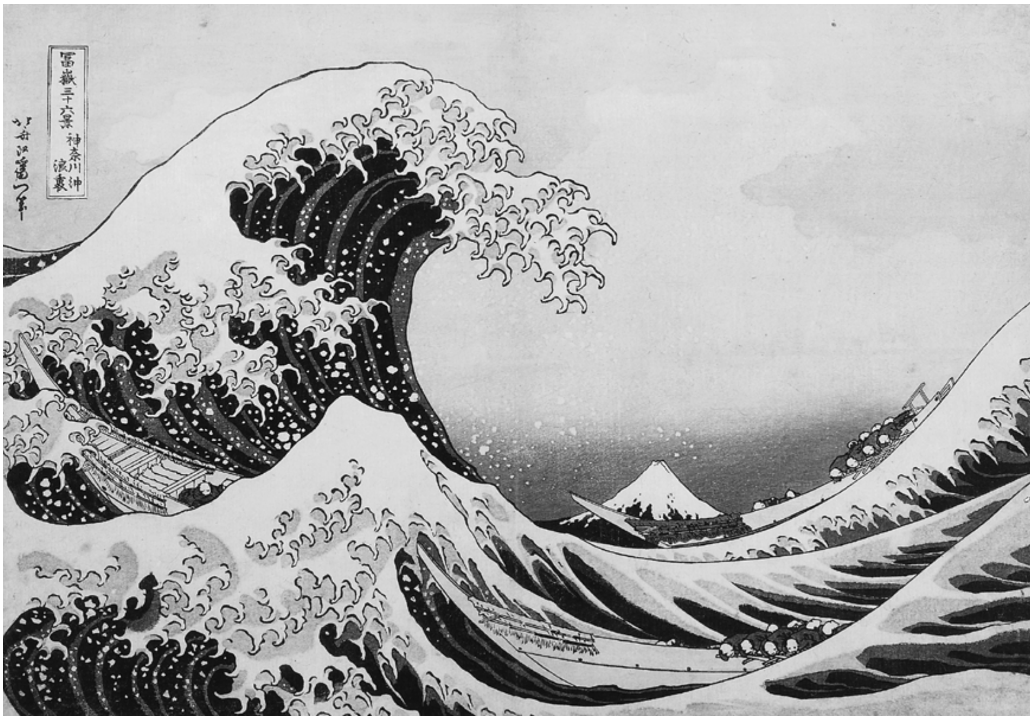

Part II

SOAP and WSDL

At the heart of Web services today are SOAP and WSDL, so it's important that you have a good understanding of them and how they're used. That said, memorizing the details of SOAP and WSDL is not critical. While these technologies are central to Web services, in many cases you may not deal with them directly, as they will be hidden in the communication and deployment layer of the J2EE Web Services platform.

This part of the book covers SOAP 1.1 and WSDL 1.1. Although support for SOAP 1.1 and WSDL 1.1 is required by the WS-I Basic Profile 1.0, support for

SOAP Messages with Attachments is not. SwA is a significant feature of Web services practice today, however, and it's supported by J2EE Web Services, as well as a future version of the BP, version 1.1, so it's covered in Appendix E.

Once you have read Part II, you should have a pretty decent understanding of SOAP 1.1 and WSDL 1.1. If you desire more detailed knowledge, I suggest you read the Notes describing these technologies, published by the World Wide Web Consortium. You must complement that reading with study of the Basic Profile, however, because the BP imposes lots of restrictions and provides many clarifications that make SOAP 1.1 more interoperable and WSDL 1.1 more portable. Still, for most developers the level of coverage in this part of the book will be more than sufficient.

In This Part

Related Appendices

Chapter 4

SOAP

SOAP was originally an acronym for Simple Object Access Protocol. (Now it's just a name.) SOAP 1.1 is the standard messaging protocol used by J2EE Web Services, and is the de facto standard for Web services in general. SOAP's primary application is Application-to-Application (A2A) communication. Specifically, it's used in Business-to-Business (B2B) and Enterprise Application Integration (EAI), which are two sides of the same coin: Both focus on integrating software applications and sharing data. To be truly effective in B2B and EAI, a protocol must be platform-independent, flexible, and based on standard, ubiquitous technologies. Unlike earlier B2B and EAI technologies, such as CORBA and EDI, SOAP meets these requirements, enjoys widespread use, and has been endorsed by most enterprise software vendors and major standards organizations (W3C, WS-I, OASIS, etc.).

