lost.

Data Type Mapping

The SOAP encoding style is intended to be generalizable across any programming language or database management system. However, if the SOAP encoding style does not fit a particular software system's requirements, another encoding system can be used. SOAP is completely flexible and extensible in this regard, but both sender and receiver have to use the same encoding scheme.

SOAP data types can be simple, such as integer, float, text, or date, as defined in XML schema language. Or, they can be complex, such as structures and arrays, as defined in the SOAP encoding schema or alternative.

Simple SOAP data types are the same as schema data types

Data types are defined using a schema, which is used to generate an XML grammar for the data types. Given the type system schema and a graph of values conforming to that schema—that is, a buffer area containing the data to be mapped—an XML instance can be generated. In reverse, given an XML instance produced according to the rules and given access to the original schema, the original data types can be reconstructed.

XML instances are generated by mapping input data to schemas

Given an existing data type definition, Web services are defined so that the types can be mapped to an XML schema. If output from an existing system into a file, for example, or a memory area, the data can be used as a source of input for generating an XML instance. First, the existing schema or data type definition is mapped, or transformed, into a corresponding XML schema. Next, the XML schema is used with the data output to a file or memory to generate an XML document containing those values. Finally, the XML document is transmitted from sender to receiver, using SOAP. The encoding namespace has to be used to qualify element and attribute names if the SOAP encoding is used.

Compound values defined in the SOAP encoding have accessors that point to a specific value within the structure or the array. Arrays can be nested or can contain structures; nested arrays can be of types different from the enclosing arrays. Arrays and structs are used to contain the information required to describe an RPC-style interaction—in particular, how the arrays and structures are accessed.

Although it is possible to use the `xsi:type` attribute such that a graph of values is self-describing both in its structure and the types of its values, the serialization rules permit that the types of values may be determinate only by reference to a schema. In other words, it's possible, but not required, to restrict type definition to a schema.

HTTP Binding

The SOAP specification allows for the possibility of multiple transport bindings, although binding to HTTP is the only one currently defined. Other mappings have been successfully defined for SOAP to SMTP, BEEP, JMS, IIOP, and others, however.

The HTTP binding ensures that SOAP messages are consistent with HTTP's message model and indicate to HTTP servers that a SOAP message has arrived. HTTP servers can therefore act on a

SOAP message by knowing that it's SOAP, rather than simply an XML document carried the way an HTML document would be carried.

The HTTP binding ensures that SOAP messages are carried correctly over HTTP

The binding works for HTTP `POST` requests. A SOAP intermediary is not the same as an HTTP intermediary. Only an HTTP-origin server can be a SOAP intermediary, a fully functional SOAP processor capable of receiving and sending SOAP messages. A SOAP Action header field, if any, cannot be computed; the sender must know it in advance. The SOAP Action indicates the intent, not the definition, of the message.

The SOAP Action is like a processing instruction to the SOAP processor, triggering a certain predefined action associated with the message, such as routing it to a JMS queue or forwarding it to another SOAP processor behind the firewall. In the v1.2 specification draft, information about the target of the SOAP message is contained within the HTTP header itself, and the SOAP Action header field is not needed.

Version Control

SOAP, like any other protocol, evolves over time. SOAP v1.1, for example, is the version originally submitted to the W3C, whereas SOAP v1.2 is the current working draft version published by the W3C after the XML Protocols Working Group made some revisions to the specification.

More than 80 implementations of the v1.1 specification exist or have been released on the market. Many of these implementations will adopt the v1.2 specification, but presumably not all at once or in a coordinated fashion. Furthermore, applications may have been deployed already based on v1.1 implementations, whereas other, newer applications may be deployed based on v1.2 implementations. What happens when a v1.1 implementation sends a message to a v1.2 implementation and vice versa?

SOAP versioning is handled via namespace revision

This is, of course, a very common problem for standards and technologies that evolve. Usually, standards and technologies strive to be upwardly compatible, meaning that a newer version can interoperate with an older version. That an older version interoperates with a newer version is less often possible. How can the older version know what changes have occurred in the newer version?

To handle this problem, SOAP defines new namespaces.

- The v1.1 SOAP envelope namespace is http://schemas.xmlsoap.org/soap/envelope.
- The v1.2 draft SOAP envelope namespace is http://www.w3.org/2001/12/soap-envelope.

The v1.2 namespaces are changed as new specification drafts are published. For example, the "2001/12" indicates the December 2001 working draft.

When the envelope schema changes for a new version of SOAP, both the sender and the receiver will have to understand it in order to take advantage of new features in the revision. As SOAP

evolves and changes, the SOAP envelope schema has to change to add the new features, corrections, enhancements, and so on and must do so in an upwardly compatible way.

Schema changes identify version changes

A SOAP processor is therefore responsible for checking the envelope namespace for compatibility with the current version. If it isn't compatible, the SOAP processor has to treat this as a version mismatch error and generate a `VersionMismatch` SOAP fault. A v1.2 SOAP processor, on the other hand, may optionally choose to process an upwardly compatible SOAP v1.1 message or otherwise to generate a `VersionMismatch` fault. A SOAP v1.2 processor can list the envelope versions it supports, using the SOAP upgrade http://www.w3.org/2001/12/soap-upgrade namespace identifier.

SOAP Message Processing

In general, SOAP messages like the ones in the examples must be viewed together with their associated XML schemas, which tell the consumers of SOAP messages how to understand them. The most popular format for such XML schemas is WSDL (see Chapter 3). The SOAP namespaces for envelope and encoding have schemas associated with them, too.

Schemas define how to interpret messages

The following example, taken from the SOAP envelope schema, defines the major SOAP parts:

```
<xs:schema       xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://www.w3.org/2001/12/
soap-envelope"targetNamespace="http://www.w3.org/
2001/12/soap-envelope">

  <xs:element name="Envelope" type="tns:Envelope" />
- <xs:complexType name="Envelope">
- <xs:sequence>
  <xs:element ref="tns:Header" minOccurs="0" />
  <xs:element ref="tns:Body" minOccurs="1" />
  </xs:sequence>
  <xs:anyAttribute namespace="##other"
      processContents="lax" />
  </xs:complexType>
```

At the top of this example is the SOAP envelope schema, found at the SOAP envelope namespace URI: http://www.w3.org/2001/12/soap-envelope. The namespace is used in the schema to qualify SOAP envelope schema names. The top-level element is the `Envelope` element, which contains the `Header` and the `Body` elements.

The `Header` element is optional and therefore has a `minOccurs` of `0`; the `Body` is required and has a `minOccurs` of `1`. SOAP processors are required to obtain and to use this and other schemas defined for SOAP when handling messages.

The sender and the receiver of SOAP messages must use the same serialization format, in order to correctly parse and interpret the message. The sender and receiver need the schema for the SOAP envelope itself; a schema for the optional SOAP encoding mechanism, if any; and schemas for any optional headers used. And, of course, there are schemas for application-specific data elements in the body. All application-specific schemas associated with SOAP messages have to be self-contained; that is, SOAP processors aren't allowed to import application-defined attributes, values, or processing instructions from the application-defined schemas.

Sender and receiver need to have the same schemas

The SOAP encoding-style schema is available at the encoding-style namespace URI: http://www.w3.org/2001/12/soap-encoding. The encoding schema contains definitions for all simple and complex data types that can be used with the optional SOAP encoding, including a base64 type for serializing binary data.

The encoding style is defined in a schema

The schemas for validating the various parts of the SOAP message are normally posted on the Internet by the SOAP receivers and have to be downloaded by any party in an exchange of messages to ensure that they are generated and understood correctly. The schemas may also be discovered in a UDDI or an ebXML registry (see Chapters 5 and 6) or by using the Web services inspection language (WS-Inspection). For more on this, see Chapter 7.

Application-defined schemas validate application data

Optional header schemas can be defined to include information about the RPC-style interaction, if needed, or to carry other general information about the message, such as priority and expiration time, security information, mail-to addresses for the message, if any, or payment codes for purchase orders. Headers are intended to provide an extensibility mechanism for adding features and capabilities to SOAP. In general, both sender and receiver should support the same headers, using the same schemas or compatible schemas defined for the headers. Some SOAP implementations, however, may support a negotiation phase during which they can agree not to support the same headers.

Optional header schemas can be defined

The header and body parts of SOAP work together to ensure that what's intended to be communicated between the parties is communicated, with the appropriate qualities of service, and that SOAP messages are mapped correctly to the underlying service implementation.

Header and body are related

The SOAP Action attribute (optional in the v1.2 draft) is part of the HTTP binding and provides additional information interpreted by the receiver of the message—for example, to post the message to a message queue. In addition, an attribute can be defined to identify an intermediate destination for header information.

In the v1.2 draft, actor attributes have been expanded to indicate which SOAP node is responsible for processing which header. This is to clarify which intermediaries, if any, process which headers.

Actor attributes identify intermediaries, if any

Header Subtleties

Normally, the sender of a SOAP message will obtain all the receiver's schemas in advance. In other words, the typical Web service model involves downloading the WSDL file from the receiver and generating a SOAP message conforming to the receiver's schema(s). So it would seem that the sender would have to know in advance whether the receiver supported certain headers, making the `mustUnderstand` attribute irrelevant. And it would also seem that without explicitly defined support in WSDL for headers, actors, and intermediaries, these mechanisms won't be used.

Finally, the idea of supporting intermediaries independent of SOAP sender/receiver pairs likely will be realized only after SOAP itself is widely adopted and used. All the functionality in Web services fundamentally seems to require negotiation and agreement between sender and receiver in advance rather than at connection or discovery time, throwing doubt onto the usefulness of the SOAP header model for intermediary processing.

As noted previously, intermediaries along the message path can operate on message headers.[5] For example SOAP headers can be extracted or supplemented at an intermediary, and parts of the SOAP header can be used to indicate which intermediary handler is to be used to process which header.

[5] The SOAP specification does not, however, include any routing protocol. IBM and Microsoft have jointly published WS-Routing, a proposal for that purpose, however. See Chapter 7 for a summary of WS-Routing.

Messages can be routed through intermediaries that process headers

A SOAP *node* is the entity that processes a SOAP message according to the SOAP specification rules to access the services provided by the underlying protocols through SOAP bindings. The SOAP specification defines a correlation between the parts, or blocks, of a SOAP message and software handlers are designed to handle, or process, each part of the message in the appropriate way.

Message processing involves mapping to the underlying services

Some implementations may develop specific handlers for each part of the SOAP message and work from clues in the major SOAP elements to determine which parts of the message are appropriate for which handler. In general, SOAP block handlers key off of element and attribute names that associate the semantics in the schema with the elements enclosed within the blocks.

Implementations may handle blocks differently

A SOAP processor can be a sender, a receiver, or an intermediary, which is both a sender and a receiver; an attribute can indicate that a given header is intended for an intermediary. A conforming SOAP processor has to validate the SOAP envelope, using the SOAP envelope schema. The SOAP processor has the implicit requirement to understand the SOAP schema and to raise errors if DTD elements and processing instructions are used—in other words, if the message violates the subset definition of XML used in SOAP.

SOAP processors have to follow a certain order

A SOAP message is an XML document that conforms to certain rules set down in the specification, and applications that process SOAP messages have to conform to other rules set down in the SOAP specification. For example, the specification says that implementations of SOAP processors can provide semantic *equivalence* to the order in which the message must be processed. That is, as long as the implementation comes up with the same results, the parts of a message can be processed in an order other than that defined in the specification.

SOAP processors have to enforce the XML subset used for SOAP messages

A SOAP processor must first check to see whether the SOAP message contains any header with the `mustUnderstand` attribute set to `1` or `True`, meaning that the processor must understand the header. If the message contains a header with such an attribute and the processor doesn't understand the header, the processor must return a fault and immediately stop processing the rest of the message.

SOAP processors first have to check the mustUnderstand attribute, if any

Next, the processor must check the SOAP blocks intended for it and process them, including any headers. Unless the order is constrained in an application-specific optional header, the blocks can be processed in any order.

SOAP processors, including intermediaries, must be able to access the entire envelope during processing. Intermediaries can process, remove, or add headers if they are targeted with an attribute at the intermediaries. This is a way to forward a message and to have the intermediary redirect the message to a new address, potentially with additional or changed instructions about

how to process the message. If it does not recognize some of the blocks or elements in the message, an intermediary must ignore them and forward them. The intermediary can remove headers and add headers, as long as the removed headers are intended for the intermediary.

SOAP intermediaries can process headers but not the body

SOAP applications also have to be able to process namespaces, and all SOAP messages should be encoded in XML. The processor has to discard messages that do not have correct namespaces, but it can process SOAP messages without namespaces, if it wants to.

SOAP Use of Namespaces

SOAP messages use namespaces to qualify element names in the separate blocks, or parts, of the message. Application-specific namespaces are used to qualify application-specific element names. The current v1.2 SOAP-defined namespaces are

- **http://www.w3.org/2001/12/soap-envelope:** for elements and attributes defined for the main or mandatory parts of the document
- **http://www.w3.org/2001/12/soap-encoding:** for elements and attributes defining the data types of the optional serialization mechanism
- **http://www.w3.org/2001/12/soap-faults:** to qualify fault code names used in fault messages
- **http://www.w3.org/2001/12/soap-upgrade:** to qualify the list of versions that a SOAP processor supports

Namespaces qualify block names and specify attributes

A SOAP message—an XML document with a mandatory envelope, an optional header, and a mandatory body—uses XML namespaces to qualify element and attribute names within the various parts of the document. Some element names are global, or referenced throughout the entire SOAP message, but most are local, or scoped via namespaces to the particular part of the message in which they are found.

XML namespaces scope local element use within SOAP parts

SOAP uses the local, unqualified attribute of type `ID` to specify the unique identifier of an encoded element. SOAP uses the local, unqualified attribute `href` of type `anyURI` to specify a reference to that value. Relative URIs are resolved using XML Base.

Changes in the v1.2 Draft

The major change for v1.2 is that the SOAP specification is now based on an abstract XML Information Set definition, including serialization rules. Ensuring that SOAP specification syntax has an Information Set definition helps other Web services technologies more easily consume, understand, and extend SOAP messages. Additional changes in v1.2 are as follows:

- The SOAP acronym is not spelled out.
- The specification is split into two major normative parts and an informative primer: Part 0, Primer; Part 1, Messaging Framework; Part 2, Adjuncts.[6] Generally speaking, Part 0 provides a tutorial on the SOAP specification, Part 1 describes the required parts of SOAP, and Part 2 describes the optional parts.
    - [6] See http://www.w3.org/TR/soap for draft updates. The v1.2 specification is in draft form as of this book's publication date, and is subject to further change.
- No element is permitted after the body element.
- New fault codes have been added for `MustUnderstand` and `DTDNotSupported`.
- In the HTTP binding, `text/xml` has been replaced with `application/soap+xml` to create a SOAP content category explicitly for HTTP. (HTTP filters can therefore more easily identify SOAP messages.)
- Fault codes no longer use dot notation for qualifying the names, and detail has been added for helping to interpret SOAP faults and HTTP status codes.
- An explicit `<response>` element was added to the RPC mapping to make it easier to correlate requests and responses for this interaction style.

SOAP v1.2 introduces several changes

In general, as SOAP is being developed and improved, its relation to other XML specifications is being clarified.

Difficulty in Writing about Evolving Specs

This chapter mixes examples and descriptions from the SOAP v1.1 and v1.2 draft specifications. It's difficult to write about a specification in progress. It appears that the major controversies include the division of required and optional functionality, the definition and role of headers relative to body information, and appropriately defining a message path and the role of SOAP processors along it.

As is often the case with committee drafts, many of the changes in the specification are the result of compromise, which raises the question of whether v1.2 will be a true improvement over v1.1 and the extent to which specification changes will, in fact, be adopted. This chapter takes a best guess at which changes will make the final cut and incorporates them into the examples and text.

SOAP Multipart MIME Attachments

The SOAP with Attachments specification is accepted as a note at W3C, adopted by the ebXML and RosettaNet consortia, and under consideration at the XML Protocols Working Group. This specification solves the problem of sending binary data or entire XML documents. Without the SOAP with Attachments specification, the processor has to uuencode or base64 binary data, which adds a lot of overhead.

SOAP with Attachments allows the SOAP message to be sent within a MIME envelope, with attachments linked from within the SOAP envelope. A content ID in the MIME envelope defines the SOAP message to be enclosed. For this to work, the schema defining the relationship has to be set up in advance. For example, to embed a photograph or an architectural drawing in a SOAP

attachment, the message type has to be defined in advance to indicate to the SOAP processor what's being sent and how. For example:

```
<SOAP-ENV:Body>
    <photo>
    <photo1 href=
         "cid:autumnleaves.jpg@thenewcomers.com" />
    </photo>
</SOAP-ENV:Body>
```

SOAP with Attachments sends binary data and large XML documents

The `cid` prefix tells the SOAP processor that the link is a reference to an attachment within another MIME boundary. For example:

```
Content-Type: image/jpg
Content-Transfer-Encoding: binary
Content-ID: "autumnleaves.jpg@thenewcomers.com"
...image...
```

Content IDs tell SOAP that the link is to an attachment within another MIME boundary

This approach directs processing of the payload to a separate MIME part handler. Each MIME part can be processed by a separate handler.

The ebXML and RosettaNet consortia adopted this approach for its messaging transport specification, placing the message headers into the SOAP body and referencing the actual documents via links within the body, carried as MIME attachments. (See Chapter 6 for more information on ebXML.)

Both ebXML and RosettaNet adopted SOAP with Attachments

SOAP in the Context of Existing Systems

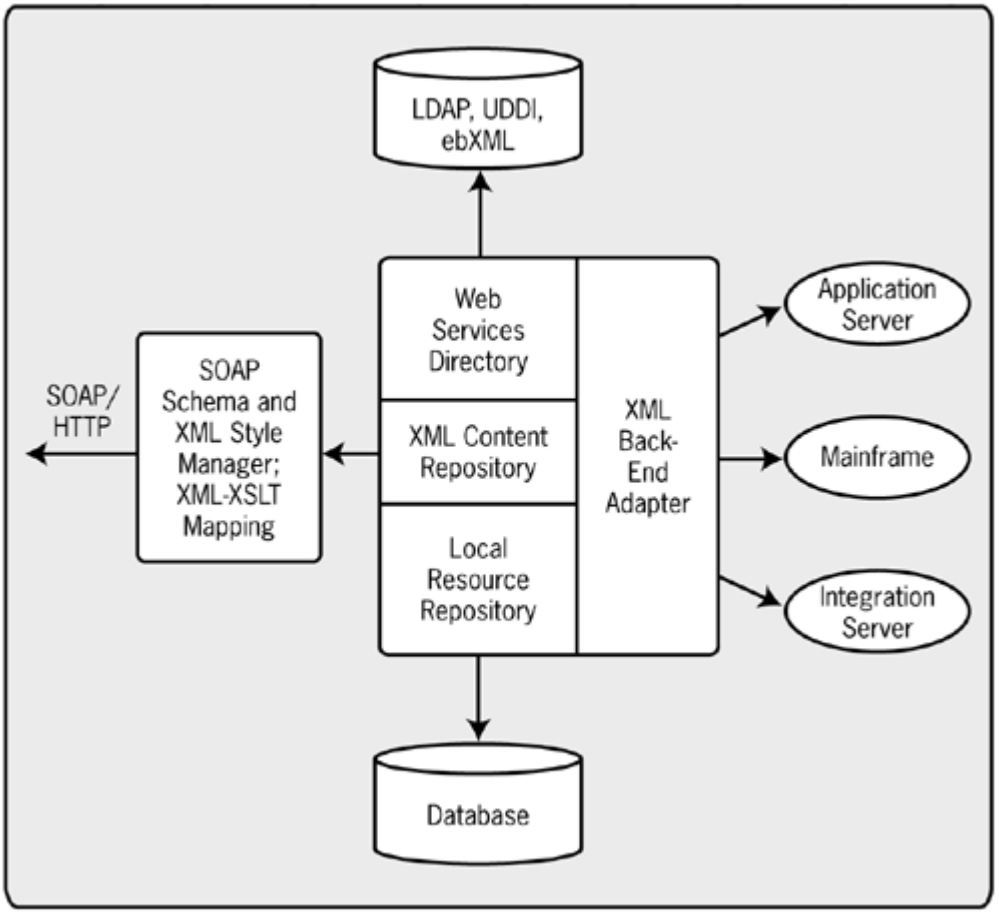
On receiving a SOAP message, perhaps formatted according to one of the various emerging XML standards, a SOAP processor may process the message and route it to the appropriate e-commerce components and/or existing applications. On the return, the SOAP processor may gather the replies from the applications and format the XML message appropriately for the type of request.

SOAP messages can be mapped to existing systems

As shown in Figure 4-2, SOAP messages may be integrated with other Web services functions, using a SOAP and an XML schema parser. The Web Services Directory manager tracks the interfaces exported to the Internet and manages them via the product's integration with the

Lightweight Directory Access Protocol (LDAP), UDDI, or the ebXML registry (see Chapters 5 and 6 for more information on UDDI and the ebXML registry, respectively).

Figure 4-2. SOAP supports existing applications.



The SOAP processor also maintains a local resource manager to allow XML fragments to be stored and retrieved as needed to create and to parse SOAP messages. The XML content repository is used primarily to manage Web service content, but XML from the repository can be extracted to produce SOAP messages. Finally, the back-end XML adapter can be used to route SOAP messages to any number of other application servers, mainframe/legacy business systems, and integration servers for complete end-to-end messaging.

SOAP processors need local metadata managers

Web services built using SOAP can easily be linked to other enterprise SOAP processors, independent of operating system, programming language, and object model. Business applications integrated with SOAP can easily and quickly exchange formatted XML messages with other businesses across the Internet.

SOAP's Future Directions

SOAP v1.2 is a simple yet extensible mechanism for bridging Web services across the Internet, regardless of platform, language, or object model. A considerable list of features and functionality

is missing, compared to a more complete distributed object system, such as CORBA or .NET, including

- Security
- Transactions
- Quality-of-service policies
- Object references
- Garbage collection
- Fault tolerance
- Work flow or business process flow

SOAP continues to be enhanced and improved

This Is Not Easy Stuff

SOAP does not define an object model, language bindings, or semantics relative to the underlying systems beyond the requirement to accept and to process an XML document in the form of a SOAP message. SOAP addresses only the wire. But this also indicates the extent to which things are left undefined relative to what's already been defined for .NET, CORBA, and J2EE, in which not only data is passed but also a method name is invoked, and the data and semantics are tied to the method name and the definition of the respective object model. Also, these distributed computing technologies include definitions for security, transactions, guaranteed messaging, and other transport-level qualities of service.

Although it's helpful to separate the concept of data mapping from the underlying software and to transport data in a manner that can be mapped into and out of virtually any existing program, the lack of completeness in the SOAP document also makes it highly subject to interpretation, susceptible to proprietary extensions, and difficult to ensure that interoperability will be preserved as extensions are defined.

Several of these features are under development at W3C, OASIS, or independent consortia; however, it may take a considerable time before all these features and functions are part of standard Web services. Once the question of SOAP standardization is settled, it is hoped that the questions of enhancements and extensions can be quickly addressed. Some of the work in progress is presented in Chapter 7.

It's unlikely that SOAP will ever address the level of detail found in existing object models and distributed computing systems, but it's very likely that more and more of the mapping to and from these systems will be automated or made easier.

Summary

The Simple Object Access Protocol (SOAP) makes it possible for Web services to exchange data, no matter where they are located in the networked environment. SOAP is mapped to HTTP by default and inherits some qualities of service from its binding to the HTTP request/response protocol. SOAP is designed to be mapped to other underlying transport protocols, from which it might inherit other qualities of service. SOAP is an evolving specification, with ongoing activity at W3C's XML Protocols Working Group focused on producing a recommended version of the specification.

The designers of SOAP intended it to provide a simple, extensible mechanism for mapping to multiple types of messaging interactions and underlying software systems. SOAP is defined using XML Infoset, schemas, and namespaces, which identify and scope the elements for its major parts: envelope, header, and body.

Chapter 5. Finding Web Services: UDDI Registry

After a Web service is set up, people must have a way to find and use the service. That's the purpose of the universal description, discovery, and integration (UDDI) registry, established by an industry consortium to create and to implement a directory of Web services. The UDDI registry accepts information describing a business, including the Web services it offers, and allows interested parties to perform online searches and downloads of the information.

To contact a business to order something, you need a way to find information about that business: street address, telephone number, Web site, or Web service address. You can obtain the information directly from a business representative, perhaps in the form of a business card, handwritten note, or e-mail. You can also look up a business name in a telephone directory and obtain the address and telephone number.

UDDI finds the Web service address and other information about a business

Similarly, the information necessary for a program running on your computer to talk to a program running on someone else's computer over the Web must be published. Although UDDI is like a White Pages or Yellow Pages for Web services, it also enables developers to interact with UDDI at both design time and runtime. In short, UDDI resources can be considered part of the Web services architecture and infrastructure.

UDDI is part of the Web services infrastructure

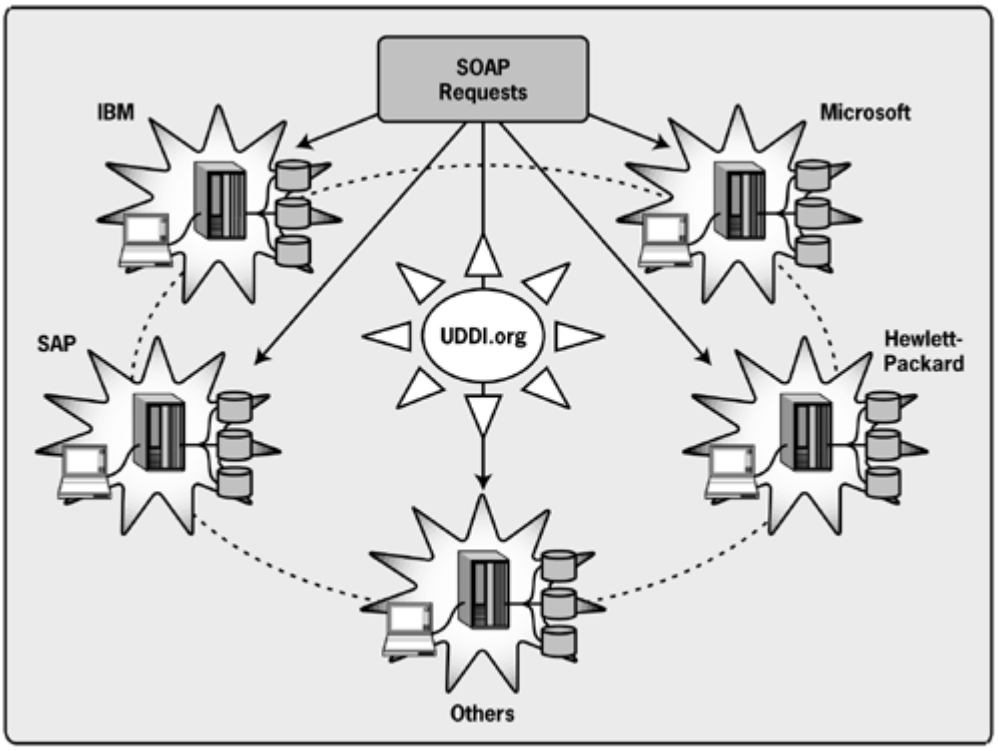
When you want to interact with a business's Web service to check inventory availability before placing an order or to check for the right hotel, car, and flight before making a travel reservation, you need to be able to find and contact the business's Web service.

UDDI supports Web services interface discovery

If you don't know the business name or if you want to compare several suppliers' terms and conditions, the problem becomes greater, and the need for a generic search and discovery mechanism becomes more evident. The UDDI registry provides such a mechanism and is therefore important to the ultimate success of Web services.

Figure 5-1 illustrates the public UDDI service hosted by IBM, Microsoft, SAP, HP, and potentially others. Each vendor provides a publicly accessible database containing business registry data, posted via SOAP requests to one of the vendor's data centers and replicated to the others. SOAP requests query the results of the posted updates to find information about businesses to be contacted, including metadata about a business's Web services.

Figure 5-1. Multiple vendors host the public UDDI service.



UDDI operators host a replicated registry database

The hosted databases are called *operator sites*, and UDDI programmers use SOAP APIs to interact with one or more of the sites. The operators do not charge for the basic UDDI service. Business data can contain pointers to Web services interface specifications, such as WSDL. Private UDDI services are often used to store a business's internal information, such as a list of available Web services.

The UDDI Organization

UDDI began as collaboration among Microsoft, IBM, and Ariba to promote the adoption and use of Web services standards. The companies founded UDDI.org and invited other companies to participate: some with founders' rights and others with advisory privileges. The companies invited as founders set the ground rules, defined the initial specifications and requirements, and decided on the eventual disposition of the work.

The three original founders were also the original operators, or hosts, of the initial public UDDI repository. Later, Hewlett-Packard joined the project and replaced Ariba as a registry host site, and SAP joined as another operator. Other operators may join over time if they meet the terms and conditions set by the original operators and if the founding membership agrees.

Covering UDDI Operator Costs

Who will pay for the operator site costs over time? Today, IBM, Microsoft, HP, and SAP are absorbing the costs of running the public registry, but as UDDI evolves and gains acceptance, the cost issues will have to be addressed. For example, the sites already offer additional services beyond the base specification requirements, such as easy-to-use browser APIs. When UDDI is transferred to an independent organization or

authority, as planned when version 3 is completed, the independent organization will be faced with covering the costs of operating the public registry. It may be possible to charge for listings, as the telephone company charges for Yellow Pages listings, or to charge for added-value services. In any case, if a public registry like UDDI becomes required for Web services to be a truly established and successful part of everyday business operations on the Internet, it's not clear how the continuing cost of the service will be borne, or by whom.

More than 300 companies joined UDDI.org in support of the effort to establish a Web-hosted business directory of services. Fifteen of the companies form the core Working Group that is responsible for setting the strategic course of the project and for resolving conflicts and making decisions. The remainder are members of the Advisory Group, which allows a company to review and to comment on specification drafts before they're made public and, on approval or invitation of one or more of the Working Group members, to join one of the specification drafting teams.

More than 300 companies belong to UDDI.org

The Working Group members have the final say on organizational issues and specification decisions, but small teams, including companies not in the core membership, perform the work on specifications. The specifications developed by the teams are sent for review and approval to the entire membership before they are published. UDDI v2, on which this chapter is based, contains enhancements, such as publishing the operator and replication specifications, improving the query capability, and providing internationalization support.

Working Group members have the final say

UDDI.org announced its intention to complete v3 of the specifications and to submit them to a standards body, perhaps W3C or OASIS, for further maintenance and enhancement. The founding members did not submit the work to a standards body originally, because it's more efficient and effective to develop the specifications in a private consortium first; more progress could be made more quickly that way, they said. Also, UDDI is unique as a specifications development group inasmuch as part of the core membership also hosts the public service based on the specifications.

UDDI v3 specifications will be submitted to an independent standards body

The operators run both a test site and a production site, allowing Web services providers and businesses to test their UDDI clients before posting real information to the production database. Of course, before being allowed to post the real information to the production database, a business needs to be authorized to do so, and the data needs to be validated. To prevent spoofing and wiretapping, communication with UDDI requires encrypted messages in the form of HTTPS. Among the work items for UDDI 3 is improved security.

Operators run both test and production sites

Ongoing Role for the Founders?

What happens to the host, or operator, part of the organization once the specifications are placed under the control of a standards organization: Will the operator sites also be placed under independent control? The issue is related as much to the disposition of intellectual property in the form of specifications and working agreements as it is to the disposition of real property in the form of computers, databases, and network equipment dedicated by private companies to a public function. Eventually, it seems clear, a public Web services registry is likely to require government regulation to ensure independent control and open access.

The Concepts Underlying UDDI

The public UDDI registry works a little like the Internet domain name service (DNS). Businesses can register with any of the hosts—IBM, HP, SAP, or Microsoft—and the information they provide will be placed into the database at that host site. At regular intervals, at least nightly, all information placed into one of the host site databases is replicated to the other host site databases, ensuring that they are all kept in synch. Users requesting data about a business can then receive the same information about a registered business from any of the host databases. When it needs to update the data, however, a business must return to the original site to which the data was placed and execute the update function.

Registry replication works similarly to DNS

For obvious reasons, security is a primary concern of UDDI.org members. Businesses are authorized to update information by one of the operators and therefore must return to the original operator to change or to delete any information they posted. Businesses wishing to register with UDDI must first obtain authorization to do so and must be approved by at least one of the operators. Approval means sending to the business an authorization token that allows the business to log onto the UDDI site and to store or to update data. Authorization to update the registry is handled by the operators individually; login information from one operator will not work for another. For all practical purposes, someone registering information with UDDI has to choose one of the operators and stick with that operator.

Security is a prime concern

Another primary concern is the quality, or validity, of the data. Someone has to ensure that the business being registered is a real business; that the business name, ID number, category information, phone number, Web page, and street address are correct; and that the business category and geographical information are correct. UDDI v2 is set up to allow third parties to do this and does validate category data.

Data validity is another primary concern

UDDI has two main parts: registration and discovery. The registration part means that businesses can post information to UDDI that other businesses can search for and discover, which is the other part. Businesses and individuals interact with UDDI by using SOAP APIs or one of the user interfaces provided by the operators or other Web services vendors. UDDI operators post WSDL descriptions of their Web services for registration and discovery. UDDI provides separate WSDL files for registration and discovery services, using its own XML document format.

The two main parts of UDDI are registration and discovery

Some UDDI SOAP APIs are used for inserting information into the registry; others, for browsing and retrieving specific information from the registry. UDDI APIs require a specific subset of SOAP; that is, UDDI does not use the optional serialization format and some other features defined in the SOAP specification. (See UDDI Support for SOAP and Unicode, later in this chapter for further information.)

SOAP APIs are used to submit and to retrieve data

How UDDI Works

UDDI information is often described as being divided into three main categories of business information:

- **White Pages:** Business name and address, contact information, Web site name, and Data Universal Numbering System (DUNS) or other identifying number.
- **Yellow Pages:** Type of business, location, and products, including various categorization taxonomies for geo-graphical location, industry type, business ID, and so on.

UDDI contains White Pages, Yellow Pages, and Green Pages information

Issues with UDDI Acceptance

Part of the problem that UDDI has to overcome is trust. In other words, will businesses really trust private vendors to host and operate such a thing? Another issue is standardization. In other words, will UDDI really become a recognized standard? Usefulness of the model may also be questioned, meaning that the proposed categorization of businesses, business information, and Web services interface information may not prove to be useful. Finally, there is a thorny legal issue with respect to doing business over the Web, including Web services. Is a business-to-business interaction over the Web equivalent legally to a paper-document exchange using a fax machine or regular mail?