Despite all the hoopla, however, SOAP is just another XML markup language accompanied by rules that dictate its use. SOAP has a clear purpose: exchanging data over networks. Specifically, it concerns itself with encapsulating and encoding XML data and defining the rules for transmitting and receiving that data. In a nutshell, SOAP is a network application protocol.

A SOAP XML document instance, which is called a **SOAP message,**[1] is usually carried as the payload of some other network protocol. For example, the most common way to exchange SOAP messages is via HTTP (HyperText Transfer Protocol), used by Web browsers to access HTML Web pages. The big difference is that you don't view SOAP messages with a browser as you do HTML. SOAP messages are exchanged between applications on a network and are not meant for human

[1] The SOAP XML document is also called the *SOAP envelope*.

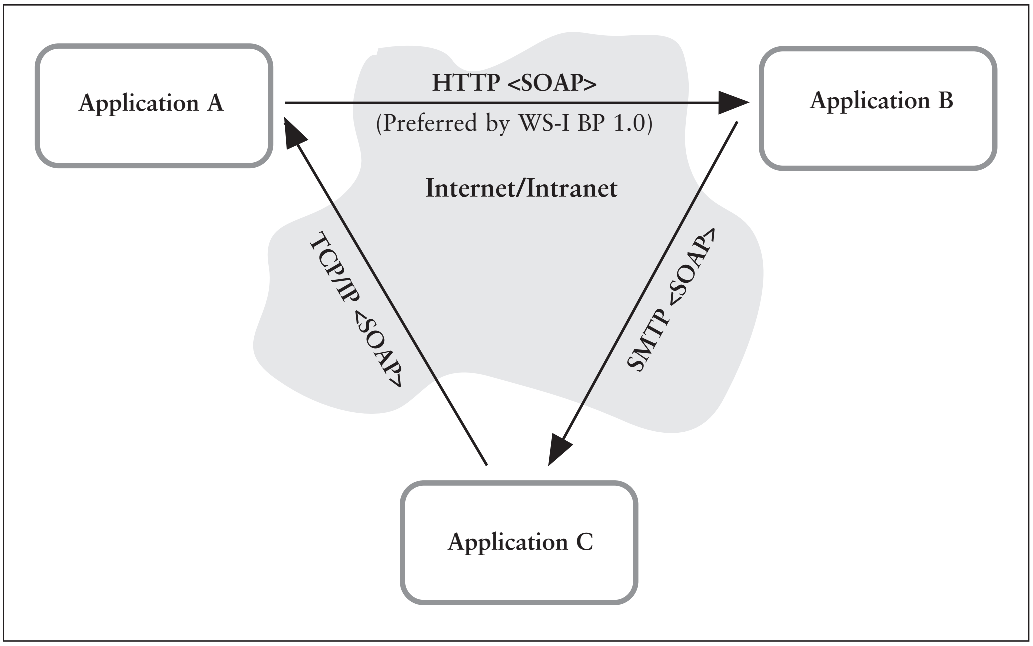

Figure 4–1 SOAP over HTTP, SMTP, and Raw TCP/IP

consumption. HTTP is just a convenient way of sending and receiving SOAP messages.

SOAP messages can also be carried by e-mail using SMTP (Simple Mail Transfer Protocol) and by other network protocols, such as FTP (File Transfer Protocol) and raw TCP/IP (Transmission Control Protocol/Internet Protocol). At this time, however, the WS-I Basic Profile 1.0 sanctions the use of SOAP only over HTTP. Figure 4–1 illustrates how SOAP can be carried by various protocols between software applications on a network.

Web services can use **One-Way messaging** or **Request/Response messaging.** In the former, SOAP messages travel in only one direction, from a sender to a receiver. In the latter, a SOAP message travels from the sender to the receiver, which is expected to send a reply back to the sender. Figure 4–2 illustrates these two forms of messaging.

SOAP defines how messages can be structured and processed by software in a way that is independent of any programming language or platform, and thus facilitates interoperability between applications written in different programming languages and running on different operating systems. Of course, this is nothing new: CORBA IIOP and DCE RPC also focused on cross-platform interoperability. These legacy protocols were never embraced by the software industry as a whole, however, so they never became pervasive technologies. SOAP, on the other hand, has enjoyed

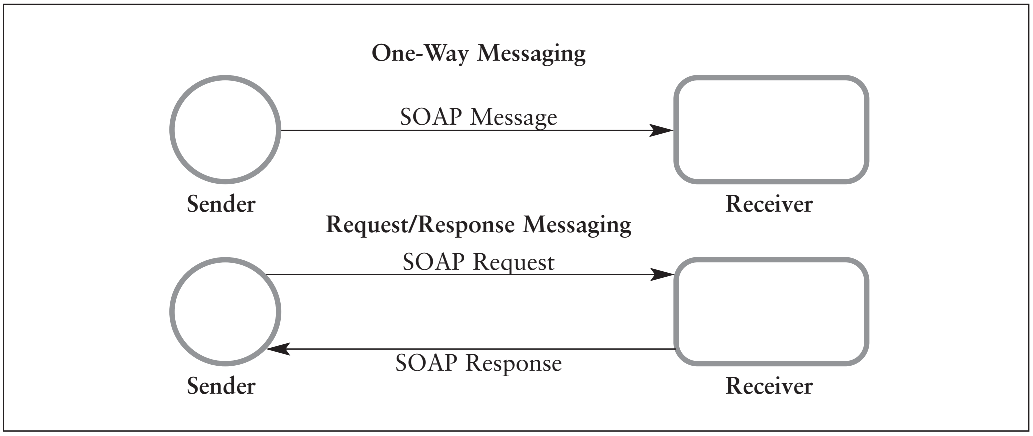

Figure 4–2 One-Way versus Request/Response Messaging

unprecedented acceptance, and adoption by virtually all the players in distributed computing, including Microsoft, IBM, Sun Microsystems, BEA, HP, Oracle, and SAP, to name a few.

The tidal wave of support behind SOAP is interesting. One of the main reasons is probably its grounding in XML. The SOAP message format is defined by an XML schema, which exploits XML namespaces to make SOAP very extensible. Another advantage of SOAP is its explicit definition of an HTTP binding, a standard method for **HTTP tunneling.** HTTP tunneling is the process of hiding another protocol inside HTTP messages in order to pass through a firewall unimpeded. Firewalls will usually allow HTTP traffic through port 80, but will restrict or prohibit the use of other protocols and ports.

> ***A port is a communication address on a computer that complements the Internet address. Each network application on a computer uses a different port to communicate. By convention, Web servers use port 80 for HTTP requests, but application servers can use any one of thousands of other ports.***

The power that comes from XML's extensibility and the convenience of using the ubiquitous, firewall-immune HTTP protocol partly explain SOAP's success. It's difficult to justify SOAP's success purely on its technical merits, which are good but less than perfect. Another factor in SOAP's success is the stature of its patrons. SOAP is the brainchild of Dave Winner, Don Box, and Bob Atkinson. Microsoft and IBM supported it early, which sent a strong signal to everyone else in the industry: "If you want to compete in this arena, you better jump aboard SOAP." The event that secured industry-wide support for SOAP was its publication by the World Wide Web

Consortium as a Note[2] in May of 2000, making it the de facto standard protocol for A2A messaging. Overnight, SOAP became the darling of distributed computing and started the biggest technology shift since the introduction of Java in 1995 and XML in 1998. SOAP is the cornerstone of what most people think of as Web services today, and will be for a long time to come.

Recently, the W3C has defined a successor to SOAP 1.1. SOAP 1.2 does a decent job of tightening up the SOAP processing rules and makes a number of changes that will improve interoperability. SOAP 1.2 is very new and has not yet been widely adopted, however, so it's not included in the WS-I Basic Profile 1.0. This exclusion is bound to end when the BP is updated, but for now J2EE 1.4 Web Services, which adheres to the WS-I Basic Profile 1.0, does not support the use of SOAP 1.2.

4.1 The Basic Structure of SOAP

As you now know, a SOAP message is a kind of XML document. SOAP has its own XML schema, namespaces, and processing rules. This section focuses on the structure of SOAP messages and the rules for creating and processing them.