Another fundamental question confronting UDDI and its users is: What does it mean that businesses are categorized? Does it somehow relate to contact information? Isn't the most important thing what a business sells rather than its geographical location? Is the location of a business in a physical sense as relevant as what it supports in terms of interactions over the Internet? With all the outstanding questions about proper and useful categorization, it's likely that the full realization of UDDI's potential is several years away if ever.

- **Green Pages:** Technical information about business services, such as how to interact with them, business process definitions, and so on. A pointer to the business's WSDL file, if any, would be placed here. Information in this category describes a service's features/functionality, including a unique ID for the service. This category is quite new and specific to the Internet.

The data structure of UDDI is expressed using complex types in XML schemas. These schemas allow for extensibility and great flexibility in the data stored for a particular business or entity.

UDDI data structures are expressed using XML schemas

Classification and identification information is useful for searching and retrieving lists and specific details about businesses. Businesses can add any number of classifications to their registrations for the purpose of assisting searches. Classification and identification information includes such things as IRS industry codes, Dun and Bradstreet DUNS numbers, product codes, geographical codes, and so on. Classifications and identifications are handled via *property bags*, or unstructured data sequences, in which virtually any type of classification and identification information can be stored.

UDDI stores classification and identification information

Flexible and Extensible Data Structures

The UDDI data structure provides so many options and extensions that it's almost impossible to predict the level of consistency that will be achieved among entries for different businesses. In other words, it may be very difficult to predict the type of detail available for a given entry. If UDDI is ever to succeed, the data will have to be normalized and regularized a good deal more than it is. Although UDDI is addressing this problem by publishing best-practices documents and providing wizards for the registration process, the problem remains.

Classification taxonomies for category information include the following:

- North American Industry Classification System (NAICS) — www.census.gov/epcd/www/naics.html
- Universal Standard Products and Services Classification (UNSPSC)—www.unspsc.org
- International Organization for Standardization (ISO) 3166— www.din.de/gremien/nas/nabd/iso3166ma (geographical regions, codes for countries, and so on)

Operator sites validate category information for industry codes via NAICS, product and service classifications via UNSPSC, and geographical codes via ISO 3166. However, including information on any or all of these categories is optional, as is checking this data when registering. Other classification taxonomies can be used, but they are not checked for validity.

Operators validate some category information

UDDI v2 supports checked and unchecked classifications and identifications. Although UDDI does not check or validate classification and identification information beyond NAICS, UNSPSC, and ISO 3166, UDDI.org does support a program that is meant to encourage third parties to provide such an ancillary service.

Registered data can be checked or unchecked

UDDI Data Model

UDDI registration information is comprised of the following five data structure types:

- `businessEntity`, the top-level structure, describing the business or other entity for which information is being registered. The other structures are related via references from this structure.
- `businessService`, the name and description of the service being published.
- `bindingTemplate`, information about the service, including an entry-point address for accessing the service.
- `tModel`, a *fingerprint*, or collection of information uniquely identifying the service specification. This data structure also supports top-level searches.
- `publisherAssertion`, a relationship structure putting into association two or more `businessEntity` structures according to a specific type of relationship, such as subsidiary or department of.

UDDI identifies five basic data structures

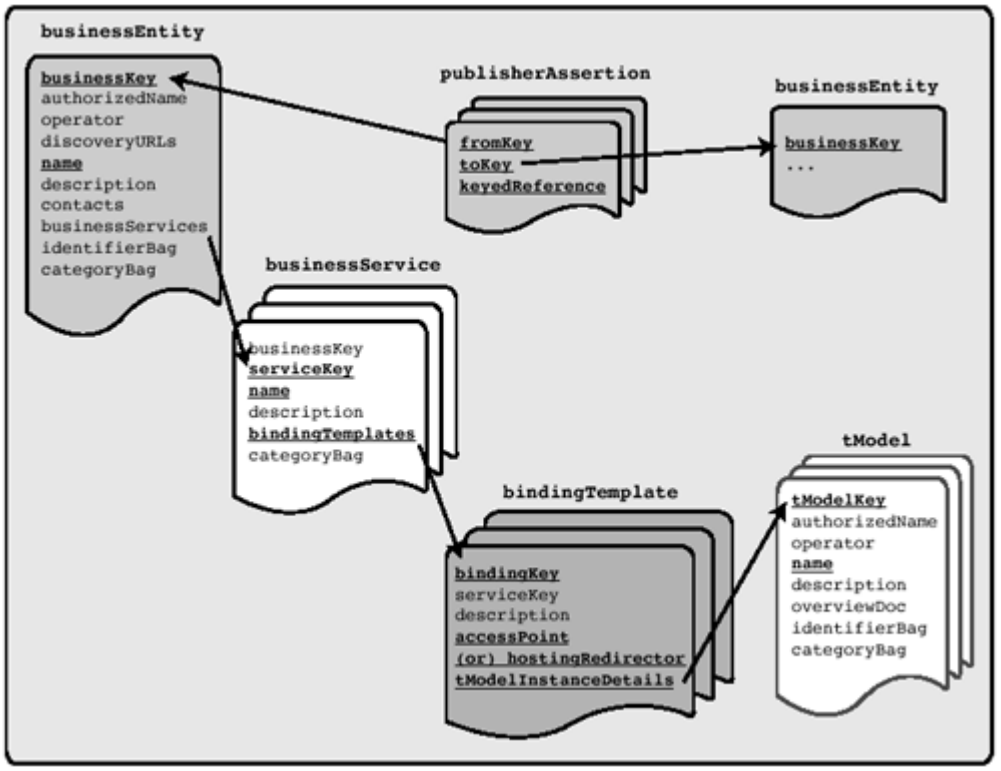
When the data is submitted for the first time, the UDDI operator assigns a unique key that identifies each of these data structures. The unique keys take the form of universally unique identifiers (UUIDs), sometimes called globally unique identifiers (GUIDs). The UUID format is derived from the Open Software Foundation (now part of the Open Group) Distributed Computing Environment; formalized now as ISO/IEC 11578: 1996 Information Technology—Open Systems Interconnection—Remote Procedure Call (see also www.iso.ch/cate/d2229.html).

UDDI operators assign unique keys for data structures

A UUID generator uses a complex algorithm, which takes into account factors, such as the current date and time, to produce a hexadecimal number that has a statistical guarantee of uniqueness. The odds of producing a duplicate number, in other words, are so great as to be impossible in practice. An example of a UUID is `C90D731D-777D-4130-9DE3-5303371170C2`.

Figure 5-2 illustrates the basic UDDI data model, in which data related to the `businessEntity` structure can be returned as results to a query. Relationships among data structures are established via key references.

Figure 5-2. The basic UDDI data model is a containment hierarchy.



In Figure 5-2, the underlined fields, such as businessKey, are required elements; that is, a request to store data will be rejected if these elements are not present, although in some cases a required element can contain zero entries. The data model is a containment hierarchy in which businessEntity is the root, or top-level, structure. The publisherAssertion schema was added for UDDI v2 to allow multiple businessEntity entries to be placed in relation with one another—for example, to accommodate large companies wishing to register their various divisions or subsidiaries.

The UDDI data model has required elements

The arrows in Figure 5-2 illustrate the elements that are used to establish the relationships. The entry in businessServices in the businessEntity schema is optional; if present, it contains one or more serviceKey fields containing key values that are present in associated businessService entries.

Relationships are established among the data structures

Similarly, the bindingTemplate element in the structure businessService contains bindingKey entries that reference any associated bindingTemplate structures. The bindingTemplate in turn references the tModel structure. Information in the bindingTemplate and the tModel structures can be combined to find a complete Web service interface.

Generic Data

UDDI is designed to accept virtually any type of Web services description, including industry-specific description languages. WSDL provides a general-purpose language for describing the interface, protocol bindings, and deployment details and as such is complementary to UDDI but not required by it (see Using WSDL with UDDI later in this chapter). In other words, the data structures established by UDDI not only predate WSDL but also are designed to be completely

extensible to contain and to reference any type of contract agreement between two parties in a distributed or networked exchange of information.

UDDI accepts virtually any type of data

Quality of UDDI Data Is a Question

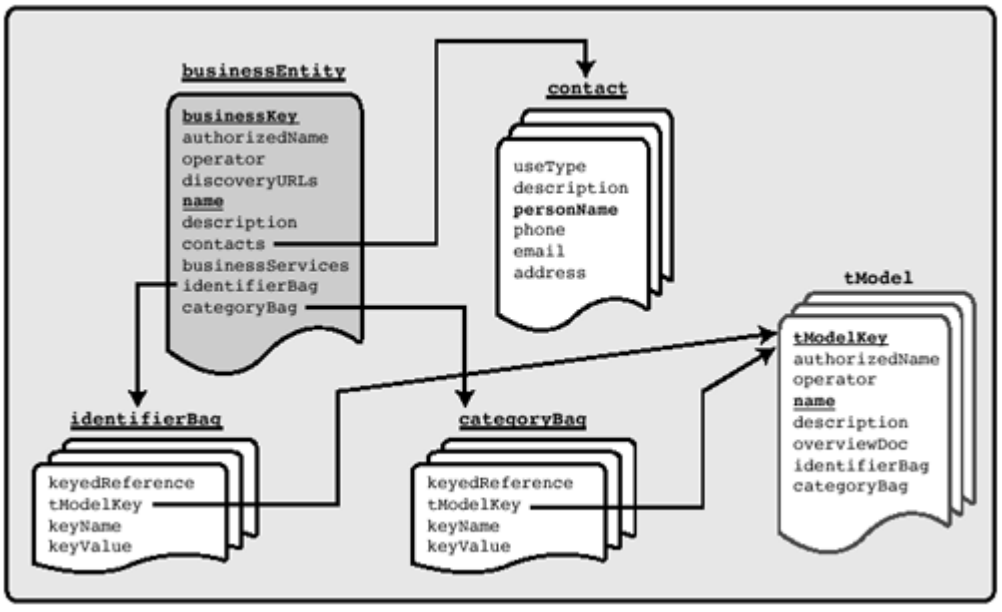
Today, even though UDDI is alive and in production, the quality of the data does not fullfill the UDDI vision. Despite provisions for third parties to work with UDDI.org members to establish validation services, the organization itself does not provide more than basic validation of category information. In other words, no mechanism has been established to ensure that the data going in is sufficient to create a truly successful Web services registry.

The XML schema structures defined for UDDI do not prescribe any underlying storage format. That is, the way in which the data is sent to a UDDI operator may be different from the way in which it is stored. This doesn't matter as long as the XML structure can be recreated from the persistent database storage format. (This is consistent with the whole XML approach to data independence, which relies on mapping into and out of XML structures that are independent from the way in which data is represented in the underlying programs and databases.)

The XML schemas don't prescribe any underlying storage format

As shown in Figure 5-3, the UDDI data structures provide several types of information, including contact, identification, and category, to be stored in association with the main `businessEntity` structure. Each of these is handled via repeating types of information sequences contained within the `businessEntity` structure. The descriptive information is basically the information that can be searched for a match and for which service information can be returned via reference to the tModel keys.

Figure 5-3. UDDI manages several types of descriptive information.



The Business Entity

The top-level structure (`businessEntity`) is where most queries start. Depending on the type of information being searched, however the queries will usually also include one or more identifiers or category keys. Queries may also start with other entities, however. Here is an example.

The business entity structure is where queries and registries usually start

```
<element name = "businessEntity">
     <complexType>
        <sequence>
           <element ref = "discoveryURLs" minOccurs = "0"/>
           <element ref = "name" maxOccurs = "unbounded"/>
           <element ref = "description" minOccurs = "0"
                                        maxOccurs = "unbounded"/>
           <element ref = "contacts" minOccurs = "0"/>
           <element ref = "businessServices" minOccurs = "0"/>
           <element ref = "identifierBag" minOccurs = "0"/>
           <element ref = "categoryBag" minOccurs = "0"/>
       </sequence>
       <attribute ref = "businessKey" use = "required"/>
       <attribute ref = "operator"/>
       <attribute ref = "authorizedName"/>
    </complexType>
</element>
```

The preceding illustrates the XML definition for the top-level `businessEntity` structure. The schema defines a complex element *sequence* constrained by the use of attribute names and usage qualifiers for repetition and required characteristics. `BusinessKey` is required, and UUID keys are assigned when the structures are created. The minimum necessary to create a `businessKey` entity, therefore, is the business name, and queries typically start with that.

Does UDDI Support Too Many Options?

With so many optional and extensible fields, UDDI is very easy to use and can contain virtually any type of information. But with all the flexibility and extensibility, how can a real pattern of registration and searching be established to support meaningful search and discovery? For example, the only required field when registering is a business name. Searches on names are difficult to get exactly correct because of spelling and other attributes of names, such as Inc. or PLC, that can be included. So it's almost certain that someone will have to review a list of names manually to pick out the right one. In addition, there's no required format for contact information or for identifying information or category information; in fact, there isn't any way to ensure consistency or appropriateness of search information.

UDDI as a concept holds great promise for Web services, and the organization is hard at work addressing the various issues. The problem UDDI is trying to solve is tremendous, and one of great potential benefit.

The Binding Template

The binding template provides information for physically accessing a Web service or other type of service registered with UDDI. A business may register multiple binding templates for a given business service and identify different access points for that service, if appropriate or useful.

A binding template contains the service access point, or URL type

The access point types in the `bindingTemplate` structure include the following `URL` types:

- mailto:
- http:
- https:
- ftp:
- fax:
- phone:
- other:

The information following the type has to be formatted correctly for the type. For example, the `mailto:` type requires a valid e-mail address; the `http:` type, a valid `URL` format; and `phone:` a valid telephone number. In this way, a business can provide the right access type for various contact mechanisms for the same or different services. No validation is done to ensure that something exists at the other end of the address; in most cases, however, this is no more of a concern than the general concern over the validity of UDDI data.

Information has to be correctly formatted for each binding type

The tModel

In UDDI terms, a *tModel* is the mechanism used to exchange metadata about a Web service, such as the Web service description, or a pointer to a WSDL file. The UDDI definition of a Web service is much broader than the examples shown in this book. The UDDI registry aims to be general enough to support any type of service accessible over the Internet. So that's why UDDI doesn't use only WSDL. If WSDL becomes popular and widely accepted, perhaps most tModels will use WSDL. For now, however, other protocols can be considered Web services by UDDI's definition, at least in terms of what's acceptable to put in a tModel.

A tModel uniquely identifies a Web service

UDDI also predates WSDL. In this book, WSDL has been used as an integral part of a Web services definition, but UDDI defines a Web service a bit differently and more broadly. UDDI defines Web services as "technical services that are exposed for either private or general use. Examples include purchasing services, catalog services, search services, and shipping or postal services exposed over transports, such as HTTP or electronic mail." Business-to-business protocols, such as RosettaNet and ebXML, can be considered Web services by UDDI's definition, although neither of them uses WSDL.

UDDI v1 predates WSDL

A tModel is roughly the UDDI equivalent of WSDL but is broader and can include pointers and references to services using addresses other than URLs and transports other than SOAP. Each tModel is assigned a UUID key by an operator when initially stored. The descriptive information stored in association with the main tModel structure is intended to help ensure that duplicate services can be identified. (Using WSDL with UDDI is discussed later in this chapter.)

WSDL is compatible with UDDI

A tModel is designed to identify uniquely interface specifications for Web services—in the broad sense in which UDDI defines them—and to help allow them to be discoverable. A tModel provides alternative entry points into the UDDI data structures in order to discover specific services and to link them to the businesses that provide them. It's also likely that multiple businesses will end up exposing the same service. Once firmly established, Web services are not going to be unique or customized to a particular business.

A tModel can be shared by multiple businesses

The tModels, representing keyed metadata about a service, are intended to ensure compatibility of interfaces or services across multiple registry entries. To be useful, a tModel should contain a pointer to a place where the user can obtain more information about the service.

Another intended function of tModels is to support registry searches for a specific service. For example, suppose that your business wants to access a catalog service or a credit card validation service. The UDDI APIs support searching for specific service definitions and listing the companies that offer compatible services. If you can obtain a tModel key value associated with the specific type of service you want, you can search UDDI for companies that offer it.

A tModel is intended to be a separate, independent entity referenceable by one or more businesses that offer a given Web service. In other words, the UDDI designers assumed that a Web service might be a generic or general-purpose service that more than one business or entity would provide. Therefore the tModel is not specific to a given business or entity and is a searchable structure in its own right, linked to one or more businesses or entities.

The tModel assumes that Web services are separately referenceable entities

The reverse is also true; a business or an entity can reference multiple tModels. In the short term, it seems likely that most businesses will expose unique Web services, but over time, it seems likely that some companies will host services for other companies, and perhaps multiple companies will get involved in the Web services hosting business.

UDDI defines tModels to look up its own services, including tModels that define the inquiry and publisher APIs for interacting with the registry and taxonomy maintenance APIs. For example, tModels are defined for NAICS, UNSPSC, and ISO 3166 classification taxonomies. These examples illustrate the use of a tModel as an abstract namespace definition.

UDDI SOAP APIs

The UDDI APIs are divided into those that register information and those that search the information in the registry. The registration APIs are used by *publishers*, that is, by businesses and/or business agents that send requests to enter the information into UDDI. The search APIs are used by registry *consumers* to find business information by category and to retrieve detailed information about one or more individual businesses that meet the search criteria.

UDDI APIs are separated into publisher and consumer APIs

In general, APIs for registering information include saving and updating current information and deleting old information. APIs for searching information include returning summary information about a group of entries or returning complete information for a specific single entry.

Anyone wishing to use any of the publisher APIs must first apply for an authentication token from an operator. Each operator distributes authentication tokens using its own mechanism; so in practice, a publisher interacts with only one operator. A publisher therefore signs up with an individual operator and is granted credentials for logging on and using the publisher APIs. The publisher APIs are required to be used over HTTPS (SSL v3.0) for encryption on the wire.

Publishers must be authorized by operators

Versioning is accomplished via namespaces. For example, the following is an attribute on the v2 APIs to indicate that the v2 APIs are being used:

```
xmlns="urn:uddi-org:api_v2"
```

Namespace revisions indicate UDDI API versions

Publisher and consumer SOAP APIs are available via HTTP `POST` only. APIs support Unicode, which requires the use of UTF-8 encoding. Publishers can register business and other descriptions using multiple human languages.

UDDI accepts HTTP `POST` operations only

Inquiry APIs

Using one or more search criteria, the inquiry APIs browse registry information and obtain specific information about a registered business or service specification once the proper unique key is obtained.

Inquiry APIs support browsing and drilling down on specific entries

Various search criteria are used to find one or more matching records. Typically, a search starts with a generic request that returns a list of businesses that match a given category or identification string. Then other inquiry APIs are used to return service and/or contact information for a given specific business. Inquiries can be done using matches on business names or identifiers or using category information, such as industry type or geographic location.

Inquiry APIs support varied criteria

Inquiry APIs are as follows:

- `find_binding`, locates specific binding information and returns a `bindingDetail` message that includes the access point, by URL type, for the service
- `find_business`, the main API for the initial search, finds information about one or more businesses and returns a business list message
- `find_relatedBusinesses`, finds businesses related to the business key and returns a business list message; checks for subsidiary or other departments for a given business
- `find_service`, finds and returns specific services listed for a specific business
- `find_tModel`, finds and returns tModel structures, providing a way to search the registry for matching services
- `get_bindingDetail`, finds and returns `bindingTemplate` information sufficient for invoking a service hosted by a specific business; returns a `bindingDetail` message
- `get_businessDetail`, gets full detail on a specific registered business and returns a `businessDetail` message, usually a second inquiry once a list of matching businesses is returned by a previous inquiry
- `get_businessDetailExt`, finds and returns an extended `businessDetailExt` message; that is the same as `get_businessDetail` but with extended information defined for after v1
- `get_serviceDetail`, finds and returns all information about a given set of registered business service information; returns a `serviceDetail` message
- `get_tModelDetail`, gets full details for a set of registered tModel data; returns a `tModelDetail` message

Inquiry APIs return no values if there's no match for one of the search keys. The APIs include an option to set a maximum number of rows to be returned and a flag to indicate that more rows exist than can be returned. Wildcard searching is available on the `find_business` API name parameter.

Inquiry operations return blank messages if there's no match for any search value

The `find_business` API, the main search API, can search via name, identifiers, categories, or tModel references and can also search for URLs. Up to five name values are supported for a search. Usually, the first search returns a list of matches to one or more of the criteria.

The `find_service` API can be used to find specific services that meet specific criteria. You can browse by name on a service, get the UUID for the service, and pass it back in to the `find_service` API to get the specific service entry.

Find specific service information

The binding-detail API gets the information you need to call a service. The information can be cached and refreshed as needed. Extension (Ext) APIs are available for compatibility when things are added. For example, `get_businessDetailExt` returns the same information as `get_businessDetail`, plus more info: extensions to UDDI for v2, in other words.

Obtain the binding detail to invoke the service

Sometimes, the results of one query feed the input for another query. For example, `get_businessDetail` might return a tModel key from the `tModelInstanceInfo` associated with the business information; then the next call can use the tModel key to obtain a specific tModel record. Finally, the tModel can be linked to one or more business entity structures to obtain the information necessary to access the service.

No authentication is required for inquiry APIs. No wire encryption is used.

Inquiries don't need security

Publisher APIs

Publisher APIs are used to store, to update, and to delete or hide information in the registry. Like all APIs that store data, the data being stored ideally takes typical subsequent query operations into account. In other words, appropriate data has to be stored to allow the inquiries to produce meaningful results.

Publisher APIs store, update, and delete registry information

Passing a blank UUID key indicates that the data is being stored for the first time. New or different information with the same UUID replaces the old information. Relationship information can be changed by making changes to the keys that define the relationship information. When `businessEntities` are registered, the operator site creates a URL that can be used to get the element being registered by using an HTTP `GET` operation.

The publisher APIs allow any number of classification codes to be stored for a business. Classifications include category codes for a business or geographical information. When registering information, a business or an agent has the option of asking for the categorization

information to be checked or to remain unchecked, that is, for UDDI to accept any categorization data the business or agent wants to store.

Any number of classification codes can be stored

UDDI operators hope that third parties will develop and provide checking services. If the checked option is used, only the name is stored unless the category information checks out.

Data can be checked when stored

Following are some publisher APIs:

- `add_publisherAssertions`, adds relationship assertions, or puts one or more businesses into relationship, such as departments or subsidiaries
- `delete_binding`, removes a `bindingTemplate`
- `delete_business`, deletes a registered `businessEntity`
- `delete_publisherAssertions`, deletes relationship information
- `delete_service`, deletes a registered `businessService`
- `delete_tModel`, hides a tModel[1]

[1] Because existing references might use them, tModels are not deleted. Hidden tModels can be reactivated by invoking the saved tModel with the original data.

- `discard_authToken`, informs the operator with which the business or the agent is registered that the authentication token is not valid anymore; that is, the business or the agent is discarding the token or is not going to use it anymore
- `get_assertionStatusReport`, a report, primarily for operator use, to check and to help validate registered relationships
- `get_authToken`, requests an authentication token from an operator site; a prerequisite for using other publisher APIs
- `get_publisherAssertions`, lists active assertions for a given publisher to define relationship information
- `get_registeredInfo`, provides a summary of all information registered on behalf of a specific user, is operator specific, and based on authentication token
- `save_binding`, either stores a new or updates an existing `bindingTemplate`
- `save_business`, either stores a new or updates an existing `businessEntity`
- `save_service`, either stores a new or updates an existing `businessService`
- `save_tModel`, either stores a new or updates an existing `tModel`
- `set_publisherAssertions`, save new assertions or replace existing ones; all elements are required

Operators can impose space limits to prevent businesses from registering too much information. Such limits are part of the negotiation with the operator you choose, but the spec calls for these limits for a new user, per account granted:

- Business entity per account: 1
- Business service definitions: 4

- Binding templates: 2
- tModels: 100
- Publisher assertions or business relationships: 10
- Maximum message size: 2MB

Operators are allowed to impose resource limits on publishers

Each registered `bindingTemplate` must contain a valid `serviceKey` that corresponds to a registered `businessService` controlled by the same account, that is, the same authentication key. It's possible to change a `bindingTemplate` relationship, however

If a `businessService` or a `bindingTemplate` is contained within more than one `businessService`, the relationships are determined by processing order: the final operation redefines the relationships for the registered information. Binding information can be moved from one relationship to another, although UDDI does not provide any validation of the resulting change. In other words, UDDI can't validate or enforce the meaning or significance of changing relationships among businesses and services.

A service can be linked to more than one business

UDDI allows relationships to be defined among businesses but has no way to reconcile, or check, duplicate entries for the same business, as when a subsidiary registers separately from the parent company. The `validate_values` API is also part of the publisher APIs but is reserved for operator use. The operator uses the API to call out to a third party with which it has agreed on using as a validation service for additional categorization taxonomies and identification schemes. This allows UDDI data categorization to be validated after a save or an update operation on the UDDI nodes when further details are required.

Subsidiaries can be linked to parent companies

Usage Scenario

Imagine that the Skateboots Company wants to register its contact information, service description, and online service access information with UDDI. The following steps are necessary:

1. Choose an operator with which to work. Each operator has different terms and conditions for authorizing access to its replica of the registry and may provide some value-added services on top of the basic UDDI services. Also, whenever you need to update or to modify the data you've registered, you have to go back to the operator with which you entered the data.
2. Build or otherwise obtain a UDDI client, such as those provided by the operators.
3. Obtain an authentication token from the operator.
4. Register information about the business. Include as much information as might be helpful to those searching for matches.
5. Release the authentication token.
6. Use the inquiry APIs to test the retrieval of the information, including binding template information, to ensure that someone who obtains it can use it successfully to interact with your service.

7. Fill in the tModel information in case someone wants to search for a given service and find your business as one of the service providers.
8. Update the information as necessary to reflect changing business contact information and new service details, obtaining and releasing a new authentication token from the operator each time.

The examples in the following sections show how the Skateboots Company would register its information and how a distributor interested in carrying the Skateboots line might find information about how to contact the company and place an order, using the Skateboots.com Web services.

Skateboots gets clearance from one of the operators

Updating the Registry

After obtaining an authentication token from one of the operators—Microsoft, for example—the Skateboots.com developers decide what information to publish to the registry and use one of the UDDI tools provided by Microsoft. If necessary, the developers can also write a Java, C#, or VB.NET program to generate the appropriate SOAP messages. Here is an example.

Skateboots can register and be discovered

```
POST /save_business HTTP/1.1
Host: www.skateboots.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "save_business"

<?xml version="1.0" encoding="UTF-8" ?>

<Envelope xmlns="http://schemas/xmlsoap.org/soap/envelope/">
    <Body>
       <save_business generic="2.0" xmlns="urn:uddi-org:api_v2">
         <businessKey="">
         </businessKey>
         <name>
         Skateboots, Inc.
         </name>
         <description>
         You know, like the ones in Batman and Robin . . .
         </description>
         <identifierBag> ... </identifierBag>
         ...
      </save_business>
    </Body>
</Envelope>
```

This example illustrates a SOAP message[2] requesting to register a UDDI business entity for Skateboots Company. The key element is blank because the operator automatically generates the UUID key for the data structure. Most fields are omitted for the sake of showing a simple example.

[2] Note that the examples in this chapter use SOAP v1.1.

Skateboots can always execute another `save_business` operation to add to the basic information required to create a business entity. Most likely, Skateboots Company will want to include one or more identifiers and category IDs and link its two Web services—for `PreSeason` and `In-season` orders—to tModel information.

Retrieving Information

After Skateboots Company has updated its UDDI entry with the relevant information, companies that want to become Skateboots distributors can look up contact information in the UDDI registry and obtain the service descriptions and the access points for the two Web services that Skateboots.com publishes for online order entry: preseason bulk orders and in-season restocking orders.

Customers retrieve the information

```
POST /get_businessDetail HTTP/1.1
Host: www.skateboots.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "get_businessDetail"

<?xml version="1.0" encoding="UTF-8" ?>

<Envelope xmlns="http://schemas/xmlsoap.org/soap/envelope/">
    <Body>
       <get_businessDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
          <businessKey="C90D731D-777D-4130-9DE3-5303371170C2">
          </businessKey>
       </get_businessDetail>
    </Body>
</Envelope>
```

This example illustrates a sample SOAP request to obtain business detail information about the Skateboots Company. Once you know the UUID, or key, for the specific business that's been registered, you can use it in the `get_businessDetail` API to return specific information about that business. Normally, to refine the initial search, you might send a message to UDDI containing the business name or a partial name with a wildcard to see whether you can get any matches.

```
<businessList generic="2.0" operator="uddi.sourceOperator"
    truncated="false"
       xmlns:="urn:uddi-org:api_v2">
    <businessInfos>
       <businessInfo businessKey="C90D731D-777D-4130-9DE3-
       5303371170C2">
          <name>Skateboots Inc.</name>
          <serviceInfos>
             <serviceInfo    serviceKey="D90D731D-777D-4130-
9DE3-...">
                <name>PreSeason Orders</name>
             </serviceInfo>
             <serviceInfo    serviceKey="E90D731D-777D-4130-
9DE3-...">
                <name>In-season Orders</name>
             </serviceInfo>
```

```
          </serviceInfos>
        </businessInfo>
        <businessInfo> ... [more Skateboot manufacturers]
    </businessInfos>
</businessList>
```

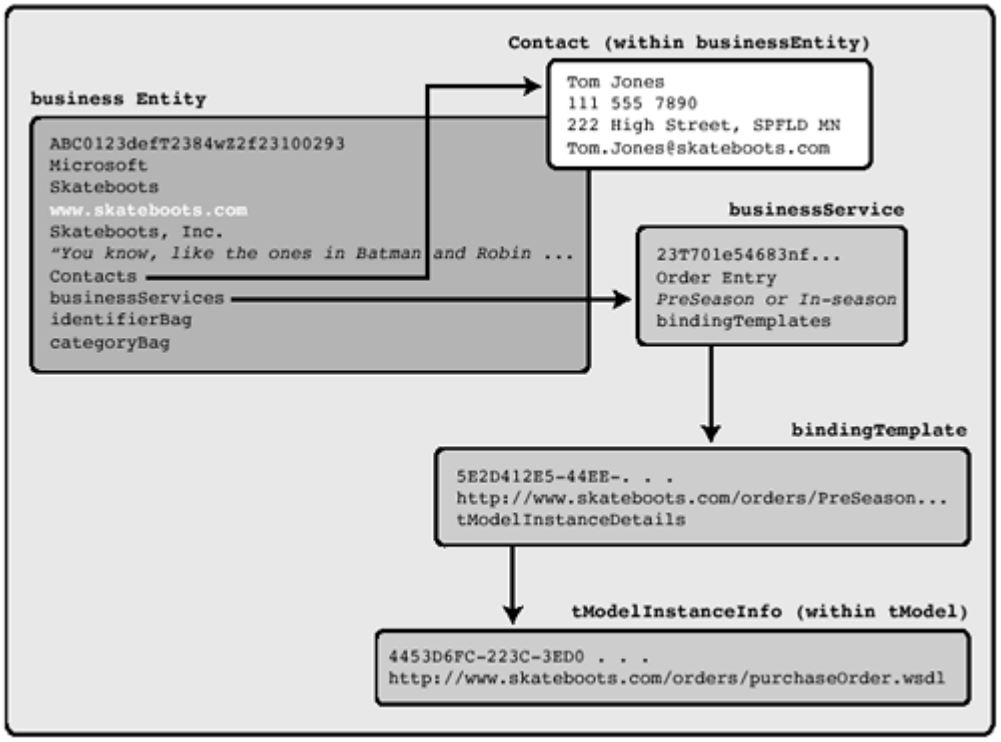
The preceeding example illustrates a sample message that might be returned from any of the UDDI operators on receipt of the `find_business` request. The service information—the `serviceKey`—is then used to locate the binding information that is needed to interact with the preseason and in-season Web services.

Another approach would have been to search for the tModel definition of the service, using identifier or category search IDs. But in this case, the distributor knows that only one company manufacturers the particular skateboots it wants to sell, so it's unlikely that a generic service exists for ordering from the factory.

Alternatively, search using the tModel

Figure 5-4 illustrates sample information stored in UDDI for the Skateboots Company's Web services after it's done registering. A more complete entry would include identifiers and category taxonomy information, which also contain pointers to tModel structures, providing an alternative traversal path for obtaining service information.

Figure 5-4. This sample shows a partial UDDI entry for Skateboots.com.



The sample in Figure 5-4 shows a straightforward traversal path, starting with an inquiry on the `businessEntity` for Skateboots and from that entry is able to find the contact information—

stored within the `businessEntity` structure—and locate the Web services information via the binding template.

Using WSDL with UDDI

The UDDI data model defines a generic structure for storing information about a business and the Web services it publishes. The UDDI data model is completely extensible, including several repeating sequence structures of information. Although it was designed and specified before WSDL, UDDI did anticipate the need for something like WSDL to be included. However, UDDI is designed and intended to work with any service description language. UDDI includes data structures appropriate for storing protocol bindings and network services.

WSDL is fairly straightforward to use with UDDI

WSDL is represented in UDDI using a combination of `businessService`, `bindingTemplate`, and `tModel` information.

As with any service registered in UDDI, generic information about the service is stored in the `businessService` data structure, and information specific to how and where the service is accessed is stored in one or more associated `bindingTemplate` structures. Each `bindingTemplate` structure includes an element that contains the network address of the service and has associated with it one or more `tModel` structures that describe and uniquely identify the service.