A SOAP message is analogous to an envelope used in traditional postal service. Just as a paper envelope contains a letter, a SOAP message contains XML data. For example, a SOAP message could enclose a `purchaseOrder` element, as in Listing 4–1. Notice that XML namespaces are used to keep SOAP-specific elements separate from `purchaseOrder` elements—the SOAP elements are shown in bold.

Listing 4–1

A SOAP Message That Contains an Instance of Purchase Order Markup

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <po:purchaseOrder orderDate="2003-09-22"
     xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO">
      <po:accountName>Amazon.com</po:accountName>
      <po:accountNumber>923</po:accountNumber>
      <po:address>
        <po:name>AMAZON.COM</po:name>
        <po:street>1850 Mercer Drive</po:street>
        <po:city>Lexington</po:city>
```

[2] In the W3C standardization process, a Note does not represent commitment by the W3C to pursue work related to the technology it describes, but the W3C has taken responsibility for SOAP 1.2 and is working to make it an official recommendation, which is the highest level of endorsement offered by the W3C.

```
        <po:state>KY</po:state>
        <po:zip>40511</po:zip>
      </po:address>
      <po:book>
        <po:title>J2EE Web Services</po:title>
        <po:quantity>300</po:quantity>
        <po:wholesale-price>24.99</po:wholesale-price>
      </po:book>
    </po:purchaseOrder>
  </soap:Body>
</soap:Envelope>
```

This message is an example of a SOAP message that contains an arbitrary XML element, the `purchaseOrder` element. In this case, the SOAP message will be One-Way; it will be sent from the initial sender to the ultimate receiver with no expectation of a reply. Monson-Haefel Books' retail customers will use this SOAP message to submit a purchase order, a request for a shipment of books. In this example, Amazon.com is ordering 300 copies of this book for sale on its Web site.

A SOAP message may have an XML declaration, which states the version of XML used and the encoding format, as shown in this snippet from Listing 4–1.

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

If an `xml` declaration is used, the version of XML must be 1.0 and the encoding must be either UTF-8 or UTF-16. If `encoding` is absent, the assumption is that the SOAP message is based on XML 1.0 and UTF-8. An XML declaration isn't mandatory. Web services are required to accept messages with or without them.[BP] (Remember that I said I'd use a superscript [BP] to signal a BP-conformance rule.)

Every XML document must have a root element, and in SOAP it's the `Envelope` element. `Envelope` may contain an optional `Header` element, and must contain a `Body` element. If you use a `Header` element, it must be the immediate child of the `Envelope` element, and precede the `Body` element. The `Body` element contains, in XML format, the actual application data being exchanged between applications. The `Body` element delimits the application-specific data. Listing 4–2 shows the structure of a SOAP message.

Listing 4–2

The Structure of a SOAP Message

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <!-- Header blocks go here -->
```

```
  </soap:Header>
  <soap:Body>
    <!-- Application data goes here -->
  </soap:Body>
</soap:Envelope>
```

A SOAP message adheres to the SOAP 1.1 XML schema, which requires that elements and attributes be fully qualified (use prefixes or default namespaces). A SOAP message may have a single `Body` element preceded, optionally, by one `Header` element. The `Envelope` element cannot contain any other children.

Because SOAP doesn't limit the type of XML data carried in the SOAP `Body`, SOAP messages are extremely flexible; they can exchange a wide spectrum of data. For example, the application data could be an arbitrary XML element like a `purchaseOrder`, or an element that maps to the arguments of a procedure call.

The `Header` element contains information about the message, in the form of one or more distinct XML elements, each of which describes some aspect or quality of service associated with the message. Figure 4–3 illustrates the structure of a basic SOAP message.

The `Header` element can contain XML elements that describe security credentials, transaction IDs, routing instructions, debugging information, payment tokens, or any other information about the message that is important in processing the data in the `Body` element.

For example, we may want to attach a unique identifier to every SOAP message, to be used for debugging and logging. Although unique identifiers are not an integral

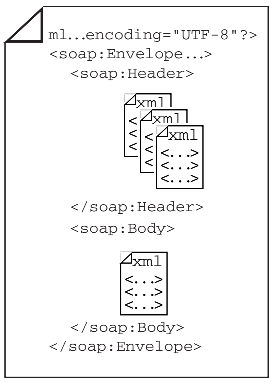

Figure 4–3 The Structure of a Basic SOAP Message

part of the SOAP protocol itself, we can easily add an identifier to the `Header` element as in Listing 4–3.

Listing 4–3

A SOAP Message with a Unique Identifier

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id" >
  <soap:Header>
    <mi:message-id>11d1def534ea:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
  </soap:Header>
  <soap:Body>
    <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

The `message-id` element is called a **header block,** and is an arbitrary XML element identified by its own namespace. A header block can be of any size and can be very extensive. For example, the header for an XML digital signature, shown in bold in Listing 4–4, is relatively complicated.

Listing 4–4

A SOAP Message with an XML Digital-Signature Header Block

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id">
  <soap:Header>
    <mi:message-id>11d1def534ea:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
    <sec:Signature >
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm=
           "http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
          <ds:SignatureMethod Algorithm=
           "http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
```

```
          <ds:Reference URI="#Body">
            <ds:Transforms>
              <ds:Transform Algorithm=
               "http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
            </ds:Transforms>
            <ds:DigestMethod Algorithm=
             "http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>u29dj93nnfksu937w93u8sjd9=
            </ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>CFFOMFCtVLrklR…</ds:SignatureValue>
      </ds:Signature>
    </sec:Signature>
  </soap:Header>
  <soap:Body sec:id="Body">
    <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

You can place any number of header blocks in the `Header` element. The example above contains both the `message-id` and XML digital signature header blocks, each of which would be processed by appropriate functions. Header blocks are discussed in more detail in Section 4.3.

4.2 SOAP Namespaces

XML namespaces play an important role in SOAP messages. A SOAP message may include several different XML elements in the `Header` and `Body` elements, and to avoid name collisions each of these elements should be identified by a unique namespace. For example, a SOAP message that contains the `purchaseOrder` element as well as `message-id` and XML digital-signature header blocks would include no fewer than six different namespaces, as shown in bold in Listing 4–5.

Listing 4–5

Using XML Namespaces in a SOAP Message

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
```

```
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id">
  <soap:Header>
    <mi:message-id>11d1def534ea:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
    <sec:Signature>
      …
    </sec:Signature>
  </soap:Header>
  <soap:Body sec:id="Body">
    <po:purchaseOrder orderDate="2003-09-22"
     xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     …
    </po:purchaseOrder>
  </soap:Body>
</soap:Envelope>
```

The use of XML namespaces is what makes SOAP such a flexible and extensible protocol. An XML namespace fully qualifies an element or attribute name, as you learned in Section 2.2. Because their use was discussed in detail there, the basic mechanics of XML namespaces aren't covered here.

Of the six namespaces declared in Listing 4–5, the first, declared in the `Envelope` element, defines the namespace of the standard SOAP elements—`Envelope`, `Header`, and `Body`—as shown in bold in the following snippet from Listing 4–5.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id">
 …
</soap:Envelope>
```

This namespace determines the version of SOAP used (1.1 at this point). SOAP messages must declare the namespace of the `Envelope` element to be the standard SOAP 1.1 envelope namespace, `"http://schemas.xmlsoap.org/soap/envelope/"`. If a SOAP application receives a message based on some other namespace, it must generate a fault. This rule ensures that all conforming messages are using exactly the same namespace and XML schema, and therefore the same processing rules.[BP]

The second, third, and fourth namespaces declared in the `Envelope` element are associated with XML elements in the header blocks:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id">
  <soap:Header>
    <mi:message-id>11d1def534ea:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
    <sec:Signature>
      <ds:Signature>
          …
      </ds:Signature>
    </sec:Signature>
  </soap:Header>
  <soap:Body>
    <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

Each header block in the `Header` element should have its own namespace. This is particularly important because namespaces help SOAP applications identify header blocks and process them separately. A variety of "standard" header blocks that address topics such as security, transactions, and other qualities of service are in development by several organizations, including W3C, OASIS, IETF, Microsoft, BEA, and IBM. All of the proposed standards define their own namespaces and XML schemas, as well as processing requirements—but none of these "standard" header blocks is addressed by J2EE 1.4 Web Services yet.