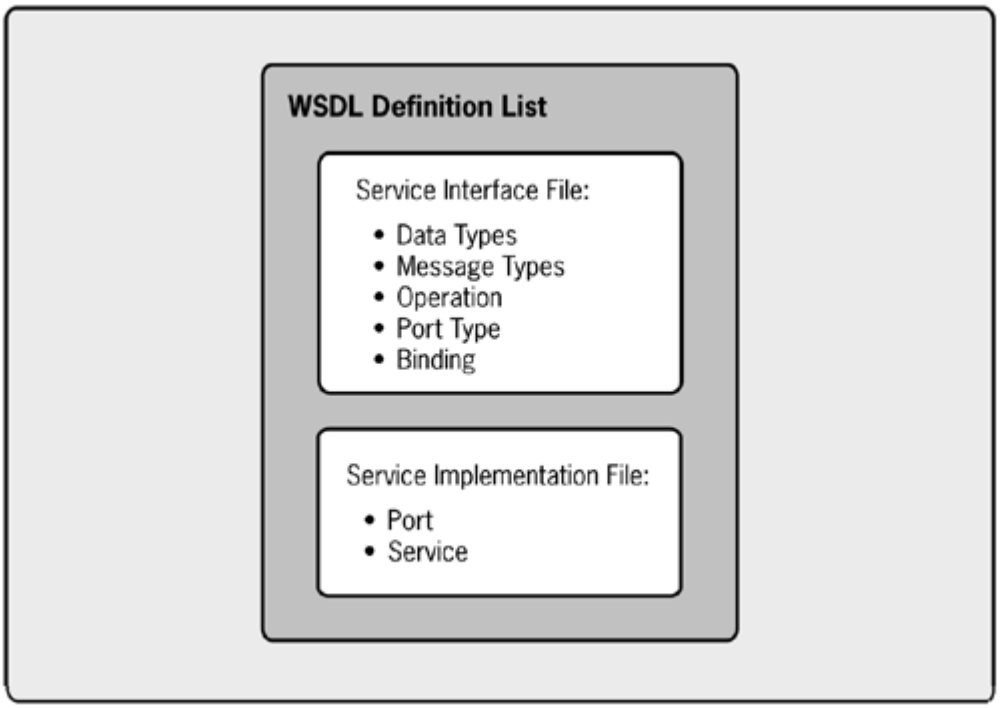
WSDL information is divided between generic service information and specific binding information

The business service information describes the Web services. A binding template contains technical information about a service entry point and construction specifications. The tModel contains descriptions of the specifications for services or taxonomies, which are the basis for a technical fingerprint for the service specification.

The access point in the binding template gives the end-point address of the Web service. It's possible to store WSDL information in several ways in UDDI, given UDDI's flexibility and extensibility. WSDL best-practice authors recommend splitting the reusable information from the information specific to a given service.

Although WSDL is not required for registry with UDDI, IBM's toolkit anticipates the usefulness of registering WSDL in UDDI by dividing WSDL into two main parts for proper fit into UDDI: service interface and service implementation. Figure 5-5 shows how the WSDL parts fit into this split. The split is recommended so that information common to a category of business, such as message formats and data types, port types, and protocol bindings, are in the reusable portion, whereas the information about a particular service end point; that is, the port definition, is included in the service implementation definition portion.

Figure 5-5. The WSDL parts list can be grouped according to recommended UDDI split.



As described in Chapter 3, WSDL is composable of separate XML files, or elements, that can be included together to form a complete WSDL file. WSDL service interface definitions are registered as UDDI tModels; the `overviewDoc` field in the `tModel` points to the corresponding WSDL document. Using UDDI, the Web service developer follows the URL stored in the `overviewDoc` element to obtain the WSDL document.

Split WSDL information along element boundaries for compatibility with UDDI

When UDDI is used to store WSDL information, or pointers to WSDL files, the tModel should be referred to by convention as type `wsdlSpec`, meaning that the `overviewDoc` element is clearly identified as pointing to a WSDL service interface definition.

For UDDI, WSDL contents are split into two major elements the interface file and the implementation file—to similarly split the abstract definition of a service from its physical implementation onto a given transport. Abstract definitions can thus be mapped onto multiple transports.

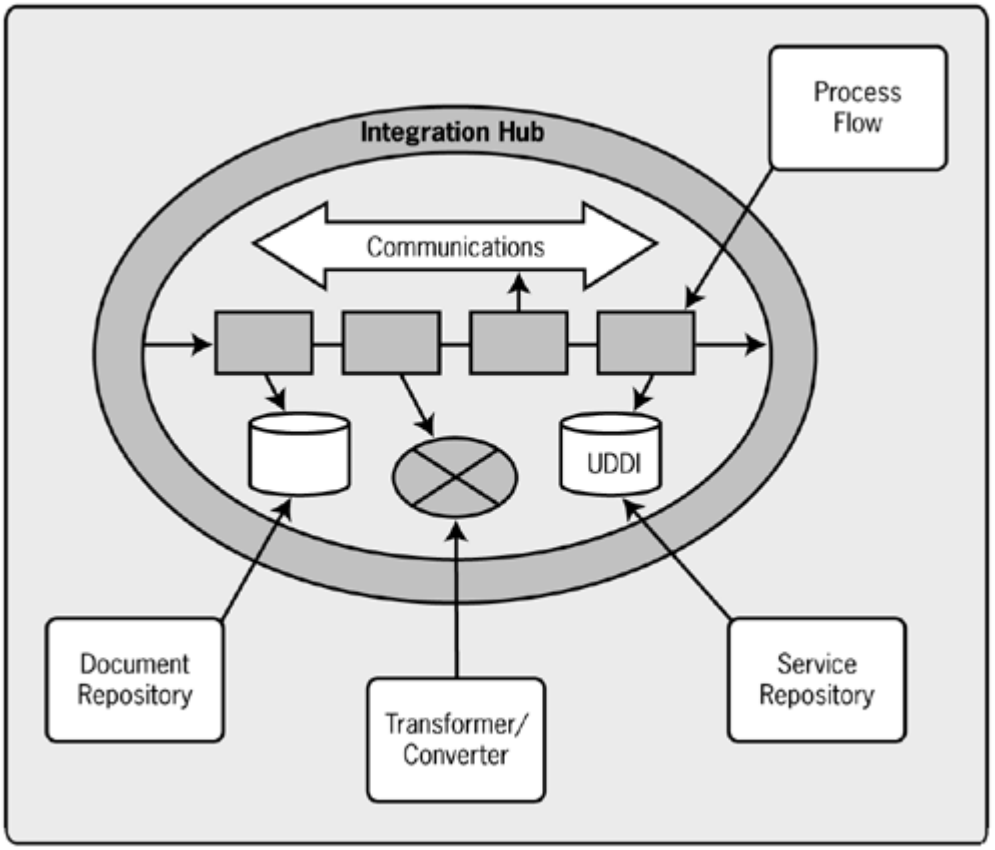
UDDI for Private Use

UDDI can be used inside the firewall as a directory of internal Web services, or internal integration points. Applications can be wrapped using XML integration technologies and registered with the internal UDDI, including extensions to the UDDI data model that make sense and/or apply to internal use scenarios. When using Web services technology for integrating internal applications, registering the integration points in a common registry makes it easier for other departments to come along later and discover applications ready to be integrated with, just as it's done on the Web.

UDDI is also suitable for inside-the-firewall storage of metadata

Figure 5-6 shows the role of an internal UDDI registry as a service metadata store in corporate integration projects. Because the UDDI data structure is extensible and because virtually any information can be stored for the identity and the category bags, it's easy for a business to adapt the registry for internal use.

Figure 5-6. UDDI can be used for internal integration architectures.



A local, or private, UDDI can be used to store metadata about the end points of an organization's applications that expose interfaces to other applications for the purpose of integrating them. For example, a bank may have customer data in the retail savings account application, mortgage account division, and money market or mutual fund division.

A private UDDI can register and discover application integration metadata

A customer relationship management system used to consolidate fees for a customer with several types of accounts would have to obtain information from these three different applications. If Web services or another service-oriented mechanism is used to accomplish the integration of these applications, metadata about the interfaces could be stored in the private UDDI registry. If, say, a marketing application wanted some or all of the same information from the other applications, developers could search the private registry to find metadata and interfaces about the existing, integrated applications. UDDI therefore can provide a service that is useful in the context of internal integration architectures comprised of other services such as an integration hub, communications manager, document repository, data transformation, and process flow.

UDDI Support for SOAP and Unicode

This section describes the specific requirements for using SOAP with UDDI, including how UDDI treats the SOAP Action header, and prohibits the use of SOAP headers and the optional SOAP encoding. This section also describes the UDDI requirements for Unicode support.

SOAP

UDDI v1 required an empty string value for the SOAP Action HTTP header field. However v2 allows the field to include the name of the UDDI API referenced within the SOAP message. For example:

```
SOAPAction: "save_business"
```

UDDI v2 permits the SOAP Action HTTP header

UDDI does not support SOAP headers. However, UDDI implementations are allowed to ignore any headers in the SOAP message as long as they do not include the `mustUnderstand` attribute, which, of course, requires a SOAP processor to understand the header within which the attribute is specified. If a UDDI operator receives a SOAP message that includes a required header, the operator will reject the message and return a SOAP `mustUnderstand` fault. Consequently, the actor attribute of a SOAP header is not supported, either.

No SOAP headers or actor attribute

UDDI does not support the optional SOAP encoding. Any SOAP message that does contain the SOAP encoding namespace attribute will fail and return a SOAP fault, a UDDI error code of `E_unsupported`, and a SOAP `errinfo` text that describes the error.

No SOAP encoding

SOAP request and response documents sent to and received from UDDI operators use the default SOAP namespace without an XML prefix. For example:

```
xmlns="http://schemas/xmlsoap.org/soap/envelope/"
```

Default namespace required

Unicode

UDDI operators decided to use UTF-8 encoding for all requests and require that all operator sites support all the characters defined for the Unicode standard for multiple-language support. Messages can therefore contain description text in multiple languages for business information that needs to be registered in more than one human language.

Unicode is required for all requests

Summary

UDDI provides a flexible, powerful, and extensible mechanism for registering and discovering business information over the Internet. UDDI also has many advantages for use inside the firewall. The public UDDI registry is hosted by private companies, and the specifications will be controlled

by those companies within the UDDI.org consortium until they are turned over to an independent standards body for maintenance.

UDDI represents its structured data by using XML schemas and supports SOAP APIs for storing and retrieving information. UDDI operators host both a test and a public registry. Information in the UDDI repository is not, and may never be, consistent or comprehensive enough to solve the worldwide categorization problem. However, UDDI provides a strong, widely adopted and used model and set of services for Web services metadata management.

Chapter 6. An Alternative Approach: ebXML

An international group jointly sponsored by the United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT)[1] and the Organization for the Advancement of Structured Information Standards (OASIS)[2] drove the initial electronic business XML (ebXML) activity. The ebXML group's charter was to research, develop, and promote global standards based on XML to exchange business data electronically and by doing so to improve the efficiency and lower the cost of doing business worldwide.

[1] See http://www.unece.org/cefact/. UN/CEFACT also sponsors the EDI standardization work.

[2] See http://www.oasis-open.org.

The ebXML initiative paralleled the development of the core Web services technologies and pursued the same goals.

Hundreds of companies, industry organizations, vendors of software and services, and standards bodies participated in the open process that created the ebXML specifications. Anyone with a computer, an Internet connection, and an e-mail address could take part, and often did, although primarily Electronic Data Interchange (EDI) champions and large companies with sufficient capital to invest in specification drafting guided the effort. The initial eighteen-month effort began in November 1999 and ended in May 2001 with the publication of the phase 1 specifications.

If implemented as specified, ebXML provides the ability for trading partners to exchange standard electronic business messages, such as orders, delivery schedules, receipts, and invoices. These messages might include entire XML documents or only the data that has changed or is required to fill in the document, assuming that both partners have a current copy. The goal is to allow large and small businesses to concentrate on production and marketing and to devote less time and effort in administration, particularly of connected IT systems. The assumption is that businesses will be able to directly connect—or at least map to—their IT systems over the Internet. Of course, this same assumption also lies behind the SOAP, WSDL, and UDDI initiatives; ebXML differs mainly in the degree to which it recognizes business process modeling as a core feature and includes industrial-strength quality of services.

ebXML assumes that businesses can connect their IT systems over the Internet

Late in the initial drafting cycle, the ebXML messaging team voted to adopt SOAP with Attachments, signaling convergence between ebXML and SOAP, WSDL, and UDDI. In contrast to those three specifications, the ebXML specifications reflect requirements from the business community for a more robust, secure communication environment and include a business process modeling methodology and schema.

In phase 2, OASIS assumed responsibility for maintaining and enhancing the infrastructure specifications, including the work of making the ebXML registry compatible with Web services metadata. UN/CEFACT continued with the development of metadata and information models, including the business process information model (BPIM), the core components (CC), and business information object (BIO) libraries.

Ongoing work is divided between OASIS and UN/CEFACT

Under UN/CEFACT, the ongoing ebXML effort comprises ten new projects defining BPIM metadata to ensure consistent business models, using the Unified Modeling Language (UML) and the Unified Modeling Methodology (UMM). This goal is driven by the need to define reusable processes and information blocks. It should also be noted that UN/CEFACT's BPIMs are technology neutral; that is, they can be used for ebXML, EDI, and any other packaging and transport syntax. As to the work on CC, it is a bridging tool for the migration of data-centric, bottom-up, EDI to the process-centric, top-down, BPIM world, but is not a mandatory component.

Overview of ebXML

Fundamentally, SOAP, WSDL, and UDDI grew out of an interest among Internet community members in establishing a remote procedure call (RPC) mechanism for exchanging XML documents across the Web. By contrast, ebXML grew out of an interest among existing EDI community members to create a better way to exchange business documents using XML and to transport them over the Internet, rather than using expensive, private value-added networks (VANs). Later, after a document-oriented interaction style was added to SOAP, the two efforts began to converge, and ebXML adopted SOAP as its default messaging transport.

ebXML defines the business process interaction

The goal of ebXML, like that of EDI, is to eliminate the expensive process of sending paper documents through the mail or via fax and to eliminate duplicate data entry tasks in the business applications of trading partners by directly connecting their IT systems. In other words, the goal is to eliminate duplication when information is entered once by the trading partner that creates the document and again by the trading partner that receives the document and to achieve efficiencies and cost reductions by minimizing human steps within the processes. The idea is to encode business data in XML documents that can be mapped directly and automatically to existing business systems.

ebXML's goal is to streamline electronic commerce

The intention of ebXML, furthermore, is to improve on the EDI model by eliminating the painful *prior agreement* aspect, which requires intense legal activity to create a trading partner agreement (TPA) between parties wishing to exchange documents via EDI. Furthermore, ebXML seeks to eliminate the difficult EDI setup period that requires encoding the TPA into the applications on both sides.

In addition, the ebXML alternative is intended to be available at lower cost and to be easier to use than EDI. For nearly twenty years, the EDI standard has been used by a relatively small number of large businesses that could afford the start-up cost and deal with its complexity. EDI was developed by pooling many related data item definitions, thereby creating a superset of all possible data items for such documents as purchase orders, shipping bills, invoices, and so on. This information set proved unwieldy. Also, EDI required private networks; expensive, specialized software; and high-priced consultants.

ebXML is intended to be cheaper than EDI

In deliberate contrast to the EDI approach, ebXML focuses on defining the processes and procedures by which businesses might agree to exchange documents and the messaging interactions required to implement those processes and procedures.[3] Instead of identifying the various data items and many optional items in a business document, the ebXML user starts by identifying the *business process,* or interaction, between two or more parties that exchange business documents.

[3] An earlier standard for intrabusiness document flows, RosettaNet, had adopted a similar model but was focused on the electronics industry. By contrast, ebXML seeks to accomplish the same thing for general business. RosettaNet confirmed ebXML's accomplishment by announcing that it would align with ebXML.

A Simple Example

Each business that participates in an exchange of documents that is ebXML-compliant identifies the actions that it must perform to send and to receive a given document. The information needed for such a document is then derived from the pattern of exchange. From the business process, therefore, ebXML derives the business data necessary to carry out the agreed-on interaction.

Begin by identifying the business interaction

For example, a business process between two companies might be defined as follows:

1. Based on its expected volume for its normal winter season, Harry's Sports sends an advance order to Skateboots Company.
2. Skateboots Company sends an acknowledgment of the order to Harry's Sports and provides an expected shipment date.
3. Harry's Sports may send an inquiry about the status of the order at any time and receive an update, such as a message saying that the shipment will be a week later than expected.
4. Skateboots ships the order and sends a message confirming the shipment to Harry's.
5. Harry's receives the order and sends Skateboots an acknowledgment that the order arrived.
6. Skateboots sends an invoice to Harry's for the order.
7. Harry's sends payment for the invoice.
8. On receipt of payment, Skateboots sends an acknowledgment, ending the transaction.

Some aspects of this interaction may be defined as optional, such as sending acknowledgments on receipt of the order, the invoice, and the payment. Other aspects may not be defined as optional, such as sending the order and the invoice.

A business transaction, such as this purchase order, fundamentally involves an exchange of things between businesses, such as goods for money. The ebXML business process and trading-partner information allow options and requirements in such an exchange to be explicitly defined and agreed on between two or more *parties* to a business transaction and implemented electronically over the Internet.

In this case, Skateboots would have stored in the ebXML registry information about the business transactions it offers over the Internet, such as this long-running preseason order. Harry's Sports

would have discovered the information from browsing the ebXML registry and contacting Skateboots to formalize the agreement. The goals for the ebXML registry include the automatic discovery of the trading partner, using search criteria specified in the collaboration partner profile (CPP).

Use the registry to store and to retrieve business process information

Define the Documents

According to the interaction definition, the transaction depends on certain documents, including the purchase order and the invoice. With ebXML, the data and the documents are not defined; rather the interactions between the two businesses are formalized into an agreement that can be executed. The agreement assumes that the exchange of information will be carried out according to the predefined business process interaction.

The ebXML BPIM identifies the interactions among trading partners participating in a transaction. Each activity within a transaction is associated with an exchange of information, whether request or response. Information is at the root of any exchange activity, transaction, and business process. Although the various parties to an interaction may have different information requirements, ebXML does not want to standardize the information exchange; EDI showed that this approach is neither reasonable nor successful.

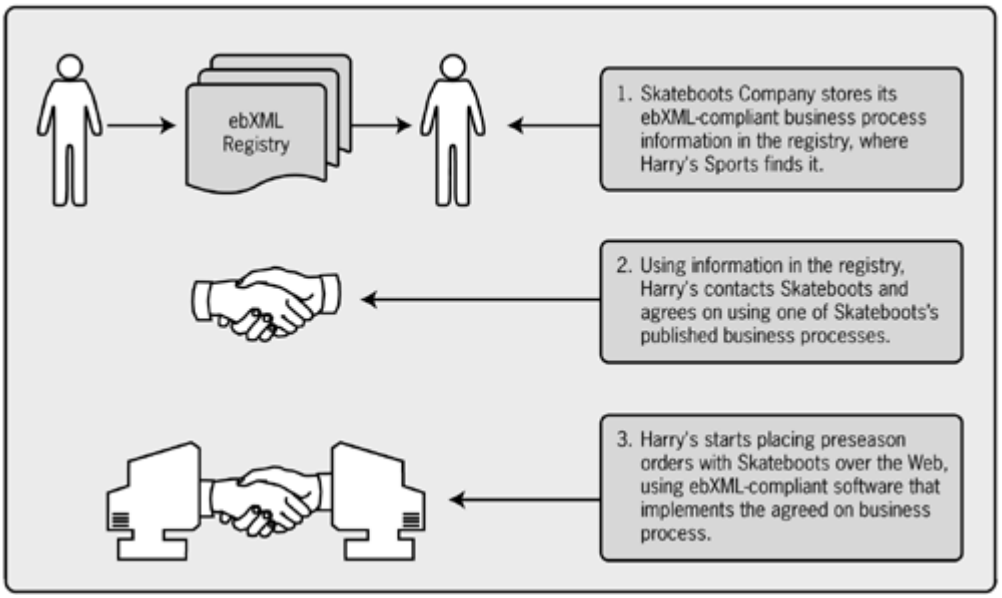
A business process may have a number of activities defined to achieve the same end result, but each of those activities typically has different constraints and different information requirements that dictate a different path through the BPIM. In other words, the ebXML BPIM allows and even assumes that multiple paths will be specified to achieve the same business interaction, depending on the trading partner or trading partners involved. Each valid path represents a separate scenario for that business process. In the CPP, an ebXML party identifies which of those scenarios it is capable of executing. For the collaboration partner agreement (CPA) to be completed, both parties agreeing to a scenario require at least one interaction pattern match.

A business process may support multiple activities

Once the interaction pattern is formalized and agreed on by the participating businesses, documents either can be chosen from one of the businesses or from one of the various standards bodies dealing with XML documents or can be based on one or more of the models stored in the registry. An ebXML assumption is that documents will be defined in a common, shareable way as companies implement ebXML and store their models and profiles in the repository.

As shown in Figure 6-1, ebXML interaction patterns contrast with those proposed by SOAP, WSDL, and UDDI in that ebXML assumes a negotiation phase between two or more businesses that use ebXML-compliant software. Similar to the SOAP/WSDL/UDDI approach, ebXML assumes that an automatic discovery and negotiation phase will eventually become part of establishing a connection between trading partners. The automatic discovery and negotiation phase would be accomplished using ebXML-compliant software that searched the registry for compatible business process metadata and quality-of-service requirements on behalf of a trading partner. On finding such compatibility, the software would contact the trading partner's ebXML-compliant software to accept the process and initiate the interaction.

Figure 6-1. An ebXML interaction starts with an agreement on the business process.



A negotiation phase is required to agree on a shared business process

The ebXML registry is designed to support browsing and ad hoc queries. Once the business searching the registry—Harry's Sports, in this case—finds the supplier it is looking for and identifies a target business process interaction—in this case, the preseason purchase order process—the client business can either contact the supplier to formalize agreement on the business process to be implemented between them or use automatic negotiation via CPP, if supported. In either case, once the process is agreed to, the final step is for both trading partners to set up their ebXML-compliant software to implement the agreed-on interaction—in this case, submitting preseason orders for skateboots.

The ebXML registry supports ad hoc queries

Legality of Electronic Documents

The legality of electronic commerce transactions is a thorny question. That is, does an electronic purchase order have the same legal standing as a paper one? This question is outside the scope of ebXML, which does not deal with the issue. However, for electronic commerce to become reality, this issue will have to be resolved. An electronic commerce transaction will have to have legal status equivalent to a paper transaction. This is already true for an automatic teller machine at a bank or an online stock trade. But these transactions use private networks and proprietary systems.

Generic purchase orders, invoices, and ebXML business processes do not yet have this status when transmitted over the Internet, unless, perhaps, when they are printed out, but such important questions as how to represent company letterhead and authorized signatures legally remain unanswered. UN/CEFACT's legal working group says that the CPP/CPA phase can form a legal contract and has published a document that descibes

the requirements for an e-commerce agreement that is fullfilled when trading parties perform a static CPP/CPA agreement. This document does not have the legal standing of UN member-country law but provides a good indication of a possible future resolution of the issue.

Some ebXML members also are proposing to establish the legality of the dynamic, or on-the-fly, trade-partner negotation via intelligent software applications. But this is a much greater problem, perhaps unsolvable in practice, as there is very little, if any, precedent for software programs to be recognized as able to establish legal contracts on their own. The huge question here, of course, is defending or validating what your software program has negotiated on your behalf.

Deploying ebXML

Interactions, such as the fairly simple pattern described earlier, can be codified using the ebXML CPA, which serves a similar purpose as WSDL. Assuming that ebXML-compliant software systems are available, the CPA drives the automatic interaction between the IT systems of trading partners that can agree on a common business process, given sufficient information in the registry.

The CPA codifies agreed interactions

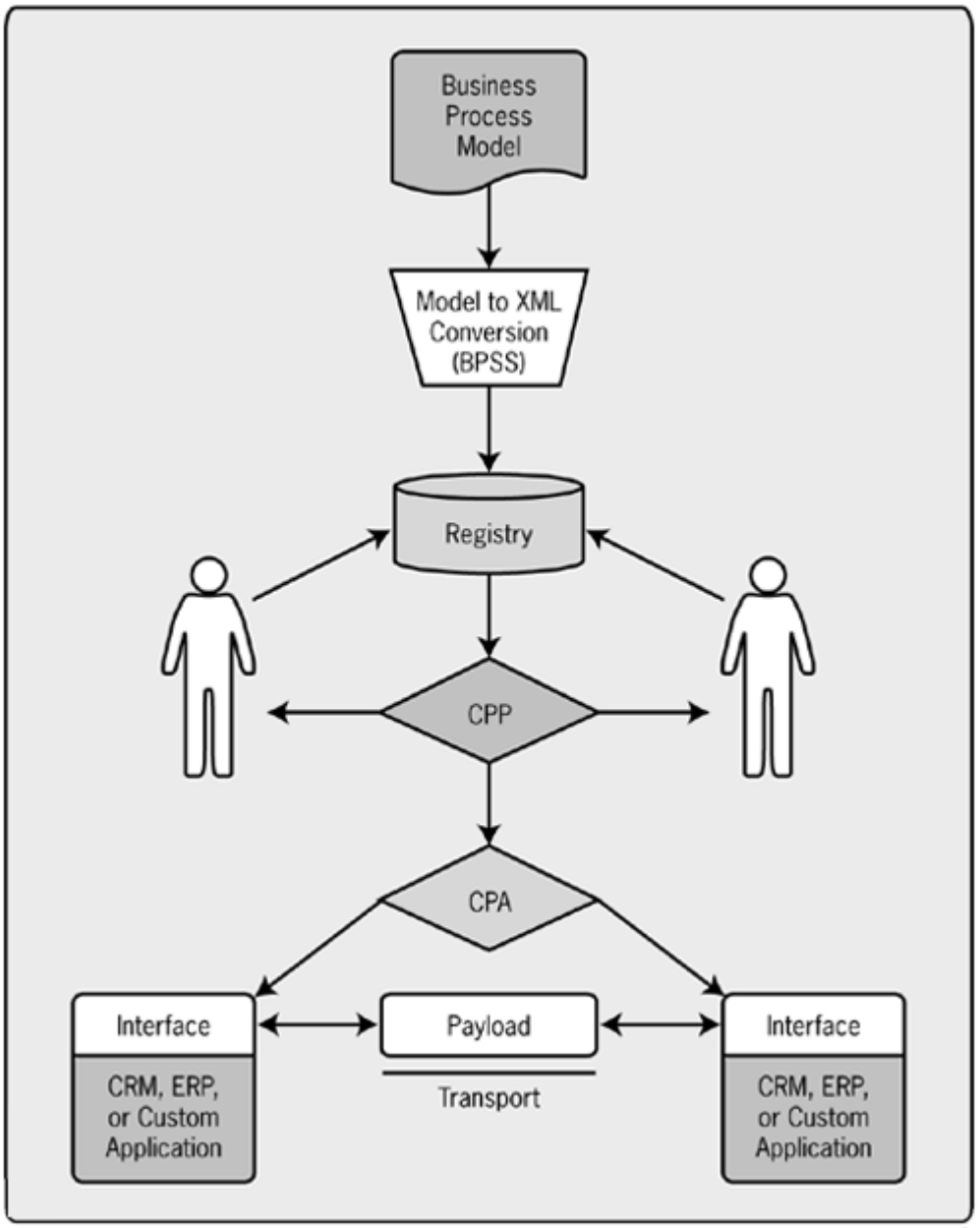
In our example, Harry's wants to find a manufacturer of skateboots and uses the ebXML registry to obtain information necessary to order from Skateboots Company. Even if Harry already knows that it wants to order from Skateboots Company, Harry's can use the ebXML registry to obtain information about the services offered by Skateboots Company for online ordering. Harry's can examine Skateboots's registered CPP and associated business process models to see whether there's a potential match with Skateboots's codified terms, conditions, and interaction scenarios.

The CPP is examined for a potential interaction pattern match

The ebXML model assumes that each business taking part in an Internet transaction will search the ebXML registry to find a qualified trading partner: one that has registered an appropriate interaction pattern, such as the purchase order and shipment process for Skateboots Company, and that a person or program at that business will explicitly decide whether the published interaction fits. After agreeing on the interaction pattern, or business process, the ebXML-compliant software at each company can begin to execute the defined interaction.

As shown in Figure 6-2, ebXML deployment requires ebXML-compliant software to implement the scenarios, or patterns, it describes and to which each trading partner agrees. The ebXML-compliant software has to understand how to derive or map interfaces to business applications, using the CPP, which defines all potential collaboration mechanisms that a given business supports. The CPA defines a specific collaboration on which the two or more trading partners agree; in other words, a CPA represents the intersection of two or more CPPs.

Figure 6-2. The ebXML deployment model requires ebXML-compliant software.



The modeling information is recommended but not required to be defined using the UMM, a subset or profile of the UML standard from the Object Management Group (OMG). The UMM or other modeling information is mapped to a business process specification schema (BPSS) and stored in the registry, where it can be referenced from a CPP.

Modeling is typically done using UMM

The information in a CPP, which is also intended to be stored in the repository, identifies which transports are available for the interaction, such as SMTP or HTTP, and security requirements, such as encryption, nonrepudiation, and digital signatures. The CPP also identifies the business processes available for engagement: Is it just order placement or a full supply-chain interaction? Does a business process have various scenarios?

SOAP versus ebXML

Skateboots Company and Harry's Sports could certainly have chosen to implement the same business process scenario using SOAP, WSDL, and UDDI. The ebXML specifications and the SOAP/WSDL/UDDI specifications significantly overlap. In fact, the ebXML messaging service is mapped to SOAP with Attachments, and UDDI can be linked to the ebXML registry. UDDI may not implement all the features of the ebXML registry, but UDDI is flexible and extensible enough to support search and discovery of ebXML metadata. With the future direction and standardization of the UDDI and ebXML registries still an open question, the convergence of these two industry efforts would seem to be in everyone's interests.

However, the sad truth is that primarily for political reasons, the two efforts maintain a conscious and forced divergence. Sun Microsystems supports ebXML while Microsoft does not, for example. Some of Microsoft's proposals for extending the core Web services specifications overlap with functionality defined in ebXML, which Sun prefers to reference.

Therefore adherents of each tend to emphasize the differences rather than the similarities, which is too bad, as the similarities are more significant than the differences. Given the inherent extensibility of WSDL, it's also possible to include CPA and CPP information in WSDL and to define or to advertise an ebXML-compliant business service using WSDL. Similarly, one can easily imagine an ebXML-compliant system that exposes Web services interfaces, that is, using WSDL instead of the collaboration protocol in the CPA schema.

The CPP allows a business to provide as many or as few details as it is willing to disclose publicly. Some businesses may provide a link to their complete product catalogs; other businesses may provide simply a high-level description of product categories or services. However, at least one business process model scenario has to be identified if an ebXML-compliant interaction is to be supported.

The CPA includes the agreed-on transport and security options, as well as the selected business scenario, via a binding to a business process specification schema (BPSS). The target party can either accept or reject an offered CPA. After the party accepts a CPA, the selected business scenario can be initiated by sending the first business information message in a payload over the agreed transport. Then the rest of the scenario is executed, and the corresponding messages are exchanged.

The CPA also includes agreed-on security and transport options

After the business process modeling is done, the models are mapped to XML—that is, using the BPSS format—and stored in one or more registries. The business service interfaces are registered, as are the CPPs, to describe the interface implementations. Vendors and businesses providing ebXML-compliant software then use information extracted from the registry to build the business service implementations that wrap existing or new business applications. The CPA defines the specific payload contents for each interaction, which by default is delivered using SOAP with Attachments as the transport, but other transports are possible.

Business process models are mapped to XML and stored in the registry

The ebXML Specifications

Phase 1 ebXML work consists of three major parts:

- Transport, or messaging specification, based on SOAP with Attachments
- Registry, using either UDDI or native ebXML—stores business process model definitions and collaboration protocol definitions
- Collaboration, meaning the automatic execution of models and interactions among business partners

Implementations are started with the transport

Vendors are typically implementing ebXML in this order, although some are concentrating first on modeling tools.

Businesses can use ebXML to find out about one another using the registry, to define TPAs, and to exchange XML messages in support of business operations. The goal is to eventually perform all these activities, without human intervention, over the Internet. Work on ebXML is not yet completed, however. Work on the specifications and compliant software continues at UN/CEFACT, OASIS, W3C, and other consortia, and at various systems and software companies. Although phase 1 has been completed and implementations are on the market, the ebXML specifications remain works in progress.

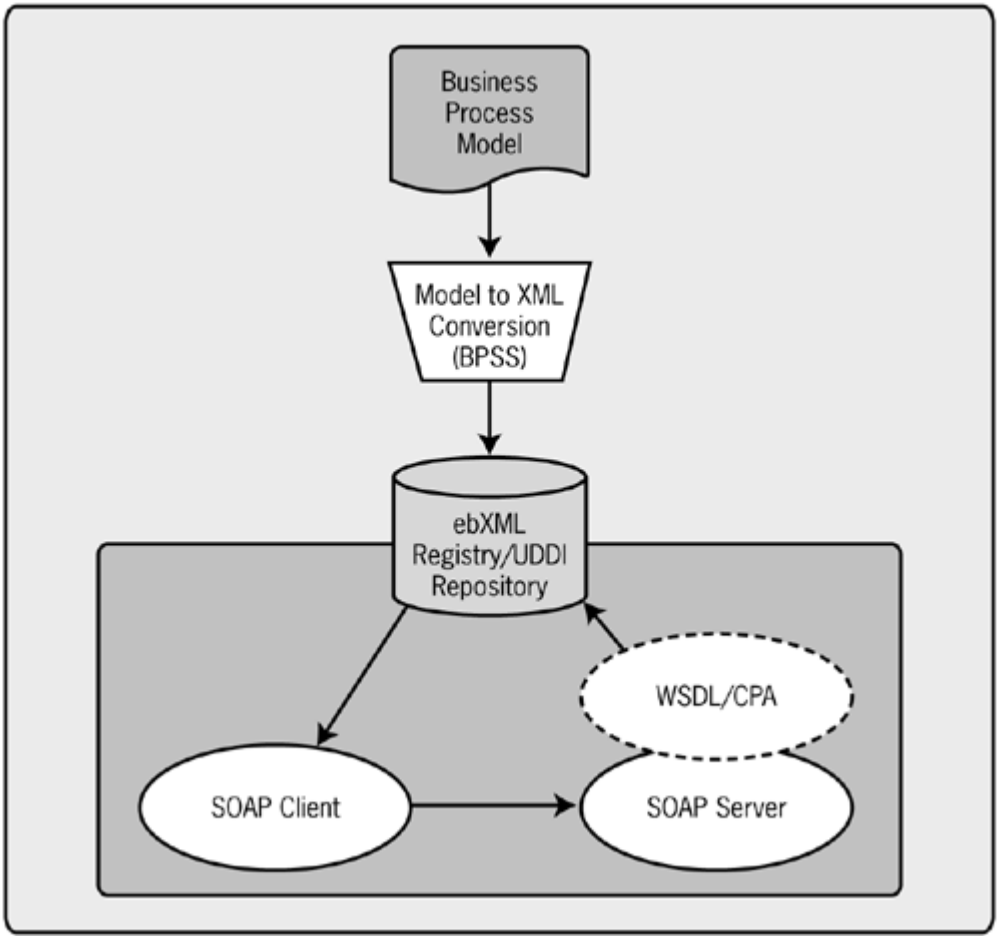
ebXML specification work continues

Many similarities exist between ebXML and SOAP/WSDL/UDDI, as both efforts sprang from very similar requirements and objectives. Because ebXML started from the business process model and SOAP/WSDL/UDDI started from the RPC model, the overlap is not complete. It's more like an intersection; ebXML maps down to implementations at the transport level, and SOAP/WSDL/UDDI map up to the document-oriented interaction style. Much commonality is also seen between UDDI and the ebXML registry.