In Listing 4–5, two other namespaces are declared in the immediate child of the `Body` element, as shown in the following snippet. The first namespace declaration belongs to the Purchase Order Markup Language defined by Monson-Haefel Books (see Part I: XML).

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id" >
  <soap:Header>
    <!-- Header blocks go here -->
  </soap:Header>
  <soap:Body sec:id="Body">
    <po:purchaseOrder orderDate="2003-09-22"
     xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     …
    </po:purchaseOrder>
  </soap:Body>
</soap:Envelope>
```

All of the local elements of a SOAP message must be namespace-qualified (prefixed with the SOAP 1.1 namespace), because the XML schema for SOAP 1.1 specifies the `elementFormDefault` attribute as `"qualified"`. In addition, the Basic Profile 1.0 requires that all the application-specific elements contained by the `Body` element must be qualified.[BP] Unqualified elements in a SOAP `Body` element create too much ambiguity as to the meaning and proper structure of elements. (See Section 3.1.6 for details about qualified and unqualified local elements.)

The `xsi:schemaLocation` attribute (the attribute that provides the URL of the schema) may be declared for validation, but in most cases the **SOAP stack** will have handled this matter at design time, so that explicit declaration of the `xsi:schemaLocation` in a SOAP message is not necessary.

A SOAP stack is a library of code designed to process and transmit SOAP messages. For example, Apache Axis, J2EE 1.4, Perl::Lite, and Microsoft .NET all have their own SOAP stacks, their own libraries of code for processing SOAP messages.

Some SOAP stacks make extensive use of the XML schema-instance namespace to indicate the data types of elements (for example, `xsi:type = "xsd:float"`). Other SOAP stacks do not, though, which causes problems when the receiver expects elements to be typed but the sender doesn't type them. According to the BP, the `xsi:type` attribute must be used only to indicate that a derived XML type is being used in place of its base type—for example, a `USAddress` in place of an `Address`.[BP]

As you learned in Section 2.2, the real power of XML namespaces goes beyond simply avoiding name collisions, to proper versioning and processing. Using fully qualified names for the SOAP and application-specific data tells the SOAP receiver how to process the message, and which XML schemas to apply in order to validate its contents. Differences in a particular version of a header block, for example, can affect how a receiver processes messages, so identifying the header-block version by its namespace enables a receiver to switch processing models, or to reject messages if it doesn't support the specified version. Similarly, properly identifying the types of XML elements contained in the `Body` element enables a SOAP receiver either to process those elements using the appropriate code modules or possibly to reject the message if it doesn't support the specified namespace.

The term "code module" is used to express an aspect of computer code that performs some function. A code module may be a separate code library, a service, or simply a branch of logic within a larger set of code.

For example, if a new algorithm is used to generate the `message-id` header block, then the namespace of the `message-id` header could change to reflect the use of the new algorithm. The SOAP message in Listing 4–6 contains a `message-id` header block with a new namespace, which indicates that it's different from the `message-id` header block used in previous examples.

Listing 4–6

Changing the Namespace of a Header Block

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 xmlns:mi2="http://www.Monson-Haefel.com/jwsbook/message-id_version2/">
  <soap:Header>
    <mi2:message-id>1-203950-3485-30503453098</mi2:message-id>
    <sec:Signature>
      …
    </sec:Signature>
  </soap:Header>
  <soap:Body>
    <!-- Application-specific data goes here -->
  <soap:Body>
</soap:Envelope>
```

Namespaces enable a SOAP receiver to handle different versions of a SOAP message, without impairing backward compatibility or requiring different Web service endpoints for each version of a particular SOAP message.

As you can see from the previous examples, a SOAP message may contain many different namespaces, which makes SOAP messaging very modular. This modularity enables different parts of a SOAP message to be processed independently of other parts and to evolve separately. The version of the SOAP `Envelope` or header blocks may change over time, while the structure of the application-specific contents in the `Body` element remains the same. Similarly, the application-specific contents may change while the version of the SOAP message and the header blocks do not.

The modularity of SOAP messaging permits the code that processes the SOAP messages to be modular as well. The code that processes the element `Envelope` is independent of the code that processes the header blocks, which is independent of the code that processes application-specific data in the SOAP `Body` element. Modularity enables you to use different code libraries to process different parts of a SOAP message. Figure 4–4 shows the structure of a SOAP message and the code modules used to process each of its parts. The code modules in gray boxes are associated with

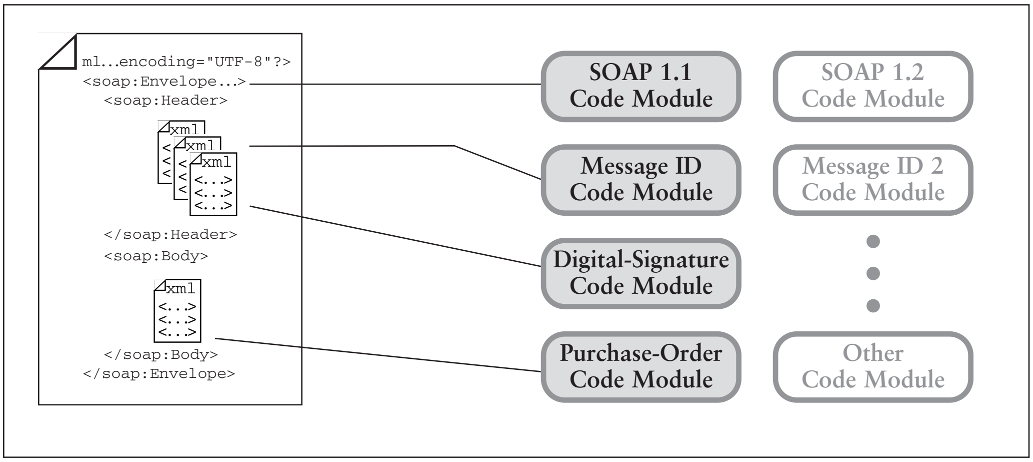

Figure 4–4 Using the Appropriate Code Modules with SOAP Namespaces

namespaces used in this SOAP message. The code modules in white boxes represent alternatives; they are associated with different namespaces, used to process alternative versions of the SOAP message.

In all the examples so far, the namespaces of the header blocks have been declared in the `Envelope` element. Doing so is not required; we could just as easily declare those namespaces in the `Header` element or the header blocks. As you learned in Section 2.2, namespaces are always locally scoped and can be declared at any level as long as the elements in question are within that scope (the element the namespace is declared in, and its subelements). For example, we could declare the header-block namespaces in the `Header` element, as shown in the boldface code lines in Listing 4–7.

Listing 4–7

Declaring XML Namespaces in a `Header` *Element*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Header
   xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
   xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
   xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id" >
    <mi:message-id>11d1def534:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
    <sec:Signature>
      …
```

```
    </sec:Signature>
  </soap:Header>
  <soap:Body>
    <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

We could also declare each namespace in its own header block as in Listing 4–8.

Listing 4–8

Declaring XML Namespaces in Header Blocks

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Header>
    <mi:message-id
      xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id" >
      11d1def534ea:b1c5fa:f3bfb4dcd7:-8000
    </mi:message-id>
    <sec:Signature
      xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
      xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      …
    </sec:Signature>
  </soap:Header>
  <soap:Body>
    <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

Although application-specific elements in the `Body` element must be qualified by prefixes, there is no such requirement for the elements contained within a `Header` element. Local elements of header blocks may be qualified or unqualified.

The way to declare namespaces is really a matter of style. As long as you adhere to the conventions and limitations of namespace declarations as they're described in the W3C *Namespaces in XML* recommendation,[3] you can use any style you wish. Table 4–1 shows the namespace prefixes used in this book and in the WS-I Basic Profile 1.0.

[3] World Wide Web Consortium, *Namespaces in XML*, W3C Recommendation, 1999. Available at http://www.w3.org/TR/REC-xml-names/.