Conceptual overlap with Web services technologies

The shaded area of Figure 6-3 illustrates the conceptual overlap between the ebXML specifications and the SOAP, WSDL, and UDDI Web services specifications. Both specify a type of service interface that gets stored in a registry, and both specify a transport using SOAP. Although WSDL is not really equivalent to CPA, the ideas are very similar.

Figure 6-3. ebXML overlaps with SOAP, WSDL, and UDDI.



Although originally SOAP/WSDL/UDDI and ebXML started out as separate, competing efforts, today the ebXML specifications can be seen as a source of requirements for the continuing W3C work on SOAP and related specifications. The ebXML specifications define features, functions, and qualities of service beyond those in the basic Web services specifications. Several vendors are supplying ebXML-compliant products, but it's possible that the convergence will overtake the momentum of ebXML and that all its features and functionality will be incorporated into Web services. The Java Community Process (JCP) initiative called Java APIs for XML Registry (JAXR)[4] defines a Java API common to both the ebXML repository and UDDI, for example.

[4] See http://java.sun.com/xml/jaxr/index.html.

ebXML can be a source of requirements for Web services

Architectural Overview

The ebXML specifications, created separately by individual authoring teams, were intended to fit within a comprehensive technical architecture, as described in the technical architecture specification. The ebXML technical architecture builds on the experience of EDI but takes advantage of the flexibility of XML and the ubiquity of the Internet. The ebXML architecture maps a business process and information model to XML documents and defines requirements for the software that processes the documents and exchanges them among trading partners.

ebXML is intended to fit within a comprehensive technical architecture

Adoption of ebXML

Because ebXML aligned with SOAP, it seems likely that the ebXML transport will be adopted and implemented. However, the bigger question centers on business process, which is where ebXML started and how ebXML is different from EDI. Standardizing business processes is at least as big a job as standardizing business data. Many attempts at the latter have been made, and many have failed. Business processes, like the businesses that invented them, are often unique and sometimes are part of a company's competitive advantage. It's not clear that businesses will agree to standardize them or even to disclose them in sufficient detail for standardization. And if this doesn't happen, ebXML may fall flat.

The ebXML architecture defines the following:

- Business processes and their associated messages and content
- A registry and discovery mechanism for publishing business process sequences and related message exchanges
- Company profiles
- Trading-partner agreements
- A uniform message transport layer

Conformance to the ebXML architecture is defined for each individual specification and also, generally, in the technical architecture specification. Conformance appears to require implementation of all the specifications, although many of the features and functions are listed as optional within the individual specifications.

Conformance requirements vary

The main ebXML specifications[5] are as follows:

[5] See http://www.ebxml.org/specs/.

- **Technical Architecture:** an overview and detailed summary of the complete architecture
- **Business Process and Information Modeling:** definition of the BPSS and a modeling methodology for producing a BPSS
- **Collaboration Protocol Profile and Agreement Specification:** definition of trading partner information and how it's codified in XML documents
- **Registry and Repository Services:** for storing and retrieving business process models and specifications, trading-partner identity, and technology requirements
- **Messaging Service:** definition of how ebXML-compliant documents are transported across a network
- **Core Components and Core Library:** shared components across industries, including a business document overview

ebXML consists of six main specifications

Subsequent subsections detail the business process, collaboration protocol, registry, messaging, and core components specifications and provide some examples and illustrations of the major points.

Some people say that ebXML can be implemented in stages, but formal conformance is defined as comprising all the architectural components of the ebXML infrastructure. Sorting out which vendor confirms to what specification can be a complex task, however.

Business Processes and Information Modeling

The steps necessary to execute a business transaction with one or more business partners define a business process. A business process is first defined and then executed. The execution of a process consists of a series of messages exchanged among one or more partners. The messages usually include business documents, but they often are acknowledgments or other informational messages as well.

Business process steps execute a business transaction

A business process is defined using a combination of a trading-partner profile, which lists the business transactions supported by a particular *party* to the exchange, and the trading-partner agreement, which identifies the specific transaction to be executed by the parties.

The central idea of ebXML is that the parties to a business transaction will first characterize the transactions that they generally offer and then negotiate an agreement with each other—either manually or automatically, if automatic negotiation is supported—to execute one or more of the generally available transactions.

Parties first describe their transactions and then negotiate mutual agreements

Specification versus Implementation

One of the things that distinguished the networking standards battle between the Open Systems Interconnection (OSI) standards of the International Organization for Standardization (ISO) and the Transmission Control Protocol (TCP) of the Internet Engineering Task Force (IETF) during the 1990s was that the OSI specification was developed first, and then vendors were expected to implement it. IETF, on the other hand, required reference implementations before a specification could be adopted.

The IETF way of working turned out to be much faster and more effecient and helped result in the victory of TCP over OSI networking as the adopted standard. Of course, there were other reasons having to do with simplicity, ease of implementation and adoption, and feature match with requirements, but the parallels between ebXML and SOAP are clear.

Today, ebXML exists primarily as a collection of specifications, whereas SOAP, WSDL, and UDDI are both preliminary implementations and specifications. If ebXML succeeds, it will show that the business process interaction is more important to define than is the data or the document. Using the Internet for business is going to require the adoption of standards that make sense to businesses and that offer the level of features, functionality, and qualities of service they have come to expect and rely on in their

proprietary networks and LANs.

If ebXML succeeds and is widely adopted and implemented, ebXML will become such a standard. Meanwhile, however, SOAP, WSDL, UDDI, and other Web services specifications are evolving toward what's defined by ebXML. Which one will win may depend on who gets there first; given the complexity and uncertainties surrounding the critical ebXML registry specification, this may well be the telling battle.

The ebXML specification for business process and information modeling contains a methodology for describing business process scenarios and an XML schema, BPSS, for encoding those scenarios. Each business process scenario defines trading-partner roles, relationships, and responsibilities for a given business transaction. The BPSS is capable of representing a scenario for a multiple-party exchange, as well as for a simple exchange between two parties.

To set up a business process scenario, a party creates a trading-partner profile to list all the transactions to be generally offered and then creates a CPP for storage in the ebXML registry. The business process *models*, also stored in the registry, use the BPSS and are bound to the CPP when it's created. When a new trading partner is enlisted for a party, a CPA is created, theoretically as the matching subset, or compatible intersection, between the two parties' CPPs and referenced BPSSs. Following an exchange of CPPs and the discovery of a mutually compatible subset of the CPP definitions, the CPA is created and proposed by the initiating party to the receiving party. The receiving party can reject an unwanted collaboration.

Parties list their transactions in a CPP

The BPSS includes standard business collaboration patterns that can be used to determine the exchange of messages and signals between trading partners. The BPSS also contains configuration information for runtime system deployment. The BPSS is available as a UML profile and as an XML DTD or schema. Information in the BPSS is the primary input for the CPPs and CPAs. Here is an example.

The BPSS includes standard collaboration patterns

```
<BusinessTransaction name="Create Order">
    <RequestingBusinessActivity                    name=""
isNonRepudiationRequired="true"
        timeToAcknowledgeReceipt="P2D"
        timeToAcknowledgeAcceptance="P3D">
    <DocumentEnvelope isPositiveResponse="true"
        businessDocument="Purchase Order"/>
    </RequestingBusinessActivity>
    <RespondingBusinessActivity                    name=""
isNonRepudiationRequired="true"
        timeToAcknowledgeReceipt="P5D">
    <DocumentEnvelope             isPositiveResponse="true"
businessDocument="PO
        Acknowledgement"/>
    </RespondingBusinessActivity>
</BusinessTransaction>
```

The example illustrates business process characteristics in a BPSS, such as nonrepudiation—that is, to ensure that the document is from whom it's supposed to be from—and a requirement for an acknowledgment of the message within a certain time period. These characteristics are defined using attributes of the `RegistryBusinessActivity` element. The BPSS also includes the basic request and response messages that define the business process scenario itself, that of sending a purchase order and receiving the acknowledgment that the purchase order was received. The security and timeout features are good examples of the type of business requirement ebXML definitions include beyond what's found in the basic SOAP, WSDL, and UDDI specifications.

The use of the UMM is specified for graphically depicting a business interaction and its associated flow of business data among trading partners. Business processes are then used to define the message sequence depicted in the graphical model. A BPSS can be generated from its UMM depiction.

Business processes can be defined graphically

Trading-Partner Profiles and Agreements

An important feature of ebXML is the specification that defines a business's ability to conduct business with a trading partner over the Internet. An exchange of electronic information between trading partners requires each partner to have sufficient detail of the other partner's environment and the technology with which the other partner sends and receives messages. Although ebXML is aimed, in general, at solving this problem, specific definitions of trading-partner tools, technologies, and transport options are required.

As mentioned earlier, the BPSS serves as a primary input for the CPP and the CPA. Included in the CPP and the CPA are industry information; details of partner requirements for transport, messaging, and security; and binding to a BPSS. Trading partners that agree on the transport, messaging, security, and BPSS formalize the agreement by generating a CPA and then execute the CPA electronically to initiate a transaction. In short, these profile and agreement documents represent a methodology, or approach, to solve the problem of how to identify business transactions to be executed over the Internet between two or more trading partners easily and perhaps automatically.

BPSS serves as primary input for the CPP and the CPA

A CPA and the BPSS it references define the nature of the *conversation* between two or more trading partners. The conversation represents a single unit of business and consists of one or more business transactions. The BPSS defines the sequence of request and response messages comprising a particular business transaction.

Conceptually, therefore, a business-to-business (B2B) server[6] at each partner's site implements the CPA and the BPSS. The B2B server must include the runtime software that supports communication with the other partner's B2B server, execution of the functions specified in the CPA, interfacing to each partner's back-end systems, and logging the interactions between the partners for audit and recovery.

[6] B2B servers can be built in a variety of ways, such as on top of application servers, using native Java code with servlets, or using .NET servers.

B2B servers implement the CPA

The CPA is derived from the intersection of two or more CPPs, mutually agreed on by trading partners wishing to exchange business documents electronically using ebXML. The definition of exactly how to create a CPA from mutually agreed on CPPs is left up to the implementer, which means that a variety of mechanisms will be provided. Both XML DTD and schema content

models are available for the BPSS, CPA, and CPP documents. CPPs and CPAs are validated against the following XML schema:

http://www.ebxml.org/schemas/cpp-cpa-v1_0.xsd

The CPA is derived from two or more CPPs

The model of ebXML states that a formal CPP description is to be published and then universally understood by interested parties. Publishing a CPP is achieved by posting the document to UDDI or to an ebXML global registry. A CPA contains the messaging service interface and implementation details pertaining to the mutually agreed-on business process scenarios. Trading partners may, but don't have to, register their CPAs in an ebXML registry. Once a CPA has been agreed on, however, electronic business interaction may begin. Here is an example.

```
<tp:PartyInfo>
    <tp:PartyId tp:type="DUNS">123456789</tp:PartyId>
    <tp:PartyRef xlink:href="http://example.com/about.html" />
  - <tp:CollaborationRole tp:id="N00">
    <tp:ProcessSpecification tp:version="1.0" tp:name="buySell"
        xlink:type="simple"
        xlink:href="http://www.ebxml.org/processes/buySell.xml" />
    <tp:Role tp:name="buyer" xlink:type="simple"
        xlink:href="http://ebxml.org/processes/buySell.xml#buyer" />
    <tp:CertificateRef tp:certId="N03" />
  - <tp:ServiceBinding tp:channelId="N04" tp:packageId="N0402">
    <tp:Service tp:type="uriReference">
       uri:example.com/services/buyerService
       </tp:Service>
    <tp:Override tp:action="orderConfirm" tp:channelId="N08"
        tp:packageId="N0402"
        xlink:href="http://ebxml.org/processes/buySell.xml#orderConfirm"
        xlink:type="simple" />
    </tp:ServiceBinding>
  </tp:CollaborationRole>
```

This example contains an excerpt from a simple CPA published in the ebXML Collaboration-Protocol Profile and Agreement Specification. The excerpt contains trading-partner, or *party*, identity information, including the DUNS ID number for the partner, the link information for the XML document to be exchanged, a role definition for the partner ("buyer," in this case), a service binding, and the definition of an order confirmation requirement.

Migration of traditional EDI-based applications and other legacy applications to platforms based on the ebXML specifications is facilitated. In particular, the CPP and the CPA are aimed at the migration of applications based on the ANSI X12 838 Trading-Partner Profile used in EDI. Because they define the technical requirements for the electronic exchange of business documents CPAs are similar to the EDI trading-partner agreements, on which they are based. Unlike EDI TPAs, however, the CPAs are not paper documents but XML documents, which themselves can be electronically exchanged.

ebXML supports migration from traditional EDI applications

Mandatory Components

It's very strange that language in the ebXML specifications describes the CPP and CPA as optional and that a company *may* or may not support it. Even though the success of ebXML most likely depends on the widespread use of the registry, storing CPPs and CPAs in the registry is itself optional, as are the CPPs and CPAs in the first place. Also, the messaging system is not required to be SOAP—nor is any particular middleware or

other software assumed to exist—although something must be there to execute things. Furthermore, business process and information modeling is not mandatory. However, if implementors and users select the Business Processes and Information specification, they *shall* use the UN/CEFACT Modeling Methodology, which uses UMM. This means that one can directly register XML, but it isn't clear from the specs. It would seem especially important to require the CPP and the CPA, as the successful interaction between trading partners basically depends on their existence or the existence of something like them. But if they aren't there, what is to be used in their place isn't clear: potentially, some form of private agreement, or perhaps SOAP/WSDL.

Registries and Repositories

The ebXML Registry and Repository Service specification defines a set of services that store and retrieve the business process, message, and vocabulary definitions used in the transactions executed among trading partners. Companies can link private and public ebXML registries to store their business process models, BPSSs, CPPs, CPAs, and related information.

Registry services store and retrieve the process model, message, and other transaction definitions

The intent of the ebXML registry is to allow companies to search for and to discover other companies with which they wish to enter into online trading partnership. The ebXML registry services are designed to share service descriptions, discovery links, and catalogs of business collaboration definitions. If the goal of the Core Component specification is realized, reusable business documents also will be available through registry services.

In contrast to the UDDI registry, the ebXML registry is designed to support ad hoc queries. UDDI can, however, be used to implement an ebXML registry, and ebXML.org publishes a whitepaper describing how to do this.[7] UDDI and ebXML registry services can complement each other. For example, it seems likely that businesses will be able to discover services that are published in UDDI registries and that link to ebXML registries.

[7] See http://www.ebxml.org/specs/rrUDDI.pdf.

The ebXML registry does not assume any runtime access by the applications that use it; all information needed for electronic business interaction must be retrieved from the registry ahead of time and cached. Once authenticated, registry users are allowed access, using an XML signature, and authorization for registry users is defined by an access control policy with default roles for `admin`, `user`, and `owner`.

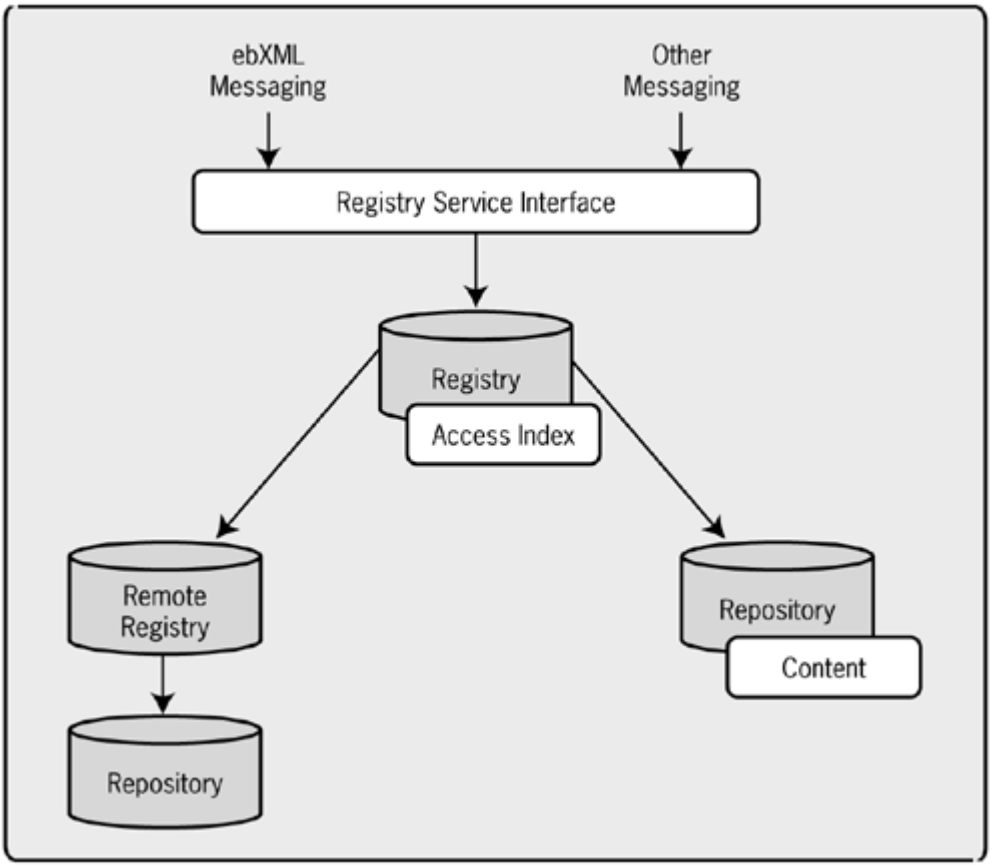
The registry does not assume runtime access

Once a registry is set up, ebXML-compliant software can access information in the registry to find trading partners and other parties with which to do business. This could take the form of a search for a particular product or service, such as a local flower shop, bolts for an airplane, or a generic search for a business in a particular industry, such as furniture manufacturing. Access to the ebXML registry can be achieved via the ebXML messaging service, but this is optional, and access can also be provided via any number of other transports and protocols.

Figure 6-4 illustrates the ebXML registry and repository architecture, in which ebXML messaging and other messaging protocols access the main registry by using the service APIs. The main registry maintains an access index pointing to the repository or repositories in which content is stored. The service interface also can be used to map to UDDI when UDDI is used as the repository. Registries and repositories can be remote, and multiple repositories can be linked

together. In the case of multiple repositories, the access index in the primary repository—the first one interacted with—provides the location of the remote registry.

Figure 6-4. The ebXML registry architecture separates the repository from the registry using an access index.



Repositories store content; registries point to repositories

Registry services create, modify, and delete registry items and their metadata. The ebXML registry is designed for trading-partner communities rather than for use by the entire Web.

The registry is designed for trading-partner communities

As with UDDI, each item stored in the ebXML registry must have a unique ID assigned to it, especially so that remote registries can unambiguously locate the item. Registry items also must have a name, a description, administrative and access status, persistence and mutability characteristics, classification according to predefined schemes, file representation type, and name of the submitting and responsible organization.

Messaging

The ebXML Messaging Service specification defines a secure, consistent, and reliable mechanism for exchanging messages among users of ebXML-compliant software. Although ebXML messaging maps by default to SOAP with Attachments, other transports can be used. SOAP with Attachments defines a mechanism by which any document can be attached to a SOAP message,

much as a Word or a PowerPoint document can be attached to an e-mail message. For ebXML, attachments also are often large XML documents, engineering drawings, or illustrations related to an XML document.

ebXML messaging maps to SOAP with Attachments

Messaging in ebXML is designed to transport payloads of any type over any protocol, sequencing payloads in complex communications and supporting comprehensive security mechanisms, such as nonrepudiation, while enforcing the rules of engagement defined in the active CPA. The messaging specification is independent of either the payload or the communication protocol used, although appendixes to the messaging specification describe mappings to SOAP with Attachments, HTTP, and SMTP.

Does Success Depend on the Registry?

As the success of UDDI depends on its use of meaningful categorization information, the success of ebXML depends on businesses' registering and storing their business scenarios in the repository. Instead of each individual business modeling its view of how it thinks a particular business goal should be achieved, ebXML advocates that standards organizations, such as UN/CEFACT, and industry user groups will create the models. These models would contain all possible activities that pertain to a particular business goal. *This is a very important point, as it assumes that many of the models stored in the registry are common to multiple businesses.*

However, the ebXML registry, like UDDI, does not constrain information placed in it to ensure uniformity. Both depend on the right thing happening naturally or on someone working it out later, which is probably a lot to expect. The ebXML specifications are supposed to provide only the building blocks and supporting infrastructure for messages, but the development of standard messages has been left to industry groups and other standards bodies, which may or may not produce them.

So, despite the general success of the ebXML initiative in terms of creating a comprehensive body of agreed-on specifications, the reality of ebXML's adoption is left largely to implementers and other standards bodies, which may or may not end up completing the job. Meanwhile, the W3C is advancing its family of Web services specifications toward many, if not all, of the same goals.

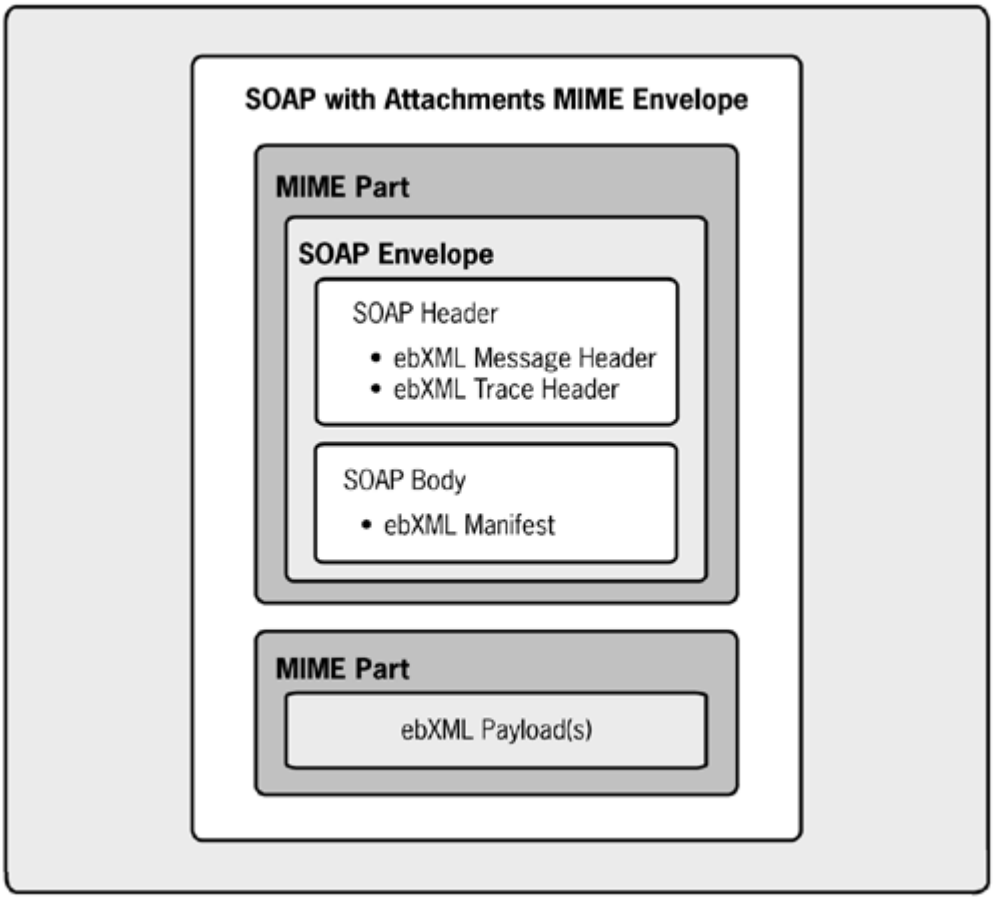
The ebXML registry is even more free-form than UDDI. Work to fill in the gaps is ongoing, but so far, the success of the ebXML registry is far from ensured. Without tighter definition of required content, reasonable and successful searches can't be performed. And without more concrete API definitions, implementations will vary. Unlike UDDI, no organization to host a central or common registry has been established. It's a classic chicken-and-egg problem—which UDDI also faces, of course—in which success depends on widespread adoption and use. The registry has to be populated to be useful and has to be proved useful to be populated.

The ebXML messaging specification adds several requirements for such qualities of service as security, guaranteed delivery, and nonrepudiation. The messaging specification also defines how communication between two ebXML-compliant parties can configure a connection using an agreed-on CPA; that is, how parties to a business transaction can start exchanging messages, given an active CPA.

The messaging specification includes additional quality-of-service requirements

Figure 6-5 illustrates the ebXML messaging specification's mapping to SOAP through the SOAP with Attachments specification. SOAP with Attachments allows a document defined in a CPA to be attached to a SOAP message, using a link to the document contained within the body of the SOAP message. The SOAP body contains a message manifest that lists all MIME parts and attachments that comprise the entire message. The message recipient can examine the manifest to determine whether all parts of the message arrived successfully. If they did not, the recipient signals an error condition, using a SOAP fault message or other application-defined message. The business data is fully contained within the attachment. The ebXML message and trace headers are mapped to SOAP headers.

Figure 6-5. The ebXML messaging specification's use of SOAP.



The SOAP body contains a manifest

Core Components

The ebXML Core Component specification identifies the data items that are most often used across industries, assigning them neutral names and unique identifiers. Core components are intended to provide interoperability among industries and business functions, but they, in fact, represent the way to define a library of XML business document parts out of which specific business documents can be assembled for individual transactions.

Core components identify common data items

Core components can relate data used in one industry to data used in another industry or from one XML vocabulary to another, such as previously defined EDI transactions. Core components, however, are still a work in progress. The EDI standards bodies are in the process of mapping their data dictionaries to the ebXML core components, for example.

Core Components capture information about a real-world business object and its relationship to other business objects and contain descriptions of how an object may be used in a particular ebXML scenario. Core components are stored in the ebXML registry and are required to contain a minimum set of information so they can be joined appropriately with other core components.

A business document used in an ebXML-compliant business transaction might therefore consist of one or more shared XML core component elements—perhaps a common ship-to address definition—and custom XML elements. Core components are therefore similar to a shared code library from which a programmer imports certain shared, reusable routines to accomplish general functions.

A business document might include one or more core components

Summary

The ebXML specifications are the result of an initial eighteen-month initiative by a large community of companies and individuals who are dedicated to defining a mechanism for Internet commerce based on XML. The specifications include an overall technical architecture, messaging and transport, registry, and business process modeling. The ebXML approach starts by defining a business process, which is modeled as a series of interactions among interested trading partners. When multiple business process definitions are available in the registry, they can be compared for compatibility. A negotiation phase that follows establishes agreement among trading partners to start executing one or more of the predefined processes.

The ebXML architecture does not define the documents to be exchanged but instead relies on the identification of the interaction patterns to drive information requirements. Many vendors are implementing ebXML, typically starting with the message transport, which is based on SOAP with Attachments.

Web services and ebXML technologies are converging in the areas of transport and registry and overlap conceptually in many areas. The ebXML specifications can be considered as a potential source of enhancements to Web services technologies that would make them usable for a broader range of business-critical applications.

Chapter 7. Web Services Architecture: Additional Technologies

After the core standards are in place for data definition, service description, message transport, and service discovery, additional technologies are needed to complete a Web services architecture that supports a broad variety of applications. Although the core standards are sufficient for many purposes, complex and critical business applications require additional features and functions.

SOAP, WSDL, and UDDI supply the basic infrastructure, like the tracks of a railroad. But a railroad also requires a system of signals, switches, and stations in order to run effectively; Web services need security, process flow, transactions, guaranteed messaging, and so on to represent a complete computing environment.

Additional technologies are needed to complete the Web services architecture

This chapter describes some of the major proposed Web services specifications and technologies aimed at completing a Web services architecture:

- **Security,** for Web services confidentiality, integrity, authentication, and authorization
- **Process flow,** for orchestrating the flow of execution across a series of Web services
- **Transactions,** for coordinating the results of multiple Web services
- **Messaging,** for configuring message paths and routing messages reliably across multiple network hops, including intermediaries

Security, process flow, transactions, and reliable messaging are major areas of work

Microsoft and IBM continue to play a major role in Web services research, drafting and publishing specifications for additional technologies, and proposing ideas for a comprehensive Web services architecture. Although the two companies agree on many of the proposals, they disagree in other cases. One of the most difficult aspects of evaluating proposals for additional technologies is ascertaining the potential adoption levels for them. As with the core standards, these additional technologies and the requisite reference architecture proposals need independent supervision and control to become widely accepted.

Microsoft and IBM often, but not always, collaborate

This chapter also describes two seminal technologies that predate Web services and that continue to serve as a source of inspiration for Web services:

- RosettaNet, a trailblazer in Web services, especially business-critical aspects of XML processing over the Internet, including content management, process flow, security, and guaranteed messaging
- XML-RPC, another trailblazer for Web services, especially in the areas of simple RPC-style interaction

RosettaNet and XML-RPC inspired Web services

The specifications described in this chapter are based mainly on extensions to SOAP and WSDL. Beyond this, however, is the need for an architecture that puts the additional services and technologies into their proper context. If you add security and transaction support to SOAP and WSDL, do you check the security information before processing the transaction? Do transaction control attributes belong in WSDL, or is it sufficient to incorporate them in SOAP headers? Can the security mechanism be used to secure the transaction context? And how does process flow relate to secure, transactional operations? Do process flows automatically initiate transactions? Furthermore, how can anyone reasonably decide which additional technologies are needed to complete the reference architecture, and which ones are out of scope or unnecessary?

A Web services reference architecture is needed

Because Web services are based on XML, they are fundamentally extensible to include other specifications and technologies. However, this flexibility introduces the problem of ensuring that the participants in a Web services interaction are using the same set of additional, compatible technologies and services.

Figure 7-1 illustrates a potential Web services architecture based on the container concept. The container is a deployment vehicle for Web services, which are described and advertised using WSDL. Within the container is a dispatcher to invoke the Web service implementations in C#, Java, or other programming language, together with configuration parameters detailing container options, such as automatic handling of security, transactions, threading, messaging, multiprotocol mapping, and metadata handling. Additional Web services technologies that provide these features and functions are available to the container, perhaps as local program libraries or as remote Web services implementations. Administration and management services are also available for configuring, monitoring, and controlling the execution environment for deployed Web services.

Figure 7-1. This sample Web services architecture is based on the container concept.

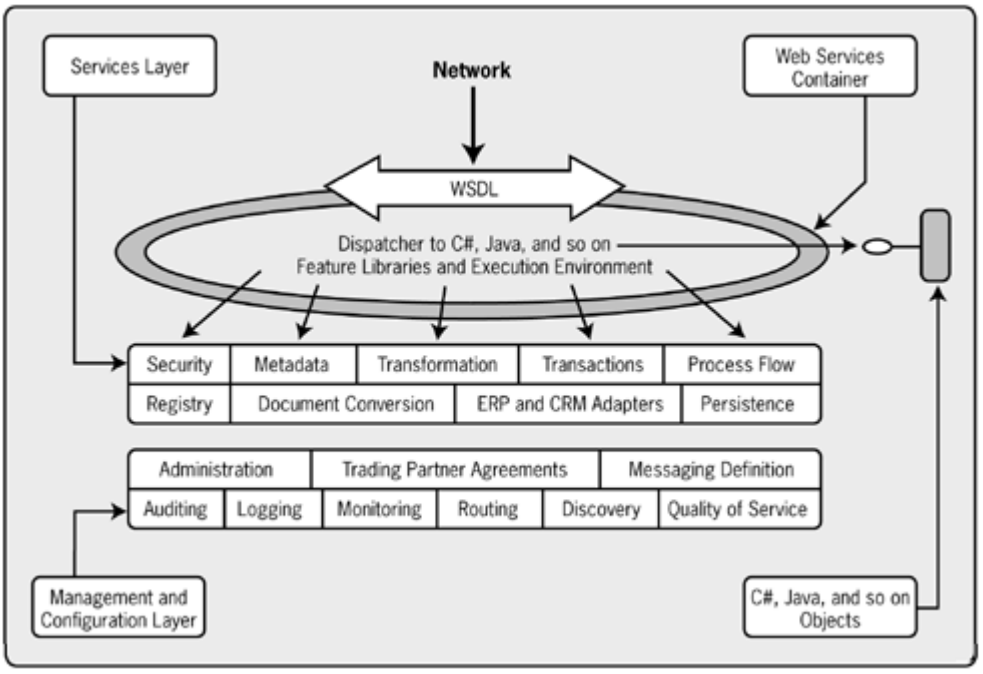


Figure 7-1 identifies the kinds of additional features and functionality that are needed to complete the Web services computing platform. The figure does not represent industry agreement on the overall model or on the specific set of features and functionality shown. It's one way, however, to gather together in a single diagram a representation of the types of additional technology proposals that are under consideration and that might end up in a comprehensive Web services architecture. Microsoft, IBM, and others have presented similar architectural proposals at a W3C forum on Web services, and the papers submitted to that forum provide a good source of ideas for completing the picture.[1] The W3C Web Services Architecture Working Group is considering these papers and other proposals as it defines such a reference architecture.

[1] W3C Workshop on Web Services. April 2001—see http://www.w3.org/2001/01/WSWS.

Security

Security is one of the most important and most complex issues confronting the Internet and Web services. Security is something you can't have enough of, but you have to draw a line somewhere, or you would never get anything done.

Security is fundamental

Basic security issues include data confidentiality and integrity—to ensure that no one steals your credit card number, for example, or tampers with the content of your message—and authentication/authorization, which deals with the rights of individuals or groups to access a certain resource, such as a given Web service interface. For confidentiality and integrity, the starting point is SSL/TLS,[2] which is implemented as HTTP over SSL (HTTPS). For authentication/authorization, the starting point is the simple checking of username/password, common to many Web sites, and proprietary mechanisms developed by such security vendors as Netegrity and Entrust. Another critical and fundamental aspect of Internet security is the use of firewalls, which have a bearing on Web services because they can be configured to inspect messages.

[2] The leading encryption technology, known as Secure Sockets Layer (SSL v2/v3)—see http://www.netscape.com/eng/ssl3/—is based on the Transport Layer Security (TLS) v1 standard from IETF (see http://www.ietf.org/ids.by.wg/tls.html).

Basic security protects data

Standardization: A Long, Slow Road

The standardization process for Web services has followed a strange route and is not guaranteed to succeed with respect to the basic Web services technologies, let alone the additional technologies needed for industrial-strength Web services deployment. Part of the reason is that the W3C acts deliberately and carefully—and therfore slowly. But, in addition, the W3C has not always succeeded in establishing its specifications in the marketplace. In particular, the HTTP-NG (next generation) specification, intended to provide much the same functionality as SOAP, fell flat, in part because of vendor competition. The original SOAP specification was developed outside of the W3C by a group of cooperating vendors that later joined with several others to submit the specification to W3C. Most of these other additional technologies have followed a

similar route. The W3C then initiated the XML Protocols Working Group, which resulted in SOAP v1.2. However, it took the working group a long time to develop the SOAP specification, and little substantive change resulted from the W3C process. True adoption of Web services, especially the additional technologies and reference architecture required to make it useful technology for business, must follow an even more difficult and challenging route, as the technologies at this level are often ones on which companies compete more intensely with one another.

XML-based security standards are still evolving, but consensus seems to be converging on a proposed standard for authentication and authorization (SAML) and a proposed standard for public key management (XKMS). Also, of course, and fundamental to all Internet security, are the firewalls that protect private networks. Firewalls map a publicly known IP address to another IP address on the internal network, thus establishing a managed tunnel and preventing unwanted access.

Additional security restricts access

A password can be encoded by using base64 for minimal protection on the wire, even without SSL/TLS, but this does not provide any encryption and is therefore not very secure. HTTPS can be used to ensure an additional level of protection. HTTPS tunnels HTTP messages over a secure network connection protected by using the SSL/TLS protocol. Browsers typically notify users when such a protected connection is being used. SSL/TLS, and therefore HTTPS, is supported by most of the popular Web services–enabled application servers, such as BEA's WebLogic, IBM's Websphere, and IONA's J2EE Edition.

Consistent Security?

Many XML security standards are being proposed and are still evolving for use with Web services. Security is a complex problem encompassing data confidentiality and integrity, authentication, and authorization, and putting all the pieces in place takes time. Meanwhile, basic encryption standards are available to secure transmissions on the wire using public keys, S/MIME, and HTTPS. Past this point, some of the security proposals start to diverge: Microsoft, for example, may go its own way with security, which could lead to difficulties realizing the end-to-end security vision for Web services when .NET platforms are included. Other vendors that focus on security exclusively, such as Netegrity and Entrust, may end up with implementations of basic authentication and authorization services available over the Web and offer alternatives to Microsoft's Passport such as Liberty.

As with any standardization effort, however, things are managable when there is a small enough number of them. For example, Society of Automotive Engineers (SAE) and metric measuring systems; these two are alternative standards for automobile parts tooling. The market is able to support both. With security, a single standard would be most effective, but two may be managable as long as conversion or mapping between them is possible.

SOAP over HTTPS with basic HTTP authentication will be the most common form of secure messaging for Web services. But this isn't sufficient because Web services are essentially exposing access to programs and data stores. Furthermore, with complex Web services, conversations may span multiple services that have been discovered dynamically or composed into a larger interaction, such as a process flow. Web services will need an end-to-end security model for the entire conversation because sensitive information could be passed from service to

service. A Web service interaction also might involve multiple parties using different security-related technologies. These requirements can be met with a combination of HTTP basic authentication and HTTPS, but a Web service may have to authenticate the request over and over again when multiple parties are involved. SAML provides a way for a Web service to reuse an assertion made by a trusted entity, although the user still has to satisfy each individual Web service's protocol. For example, if a service requires S/MIME formatting, the requester must adhere to S/MIME, even with a valid SAML assertion.

HTTPS is not sufficient

SAML

Security Assertions Markup Language (SAML), an OASIS initiative, provides a standard way to profile information in XML documents and to define user identification and authorization information. SAML implementations provide an interoperable XML-based security solution, whereby user information and corresponding authorization information can be exchanged by collaborating services. SAML defines standard XML formats for authentication and authorization information that can be propagated along a call chain, using any transport technology. SAML enables single sign-on and end-to-end security for Web services. Users can travel across sites with their entitlements so that companies and partners in a trusted relationship can deliver single sign-on across sites. Here is an example.

```
SAML Request:
< ?xml version="1.0" encoding="UTF-8"?>
< Request
  xmlns="http://www.oasis-open.org/committees/security/docs/
         draft-sstc-sch
  ema-protocol-19.xsd" xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns:saml="http://www.oasis-open.org/committees/security/docs/
         draft-sstc-schema-assertion-19.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oasis-open.org/committees/security/
         docs/draft-sstc-schema-protocol-19.xsd
         d:/platform/draft-sstc-schema-protocol-19.xsd"
  RequestID="String"
  MajorVersion="0" MinorVersion="0" >
  < AuthenticationQuery>
   <saml:Subject>
    < saml:NameIdentifier Name="admin" SecurityDomain="UserID"/>
    < saml:NameIdentifier Name="admin" SecurityDomain="Password"/>
   < /saml:Subject>
  < /AuthenticationQuery>
</Request
< Response
  xmlns="http://www.oasis-open.org/committees/security/docs/
         draft-sstc-schema-protocol-19.xsd"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns:saml="http://www.oasis-open.org/committees/security/docs/
         draft-sstc-schema-assertion-19.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oasis-open.org/committees/security/
         docs/draft-sstc-schema-protocol-19.xsd
         d:/platform/draft-sstc-schema-protocol-19.xsd"
  ResponseID="String"
  MajorVersion="0" MinorVersion="0" StatusCode="Success"
  InResponseTo="requestId">
< /Response>
```

The example illustrates the use of SAML request and response. This format can be propagated using SOAP headers to ensure that authentication information is available when a message is propagated from one point to another along a Web services call chain, so that the appropriate authorization checks can be made at each point. The response indicates a successful match for the username and password submitted in the request.

SAML propagates authentication information

XKMS

The XML Key Management Specification (XKMS) defines a protocol for distributing and registering public keys used in encrypting and decrypting messages transmitted using SOAP and other transports. XKMS is designed for use with the W3C XML Signature specification (XML-SIG), developed jointly by W3C and IETF. XKMS includes two major parts: the XML Key Information Service Specification (X-KISS) and the XML Key Registration Service Specification (X-KRSS).

XKMS mechanisms manage public key encryption

X-KISS defines a mechanism to resolve public key information contained in XML-SIG elements. X-KISS allows a client to delegate part or all of the tasks required to process key information elements to an underlying service implementation. Applications thereby can be abstracted from the underlying Public Key Infrastructure (PKI) implementation used to establish trust relationships. The underlying PKI may be based on any one of a number of specifications, including PKI for X.509 certificates (PKIX), Simple Public Key Infrastructure (SPKI), and Pretty Good Privacy (PGP).

X-KRSS defines a Web service that accepts registration of public key information. Once registered, the public key may be used with other Web services, including X-KISS.

X-KISS defines a wrapper around an underlying PKI service, whereas X-KRSS defines a Web service for registering PKI information. Both protocols are defined using XML schemas, SOAP, and WSDL. Expression of XKMS in other compatible object encoding schemes is also possible.

Trust services wrapped by X-KISS can manage private/public key pairs for digital signatures. The entire process of registering, fetching, and revoking keys can be delegated to the trust service. XKMS specifies an XML protocol to access the key management functions of a trust service. Trust service functions include all the features necessary to sign and seal documents digitally, so that only the intended recipient can view and manipulate the content.

X-KISS extends basic application server functionality

Application servers, such as BEA's WebLogic and IONA's J2EE Edition, typically store keys but do not provide management functions, such as key renewal and revocation checking. Services hosted on such a server fetch keys from a local key store and use them to sign a document, using

the syntax specified by the XML Digital Signature specification.[3] X-KISS extends the basic functionality to include key registration, fetching, and revoking.

[3] See http://www.w3.org/2000/09/xmldsig.

Digital signature helps protect confidentiality

A digital signature is produced by using a document hash function and the private key so that the receiver can independently compute the hash and compare it to the value produced via decryption using the public key. It is also possible to include the credentials in messages using syntax other than XML, over protocols other than SOAP, and through a definition language other than WSDL, although such expression is outside the scope of the specifications.

WS-License and WS-Security

The Web services license language (WS-License) and the WS-Security specifications[4] are Microsoft proposals that are designed to work together. The WS-License specification defines formats and encodings for security credentials used by WS-Security for preserving message confidentiality and integrity. A license is a special type of credential that contains a set of related assertions signed by an authority, such as a public or private key. Both X.509 certificates and Kerberos tickets are considered valid licenses.

[4] These specifications, along with WS-Inspection, WS-Referral, and WS-Routing, comprise Microsoft's Global XML Web Services Specifications (GXA), released in November 2001. See http://msdn.microsoft.com/ws/2001/10/License/ for WS-License and http://msdn.microsoft.com/ws/2001/10/Security/ for WS-Security.

The main purposes of WS-License are to define a set of commonly used license types and to specify how they can be included within the WS-Security `<credentials>` tag. WS-Security provides credential exchange, message integrity, and message confidentiality for SOAP messages using the credentials defined by WS-License. WS-Security associates licenses with messages for confidentiality. Integrity is provided by using XML Signature and licenses to ensure that messages are not tampered with. Confidentiality is ensured by combining XML encryption with licenses. WS-License and WS-Security are designed to be used with SOAP, but their relationship to other, similar security efforts, such as SAML, is not defined.

WS-License defines security token formats for WS-Security

Process Flow

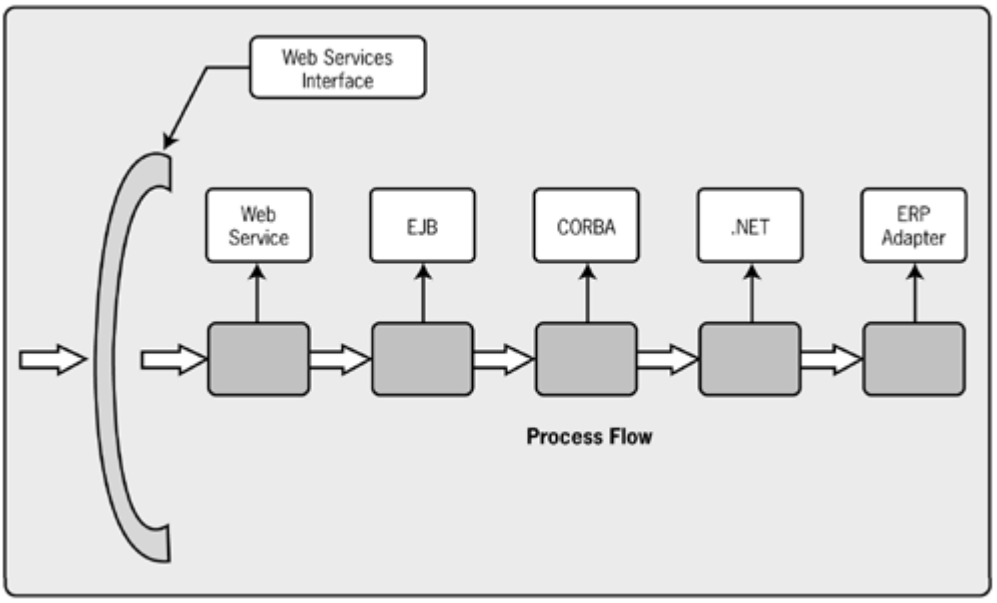
Businesses that start to interact with one another by using such Web services technologies as SOAP, WSDL, and UDDI will need to establish formal processes and procedures for handling access to information technology systems, as they already do for internal operations. Once the formal processes and procedures are in place, they can be defined as Web services, and a choreography, or business process flow definition, can be placed around them to put them in an agreed on sequence for automatic execution over the Web.

Once program-to-program communication is established using Web services, the next step is to implement formal trading-partner agreements and collaborations as process flows, or orchestrations. Two proposals for defining this sequence of messages into a business process framework are XLANG from Microsoft and Web Services Flow Language (WSFL) from IBM.

Orchestration places Web services into relationships

As shown in Figure 7-2, a process flow puts into a defined sequence tasks that invoke Web services, JavaBeans, CORBA objects, .NET classes, and ERP adapters, among others. Most process flow implementations allow callouts to any type of program for external access. Many such implementations are focused entirely on orchestrating a series of Web service invocations. In any case, a process flow definition itself can be exposed as a Web services interface such that sending a message to that interface initiates the automated flow of execution. A typical application for a process flow is to automate a business transaction, such as filling a purchase order or processing an insurance claim.

Figure 7-2. Process flow definition entails a sequence of tasks.



XLANG

The XLANG specification defines a message-centric flow language similar to RosettaNet as an extension to WSDL. RosettaNet process flow definitions originated Web-oriented business process and protocol specifications. Similarly, XLANG defines in WSDL terms a sequence of messages to implement a business process.

XLANG defines XML elements for branching and switching, exception handling, and grouping WSDL operations. The XLANG schema is intended to be processed or executed in conjunction with the WSDL it references. In other words, XLANG assumes the existence of WSDL for defining the operations it puts in sequence.

XLANG is based on BizTalk

XLANG also defines a way to carry context information in WSDL-defined messages such that information about the current instance of a business process is identified and messages can be related back to the right instance of a suspended or active process flow when the messages arrive

at a given trading partner. XLANG assumes statically configured conversations. Here is an example.

```
<?xml version="1.0"?>
<definitions name="StockQuoteProvider"
  targetNamespace="http://example.com/stockquote/provider"
  xmlns:tns="http://example.com/stockquote/provider"
  xmlns:xlang="http://schemas.microsoft.com/biztalk/xlang/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <portType name="RequestReceivePortType">
    <operation name="AskLastTradePrice">
      <input message="..."/>
    </operation>
  </portType>
  <portType name="ResponseSendPortType">
    <operation name="SendLastTradePrice">
      <output message="..."/>
    </operation>
  </portType>
  <binding name="RequestReceivePortBinding"
    type="tns:RequestReceivePortType">
    <!– details omitted –>
  </binding>
  <binding name="ResponseSendPortBinding"
    type="tns:ResponseSendPortType">
    <!– details omitted –>
  </binding>
  <service name="StockQuoteProviderService">
    <port                                  name="pGetRequest"
binding="tns:RequestReceivePortBinding">
      <soap:address location="mailto:quote@example1.com"/>
    </port>
    <port                                name="pSendResponse"
binding="tns:ResponseSendPortBinding">
      <soap:address
location="mailto:response@example2.com"/>
    </port>
    <xlang:behavior>
      <xlang:body>
        <xlang:sequence>
          <xlang:action operation="AskLastTradePrice"
            port="pGetRequest" activation="true"/>
            <xlang:action operation="SendLastTradePrice"
              port="pSendResponse"/>
            </xlang:sequence>
          </xlang:body>
        </xlang:behavior>
  </service>
</definitions>
```

As shown in the example, XLANG adds elements into the WSDL file to put previously defined port operations into a sequence. This is a simple example of a stock trade service, extracted from the specification, in which the `AskLastTradePrice` operation defined for the

`pGetRequest` port is defined as part of a sequence with the `SendLastTradePrice` operation defined for the `pSendResponse` port. In this example, the ports are implemented using e-mail.

WSFL

The Web Services Flow Language (WSFL) specification[5] is an IBM proposal that describes process flows in terms of two types of Web service *compositions:*

[5] *See* http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

- Internally focused, or Web service aggregation flows, putting multiple internal Web services into relationship
- Externally focused, or flows between Web services exchanging messages between trading partners

In the first case, WSFL defines how Web services can be aggregated to implement a larger function, such as an internal business process for preparing a purchase order. For this type of flow model, WSFL defines the control and data flow between the associated Web services almost as if they were subroutines called from the same main program. This type of flow is also sometimes called a *micro*-process flow.

In the second case, WSFL describes how a purchase order might be sent from one company's Web service to another. For this type of flow model, no execution sequence of Web services is specified. Rather, the interaction between Web services is identified in the flow and is modeled as a link between end points, almost as if they were adapters in an EAI type of integration flow. This type of flow is also sometimes called a *macro*-process flow.

WSFL is derived from IBM's MQ Series product line

WSFL supports recursive composition of Web services, meaning that the flows can reference another flow or a specific end point. In contrast to XLANG, the WSFL defines its flows in a separate XML document rather than as extensions to WSDL.

WSFL is layered on top of WSDL, which WSFL uses to describe the end points for business operations, the description of service interfaces, and their protocol bindings. Here is an example.

```
<flowModel                              name="totalSupplyFlow"
serviceProviderType="totalSupply">
  <serviceProvider name="mySupplier" type="supplier">
    <locator type="static" service="qualitySupply.com"/>
  </serviceProvider>
  <serviceProvider name="myShipper" type="shipper">
    <locator type="static" service="worldShipper.com"/>
  </serviceProvider>
  <activity name="processPO">
    <performedBy serviceProvider="mySupplier"/>
    <implement>
      <export>
        <target                      portType="totalSupplyPT"
operation="sendProcOrder"/>
```

```
      </export>
    </implement>
  </activity>
  <controlLink                                source="processPO"
target="acceptShipmentRequest"/>
    <dataLink                                 source="processPO"
target="acceptShipmentRequest">
      <map sourceMessage="anINVandSR" targetMessage="anSR"/>
    </dataLink>
</flowModel>
```

XLANG versus WSFL

IBM and Microsoft manage to agree on many Web services specifications but not on the process flow language. This may stem from the different basis for the proposals in their respective products, which are not compatible. From a purely Web services viewpoint, however, XLANG seems more natural, as it bears a relationship to RosettaNet, one of the major influences on Web services specifications, and is simpler, easier to implement, and more naturally an extension of WSDL. But perhaps the real solution will have to come from a completely new and independent source.

This WSFL example depends on an associated WSDL file containing the port type and operation definitions referred to as `supplier` and `shipper`. The `totalSupplyFlow` model specifies collaboration with two service providers offering their joint customers a complete business process. Each service provider is represented using a separate `<serviceProvider>` element. One service provider is of type `supplier` and is referred to as `mySupplier`. The other service provider is of type `shipper` and is called `myShipper`. The business process represented by the `totalSupplyFlow` flow model consists of three business tasks, called activities, that have to be performed in the specified order to complete the business process successfully. A purchase order has to be processed, a shipment request must be accepted, and money has to be received.

Transaction Coordination

When multiple Web services are put into relationship—for example, in a long-running automated business process flow—results of related operations may need to be coordinated to ensure that the desired outcome was achieved. A classic example of this sort of problem is the travel reservation, in which multiple related operations are executed to complete an itinerary: hotel, flight, and rental car reservations. If one of these operations fails, the others should not be completed, as travel is not practical without all the reservations being made.

In established distributed computing technologies, such as CORBA, DCOM, and EJB, transaction coordination services provide an all-or-nothing protocol called two-phase commit, in which all operations on data either succeed or fail as a unit.[6] Software-based transaction coordination ensures accurate recording for business operations involving multiple operations on data, despite hardware or software failures. However, remote execution of the existing transaction coordination protocol depends on the existence of a connection-oriented transport, which is not the case for Web services, at least not until or unless a reliable, session-oriented transport is defined and used in place of HTTP.

[6] For a good introduction to transaction processing concepts, see *Principles of Transaction Processing* (Bernstein and Newcomer, 1997).

Transactions ensure coordinated results

Various suggestions and proposals have been considered to address this key problem. One is to use a connection-oriented messaging protocol, such as HTTPR or BEEP, when transaction coordination is required, and then to use a form of traditional two-phase commit, such as the Transaction Internet Protocol (TIP).[7] Another is to define an alternative to the traditional two-phase commit protocol for use over current, disconnected Web transports.

[7] See http://www.ietf.org/rfc/rfc2371.txt?number=2371.

BTP

The Business Transaction Protocol (BTP)[8] specification defines an XML protocol for placing long-running business-to-business (B2B) and other similar transactions into a choreographed sequence. BTP is based on a multilevel transaction model that provides independence from coordinating the underlying resource managers. In other words, BTP is an alternative to traditional two-phase commit.

[8] See www.oasis-open.org/committees/business-transactions/index.shtml.

BTP is designed to reliably manage the results—success or failure—of complex B2B transactions and to propagate those results to all applications involved in managing the underlying resources: database systems, files, queues, and so on. Unlike traditional transaction management systems, BTP does not coordinate or drive actions in response to the results and does not coordinate recovery after failure. In the BTP model, interpreting the results, including recovery from failure, remains the responsibility of the participants. BTP does not specify the business protocol governing the business transaction but simply provides facilities and semantics for a reliable termination mechanism to achieve a shared agreement on the outcome of a business transaction.

BTP provides an alternative to traditional two-phase commit

Alone, BTP cannot provide the attributes of an ACID transaction: atomicity, consistency, isolation, durability. Systems using this protocol must decide separately and individually how to manage their local resources according to the propagated results. For example, on termination owing to failure, systems might execute an appropriate compensating action.

BTP messages can be transmitted using any B2B protocol, including SOAP and ebXML. A header can be added to the ebXML message envelope, or a SOAP header can be defined to carry the required transaction propagation context information. Examples of BTP system messages include *startTransaction* or *terminateTransaction* and can be sent using standard ebXML or SOAP messages.

A J2EE, .NET, OMG Object Transaction Service (OTS), or native platform transaction coordinator can participate in a long-running transaction as long as a mapping is created between the BTP messages and the underlying transaction coordinator. Of course, the mapping could also be performed directly onto the resource manager. However, BTP also introduces the notion of a compensating transaction that can be used by each participating resource manager to compensate for the changes done during the transactions if the outcome is a rollback.

Web Services Can't Use Traditional TP Technology

BTP evolved because conventional ACID transactions are not well suited to Web services and other B2B protocols. B2B transactions tend to be long running and

asynchronous and to use XML protocols over HTTP or SMTP, which are not connection-oriented protocols. Traditional two-phase commit protocols assume the use of connection-oriented protocols, and the loss of a connection is defined as a rollback. Clearly, in the often disconnected world of the Internet, triggering a rollback on the loss of a connection is not possible. If the Web forced a rollback every time a connection were lost, work would rarely get committed. And if database locks were held open while waiting for two-phase commit messages over the Internet, blocking would be a serious problem.

Despite many efforts to solve the problem, nothing has stuck. Even BTP, which has a design and architecture appropriate to the Web, may well not succeed for lack of widespread support. Gaining consensus on TP technology is much more difficult than you might think.

Extended Transactions

Work at the Object Management Group (OMG) and in the Java Community Process defined an extended transaction model suitable for long-running transactions and intended for use in transaction protocols other than the established two-phase commit protocol. An example in the OMG specification describes an open nested transaction model that allows resources to commit independently, much as participants in a BTP transaction are allowed to do. The OMG extended transaction model is based on a generic coordination state machine, however, which is capable of driving any protocol, including BTP and traditional two-phase commit. A generic coordination mechanism such as this holds potential as a Web services transaction control mechanism, also.

The OMG extended transaction model may apply to Web services interactions

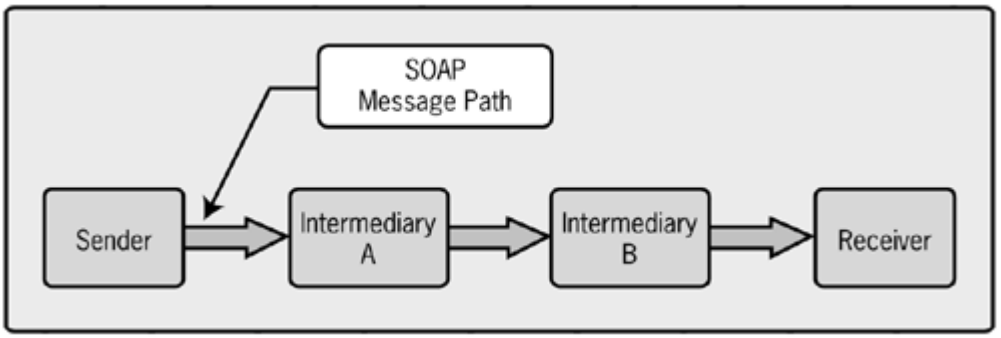
Messaging

Many proposals for additional Web services technologies center on the messaging model, especially because qualities of service can be associated with the underlying transport. In traditional distributed computing environments, additional technologies, such as security and transactions, are often directly related to the transport layer. Because Web services messages are fundamentally one-way asynchronous messages that are potentially transmitted over multihop Internet networks, including intermediaries that can inspect and process SOAP headers, additional technologies in this area focus on improving reliability and ensuring that the multihop messages are routed correctly.

Messaging proposals define routing and reliability

Figure 7-3 illustrates a simple example of a SOAP message path that includes an originator of a message and two intermediaries, which process part of the message headers enroute to the ultimate receiver of the message. This simple example shows that a SOAP message can be routed via multiple network hops in order to reach its ultimate destination. There-fore a mechanism is needed to clearly identify the entire route, to ensure successful transmission of the message along the entire route, and to report failure or fault information reliably to the original sender.

Figure 7-3. This sample SOAP message path includes intermediaries.



Routing Information in the Messages or External to Them?

In general, information about the destination of a message and the path it follows between sender and receiver, including any intermediary hops, can be contained within the message or described in associated routing tables. When the information is contained within the message, the problem arises of changing the contents of the message whenever the routing path for the message changes. When the information is maintained within an independent table, environment variable names or in URIs that associate the information with the message, the routing information can be changed independently of the message contents. Established distributed computing environments, such as CORBA and DCOM, support the external translation of destination names to addresses and model intermediaries as forwarding agents transparent to the sender and receiver. Proposals for adding capabilities to the Web services messaging layer tend to focus on incorporating additional information into the message itself and trying to model or define the roles of all network hops in the message path. However, external definition provides the application with a consistent, uniform view of a sender/receiver role, as the existence of intermediaries is transparent to the message. Once the existence of intermediaries is exposed to the application layer of the protocol—or in this case, exposed to the message definition—specifications need to define the behavior in sufficient detail to ensure that all implementations are consistent. This applies to both normal and error cases. Thus Web services messaging extensions continue to struggle with the concrete definitions of the roles of sender, receiver, and intermediary.

WS-Inspection

The Web services inspection language (WS-Inspection) is an IBM/Microsoft proposal that defines a way to represent and to retrieve a list of Web services published at a particular address.[9] That is, WS-Inspection is designed to be used when the sender already knows the receiver's address and wants to discover what services the sender offers and obtain the service description formats. WS-Inspection provides a more focused discovery and download capability for Web services metadata than do the UDDI and ebXML registries, which require a certain amount of browsing in order to obtain the information. Furthermore, WS-Inspection operations are focused on a single target server.

[9] See http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html or http://msdn.microsoft.com/ws/2001/10/Inspection.

WS-Inspection retrieves Web services descriptions from a known address

WS-Referral

WS-Referral provides a dynamic definition of the SOAP nodes in a message path, including intermediaries. The WS-Referral specification works in conjunction with the attributes of SOAP headers to ensure that the requisite nodes are configured along the message path to perform the roles identified for them. In other words, the SOAP attributes identify which nodes are responsible for processing which headers. The WS-Referral specification provides a way to define those nodes within the message path and to delegate part or all of the responsibility for processing a given SOAP message to one or more of them.

WS-Referral defines intermediate nodes that SOAP messages require

WS-Referral uses SOAP messages and headers to exchange routing entries among message path nodes and can change routes independently of the HTTP or other underlying transport routing tables. WS-Referral messages can establish and modify a routing path for a message. SOAP routers in such a configuration can be set up to provide specialized application services, such as load balancing, mirroring, caching, and authentication.

WS-Routing

WS-Routing defines a complete message path for routing SOAP messages across a multihop network environment. The message path is defined within the SOAP envelope. WS-Routing supports message interaction patterns for one-way, request/response, and peer-to-peer conversations. Although it defines the roles of the original sender, the intermediary, and the ultimate receiver, the SOAP specification does not provide any means to specify an ordered path or route from one to another of these types of SOAP nodes. Multiple instances of SOAP nodes in each role are possible.

WS-Routing defines message path hops

Delegate or Obfuscate?

The notion of delegating SOAP message header processing to specific nodes that are configured to provide specific qualities of service for a message is intriguing, but the idea also tends toward confusion. The interest of an application or a user in the contents of a message is quite distinct from the interest of the same application or user in the qualities of service applied to the contents of the message. Therefore it's important to ask whether splitting a message for processing at multiple nodes along a message path is helpful or harmful to the keep-it-simple tenant of original Web services specifications.

WS-Routing defines a SOAP header that contains elements identifying

- The message originator
- The ultimate receiver
- The next hop on the path to which to forward the message
- An identification of the reverse message path along which the response, if any, can be sent

The forward and reverse elements are redefined at each hop, eliminating the requirement for any node to know the complete message path, although it might be contained within a WS-Referral element. At each hop, the path back to the sender is dynamically refreshed, so that a message can always retrace its steps through any number of intermediaries back to the original sender.

BEEP

The Blocks Extensible Exchange Protocol (BEEP) is an IETF proposal for a generic connection-oriented asynchronous Internet protocol. BEEP permits simultaneous and independent exchanges of messages between peers that have established a connection. The connection is defined in terms of a *channel*, which is a binding to an application providing a service, such as transport, security, transaction management, or data exchange. BEEP is comparable to a conversational style of communication protocol, such as those provided by LU6.2 and Tuxedo.

BEEP supports both text and binary messages and can be used to help address and resolve application-level issues resulting from the connectionless nature of HTTP. In other words, certain qualities of service in Web services and B2B message exchanges can rely on the features of the BEEP communications protocol instead of being dealt with by the application. For example, BEEP connections could be used to help ensure that all parties to a business transaction reliably receive notification of the transaction outcome; with HTTP, by contrast, this level of reliability has to be built within the application.

BEEP is a connection- oriented Internet protocol proposal

BEEP defines a *framing* mechanism for exchanging arbitrary MIME and XML messages between peers. Profiles are defined for the channels to help accomplish the communication transport goals.

BEEP sessions are mapped to an underlying transport, typically TCP. Once a session is established, each peer advertises the profiles it supports. When a channel is created between peers, the client and the server negotiate agreement on one or more mutually compatible profiles. Channels then initialize the session and maintain the negotiated profiles and the session characteristics for the duration of the connection. Data exchange profiles are usually created after the initial tuning channels have completed their negotiations. Only one tuning channel can be active at the same time, but multiple data exchange channels can be active at the same time.

When a BEEP session is established, each peer is given either a listening or an initiating role. Typically, the BEEP peer that initiates the exchange is called the client, and the listening peer is called the server. But because BEEP is peer to peer, this relationship is not required.

BEEP provides a peer-to-peer communication model

BEEP messages are grouped into three styles of exchanges:

- **MSG/RPY,** in which a client sends a message to the server and the server returns a reply after performing a task
- **MSG/ERR,** in which the client sends a message but the server does not perform any task and replies with an error message

- **MSG/ANS,** in which the client sends a message and the server performs a task during the course of which it returns zero or more answer messages followed by a null termination message

BEEP supports multiple message interaction styles

MSG/RPY and MSG/ERR are called one-to-one positive and negative exchanges, respectively; MSG/ANS supports one-to-many exchanges.

Messages are usually sent in a single frame but can be split across multiple frames if necessary or convenient, that is, if only part of the message is ready. A frame comprises a header, a payload, and a trailer. The header and the trailer are defined using ASCII characters, but the payload can be anything.

The following example illustrates a BEEP message contained in a single frame that consists of a header, a trailer, and a payload of 120 octets:

```
C: MSG 0 1 . 52 120
C: Content-Type: application/beep+xml
C:
C: <start number='1'>
C:   <profile uri='http://iana.org/beep/SASL/OTP' />
C: </start>
C: END
```

A BEEP binding to SOAP that shows how SOAP envelopes are transmitted using a BEEP profile has been defined.[11]

[11] See http://clipcode.org/beep/soap.

Reliable HTTP

Reliable HTTP, or HTTPR,[12] is an IBM proposal that layers a conversational protocol on HTTP request/response interactions. HTTPR defines commands to `push`, `pull`, and `exchange`—a combination of `push` and `pull`—SOAP messages reliably between sender and receiver pairs. HTTPR, like many of the Web services extension proposals, works by inserting information in SOAP headers, including a correlation ID for matching requests with replies. The reliable part of HTTPR persists information about the message and correlates acknowledgments and replies with the original requests. Each party in the message path persistently stores information about the message along with its unique correlation IDs so that it's possible to trace the flow of a message and to discover what happened in the event of failure. Similarly, persistence information can be used to retransmit a message if the first attempt fails. Push and pull commands can be batched if the application wants to transmit bulk data.

[12] See http://www-106.ibm.com/developerworks/webservices/library/ws-phtt/httprspecV2.pdf.

HTTPR defines how metadata and application messages are encapsulated within the payload of HTTP requests and responses, such as those used for SOAP messages. HTTPR can also ensure that each message is delivered to its destination application exactly once or is reliably reported as undeliverable. HTTPR specifies the commands, the state information that needs to be stored safely, and when to store the state information in order to ensure reliable message delivery.

Reliable HTTP defines persistence for SOAP message paths

HTTPR requires HTTP v1.1. A sender initiates an HTTPR interaction by generating an HTTP `POST` request that includes an HTTPR command and any associated messages. The receiver replies using an HTTP response that includes status information about the message received and, optionally, a batch of messages intended for that sender. The sender assigns each message interaction an identifier, which is placed in stable storage along with information about the current state of message processing. In the event of a failure, this state information is recovered from stable storage and used to ensure qualities of service, such as exactly-once delivery, reliable transfer following interruptions and failures, or preserving the order of messages in a multihop communication path. Additional commands allow senders to inquire on the status of a message, given the unique ID, and to report on the result of processing a message with a given ID.

Web Services Foundations

RosettaNet and XML-RPC are technologies that predate and significantly influenced Web services development. RosettaNet specifications and technologies also provided input to ebXML and continue to serve as a source of inspiration for Web services proposals. It may help to understand a bit about these technologies to appreciate the design center for Web services specifications, the focus on keeping SOAP simple, and what kinds of additional technologies make sense to add to a simple transport layer.

XML-RPC predates SOAP, and it was used as a primary source for the original SOAP specification. In general, SOAP can be viewed as emerging from a combination of the concepts behind these two original XML protocols: one as the source of the RPC-style interaction and the other as the inspiration for the document-oriented style of interaction. Both technologies are in significant use today, which is likely to continue until Web services mature and supersede them.

XML-RPC pioneered the RPC-oriented Web service style

RosettaNet

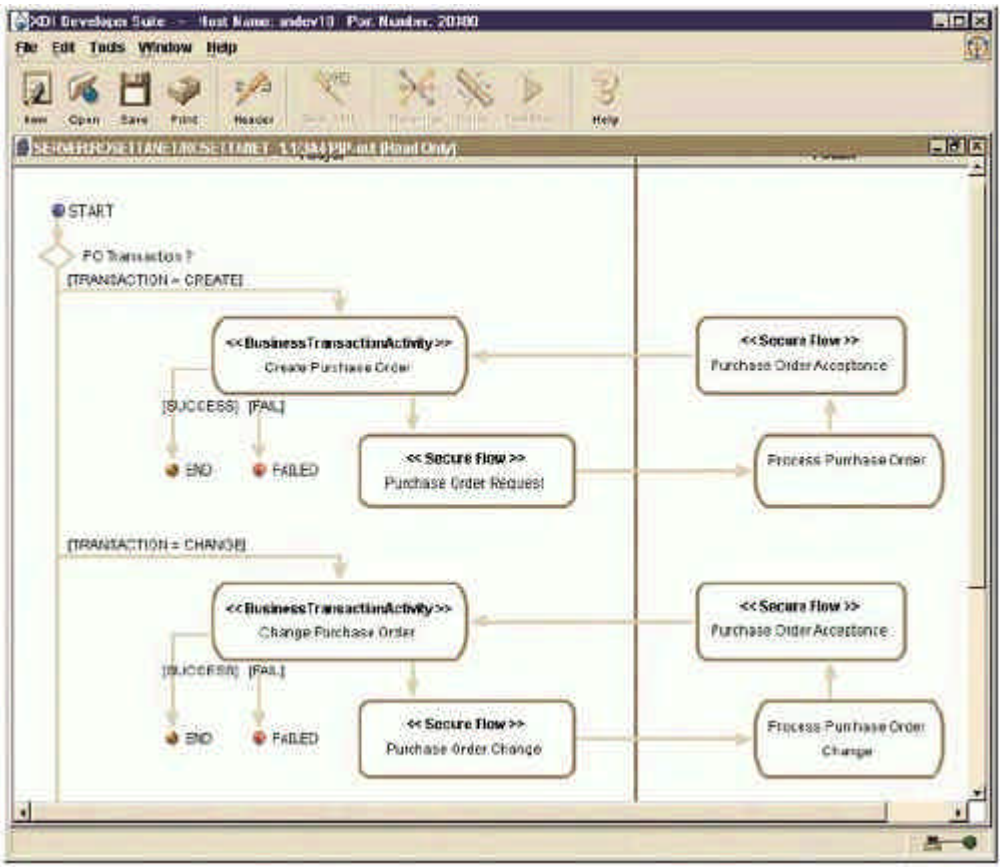
The RosettaNet consortium was launched in June 1998, primarily as an organized response in the electronics industry to generalize the success of such companies as Dell Computer, which had begun efficiently managing its supply chain by using the Internet. Personal computer vendors are typical of electronics companies that assemble finished consumer and industrial products using standard parts from a variety of suppliers. Closely tracking the availability of parts for specific products provides the best opportunity for just-in-time manufacturing on demand, as Dell has done so successfully.

RosettaNet is independent, self-funded, and nonprofit. Its members comprise information technology, electronic components, and semiconductor manufacturing vendors. RosettaNet's goal is to develop and to deploy electronic commerce standards for business process automation among partners in the high-tech supply chain. RosettaNet pioneered XML-based standards for B2B electronic commerce, and its suite of specifications, technologies, and XML documents encompass the entire high-tech supply chain. RosettaNet specifies common business processes using Partner Interface Processes (PIPs) and an XML protocol for networked application access: the RosettaNet Implementation Framework (RNIF).

RosettaNet pioneered the document-oriented Web service style

As shown in Figure 7-4, which was captured from IONA's Orbix E2A Collaborate Developer Suite, a RosettaNet PIP defines a business transaction flow, such as sending a purchase order request and receiving an acknowledgment or sending a purchase order change request and receiving an acceptance of the change. The rule toward the middle of the diagram separates the flow between the purchaser and the supplier.

Figure 7-4. A RosettaNet PIP defines a business transaction flow, such as this purchase order transaction.



The PIPs define the document types, such as specific purchase order forms, acknowledgment and error report forms, invoicing and shipping forms, and so on, as well as the business processes required to complete the PIP. PIPs are completed by ending the chain of document messages, or choreography of messages, defined for the PIP. RosettaNet implementations typically provide graphical displays of PIP interaction flows. PIP implementations are interoperable: As long as each vendor in the interaction conforms to RosettaNet, the trading partners can exchange documents according to the steps defined for the agreed-on PIP.

This exchange of documents among trading partners is also called business process choreography. Examples of business process choreography supported by PIPs include purchase order management, distribution of new-product information, and invoice management. A PIP specifies not only the XML business document structure and content format but also the time, security, authentication, and performance qualities of service for a given type of document-oriented interaction.

PIPs represent publicly visible document interactions but do not define how the business processes and documents are mapped to internal IT systems or how the XML documents are mapped to native data formats. However, PIP processing requirements assume the existence of such internal, or *private,* processes behind the public ones.

RNIF defines the XML protocol envelope that carries within its payload all document exchanges defined for a given PIP. RNIF's envelope format is independent of the specific transfer protocol used to transmit the message between partner nodes, although most often it's implemented over HTTP and MIME. RNIF messages also include a header that carries information about the PIP driving the message interaction sequence and that contains information about the specific step within that sequence that's active or next. In this way, RNIF messages are self-describing and provide information necessary for implementers to maintain the proper order of steps in an orchestration defined for a given PIP. RNIF features security mechanisms to digitally sign and/or encrypt all RosettaNet messages and a reliable messaging mechanism based on acknowledgments to ensure that messages are sent and received securely and surely.

RNIF is an XML protocol for RosettaNet documents

Like SOAP with Attachments, RNIF v2 specifies the use of MIME multipart/related types for the basic enveloping construct and the S/MIME multipart/signed and the application/pkscs7-mime enveloped-data types for digitally signing and content enveloping purposes, respectively. The similarity of RNIF and SOAP with Attachments allowed RosettaNet to align with ebXML as of RNIF v3.

XML-RPC

XML-RPC was developed by Dave Winer and others at UserLand,[13] an Internet publishing company looking for a more efficient way of transferring XML documents across the Web and developing interactive Web sites. XML-RPC is a very simple and straightforward definition of remote procedure call semantics in XML. As with SOAP, an XML-RPC message is carried as an HTTP `POST` request, and the response is carried as a normal HTTP response. The XML-RPC syntax represents a simple mapping to the RPC method and argument name format and is relatively uncomplicated compared to SOAP, without headers, complex serialization formats, or abstractions suitable for document-oriented interaction. The XML-RPC specification simply requires that the XML message carried in the HTTP `POST` be mapped onto a procedure execution at the target host and that the remote procedure be executed; the XML message then returns its results on the HTTP response message. Procedure arguments can be formatted using simple data types, such as scalars, numbers, strings, and dates or can use complex record and list structures. For example:

[13] See *Programming Web Services with XML-RPC* (St. Laurent, Dumbill, and Johnston, 2001).

XML-RPC: More Real Than SOAP?

XML-RPC is much simpler than SOAP and, without WSDL and associated specifications, much easier to understand. More than 30 implementations of XML-RPC are in use. It's a bit like the RosettaNet versus ebXML issue; one is real, implemented, and used within its world, but the other holds greater promise.

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
```

```
Content-Type: text/xml
Content-length: 181
<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
         </param>
      </params>
   </methodCall>
```

The URI in the HTTP header shown in the example includes `/RPC2`, which can be used to tell the server to route the request to the RPC2 responder, thereby bypassing the Web server, or going straight to the RPC mapper. The message payload, which is a simple XML document, is contained using a top-level `<methodCall>` element. The `<methodCall>` element contains a simple `<methodName>` element to identify the procedure to call and a `<params>` element to contain the data for the procedure arguments.

XML-RPC messages map to methods

XML-RPC does not define how to map the `<methodName>` to an actual method or procedure; that's up to the implementation. As long as an implementation is capable of accepting an XML-RPC message, mapping it to a procedure or method, and returning the result correctly formatted as an XML-RPC response, the remainder of the details can be implementation specific.

The `methodName` could be the name of a file containing a script that executes on an incoming request, the name of a cell in a database table, or a path to a file contained within a hierarchy of folders and files. For example:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT
<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><string>South Dakota</string></value>
         </param>
      </params>
   </methodResponse>
```

The example illustrates the response to an XML-RPC message. As in SOAP, the HTTP `200 OK` in the response message indicates success.

If any kind of error occurred in executing the procedure or handling the message, the response must contain a `<fault>` element to identify the problem. The body of a response can contain either a `<params>` element or the `<fault>` element but not both. For example:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 426
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:02 GMT
Server: UserLand Frontier/5.1.2-WinNT
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
           <name>faultCode</name>
           <value><int>4</int></value>
        </member>
        <member>
           <name>faultString</name>
           <value><string>
             Too many parameters.
           </string></value>
         </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

The example illustrates a possible fault message response. This type of error reflects an application-level error rather than a protocol-level error, which would be returned as an HTTP error 400 or 500. Note the use of embedded XML elements for a data type definintion.

Summary

Many additional technologies are needed to complement the core Web services specifications for use in complex and demanding business applications. To be viable, Web services specifications need security, process flow, transactions, messaging improvements, and other qualities of service. A reference architecture for additional Web services technologies is needed to guide the development of these additional technologies with respect to clearly defining what's needed and what isn't and how additional technologies interact with one another and with the core specifications.

Efforts are under way to define security standards to protect document confidentiality and integrity and to create mechanisms to control who is allowed to access which Web service interfaces. Sample specifications and proposals are available on the Internet for security technologies, reference architectures, process flow, transaction coordination, message routing, and reliable messaging. A combination of these and perhaps other, as yet undefined, specifications will emerge to provide the next layer of functionality for Web services technologies, making them suitable for more and more types of mission-critical applications. Some sense of where Web services came from and where they are going can be gleaned from studying RosettaNet and XML-RPC, two widely used technologies that predate Web services.

Chapter 8. Implementing Web Services

To be useful, SOAP, WSDL, UDDI, and ebXML need to be widely implemented and available in software products. Only then can the goal of achieving interoperability and integrating disparate software systems be attained. Many software vendors have already begun to implement these and other key Web services specifications, and many more of the vendors have pledged to provide more complete implementations in the future.

Web services technologies need wide adoption to be useful

Implementation efforts tend to fall into two major categories, leaving Web services developers and deployers with a choice. In the software industry, one large community revolves around the Microsoft environment, and another revolves around the Java environment. Both environments provide support for Web services development and deployment, and therefore developers and Web services users in general have a choice to make. Of course, because Web services are all about interoperability and integration, these two environments are compatible, at least to the extent that they both implement the same specifications and interpret them the same way.

Developers have a *choice*

Microsoft is focusing its implementation within its .NET initiative, whereas Sun Mircrosystems and other Java vendors are focusing their implementations within the Java 2 Platform, Enterprise Edition (J2EE) initiative. As the leader of the Java initiative, Sun Microsystems has published its Open Network Environment architecture (SunONE) as a blueprint for the evolution of Web services. Microsoft has defined its Global XML Architecture for extending Web services support in the .NET Framework and in general through standardization. Now that the core technologies have been established as widely adopted specifications, the next step is to define a comprehensive architecture for additional technologies, such as security, process flow, reliable messaging, and transactions. The W3C Web Services Architecture Working Group is doing just that (see www.w3.org/2002/ws/arch).

Both .NET and J2EE support Web services

Because Web services solve enterprise integration problems, integration broker vendors support Web services via adapters and integration components, or building blocks. Database vendors support Web services interfaces directly to database management systems. Enterprise resource planning (ERP) and customer resource management (CRM) packaged applications support Web services interfaces for integration with other packages and software products. Finally, a number of vendors are producing products aimed purely at implementing a Web services layer.

Integration brokers, packaged applications, and database products also support Web services

The main categories of Web services implementation activities therefore are

- **Microsoft's .NET Framework:** Windows is well established as the most popular desktop environment, and many developers use Microsoft tools for both client-and server-side applications. Microsoft is adding Web services support throughout its .NET platform, in

which every program is potentially a Web service, and all transports can use XML protocols.

- **Application servers:** The J2EE community, comprised of application server vendors, is adding Web services application programming interfaces (APIs) to Java servlets, classes, and beans, making XML integration and Web services part of the core application server definition.
- **Integration brokers:** Vendors in this middleware market segment are using Web services for enterprise application and business-to-business (B2B) integration, bridging Web services applications inside and outside the firewall.
- **Database vendors:** Vendors in this group are focusing on the use of Web services to provide a means of accessing database tables and stored procedures.
- **ERP, CRM, and others:** These packaged application vendors are adding Web services interfaces for integrating those packages with other packages and software systems.
- **Web services platform:** Vendors in this market segment are developing and supplying Web services infrastructure products as independent, or self-contained, products.

Web services implementations fall into six categories

Some vendors offer products in more than one category. IONA, for example, provides a Web services platform product, an application server product with Web services support, and an integration broker product with Web services support. If nothing else, the diversity of implementation highlights the flexibility and extensibility of Web services. Web services are defined at a sufficiently high level of abstraction to allow multiple applications for a broad variety of products. Vendors, whether focused on .NET, J2EE, or integration broker middleware, however, tend to follow a number of basic architectures for Web services implementation.

Vendors offer products, in multiple categories

Implementation Architectures

Implementation architectures vary according to the degree to which the Web services layer is apportioned value in the overall solution. In providing a Web services interface to a database management system, for example, the primary value of the solution remains apportioned within the database layer rather than in the Web services layer. The Web services layer becomes one of many options for interacting with the data in the database. On the other hand, the Microsoft My Services initiative apportions significant value to the ability of multiple Web services to interact in combination and to create applications differently or more quickly by using them.

Implementations vary in the value assigned to the Web services layer

Because Web services are not executable, much of the value in the development environment, such as J2EE and the .NET Framework, remains within the programming languages beneath the Web services. Web services represent another means of exchanging information with the application server, which still performs the main job of application development and integration. A certain value exists in the Web services layer itself, primarily for integrating, or orchestrating, multiple Web services components, which are implemented using a variety of languages, middleware systems, database management systems, and packaged applications.

Web services map messages to server objects

Figure 8-1 illustrates the Web services implementation architecture in which .NET and J2EE application servers expose programs and classes as Web service components. The programs and classes represent a variety of back-end technologies accessed using the application server, including Enterprise JavaBeans, .NET classes, message queues, and CORBA objects, to name a few.

Figure 8-1. An application server exposes back-end technologies using a Web services interface.

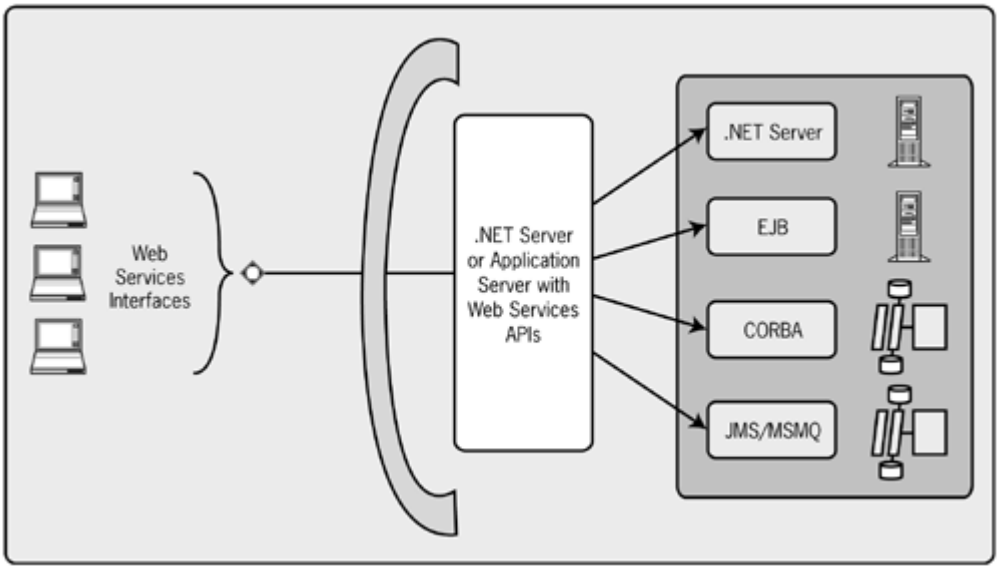
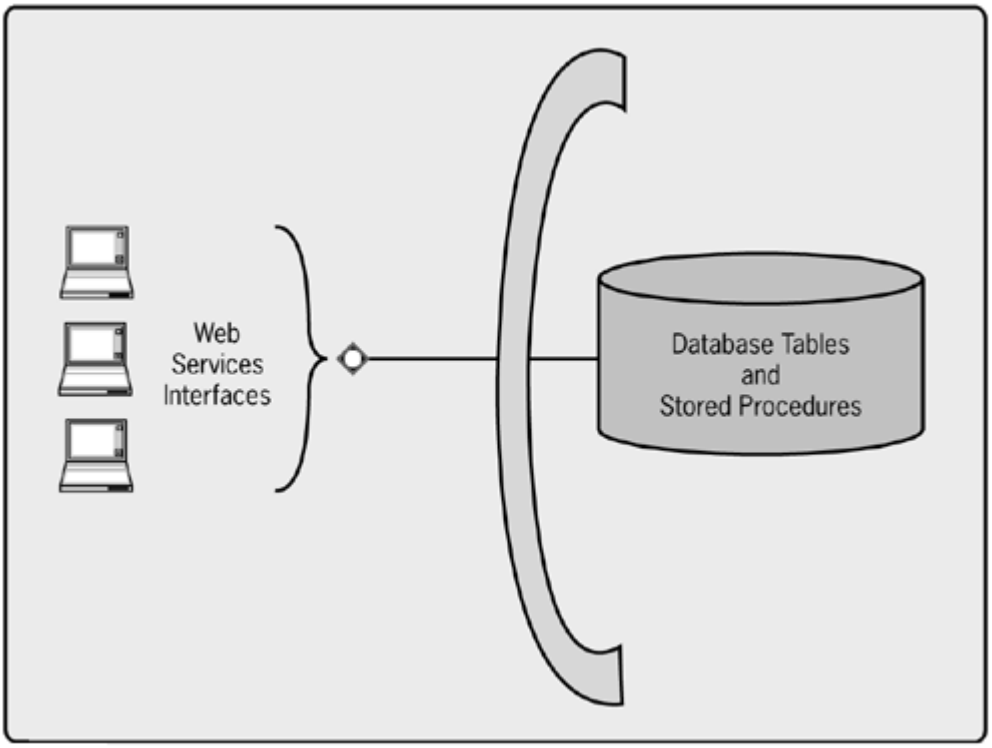


Figure 8-2 illustrates the Web services implementation architecture in which a Web services interface is used to access a database table or a stored procedure. This type of access can be thought of as a two-tier architecture, unlike the application server architecture, which includes a middle tier for business logic that is developed and managed independently of the back tier for data access.

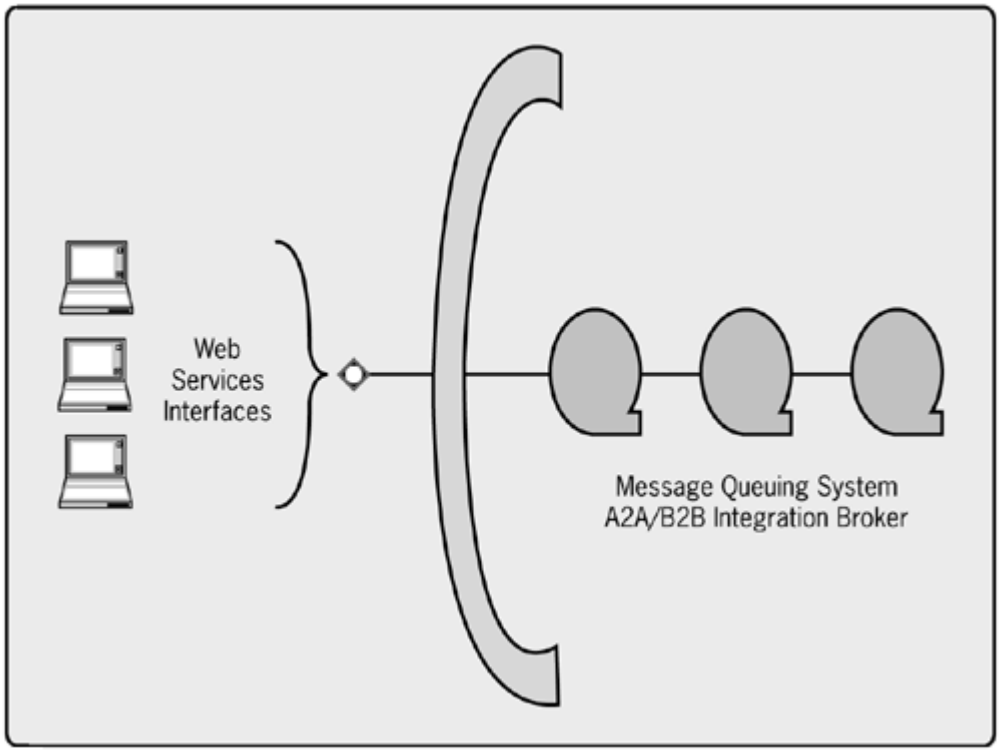
Figure 8-2. A Web Services interface is used to access database tables and stored procedures.



Web services map messages to databases

Figure 8-3 illustrates the Web services implementation architecture in which a Web services interface maps directly to a queued message system, such as Java Messaging Service (JMS) or MQSeries, or to a B2B server, such as IONA's Orbix E2A Collaborate. In both cases, the SOAP message is treated as input to an asynchronous communication system for processing. The message is stored in a persistent queue or database and is forwarded to another queue for processing. Finally, results are written to a reply queue. In this way, Web services represent another way into and out of existing B2B and application-to-application (A2A) integration broker products. For an integration broker vendor, the main value remains within the adapters, transformers, routers, and other parts of the toolkit.

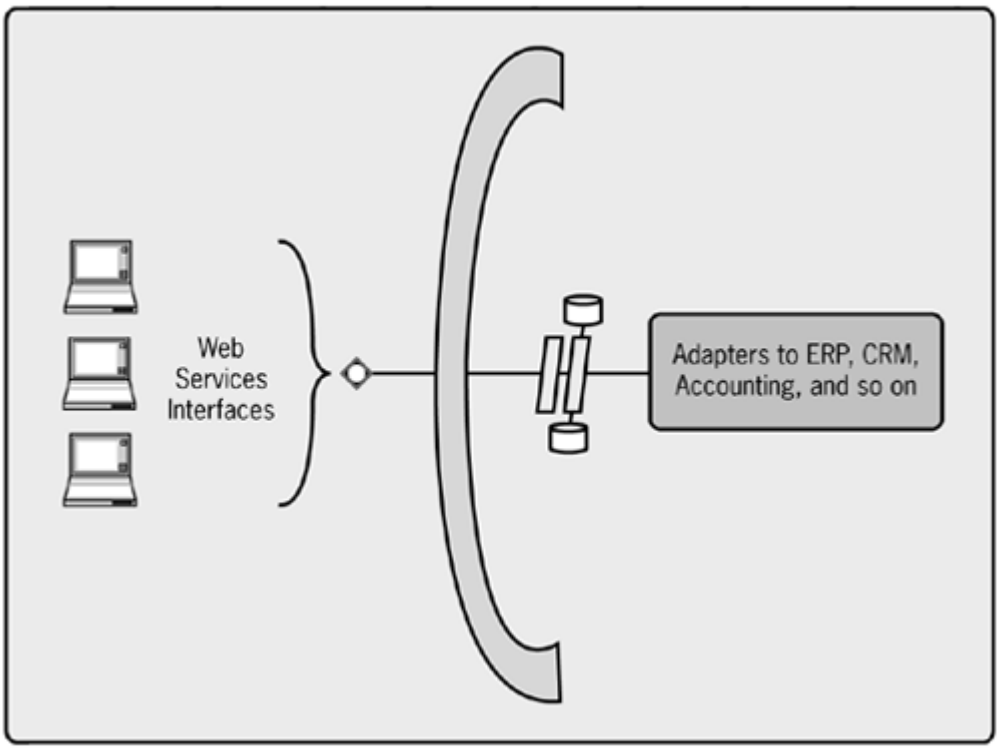
Figure 8-3. In an integration broker architecture, a Web services interface maps to A2A and B2B products.



Web services map messages to queues

Figure 8-4 illustrates the Web services implementation architecture for packaged application software such as ERP, CRM, and accounting/billing systems. For an ERP or a CRM system, the primary value of the application remains in the features and functions of the software package as related to business operation support. Web services, therefore, basically represent another means of getting data in and out, albeit a widely adopted and supported means. Packaged-application software vendors, such as Baan, PeopleSoft, SAP, Siebel, and others, also are starting to offer integration products using Web services technology.

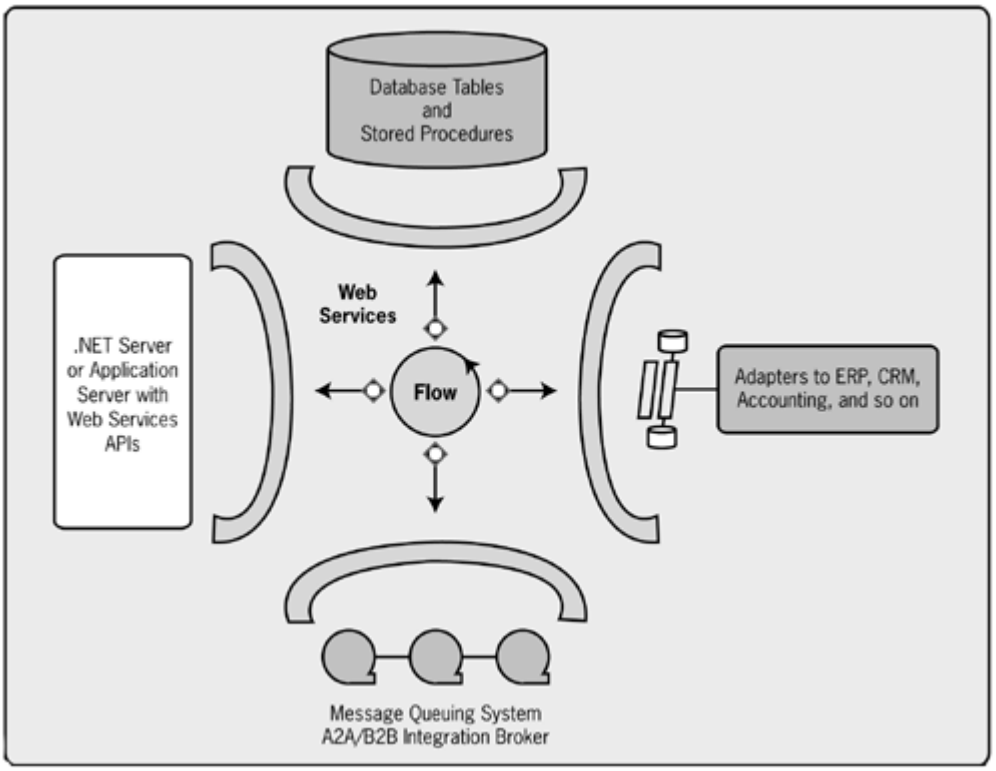
Figure 8-4. A Web services interface can be implemented for a packaged application architecture.



Web services map messages to packaged applications

Figure 8-5 illustrates the Web services implementation architecture that focuses on the value inherent in the Web services layer. A business process flow engine or other means of orchestration lies at the center of the value proposition in this architecture, which depends on the relationship among Web services to achieve or to create new applications at a higher level of abstraction. In other words, Web services have value because they are widely adopted and implemented, allowing multiple disparate software domains to be integrated. The process of integrating such domains requires certain functionality above the core Web services standards, such as process flow, security, reliable messaging, transactions, and so on. Web services brokers, such as IONA's Orbix E2A XMLBus Edition, are built to provide this value. In this way, disparate software domains are bridged, multiple vendor Web services implementations are joined, and new applications are created from a combination of old ones.

Figure 8-5. A Web services broker focuses on value in the Web services layer.



Web services brokers integrate them all

The Major Implementation Streams

The variety of vendor views and implementation architectures is understandable, as established vendors tend to view Web services in terms of additions or extensions to existing, successful product lines. Web services technologies are designed to be extensible, meaning that different vendors are going to have different interpretations, and many different types of products are going to emerge.

Vendors represent a variety of views

One key aspect of Web services therefore is achieving and maintaining interoperability and compatibility. For Web services truly to succeed, vendor products must become and remain as compatible with one another as today's browsers, Web servers, and HTML tools. This concept is being applied to Web services proposals and must be adhered to for Web services to succeed as widely as HTML and Web servers have.

Interoperability needs to be maintained

Many unresolved problems remain within the Web services community, as shown by the additional proposals in Chapter 7 and in the continuing work to complete ebXML. Different vendors focus on different value propositions and are therefore likely to implement different sets of additional Web services technologies.

Unresolved issues exist mainly at the level of additional technology

Until a comprehensive Web services architecture is available, like that created to guide the Object Management Group's success with CORBA, the vendor community will stay divided by differences of opinion and interpretation. Meanwhile, the two most significant and influential threads of activity around Web services remain .NET and J2EE.

.NET and J2EE represent significant implementation activities

Microsoft's .NET

It's fair to say Microsoft initiated industry focus on Web services, and Web services support is perhaps the most significant aspect of .NET, a broad-based initiative through which a variety of products is being developed and delivered to market. This effort also includes enhancements to Windows, COM+, and a variety of specialized server products.

Web services are a significant part of .NET

Like previous generations of Microsoft technology for application development and deployment, .NET includes programming tools designed for optimal use of Windows operating system features and functionality. The Visual Studio .NET integrated development environment (IDE) is tightly integrated with Windows, automatically generates code for .NET servers, and takes full advantage of significant Windows features.

.NET makes optimal use of Windows

Although .NET includes Web services, it is also a brand that applies to a group of technologies. Some of these technologies are new; other technologies have been updated to provide a new approach to creating Windows applications. Still other parts of .NET are rereleases of existing technologies.[1]

[1] See *Understanding .NET* by David Chappell (Addison-Wesley, 2002).

.NET is a brand applied to a range of technologies

.NET and the Web

Although .NET definitely supports Web services and is perhaps best known for that, it also supports other things, such as developing new applications and deploying them on a variety of servers. Most other approaches to Web services simply wrap applications with

mappings to and from XML data structures and formats. The .NET initiative goes further, perhaps because it's intended to represent the next generation of VB and Windows and adds to the .NET framework things needed for Web services. Microsoft has to be sure to avoid getting into the business of requring its software to be at each end of the communication, which would be antagonistic to the Web's origins as a free, open, and completely standard place. To Microsoft's credit, however, it has remained pretty sincere about helping to standardize the base protocol, and continues to be very active at W3C.

The following are the major parts of .NET:

- **.NET Framework:** Includes the Common Language Runtime (CLR) and the unified class library. The CLR is a standard foundation for building a range of new applications, whereas the unified class library provides many new application services. Among the technologies in the library are ASP.NET, which is the next generation of Active Server Pages (ASPs); ADO.NET, the next generation of ActiveX Data Objects; support for implementing Web services; and much more. Microsoft is also releasing a trimmed-down version called the .NET Compact Framework, intended for use in smaller devices, such as set-top boxes, personal digital assistants (PDAs), and wireless phones.
- **Visual Studio .NET:** Provides a visual programming environment for the programming languages that can be used with the .NET Framework: Visual Basic, an enhanced version of C++, and a new language called C#, which was designed explicitly for use in the .NET Framework.
- **.NET Enterprise Servers:** Host the server applications that the .NET Framework typically invokes, although they don't all necessarily make use of it. These servers include BizTalk Server 2000, Application Center 2000, Commerce Server 2000, Host Integration Server 2000, SQL Server 2000, Exchange Server 2000, Mobile Information Server 2001, and Internet Security and Acceleration Server 2000. These products are largely independent of the other .NET technologies listed here.

.NET has three major parts

ASP .NET is the recommended technology for implementing Web services based on the .NET Framework. ASP.NET Web services process service requests using SOAP over HTTP, as well as HTTP `GET` or `POST`. ASP.NET Web services automatically generate WSDL and Disco (precursor of WS-Inspection) files for Web services. You can use ASP.NET Web services to implement a Web service listener that accesses a business façade implemented as a COM component or managed class, typically hosted on a .NET server. The .NET Framework development kit also provides tools to generate proxy classes that client applications can use to access Web services.

ASP .NET is the recommended technology for Web services

Note that ASP .NET Web services—like any Web services—do not expose the server-side data types to client applications. These are completely hidden inside the Web service. The .NET Web services tools assume a stateless programming model—that is, each incoming request is handled independently. The only state maintained between requests is anything persisted in a data store.

Web services are stateless

.NET remoting, however, supports a more tightly coupled, object-based programming model between client and server, which provides remote access to server-side objects with full data type fidelity. Clients can also obtain references to server-side objects and control the lifetime of those objects for stateful interaction. If you use these object lifetime services, however, client applications will also need to be implemented using .NET Remoting.

.NET Remoting supports stateful interactions

In addition to the features provided by the .NET Framework, Visual Studio .NET provides tools to help you build, deploy, and consume Web services. For example, the IDE supports UDDI and Disco for locating Web services and understands how to generate client-side proxies from WSDL files. Visual Studio .NET also includes the Active Template Library (ATL) server, which C++ developers can use to construct Web service listeners that connect to a business façade implemented as a C++ class. ATL Server supports SOAP over HTTP, will automatically generate WSDL files for your Web services, and also provides tools to generate C++ proxy classes that client applications can use to access Web services.

Visual Studio supports UDDI, WSDL

Off-Platform .NET

Microsoft has been backing several initiatives designed to make .NET more open and more acceptable to customers concerned about the potential lock-in on Windows platforms. Off-platform versions of .NET are underway, although it must be noted that previous efforts to deliver off-platform implementations of COM and COM+ were not commercially successful. Ximian[2] is sponsoring a project to build an Open Source version of .NET. In addition, .NET specifications have been submitted to the European Computer Manufacturers Association (ECMA), which also engaged in COM API standardization.

The submitted specifications include the Common Language Runtime, the Microsoft Intermediate Language (MSIL), and many of the .NET Framework classes. Still, it remains to be seen whether independent implementations and standardization of .NET will ever get off the ground and whether reimplementations of .NET, such as the Ximian open source version, will ever succeed. It's traditionally been very difficult to "me-too" Microsoft and to keep off-platform implementations up to date. Also, many of the .NET features rely on Windows features, especially those for the client side and development environment. Will off-platform .NET be able to succeed without those features? No one imagines that anyone else will be able to supplant or even to duplicate Microsoft's presence on the desktop. Perhaps interoperability solutions will be enough, however, to allay fears of lock-in.

[2] A vendor of open source desktop software, including GNOME. The .NET project is called Mono—see http://www.ximian.com/devzone/projects/mono.html.

Does .NET Mean an Internet Operating System?

Bill Gates's announcement of .NET included a description of .NET as the equivalent of Windows for the Internet. If you had a service you relied on within the Windows operating system in the past, he said, in the future you would rely on the equivalent service over the Internet. If you needed to store a file, set a date in your calendar, send e-mail, or invoke any type of programmatic service, you would do so using .NET or Web services instead of using an operating system service.

This vision is as broad and compelling as it is fantastic. Imagine depending on the Internet or a Web service the way you'd rely on a desktop operating system service! Still, the idea of assembling applications—and computing platforms—out of standard components, the way you'd put together a PC, signifies great potential change in software development and deployment.

Microsoft's vision of Web services is that they will integrate services typically thought of as desktop, or local operating system, services with services accessed over the Web. Therefore .NET supports, or will support, all types of Web services interaction styles. Multiple vendors, such as IONA, Metratech, and Santra, are also providing generic Web services, or Web services *building blocks.*

.NET will support all types of Web services interaction styles

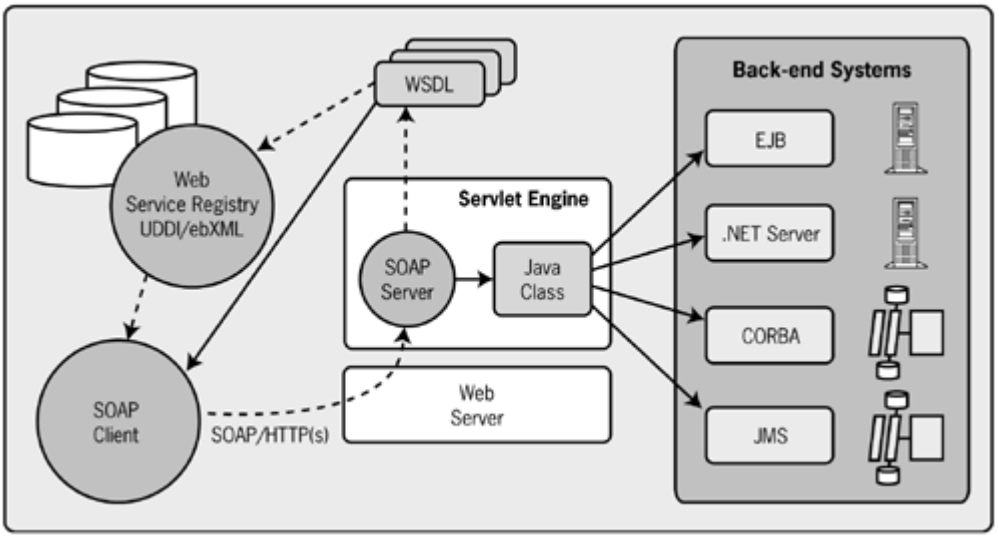
J2EE and Application Servers

Application server vendors are extending their products with Web services capabilities, in much the same way that many new technologies tend to get incorporated into J2EE, as new APIs.

J2EE vendors are supporting Web services

As shown in Figure 8-6, Web services can be integrated with application servers, such as WebLogic, WebSphere, and Orbix E2A J2EE Edition, to provide access to a variety of back-end systems, including .NET classes and COM objects, Enterprise JavaBeans, CORBA objects, MQSeries, MSMQ, and Java Messaging System queues. The SOAP messages arrive via the Web server integrated with the application server's servlet engine, in which is deployed a Java class representing, or wrapping, the back-end system to be integrated. Web services toolkits, such as IONA's XMLBus and those integrated directly with WebLogic and WebSphere, translate this Java class into a corresponding WSDL file, which can be stored in a UDDI or an ebXML registry. The SOAP client can find the location of the WSDL file from the registry or directly, using its URL, and invoke the Web service.

Figure 8-6. Web services can be integrated with application server environments.



Microsoft Support for Web Services Standards

Microsoft has pledged to support a common set of Web services standards or the railroad tracks on top of which run the trains carrying XML data. Microsoft believes, correctly, that a basic or miniumum set of agreed-on XML standards will be necessary for Web services adoption. However, a significant challenge will be to figure out where to draw the line between the minimum set and a larger set of potentially useful standards. This is a bit like the question of where to draw the line between the Web services movement and the ebXML initiative. The latter seeks to establish standards at a higher level of abstraction from the core technologies, focusing on the business process model. Microsoft is betting, of course, that if the appropriate level of Web services technologies gains acceptance as standards, the company will be able to find a good way of making a profitable business out of technologies and tools layered on them, or perhaps out of supplying content for them, just as the steel industry profited from the establishment of a national rail system (Drucker, 1999). Microsoft has led the way with many W3C submissions for this reason.

Application Server Vendor View

Sun Microsystems initiated and has been at the forefront of a developer community and execution environment set of technologies based on the Java programming language and its various editions. The Java 2 Platform, Enterprise Edition (J2EE) has become the focus for Web services technologies for the Java community, including the leading application server vendors. Sun Microsystems has provided consistent leadership for this community and is extending the leadership with respect to Web services. Sun Microsystems views Java as the most suitable language for Web services development, and the Java community has already produced many products.

Sun Microsystems provides leadership for the J2EE community process

J2EE vendors essentially view Web services as another type of application server client. All the requisite qualities of service are defined in J2EE and implemented in application servers. Unlike .NET, the idea of creating J2EE applications using Web services building blocks is not yet prevalent. The J2EE community views Java classes and beans as the building blocks, with Web

services a type of interface into and out of them. Web services may eventually support a level of quality of service equivalent to that of J2EE-based application servers, but Web services may need to take their own direction, especially when multiple, disparate software domains are to be bridged across the Internet. And, it is likely to be a long time before Web services can match the qualities of service already available in J2EE-based application servers.

Web services are clients to application servers

Some of the Java-oriented views toward Web services include the following ideas:

- J2EE applications expose EJBs and JMS destinations as Web services.
- Exposed services can use WSDL as the service description language and can provide access to components using SOAP over an HTTP transport.
- Services are published and distributed through ad hoc means.
- Private registries, possibly based on UDDI, are used to integrate with partners by some applications.
- Web services will not change existing trading-partner policy and convention agreements.

Application servers use Web services for a variety of interactions

Application Server Vendor Position

While Microsoft is placing its bet on reinventing its core services based on XML, BEA Systems, IBM, Sun Microsystems, and others are betting that the Java platform established today in products such as WebSphere, WebLogic, and iPlanet will continue to evolve as the development environment of choice for Web services. Most of the Web services work in the application server market is focused on initiatives to extend J2EE with APIs supporting Web services standards and technologies. Web services are in many ways a natural fit for application servers, as shown in Figure 8.6.

Java APIs for Web Services

The J2EE standard defines Java APIs for enterprise services, such as security, messaging, transaction management, and directory lookup. The J2EE APIs are provided in application server vendor products from BEA Systems, HP, IBM, IONA, Oracle, Sun Microsystems, and others. Several efforts are under way to extend J2EE APIs for use with Web services. These APIs facilitate parsing and manipulating XML from within J2EE servers and provide access from J2EE servers to external Web services technologies, such as UDDI.

J2EE includes new APIs for Web services

Table 8-1 shows the Java Community Process (JCP) initiatives, led by Sun Microsystems, that are relevant to Web services. The JCP initiatives shown in Table 8-2 related to Web services are led by other vendors.

JCP members define the APIs

J2EE Initiatives for Additional Technologies

Significant work is already under way within the JCP for Web services security and transactions. Additional technologies, which are shown in Table 8-3, will provide the basis for secure, reliable, robust, and complex business applications for J2EE-hosted Web services.

APIs are under way for security and transactioning

Reliable Enterprise Web Services Already in ebXML

Enterprise qualities of service specifications such as routing and reliable messaging already exist within ebXML. Many large vertical standards organizations have said that they will adopt ebXML for these purposes since ebXML already defines routing and reliable messaging on top of SOAP. The relationship between ebXML's reliable messaging and the reliable messaging specifications created for the Microsoft GXA are undefined, however. Convergence on a single solution is always better but, in this case, by no means guaranteed.

Table 8-1. Sun Microsystems-Led JCP Initiatives

Initiative	Description
Java API for XML Parsing (JAXP)	Provides users with pluggable APIs for XML parsing and transformation. For parsing, for example, an application could use either a DOM or a SAX parser through JAXP. In either case, the developer can choose to use any among many compatible DOM or SAX parsers.
Java API for XML Messaging (JAXM)	This specification defines Java APIs for exchanging business documents—XML documents and other arbitrary data—among trading partners. JAXM implementations are expected to package the documents, using SOAP with Attachments, ebXML Transport, RosettaNet, or BizTalk, and to deliver the documents to the intended destination. JAXM defines a low-level Java API for constructing and routing SOAP messages and APIs for emerging messaging standards, such as ebXML.
Java APIs for XML-based RPC (JAX RPC)	This specification defines Java APIs for SOAP and potentially other XML-based RPC-oriented protocols. The APIs include those for • Marshaling and unmarshaling arguments • Transmitting and receiving calls, including portable stubs and proxies • Mapping SOAP RPC-style interactions onto Java interfaces, classes, and methods • Mapping Java interfaces, classes, and methods onto SOAP RPC-style interactions JAX RPC defines APIs to map SOAP onto Java classes and vice versa, that is, Java APIs that perform the mapping to and from XML in the form of SOAP messages.
Java APIs for XML Registries (JAXR)	This specification defines Java APIs for access to UDDI and ebXML registries for storing and retrieving information about a business's Web services and other interfaces. Ideally, JAXR will provide one set of

Table 8-1. Sun Microsystems-Led JCP Initiatives

Initiative	Description
	APIs that can be used independently of the underlying registry implementation, much the way current Java Naming and Directory Interface (JNDI) APIs can be mapped to any underlying directory service. JAXR provides an abstraction for XML registries so that it has pluggable support for current registry standards such as the ebXML Registry/Repository and UDDI, and also for future registry standards that may emerge. By using JAXR, developers can build clients that can work with diverse registry implementations. JAXR supports the full life-cycle spectrum for managing objects and services in a registry, such as registering, updating, querying, and deleting objects or services in the registry.
The Java Web Services Developer Pack	Provides a complete set of technologies for tools vendors to incorporate support for Web services in their products. The pack combines all of the Java XML APIs and technologies with other key technologies. Consequently, developers have all the tools they need to start developing robust web services. Services developed using the Java Web Services Developer Pack can then also be deployed into a secure, reliable, and scalable J2EE environment.
SourceForge ebXML Registry/Repository Open Source Project	Sun Microsystems donated an internal implementation of the ebXML registry and repository to the open source community at SourceForge. A growing developer community from many companies and countries is working on this project (see http://sourceforge.net/projects/ebxmlrr/).

Table 8-2. JCP Initiatives Led by Other Vendors

Initiative	Description
Java APIs for WSDL	This JSR provides a standard set of APIs for representing and manipulating services described by WSDL documents. These APIs define a way to construct and to manipulate models of service descriptions.
ebXML CPP/A APIs for Java	This JSR provides a standard set of application programming interfaces for representing and manipulating information described by ebXML CPP and CPA documents.
Implementing Enterprise WebServices	This specification defines, comprehensively within the J2EE platform, an architecture for developing and executing Web services, including JAXM, JAXR, and JAX RPC. This specification also references Java APIs under development for XML Data Binding (JAXB). JAXB binds XML documents to Java objects, using XML schemas, and provides a binding framework API for accessing the generated Java objects (see http://java.sun.com/xml/webservices.pdf).

Understanding .NET versus J2EE

The main difference between Microsoft's .NET and the various forms of Web services toolkits associated with application servers is that Microsoft views Web services as the thing you develop. By contrast, application server vendors view Java classes and Java beans as the thing you develop, even when you are ostensibly developing Web services. With the .NET approach, development and deployment of a Web service occur in a single stage. Once you write a program in any one of the .NET languages, you have only to push a button to create and deploy it as a Web service. Every program is a Web service to .NET, or at least potentially so.

Table 8-3. JCP Security and Transaction Web Services Technologies

Technology	Description
Java Specification Request (JSR) 104: XML Trust Service APIs	This specification defines a standard set of APIs and a protocol for a trust service. A key objective of the protocol design is to minimize the complexity of applications using XML Signature. By becoming a client of the trust service, the application is relieved of the complexity and syntax of the underlying PKI used to establish trust relationships, which may be based on a different specification such as X.509/PKIX, SPKI, or PGP.
JSR 105: XML Digital Signature APIs	This specification defines a standard set of APIs for XML digital signatures services. The XML Digital Signature specification is defined by the W3C. This proposal is to define and to incorporate the high-level implementation-independent Java APIs.
JSR 106: XML Digital Encryption APIs	This specification defines a standard set of APIs for XML digital encryption services. This proposal is to define and to incorporate the high-level implementation-independent Java APIs.
JSR 155: Web Services Security Assertions	This specification provides a set of APIs, exchange patterns, and implementations to exchange assertions securely—with integrity and confidentiality—between Web services based on OASIS SAML.
JSR 156: XML Transactioning API for Java (JAXTX)	The specification provides an API for packaging and transporting ACID transactions and extended transactions, such as the BTP from OASIS, using the protocols being defined by OASIS and W3C.

In the Java world, Web services are a two-step process in which programmers develop classes and beans and then decide as a second step which of them, if any, are to be created and deployed as Web services. Application server developers often design new classes and beans for the specific purpose of exposing business logic as a Web service, as existing classes and beans are written at a level of granularity more appropriate to use within a tightly coupled binary protocol running in a LAN environment. Some tools also generate Java classes from Web services, but creating a Web service from a Java program is not the default assumption, as it is for the .NET developers. Application server "tiers" still communicate using RMI or RMI/IIOP, whereas .NET tiers communicate using XML protocols.

In short, .NET is a thorough, fundamental rearchitecting of a distributed computing platform based on Web services, while application server support for Web services tends to be designed more as another client, or presentation tier for the back-end system. This may change, however, as more of the JCP-defined APIs for Web services get adopted.

Vendor Views on Adoption of Web Services Technologies

A brief questionnaire was sent to leading Web services vendors to obtain their views on the current state of adoption for the basic Web services technologies—SOAP, WSDL, and UDDI—and for ebXML, and on the future direction of Web services. Support is unanimous for SOAP and the idea of Web services, but some questions remain about the adoption for WSDL, UDDI, and ebXML, in that order. All the vendors agreed that following the adoption and implementation of the core standards, the next step is to standardize the additional technologies required for more complex and robust business applications.

Vendors provided their own views

This section summarizes the responses and lists these vendors' current types of products that provide Web services, and what type. The statements about current products and future directions are taken from their replies to the questionnaire. In addition, some of the vendors took advantage of the opportunity to submit a brief analysis.

The Questionnaire

A questionnaire was sent to BEA Systems, Cape Clear, HP, IBM, IONA, Microsoft, Oracle, Sun Microsystems, and Systinet. The information in this section represents the views of the vendors as they responded to the questions. The first question was aimed at discovering the extent of support for the core Web services technologies covered in this book—SOAP, WSDL, UDDI, and ebXML—especially in terms of product implementation. The second question gave the vendors a chance to discuss next steps for Web services, either in terms of their product roadmaps or in relationship to the industry as a whole.

Questionnaire sent to major vendors

Following a brief introduction, the original text was preserved wherever possible because the replies were not consistent with respect to format or content. In some cases, the replies were almost entirely in the form of analysis. In all cases, the views stated are those of the respective companies.

Information presented as received

The Questions

1. **Basic information:** Widespread adoption and implementation seem ensured for the core Web services standards—SOAP, WSDL, and UDDI and, to some extent, ebXML—specifically the transport, CPP, and CPA subsets. Please indicate any opposing views; otherwise, the book will state agreement on this point.

   Could you please provide a list for inclusion in the book of the products that implement these standards? Provide a brief description of each product's major features and benefits; that is, what would your customers use them for and how—hand-built, generated, combination of the two, and so on.

2. **Next steps:** Following the adoption and implementation of the core standards, the next step is to standardize the additional technologies required for more complex and robust business applications. Please indicate any opposing views; otherwise the book will state agreement on this point.

   Which of the various proposed additional technologies—security, transactions, routing, reliable messaging, process flow, and so on—are the most important? And which ones, if any, are most likely to achieve a level of adoption comparable to the core standards? Which ones, if any, do you intend to implement?

In summary, it appears that SOAP is widely supported, adopted, and implemented within products. WSDL also is very widely supported, and many products already provide implementations. However, some questions remain with respect to its evolution and the extent to which it may change during formal standardization. UDDI and ebXML raised the most question marks among the respondents; however, the consensus in favor of UDDI remains strong. Several vendors are implementing parts of ebXML, but the extent to which the complete ebXML architecture will be adopted remains unclear.

Unanimous on SOAP, nearly so on WSDL

Vendors showed less agreement about next steps and the adoption of additional technologies. However, all vendors mentioned common themes in recognizing the need for these additional technologies, identifying security, reliable messaging, transaction processing, and process flow as important, although in varying priorities and orders. Sun Microsystems, for example, pointed to ebXML for many of the additional technologies, while Microsoft pointed to their GXA instead.

Difference of opinion strongest on ebXML

BEA Systems

BEA, like HP, was working on XML-based protocols and products of its own when Web services hit. Like Sun Microsystems, BEA attended the June 2000 W3C Workshop on XML Protocols in Amsterdam to help evaluate whether to support SOAP. By the time of the WSDL submission to W3C later that year, BEA had joined the effort as a cosubmitter, clearly indicating its support for Web services. Since then, BEA has become very active in the Web services community and, like other application server vendors, has begun shipping Web services support in its products. BEA is active at W3C and in the JCP efforts to define Java APIs for Web services technologies. BEA's Web service implementation centers on its WebLogic flagship application server product.

Web Services for the J2EE Professional

WebLogic Server has two ways to create and to consume Web services. Each approach is targeted at a different developer audience. For the J2EE professional developer, WebLogic Server provides an intuitive way to expose J2EE components as Web services. For developers who are not comfortable with the complexity of the J2EE programming model, a new IDE and framework, named WebLogic Workshop, provides a higher level of programming abstraction that shields the developer from the complexities of J2EE and object-oriented programming.

Web services for the J2EE professional are implemented using three types of components:

- Back-end J2EE components
- Handlers, for preprocessing and postprocessing SOAP requests
- Serializers and deserializers, for converting data between an XML representation and Java objects

These components are linked together in a pipeline model of processing. The pipeline allows developers greater control over the handling and ordering of processing of SOAP messages.

Back-end components can be one or more stateless session EJBs for RPC calls or message-driven beans for asynchronous messaging. WebLogic Server provides a simple mechanism for exposing the appropriate EJB methods as operations of a Web service and generates the associated WSDL file automatically.

Handlers are optional components that provide access to the SOAP message before and after invoking the back-end component. They typically are used for processing and manipulating SOAP header information.

WebLogic Server provides a framework for automatically converting data between XML and Java. The Web services can therefore be designed only in Java without having to process XML directly. Support for the simple data types defined in the SOAP specification are built in. Complex or user-defined data types can be converted by using optional serializer/deserializer objects. The WebLogic Autotyper utility can be used to generate a serializer and a deserializer that will handle the conversion at runtime. As input, WebLogic Autotyper will accept an XML or a Java representation of the types and will generate the equivalent Java or XML representation along with the serializer/deserializer.

The handlers and the serializers and deserializers make use of an innovative new XML parser, called a *pull parser*. A pull parser, sometimes known as a tokenizing parser, enables a developer to extract a subset of the XML document in a very high performant manner.

WebLogic Workshop

WebLogic Workshop is an IDE that is targeted at the developer who prefers a GUI-based approach to application development. Borrowing from the Visual Basic paradigm WebLogic Workshop allows developers to define a Web service and its interaction with system resources, using visual metaphors. The developer can then drop down in the code view and define the procedural logic for the Web service. WebLogic Workshop developers work within a higher level of programming abstraction that shields them from the complexities of J2EE and object-oriented programming. This exposes the power of Web services, J2EE, and WebLogic Server to a new class of developer.

Web Services Standards

WebLogic Server supports the most common Web services standards, notably, SOAP, XML schema, WSDL, and UDDI. In addition, support for conversational Web services is included.

Cape Clear

Former IONA employees started Cape Clear in Dublin in 1999 to focus on XML. Initially, they worked on the integration of XML and CORBA and then later switched to Web services.

The company's two products are CapeConnect, a Web services platform and integration; and CapeStudio, a Web service development toolset. The CapeConnect toolkit for the Windows platform generates Web services clients that interoperate with Java classes and CORBA objects on the server side. Among other things, CapeConnect connects Web services clients on Windows and other platforms with Java and CORBA servers on Windows and other platforms.

Current Situation

Cape Clear agrees with the adoption of core standards but believes that ebXML adoption will take up to a year longer than ebXML vendors are predicting. The Web services community is much wider than the CORBA or J2EE communities, and ebXML is possibly too complex for a lot of Web services users. Right now, most users are concentrating on getting the essentials right. Cape

Clear believes that only when there is a critical mass of commercial experience around these core standards will a lot of people start moving up to more complex standards.

Next Steps

Security, transactions, routing, reliable messaging, and process flow are all important, as are automated data transformation, rules processing, asynchronous messaging, and systems management. Cape Clear suggests the following order of importance and order for roll-out and/or adoption:

- **Security:** This is essential, and everybody wants this *now*.
- **Reliable messaging/asynchronous messaging:** This is essential for solving real-world problems.
- **Systems management:** So far, this topic has received far too little discussion, which is a sign of the immaturity of the technology, despite the interest in it.
- **Routing:** Today, nearly all development is fixed, peer-to-peer, but asynchronous messaging will require routing. However, a drawback of the proposed SOAP routing specification is that it requires knowing all the routing hops in advance of sending the message, which is sort of like the old UUCP mail where you had to know the physical layout of the network to send a message.
- **Automated data transformation:** XSLT provides a great way of mapping other XML dialects into SOAP/WSDL, which will become more and more important, especially in industries with a lot of XML adoption, such as finance; as well as a way of doing object transformations—WSDL–WSDL.
- **Transactions:** This area is not as important as some people think, as most of the Web-based applications will not expose transactions, which will lie encapsulated under a more abstract Web service interface. It's also difficult to see people integrating transactions with an asynchronous model, and there is not enough interoperability between transaction engines to make it work portably, particularly given that not all vendors support nested transactions.
- **Rules processing:** The Web services platform is a natural place to add platform-neutral business logic; Cape Clear believes that the easiest way to do this is via a rules-based approach, whereby business logic will be executed based on the content of incoming messages. Rules processing also ties into routing for content-based addressing or message rewriting.
- **Process flow:** This area will remain very important to some people, possibly increasing in importance in terms of Web service assembly. However, that will depend on a large number of available Web services that have been designed for easy assembly and sufficiently advanced tools to make it easy to assemble them. If it's difficult to do, it will alienate both the people without the skill and patience to use the tools as well as the skilled users who will just code the applications by hand.

Hewlett-Packard

Hewlett-Packard (HP) began early in the Web services market with its eSpeak initiative. Although eSpeak incorporated many, if not all, of the core concepts and technologies in Web services, eSpeak began before the emergence of the core Web services standards; eSpeak was based on proprietary formats that did not conform to SOAP, WSDL, and UDDI. HP's newer products do, however, and the company is bringing its experience gained with eSpeak to bear on standards initiatives as well as products. HP is very active at W3C, JCP, and OASIS.

eSpeak was an early implementation of Web services concepts

Products

HP markets several standards-based Web services products, including the ones described in Table 8-4.

Next Steps

Although simple Web services can provide significant benefit for many businesses, complex Web services capabilities are required for true B2B collaboration and hence require complex Web services, which typically involves XML document exchange or a series of document exchanges. Thus businesses will require a flexible and customizable platform that enables them to plug in emerging standards-based solutions for supporting long-duration transactions, message and sender authentication, asynchronous messaging, support for multiple business protocols, manageability, and the ability to combine services and applications into business processes easily. Such a solution also needs to support multiple platforms and to deliver true interoperability between programming environments. HP's product direction encompasses all of these areas. For example,

- **Security:** Bundled with the HP Web Services Platform is an XML Digital Signature pack that enables you to ensure message integrity and message authentication. The Digital Signature pack describes the XML syntax for representing digital signatures and provides the flexibility to sign specific portions of an XML document. SSL can be used to ensure sender authentication.
- **Asynchronous messaging and support for multiple protocols:** In addition to SOAP-RPC support, the HP Web Services Platform will support virtually any XML protocol and therefore will participate in multiple trading environments. Included in the HP Web Services Platform architecture is a listener framework for receiving both synchronous and asynchronous messages over a variety of transport protocols and a pipeline for preprocessing and routing XML documents. This framework enables the HP Web Services Platform to receive, process, and return XML documents based on requirements detailed in such business protocols as RosettaNet, ebXML, or BizTalk.
- **Management:** Via an integration with Hewlett-Parkard's market-leading management platform, OpenView, businesses will be able to extend their management view and control to include Web services.
- **Business process management:** HP will offer integration with Process Manager and Process Manager, Interactive Edition, two business process management and graphical workflow platforms. Integrating the HP Web Services Platform into these or similar business process management platforms enables both business and technical resources to reuse business logic—applications, services, processes—to create or to modify business processes.
- **Platform-neutral interoperability:** HP is dedicated to providing middleware products that are platform-and application-server–neutral. HP's Web services components will be able to operate within any J2EE-compliant application server, including HP-AS, WebLogic, and WebSphere. In addition, significant testing by HP will ensure true interoperability between Java- and .NET-deployed services.

Complex Web services capabilities are required

Table 8-4. Hewlett-Packard's Web Services Products

Product	Description
HP Web Services Platform	This is a complete, standards-based architecture for developing, integrating, deploying, and consuming Web services. This product supports today's key Web services standards such as WSDL, SOAP, and UDDI and includes the ability to easily plug in future business and technology standards. The platform has a number of key components, including HP-SOAP server, HP Service

Table 8-4. Hewlett-Packard's Web Services Products

Product	Description
	Composer, HP Registry Composer, and trail map tutorials. • HP-SOAP provides a standards-based communications platform for sending, receiving, and processing SOAP-based messages. HP-SOAP delivers a pipeline-processing model based on Cocoon2, an Apache open source framework, and includes a listener framework (HTTP, HTTPS, SMTP), envelope processing, and routing capabilities. HP-SOAP operates within a J2EE application framework in order to take advantage of the performance, reliability, and scalability aspects of J2EE. • HP Service Composer supports the full life cycle of Web services development and enables you to create, view, and modify new and existing WSDL definitions; create, view, and modify XML schema definitions; generate Web services definitions from existing Java code bases; and generate Web services client proxies, server skeletons, and deployment descriptors. • HP Registry Composer is a graphical tool that simplifies the process of registering and searching for Web services in either public or private UDDI registries. HP Registry Composer supports the latest UDDI specifications, as well as UDDI4J, the prevailing Java API for accessing UDDI registries.
HP Web Services Registry	This is an implementation of the UDDI specification that enables you to set up private registries with controlled access to available Web services. Similar to public UDDI registries, HP Web Services Registry provides both Web-based and programmatic interfaces for registering and searching for available businesses and services. The programmatic interface is based on UDDI4J, a Java API for accessing UDDI registries. Also bundled with HP Web Services Registry is the HP Registry Composer tool.
HP Web Services Transactions	This product provides end-to-end transactional integrity for Web services. Based on BTP (see Chapter 7), HP Web Services Transactions enables reliable transactioning between Web services outside the firewall and between disparate platforms, such as between Java-and .NET-deployed Web services. Supported are long-duration transactions, nested transactions, and two-phase commit capability. The product includes class libraries for migrating new or existing Web services to transaction-capable Web services.

IBM

When IBM decided to get involved in Web services by joining Microsoft as authors and proponents of the SOAP specification, it was big news and put the initiative on the map in a way that IONA's earlier support could not. Any significant collaboration between Microsoft and IBM carries a lot of weight. At the time, Microsoft and IBM also had been working on the proposal that became UDDI and later collaborated to produce WSDL. IBM is also a major contributor to ebXML and sponsors a major Web services developer site.

IBM's endorsement put SOAP on the map

IBM products either already support Web services or are rapidly moving to support them, starting with the core technologies: SOAP, WSDL, and UDDI.

Product support is moving quickly

Basic Information

In response to question 1 on the basic questionnaire sent to all vendors, IBM made the following comments:

> IBM would agree with this statement; however, we would like to modify the list in two ways. First, we would like to add XML Encryption and Digital Signature; second, we would like to emphasize that the most important of these three standards is WSDL. The description is what creates the interoperability. We are in but the first step of a rapidly emerging foundation for business—SOAP and UDDI are where we start. The future is bright for additional protocols and publication technologies. Certainly SOAP is critical to Web services, and we don't see this role diminishing. SOAP is the first, best-defined, and tested WSDL binding available. It is the protocol of choice for doing cross-enterprise B2B communication. However, it will not be the only WSDL binding in use inside or outside the enterprise. We see potential for other bindings, which are not SOAP-based, to emerge in the future and be very important as well.
>
> IBM is still fully behind UDDI both in its role as the public business and service registry and as a private, intraenterprise registry. IBM also sees the value of additional publication methodologies such as WS-Inspection. IBM sees Web services in the future as being built around WSDL with support for SOAP, UDDI, XML Encryption, Digital Signature, and other core Web services standards yet to be invented. And, yes, IBM recognizes that WSDL itself is a first step and requires evolution to meet the demands of e-business. This evolution is already under way for security and XML Encryption and Digital Signature. IBM looks forward to continuing to help guide that evolution with the W3C and OASIS.
>
> IBM believes that ebXML is coming from the business side and offers a holistic solution. Web services is coming bottom-up and offers incremental parts that are easier and cheaper to adopt. IBM believes there will be convergence of the two stacks with the business-level concerns of ebXML around business agreements and business process interaction, layered on top of the Web services base of SOAP, WSDL, UDDI, Security, and Workflow.

AlphaWorks Information

IBM hosts a developer Web site called AlphaWorks (http://www.alphaworks.ibm.com) and offers a variety of technology previews. The technologies available from IBM's AlphaWorks Web site are not official IBM products. However, the Web site offers IBM customers and developers a view of the technologies, such as SOAP, WSDL, and UDDI, that IBM believes may prove to be important in the future.

Technology previews are available at AlphaWorks

The AlphaWorks site allows customers and developers to comment on these technologies and improve them during this Alpha stage. In the case of Web services, IBM has previewed key Web services technologies on AlphaWorks and has shown how these technologies could be used via technology demonstrations.

The available AlphaWorks technologies include the ones that are shown in Table 8-5.

Table 8-5. IBM's AlphaWorks Technologies

Technology	Description
Web Services ToolKit (WSTK)	A software development kit that includes a lightweight runtime environment, a demo, and examples to aid in designing and executing Web services applications that can use SOAP, UDDI, and WSDL to find one another automatically and collaborate in business transactions without additional programming or human intervention. SOAPConnect, which is a LotusScript library and Java agent, is included; it facilitates consuming Web services from a Domino application.
Web Services Hosting Technology	A set of technologies that enables hosting support for Web services within a WSTK environment.
Web Services Invocation Framework	A tool that provides a standard application programming interface (API) for invoking services described in WSDL, no matter how or where the services are provided.
Web Services Gateway	A middleware component that provides an intermediary framework between Internet and intranet environments during Web services invocations. It offers a gateway function at the protocol level from SOAP to other protocols.
Business Explorer for Web Services	An XML-based UDDI exploring engine that provides standard interfaces for efficiently searching business and service information in UDDI registries.
Web Services Process Management Toolkit	A tool that allows you to compose Web services and to include them in a business process to achieve a defined business goal. It is the Web services enabling for MQ Workflow.
Lotus Web Services Enablement Kit (WSEK)	This technology is a blueprint, including demonstration, samples, and source code, for enhancing Lotus products with Web services.
XML Registry/Repository (XRR)	This product is a database for storing WSDL and other XML artifacts.

Products

Table 8-6 describes the official IBM products with Web services implementations.

WebSphere and DB2 support Web services

Table 8-6. IBM's Official Products

Product	Description
WebSphere Application Server 4.0	This flagship application server offering is aimed at professional Java technology developers who require J2EE and Web services functionality. It is Web services enabled with support for XML, SOAP, WSDL, and UDDI. (More information is available at http://www.ibm.com/software/webservers/appserv.)
WebSphere	An easy-to-use, integrated development environment for building, testing, and

Table 8-6. IBM's Official Products

Product	Description
Studio Application Developer	deploying J2EE applications. This application development environment, equipped to support Web services development, includes • A powerful Java development environment with wizards, content assist, debugger, refactoring, scrapbook, unit test environment, and pluggable JRE • Complete EJB development and test environment with EJB v1.1 support • Web services integration to create, build, test, publish, and discover Web service-based applications, supporting UDDI, SOAP, WSDL, and XML • XML development environment to generate DTDs, schemas, XML documents, and XSL style sheets • Wizards for servlet and JSP creation • Relational Schema Center for advanced database application support • Profiling and tracing tools for optimizing application performance • Collaborative team development support
WebSphere UDDI Registry	A UDDI-compliant registry for Web services in a private intranet environment. With the IBM WebSphere UDDI Registry, Web services developers can publish and test their internal e-business applications in a secure, private environment. The product supports multiple users in various department or companywide scenarios and also supports the SOAP-based APIs defined by either the UDDI v1 or v2 specifications and provides persistence for published entities through a relational database. A Web-based graphical user interface supports publishing and querying of businesses, services, and other UDDI-compliant entities without programming (available from the WebSphere Developer Domain at http://www7b.boulder.ibm.com/wsdd/downloads/UDDIregistry.html).
DB2 XML Extender 7.2	Enables Web services applications to access data stored in DB2 as an XML structured document, providing businesses with greater ease and efficiency in developing new dynamic e-business applications using XML interfaces only. Three kinds of Web services operations are supported: retrieve and store XML, SQL-based, query and update, and stored procedure invocations.
Web Services Object Runtime Framework	Provides Web services access to SQL queries, stored procedures, and XMLExtender functions, and enables XML Extender transform capabilities and DB2/XML to/from Web services (DADX). A DADX document specifies a Web (WORF) service using a set of operations defined by XML Extender DAD documents or SQL statements. Web services specified in a DADX file are called DADX Web services. The framework provides the following features: • Resource-based deployment and invocation • Automatic service redeployment, at development time, when defining resource changes • HTTP `GET` and `POST` bindings, in addition to SOAP • Automatic WSDL and schema generation, including support for UDDI best practices • Automatic documentation and test page generation WORF is downloadable from the IBM XML Extender Web site—http://www.ibm.com/software/data/webservices/—and is shipped with WebSphere Studio Application Developer and WebSphere Studio Site Developer (WSSD) Technology Preview.

Next Steps

In addition to advancing core technologies for Web services, IBM sees significant value in the area of enabling services, or those that an application developer, service developer, or business process developer would want to use. The Web Services ToolKit contains a few of these: notification, metering and accounting, persistence, and identification. The Web Services Hosting Technology illustrates how to use these enabling services in a solution for service provisioning. IBM hopes to see the WSDLs for a set of useful services standardized in the future to promote interoperability, availability, profitability, and reusability of Web services.

As far as the future goes, IBM believes security is the most critical functionality needed for Web services. Digital Signature and XML Encryption are the next two technologies to be added to the core. Additional security technology is being worked on to create an end-to-end security solution for Web services. Business process choreography would be IBM's next step, and it already has a proposal in this space (WSFL—see Chapter 7); however, there is no industrywide standard as yet.

Along with choreography comes the need to address quality-of-service aspects, including reliability, agreements, transactions, and management. IBM believes that the core standards list will expand to include not only security, but also those technologies created and adopted for choreography, quality of service, and reliable messaging as these are the points where interoperability is crucial. There may also be a set of technologies for Web services, which are just as important but not part of the core, specifically management provisioning and partner agreements. Of course, IBM is also making progress with these technologies for Web services simultaneously.

IBM views Web services as the critical next step for middleware. As such, its products are rapidly moving to (or already do) support Web services. Initial support has been for WSDL for describing services, SOAP for accessing services, and UDDI for discovering services. IBM's focus for Web services is that it is a standards-based, unified framework for both B2B and Enterprise Application Integration (EAI). WSDL serves the most critical role for IBM, as it's the lingua franca for describing services. In fact, IBM defines Web services as "software components described via WSDL [that] are capable of being accessed via standard network protocols, such as SOAP over HTTP."

From the business point of view, IBM sees Web services as allowing one to lower the cost of interaction to almost nothing between two applications or parties as the application describes its interfaces for anyone to connect with. From a base technology point of view, IBM feels that WSDL enables the whole area of service-oriented architectures and that SOAP/HTTP is critical when such models are used for B2B scenarios.

IONA

IONA joined Microsoft in support of SOAP in early 2000, implementing early versions of the specification and demonstrating interoperability between SOAP and CORBA. IONA also has been very active in support of ebXML and provides an implementation of ebXML in addition to comprehensive support for Web services throughout its product line, which includes the Orbix End 2 Anywhere (E2A) platforms. IONA continues to focus on interoperability, Web services creation, security, and orchestration.

IONA demonstrated interoperability in early 2000

Products

Orbix E2A Web Services Integration Platform

This platform uses Web services standards, including SOAP, WSDL, and UDDI, to simplify business process automation and information exchange for customers, trading partners, and internal departments. The Orbix E2A Web Services Integration Platform consists of the three editions described in Table 8-7.

The entire product line supports Web services

Orbix E2A Application Server Platform

The platform is the most widely deployed development platform for the most demanding distributed applications in the world, combining the scalability of CORBA and the productivity of J2EE. The Application Server Platform supports all of the major component architectures and Web services and consists of the three editions described in Table 8-7.

Next Steps

Web services are very appropriately used for wrapping business process flows. Everyone asks what they should use Web services for today, and the right answer is to help automate business process interactions. So establishing a realistic, simple, and appropriate business process flow standard is very important. Microsoft and IBM appear to be far apart on this critical technology, which may impede progress.

Focus should be on security and process flow

In the Web services world, a business process flow is the equivalent of a business transaction, and therefore Web services transactions should focus on interactions with process flows rather than on interactions with data resource managers. It should be sufficient to notify a business process flow of a cancellation or a change in the event of failure or inability to successfully coordinate the execution of multiple business process flows or of multiple steps within a single flow. Therefore once a business process flow specification is adopted, a transaction coordination specification that fits well with it will be needed.

Table 8-7. The IONA E2A Platforms

Product	Description
Web Services Integration Platform	
Collaborate Edition	A single platform for total business integration provides a comprehensive set of integration solutions for process collaboration both inside and outside the enterprise, including Web services and ebXML.
Partner Edition	This product provides a low-cost, easy-to-deploy Web services connector that enables customers to interact seamlessly with each other and with trading partners that have deployed the Collaborate Edition or other Web services integration technologies.
XMLBus Edition	A comprehensive visual environment for building, deploying, integrating, and managing Web services, including Web services integration broker options.
Application Server Platform	
Enterprise Edition	This product delivers a complete enterprise deployment platform for J2EE, CORBA, Web services, mainframe environments, and Microsoft's .NET.

Table 8-7. The IONA E2A Platforms

Product	Description
Standard Edition	This product combines the scalability and robustness of CORBA, the developer productivity of J2EE, and the open access of Web services.
J2EE Technology Edition	This product combines an extremely reliable and scalable, fully certified J2EE application server with integrated Web Services support.

Beyond these functional areas, the qualities of security and reliable messaging are critical and should be settled as soon as possible. Work on these areas can progress in parallel with the work on process flow and transaction coordination, because both security and transactions are necessary for successful process-flow deployment.

Security, transactions, and reliable messaging are also important

Strategy

E2A is driven by two simple but profound ideas. First, any application, whether firmly entrenched or newly developed, should be able to interact with any other relevant application, no matter where it resides or how it was developed. Second, application development and application integration should not be treated as separate processes. In a fully connected world, they must be part of the same process.

E2A moves far beyond end-to-end—the idea that business processes would progress unimpeded between internal applications and external business partners, regardless of platform, language, database, application, transport mechanism or anything else. For most organizations, end-to-end meant point-to-point, with a lot of points joined by too many hard-wired, inflexible connections. Each connection took too long to build, cost too much to maintain, and was too inflexible to allow for business process changes. In the end, it didn't account for how quickly the world changes. End-to-end has failed to deliver on the promise of total business integration.

Web services standards represent a breakthrough that has markedly changed this equation. Web services standards are making the process of total business integration far more cost-effective and far more flexible. Many vendors have taken tentative initial steps toward addressing the potential of Web services. IONA is going much further, combining Web services standards with its extensive integration technology. IONA delivers a solution, Orbix E2A, that revolutionizes enterprise integration by leveraging Web services at its core.

Microsoft

Much was covered in the .NET section, and .NET is broader than Web services. Microsoft's approach is to make Web services a fundamental platform technology used by multiple products. (Microsoft products are built assuming the existence of Web services, and Microsoft is in the middle of this process; some products have been adapted, and some have not.) Like many previous Microsoft initiatives, .NET has seen strong uptake, with millions of developers and thousands of customers jumping on board.

Web services are fundamental

Products

The current products that provide implementations of the core Web services technologies—SOAP, WSDL, and UDDI—are .NET, the .NET Framework, and Visual Studio .NET, as described in Table 8-8.

Web services products are part of .NET

Next Steps

The Global XML Web Services Architecture (GXA; specifications are covered in Chapter 7) is the framework for the future of XML Web services. SOAP, WSDL, and UDDI constitute a set of baseline specifications for application integration and aggregation. From these baseline specifications, companies are building real solutions and getting real value. But as they develop XML Web services, companies' solutions have become more complex and the need for standards beyond this baseline are readily apparent. The baseline specifications leave a gap that requires Web services developers to implement higher-level functionality, such as security, routing, reliable messaging, and transactions in proprietary and often noninteroperable ways.

Microsoft's GXA proposal

At the W3C Workshop on Web Services in April 2001, Microsoft and IBM presented an architectural sketch for the evolution of Web services that builds on the baseline specifications. This sketch (available at www.w3.org/2001/03/wsws-popa/paper51) was the forerunner of Microsoft's GXA, which provides principles, specifications, and guidelines for advancing the protocols of today's XML Web services standards to address more complex and sophisticated tasks in standard ways, allowing XML Web services to continue to meet customer needs as the fabric of application internetworking.

Table 8-8. Microsoft's Current Web Services Products

Product	Description
.NET	Microsoft .NET is a platform for building, running, and experiencing the next generation of distributed applications. It spans clients, servers and services and consists of: • A programming model that enables developers to build XML Web services and applications • A set of XML Web services, such as Microsoft .NET My Services that helps developers deliver a simple and integrated user experience • A set of servers, including Windows 2000, SQL Server, and BizTalk Server, that integrates, runs, operates, and manages XML Web services and applications • Client software, such as Windows XP and Windows CE, that helps developers deliver a deep and compelling user experience across a family of devices • Tools, such as Visual Studio .NET, to develop XML Web services and applications for Windows and the Web for a rich and compelling user experience

Table 8-8. Microsoft's Current Web Services Products

Product	Description
.NET Framework	This Microsoft framework is a platform for building, deploying, and running Web services and applications. It provides a highly productive, standards-based, multilanguage environment for integrating existing investments with next-generation applications and services and the agility to solve the challenges of deployment and operation of Internet-scale applications. The .NET Framework consists of three main parts: (1) the common language runtime, (2) a hierarchical set of unified class libraries, and (3) a componentized version of Active Server Pages called ASP .NET.
Visual Studio .NET	This product enables developers to write robust and dependable software quickly and delivers on the promise to address the fundamental challenges facing developers and their organizations today. A powerful, highly productive, and extensible programming environment, Visual Studio .NET unlocks the potential for application development. It provides the tools and technologies required to build applications that will power today's organizations and drive the next generation of XML Web service-based software.

Future View

The demands of today's dynamic business environment have pushed the limits of the current computing paradigm. Remaining competitive and nimble requires that information and processes be more connected and tailored to people and systems, both inside and outside the organization. The most pressing technical challenge facing companies today lies in the integration of business applications and electronic processes so that they work well to meet business needs: streamlined operations, more accurate and timely information, and better-served customers and business partners.

The global adoption of the Internet provides us with the network that we need to connect our systems, both inside individual businesses and among businesses, their suppliers, and their customers. The Internet has the potential to act as a universal, inexpensive network that can connect the applications within the business process and in people's everyday lives.

Merely having the network in place is not sufficient, however, if the software on which we build our businesses does not utilize the network's power and potential. Although the network is available, the challenge of integration remains. Today's standalone systems, applications, and Web sites create "islands" of data and functionality—typically locked inside centralized databases. Integrating these islands of information with existing systems, as well as with those of partners and customers, has traditionally been difficult, expensive, and inflexible.

In response to these challenges, the computing industry is converging on a new model for building software to make this integrated world of applications for businesses and individuals much more achievable. The model enables a standard way to connect applications and exchange information using the Internet. This new, Internet-based integration methodology, called XML Web services, enables applications, machines, and business processes to work together in ways never previously possible.

XML Web services, in conjunction with the Internet, provide the foundation to overcome these challenges. The baseline of XML Web services is in place, with wide vendor adoption and a high level of interoperability. The challenge we now face is building out the enhanced capabilities of XML Web services demanded by business in a way that keeps XML Web services interoperable and open. Microsoft's Global XML Web Services architecture provides the vision for that evolution.

Oracle

Oracle is primarily a database vendor but is also entering and competing in the application server market and will be offering Web service products based on its application server and its database. This view revives the old debate between client/server and three-tier database access; nonetheless, many developers will find directly connecting Web services to the database to be a good fit for simple, data-centric applications. Oracle was among several vendors that endorsed SOAP and the other specifications around the same time Sun Microsystems did. Oracle offers predefined Web services types to make it easier to build and produce high-performance Web services applications. A fundamental application of Web services is to provide a standard interface to database management systems, and Oracle provides this.

Both the database and the application server support Web services

Basic Information

Oracle9i Application Server (AS) Web Services, Oracle's J2EE-based Web services offering, has five distinguishing features, as shown and described in Table 8-9.

Next Steps

These following technologies will play important roles: reliable messaging, security, process flow, and transactions. In the foreseeable future, Oracle9iAS Web Services plans to fully support and implement the final specifications of the following JSRs (see also J2EE Application Servers section) or their replacements:

- JSR 109: Implementing Enterprise Web Services
- JSR 101: JAX RPC

Additional technologies are adopted through JSRs

Table 8-9. Oracle's J2EE-Based Web Services

Service	Description
Service Implementation	Oracle9iAS Web Services are implemented as any of the following: • Java classes, both stateless and stateful • Enterprise JavaBeans (EJBs), currently stateless session beans • Message-driven beans (MDBs) • PL/SQL stored procedures or functions
Service Packaging	Oracle9iAS Web Services are packaged as standard J2EE Enterprise Archive (EAR) files, including the service implementation and a servlet adapter corresponding to the specific implementation type. For scalability, Oracle9iAS supplies the following five servlet classes, one for each supported implementation type: • Java class (stateless)—the object implementing the Web service is any arbitrary Java class. • Java class (stateful)—the object implementing the Web service is any

Table 8-9. Oracle's J2EE-Based Web Services

Service	Description
	arbitrary Java class; a servlet `HttpSession` maintains the object state between requests from the same client. • EJBs—the object implementing the Web service is a stateless session EJB; the Web service is considered to be stateless. • PL/SQL stored procedure or function—the object implementing the Web service is a Java class that accesses the PL/SQL stored procedure or function; the Web service is considered to be stateless. The Oracle9iAS Web Services development commands generate the Java access class. • MDB—the object implementing the Web service is a message-driven bean; the Web service is considered to be stateless.
Publishing	This service provides automatic generation of a WSDL file and WSDL-compliant stubs. Oracle Enterprise Manager allows you to register the specific Web service and to publish its WSDL to the UDDI registry and to discover published Web services.
UDDI Registry Streaming	This service is implemented on top of Oracle9i Database and uses the following three standard classification taxonomies: • North American Industry Classification System (NAICS) • Universal Standard Products and Services Codes (UNSPSC) • ISO 3166 Geographic Taxonomy (ISO 3166)
HTML/XML	This service allows HTML/XML streams via HTTP for • Access to an XML news feed through a static URL • Programmatic access to a dynamic stream accessed through an HTML form • EJB methods for accessing and processing the XML or HTML streams

- JSR 110: JWSDL
- JSR 67: JAX M
- JSR 93: JAX R
- JSR 151: J2EE v1.4
- JSR 153: EJB v2.1

Sun Microsystems

Sun Microsystems joined Web services initiatives later than IBM and IONA and was initially more focused on ebXML. Sun participated in the W3C forum on XML Protocols in June 2000 and endorsed SOAP soon afterward, which made industry support for SOAP nearly unanimous. The company then supported several Web services-related submissions to W3C, began work on its SunONE architecture, and remains very active at both ebXML and W3C. Sun Microsystems is also leading many of the Java Community Process initiatives for Web services.

Sun Microsystems's endorsement removed final doubts about SOAP

SunONE

The Sun Microsystems's Open Network Environment (SunONE) provides a framework for all the elements necessary for defining, building, deploying, and operating sophisticated business services, including, but not limited to, Web services.

SunONE is an architecture for Web services and more

SunONE addresses the entire range of network-driven services and services on demand, regardless of which networks the service uses—phone network, Internet, wireless—either alone or in combination. The SunONE architecture is standards-based, affording customers interoperability with other vendors' products that support the same standards. SunONE products are based on the Java programming languages technology.

The Java Community Process is open and based on standards. It therefore engenders a competitive, compatible market where multiple vendors vie to produce the best implementations. History clearly shows that competitive markets produce better products. The key element is that choice creates competition and that a competitive, compatible market gives vendors the incentive to build superior products.

Java and standards-based solutions promote choice

In order to build complex and robust business applications, you must have all of the elements in the SunONE architecture: service creation and assembly, service delivery, applications and Web services, a service container, service integration, identity and policy, and a foundation platform on which to run it all.

SunONE supports complex and robust applications

- For service creation and assembly, Forte and iPlanet tools are available.
- For service delivery, the iPlanet portal server (Collaboration, Mobile Access, Personalized Knowledge, Secure Remote Acess Packs) is popular.
- For ready-to-use applications and Web services, various products from the iPlanet e-commerce (BillerXpert, BuyerXpert, MarketMaker, SellerXpert, Trustbase) and communication portpolios (Calendar, Messaging, Mail) can be used.
- For the container, iPlanet also has a Web server and an industrial-strength application server.
- For service integration, an iPlanet integration server can be used to communcate with back-end systems.
- For identity and policy, there is the iPlanet Certfication Management System, Directory Server, Proxy Server, Policy Server, and Liberty Alliance Project.

All these elements run over the proven, scalable, and reliable Solaris OS on a SPARC platform.

Future Direction

To date, much of the discussion at Sun Microsystems has focused on the low-level protocols and technology for implementing Web services. But, over the next two to three years, the lower level will fade into the background, and the upper level will take its rightful prominence. The upper level is focused on business processes and solutions that use Web services to provide increased value to the stakeholders involved in those systems. What customers are increasingly and

unequivocally saying is that they are interested in solutions that help them to provide value to their stakeholders while achieving a return on their technology investments.

Consequently, the emphasis will increasingly be on initiatives that provide these kinds of business solutions. Two kinds of intiatives directly address business solutions of interest to customers: explicit business implementation technology and formal frameworks for business technology solutions. An example of the former is the OASIS Universal Business Language (UBL) effort to define a common XML business document library (see http://oasis-open.org/committees/ubl/). UBL will provide a set of XML building blocks that will enable trading partners to unambiguously identify and exchange business documents in specific contexts. An example of the latter is the SunONE architecture and products.

Systinet

Systinet, formerly Idoox, is a software company that was founded by Roman Stanek, formerly of NetBeans. Systinet was an early adopter of Web services and has been very active at W3C, JCP, UDDI, and on OASIS committees. Systinet focuses entirely on Web services and has no other product line.

Adoption of Core Technologies

There's no doubt that SOAP and WSDL will be widely accepted and adopted. UDDI is taking a little longer to garner support but will eventually succeed. Systinet does not see very much interest in ebXML but expects SAML and XKMS to gain adoption.

Current Products

The Systinet Web Applications and Services Platform (WASP) product line supports SOAP, WSDL, and UDDI. Table 8-10 describes the four families of the Systinet product line.

Next Steps

Security is the most critical next step. Security requires interoperability at the application level, so standards are necessary. SAML is key and is likely to experience rapid adoption. Systinet is already implementing support for SAML in WASP Security. It also needs an open, standardized, federated identity system for universal single sign-on—probably that will be Liberty.

Reliable messaging is very important, but it doesn't *require* standardization because it can be implemented at the transport protocol level. Systinet supports reliable messaging today by using a reliable transport protocol: for example, JMS or a reliable network provider such as Grand Central or Slam Dunk. It might be useful to have a standard mechanism or language to express reliability requirements so that consumers of the service could discover these requirements from a Web service description. But it's not a critical need at this time; nor has the industry reached consensus on any potential standard. IBM's HTTPR fell pretty flat. (Besides, a reliability standard shouldn't be dependent on the transport protocol layer; we already have that.)

Routing is pretty important and will become more so as people try to do more interesting applications with Web services. Routing and intermediaries will be used to implement metering, billing, entitlement, auditing, reliability, security, transcoding, filtering, optimization, and so on. Standardization isn't making much progress in this area, however.

Table 8-10. The Systinet WASP Products

Product	Description
WASP Developer	Provides an integrated Web services development environment implemented as a plug-in to popular Java IDEs. The plug-in adds tools, wizards, and

Table 8-10. The Systinet WASP Products

Product	Description
	generators to make Web services development a natural part of application development and is available for Forte, Jbuilder, and Eclipse. The tools can generate WSDL from Java and Java from WSDL. They fully automate the development of SOAP interfaces and of deployment packages. Developers can deploy Web services from within their IDE, to either a local or a remote server. An integrated administration console runs within the IDE. The tools include a distributed debugger and a SOAP message monitor. The tools can also query and publish to UDDI.
WASP Server	Provides Java and C++ runtime Web services containers that can be deployed in a variety of Web servers and application servers (Apache, Tomcat, WebLogic, WebSphere, iPlanet, Orion, JBoss, and so on). The framework provides numerous typed interception points that enable easy customization to support multiple transport protocols—HTTPS, SMTP, JMS, and so on—as well as custom message transcoding, envelope processing, header processing, encoding, and serialization. Both containers support an advanced security framework that provides end-to-end security protection. It supports multiple authentication mechanisms, trusted domains, and credential delegation and enables service- and method-level authorization. Both containers support remote references and an asynchronous API for solicit/response and notification behaviors. WASP Server for Java supports tight integration with J2EE. The J2EE framework enables SOAP integration with any JNDI-registered resource. The containers include client and server libraries, a client library for JavaScript, and a set of command-line tools for all those Spartan developers who insist on developing with vi/Emacs/NotePad.
WASP UDDI	A UDDI registry service implementation intended for use as a private registry within a corporation or a community. WASP UDDI, developed using WASP Server for Java and Tomcat, supports a variety of registry data stores, including Oracle, DB2, Sybase, SQL Server, PostgreSQL, PointBase, and Cloudscape. WASP UDDI is available as either a WASP UDDI Standard, a straight implementation of the UDDI v2 specification or WASP UDDI Enterprise, which extends the UDDI v2 specification, adding advanced security and query features. WASP UDDI includes a Java client library that simplifies programmatic access to any UDDI registry. The client library maps the UDDI data structures and SOAP messages to Java objects and provides a simple RMI-style programming interface. (This library is also included in WASP Developer and WASP Server.)
WASP Security	Provides single sign-on authentication and authorization services. These servers will eventually be bundled with WASP Server. WASP Card is a single sign-on service that supports a variety of authentication methods, such as certificates, Kerberos, LDAP, and so on). Once authenticated, a user receives an authentication token that can be used to access any Web service that supports the authorization token. Systinet's plan is to make WASP Card compliant with Liberty once the specifications have been published. WASP Guard is an authorization service. Both services are accessed through open SOAP APIs. All security information is exchanged using SAML.

Process flow will take much longer. All process flow (orchestration) efforts, such as XLANG, WSFL, Business Process Modeling Language (BPML, from BPMI at www.BPMI.org), and so on, are so complex that few people will use them, although ebXML choreography has a chance. It's all about establishing the rules of engagement between two companies, and it doesn't attempt to use workflow engines to automatically execute a long-term process. According to Systinet, no process flow standard will happen for at least a couple of years.

Systinet believes that loosely coupled transaction management will follow a path similar to that of process flow. Fortunately, there appears to be only one standard effort in this space: BTP. Anyone who wants to implement loosely coupled transactions will use BTP; but Systinet thinks very few people will implement loosely coupled transactions in the near future.

Others

Many companies in the Web services space, somewhat similar to Microsoft's .NET initiative, view Web services as an important technology in its own right. These companies create products based on Web services, either starting with the Web service itself and generating stubs and proxies for adaptation to existing systems or deriving Web service interfaces from existing systems.

The market for these products remains to be validated, as part of the value proposition of Web services is that they are based on open standards that anyone can implement. Microsoft, for one, is providing the basic Web services infrastructure as a commodity part of its operating system. Assuming that Web services go the way of the Web browser and are accepted and adopted as widely, companies focusing solely on Web services may have a tough road. Also, Web services on their own are not executable and to be useful, depend on underlying infrastructure. The infrastructure vendors are providing Web service-enabled versions of their products, so why buy the Web services layer from someone else?

Shinka Systems, for example, takes the view that Web services definitions are a starting point rather than an existing program or database. Its tools define data types and messages for Web services, map them to WSDL, and generate stubs and proxies from the WSDL files. Programs are then fit to the stubs and proxies to provide the implementations behind the Web service definitions.

Implementations of ebXML

Several vendors, including BEA, HP, IBM, IONA, Oracle, and Sun Microsystems, are offering ebXML implementations. The typical progression of implementation is to provide the transport first, using SOAP with Attachments; then to provide the registry, followed by the BPSS, CPP and CPA support; and, finally, to provide the core specifications and modeling tools. However, some vendors are starting at different points in the architecture.

Most ebXML implementations will be delivered over time. Microsoft has clearly stated that it does not intend to implement ebXML. A Web services interface can easily be created for ebXML, and several of the vendors are providing this capability. A list of vendors providing ebXML solutions can be found at http://www.ebxml.org/implementations/index.htm.

Summary

Software vendors provide Web services implementations in a variety of architectures, typically depending on how they apportion value to Web services relative to existing products. Web services provide interfaces to application servers, .NET servers, object request brokers, message-oriented middleware, database management systems, and packaged applications. In addition, some vendors' products are focused on the value of the Web services standards themselves, providing interoperability across various software domains, new mechanisms for assembling applications, and orchestrations for placing multiple Web services interfaces into different combinations.

Vendors are nearly unanimous in their support for the core standards—SOAP, WSDL, and UDDI—but vary in their support for additional technologies. It is agreed that security is an essential next step, but opinions vary regarding the relative priority of transactions, process flow, and reliable messaging proposals.

With respect to ebXML, some vendors provide implementations and others do not, although, often, it isn't clear what a vendor means by ebXML support—whether it's just the transport (that is, SOAP with Attachments) or any of the specified additional qualities of service, registry, and metadata technologies. Opinion among vendors regarding additional technologies and ebXML is likely to be influenced by commercial interests, inasmuch as vendors tend to support technologies closely related to their products.

A wide variety of products is available from large and small vendors, providing considerable choice to consumers. The basic development and deployment choice, however, remains between Microsoft's .NET and the Java community's J2EE platforms.

Visions of the future direction vary also by vendor, with some vendors providing very forward-looking statements in terms of the impact of Web services on electronic commerce, and others sticking to more of a technology-centric view of where things are going. In any case, it is certainly clear that software products are significantly impacted by Web services implementations, and that Web services hold tremendous potential for change.

Bibliography

Because Web services technologies and ebXML are relatively new, most primary sources of information for this book were specifications, which can be found at the various Web sites listed here.

Books

Bernstein, Philip A., and Eric Newcomer *Principles of Transaction Processing* New York: Morgan Kaufmann, 1997

Box, Don, Aaron Skonnard, and John Lam *Essential XML: Beyond Markup* (The DevelopMentor Series) Boston: Addison-Wesley, 2000

Castro, Elizabeth *XML for the World Wide Web: Visual QuickStart Guide* Boston: Peachpit Press, 2000

Chappel, David *Understanding .NET: A Tutorial and Analysis* Boston: Addison-Wesley, 2002

Harold, Elliotte Rusty, and W. Scott Means *XML in a Nutshell: A Desktop Quick Reference* Sebastapol, CA: O'Reilly & Associates, 2000

Kay, Michael H.. *XSLT Programmer's Reference, Second Edition* Chicago: Wrox Press, 2001

St. Laurent, Simon, Edd Dumbill, and Joe Johnston *Programming Web Services with XML-RPC* (O'Reilly Internet Series) Sebastapol, CA: O'Reilly & Associates, 2001

Articles and White Papers

Berners-Lee, Tim "A Roadmap to the Semantic Web" (September 1998,) and other whitepapers http://www.w3.org/DesignIssues/

Ceponkus, Alex "XML Web Services" Greater Boston Chapter ACM Professional Development Seminar (March 2001) http://www.gbcacm.org/website/semInfo/-php?id=1

Conklin, Peter, and Eric Newcomer, "The Key to the Highway," in *The Future of Software* by Derek Leebaert (Editor) Cambridge: MIT Press, 1996

Curbera, Francisco, and David Ehnebuske, IBM; and Dan Rogers, Microsoft "Using WSDL in a UDDI Registry 1.05"—UDDI Working Draft Best Practices Document (June 25, 2001) http://www.uddi.org/pubs/wsdlbestpractices-V1.05-Open-20010625.pdf

Drucker, Peter F."Beyond the Information Revolution" *Atlantic Monthly* Online edition (October 1999) http://www.theatlantic.com/issues/99oct/9910drucker.htm

Forum 2000 "Remarks by Bill Gates" Redmond, WA (June 22, 2000) http://www.microsoft.com/billgates/speeches/2000/06-22f2k.asp

Gudgin, Martin, and Timothy Ewald (December 19, 2001) "All We Want for Christmas Is a WSDL Working Group" http://www.xml.com/pub/a/2001/12/19/wsdlwg.html

IONA Technologies PLC (January 2002) "Orbix E2A XMLBus Edition Technology Overview" http://www.xmlbus.com/learn/webserviceswp.pdf

Kirtland, Mary "A Platform for Web Services" Microsoft Developer Network (MSDN) site (January 2001) http://msdn.microsoft.com/default.asp

Shohoud, Yasser "Introduction to WSDL" http://www.devxpert.com/tutors/wsdl/wsdl.asp

Snell, James, Software Engineer, IBM Emerging Technologies (April 2001) "The Web Services Insider, Part 2: A Summary of the W3C Web Services Workshop" http://www-106.ibm.com/developerworks/webservices/library/ws-ref2/

Specifications

ebXML Specifications: http://www.ebxml.org/specs/index.htm#technical_specifications.
• Technical Architecture
• Business Process and Information Modeling
• Collaboration-Protocol Profile and Agreement Specification
• Registry and Repository Services
• Messaging Service
• Core Components and Core Library
• Using UDDI with ebXML: http://www.ebxml.org/specs/rrUDDI.pdf

IBM Specifications: http://alphaworks.ibm.com/
• WSFL: http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
• WS-Inspection: http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html
• HTTPR: http://www-106.ibm.com/developerworks/webservices/library/ws-phtt/httprspecV2.pdf

IETF Specifications: www.ietf.org
• SSL/TLS: http://www.ietf.org/ids.by.wg/tls.html
• TIP: http://www.ietf.org/rfc/rfc2371.txt?number=2371
• BEEP: http://www.ietf.org/rfc/rfc3080

Java Specifications: http://java.sun.com/webservices/

Microsoft Specifications: http://msdn.microsoft.com/default.asp
• WS-License: http://msdn.microsoft.com/ws/2001/10/License/
• WS-Security: http://msdn.microsoft.com/ws/2001/10/Security/
• WS-Inspection: http://msdn.microsoft.com/ws/2001/10/Inspection/
• XLANG: http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

Miscellaneous:
• SOAP over BEEP: http://clipcode.org/beep/soap
• Open source .NET: http://www.ximian.com/devzone/projects/mono.html
• IONA Web services information: www.xmlbus.com/learn

OASIS Specifications: http://www.oasis-open.org
• BTP: www.oasis-open.org/committees/business-transactions/index.shtml
• OASIS/ebXML Registry Services (RS) Specification v2.0: http://lists.oasis-open.org/archives/members/200201/msg00002.html
• OASIS/ebXML Registry Information Model (RIM) v2.0: http://lists.oasis-open.org/archives/members/200201/msg00003.html
• ebXML Messaging: http://www.oasis-open.org/committees/ebxml-msg/#documents
• ebXML Collaboration Protocol Profile and Agreement (CPA): http://www.oasis-open.org/committees/ebxml-cppa/
• eXtensible Access Control Markup Language: http://www.oasis-open.org/committees/xacml/

OMG Specifications: www.omg.org
• *Additional Structuring Mechanisms for the OTS Specification* (September 2000), document *orbos/2000-04-02*

RosettaNet Specifications: www.rosettanet.org

UDDI Specifications: http://www.uddi.org/specification.html
• UDDI Version 2.0 Programmer's API Specification
• UDDI Version 2.0 Data Structure Specification
• UDDI Version 2.0 XML Schema
• UDDI Version 2.0 Replication Specification
• UDDI Version 2.0 XML Replication Schema
• UDDI Version 2.0 XML Custody Schema
• UDDI Version 2.0 Operator's Specification

W3C Specifications (including XML, SOAP, and WSDL)

• Main page: http://www.w3.org/TR/
• The generic XML homepage: http://www.w3.org/XML/
• Extensible Markup Language 1.0: http://www.w3.org/TR/REC-xml
• XML Base: http://www.w3.org/TR/xmlbase/
• XML Names: http://www.w3.org/TR/REC-xml-names/
• DOM: http://www.w3.org/TR/DOM-Level-2-Core/
• XML Schema: http://www.w3.org/TR/xmlschema-1/ and http://www.w3.org/TR/xmlschema-2/
• XML Path Language: http://www.w3.org/TR/xpath
• XML Pointer Language: http://www.w3.org/TR/WD-xptr
• XML Linking Language: http://www.w3.org/TR/xlink/
• XSL Transformation: http://www.w3.org/TR/xslt
• Canonical XML: http://www.w3.org/TR/xml-c14n
• XML Information Set: http://www.w3.org/TR/xml-infoset/
• XML Inclusions: http://www.w3.org/TR/xinclude/
• XML Query: http://www.w3.org/TR/xquery/
• XML in 10 points: http://www.w3.org/XML/1999/XML-in-10-points
• Clarification on URIs and URLs: http://www.w3.org/TR/uri-clarification/
• WSDL: http://www.w3.org/TR/wsdl
• SOAP v1.2: http://www.w3.org/TR/soap/12-part0, /12-part1, and /12-part2—for updates, see also http://www.w3.org/TR/soap
• SOAP v1.1: http://www.w3.org/TR/SOAP/
• SOAP with Attachments: http://www.w3.org/TR/SOAP-attachments
• XKMS: http://www.w3.org/TR/xkms/
• W3C Workshop on Web Services (April 2001): http://www.w3.org/2001/01/WSWS
• XML Digital Signature: http://www.w3.org/2000/09/xmldsig