Table 4–1 Namespace Prefixes

Prefix	Namespace
`soap`	`"http://schemas.xmlsoap.org/soap/envelope/"`
`xsi`	`"http://www.w3.org/2001/XMLSchema-instance"`
`xsd`	`"http://www.w3.org/2001/XMLSchema"`
`soapenc`	`"http://schemas.xmlsoap.org/soap/encoding/"`
`wsdl`	`"http://schemas.xmlsoap.org/wsdl/"`
`soapbind`	`"http://schemas.xmlsoap.org/wsdl/soap/"`
`wsi`	`"http://ws-i.org/schemas/conformanceClaim/"`

4.3 SOAP Headers

The SOAP specification defines rules by which header blocks must be processed in the **message path.** The message path is simply the route that a SOAP message takes from the initial sender to the ultimate receiver. It includes processing by any intermediaries. The SOAP rules specify which nodes must process particular header blocks and what should be done with header blocks after they've been processed.

The SOAP specifications and the Web services community in general use a lot of terminology that may seem a little confusing at first, because, unlike other application protocols, SOAP is not limited to a single messaging paradigm. SOAP can be used with a variety of messaging systems (asynchronous, synchronous, RPC, One-Way, and others), which can be combined in non-traditional ways. In order to describe all the parties that participate in SOAP messaging, new terminology was invented to avoid restrictive and preconceived notions associated with more traditional terms, such as “client” and “server.” Although this new terminology wasn't introduced until early drafts of SOAP 1.2 were published, it applies equally well to SOAP 1.1.

SOAP is a protocol used to exchange messages between **SOAP applications** on a network, usually an intranet or the Internet. A SOAP application is simply any piece of software that generates or processes SOAP messages. For example, any Java application or J2EE component that uses JAX-RPC (covered in Part IV) would be considered a SOAP application, because JAX-RPC is used to generate and process SOAP messages. The application sending a SOAP message is called the **sender,** and the application receiving it is called the **receiver.** As a J2EE Web services developer you will be creating SOAP applications using JAX-RPC-enabled applications and components, which will act as receivers or senders or both.

A SOAP message travels along the message path from a sender to a receiver (see Figure 4–5). All SOAP messages start with the **initial sender,** which creates the SOAP

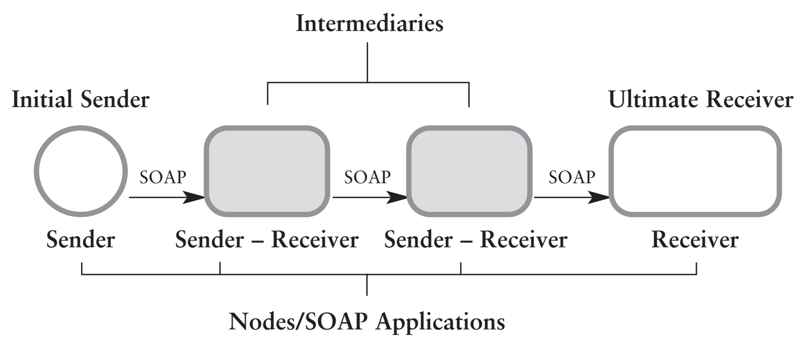

Figure 4–5 The SOAP Message Path

message, and end with the **ultimate receiver.** The term **client** is sometimes associated with the initial sender of a request message, and the term **Web service** with the ultimate receiver of a request message.

As a SOAP message travels along the message path, its header blocks may be intercepted and processed by any number of **SOAP intermediaries** along the way. A SOAP intermediary is both a receiver and a sender. It receives a SOAP message, processes one or more of the header blocks, and sends it on to another SOAP application. The applications along the message path (the initial sender, intermediaries, and ultimate receiver) are also called **SOAP nodes.**

To illustrate how nodes in a message path process header blocks, I'll use an example with two relatively simple header blocks: `message-id` and `processed-by`. The `processed-by` header block keeps a record of the SOAP applications (nodes) that process a SOAP message on its way from the initial sender to the ultimate receiver. Like the `message-id` header, the `processed-by` header block is useful in debugging and logging.

In this example, a SOAP message passes through several intermediaries before reaching the ultimate receiver. Figure 4–6 depicts the message path of a purchase-order SOAP message that is generated by a customer and processed by sales, accounts-receivable, inventory, and shipping systems.

Intermediaries in a SOAP message path must not modify the application-specific contents of the SOAP `Body` element, but they may, and often do, manipulate the SOAP header blocks.

In the present example, each SOAP intermediary is required to add a `node` element to the `processed-by` header block, identifying itself and the time it

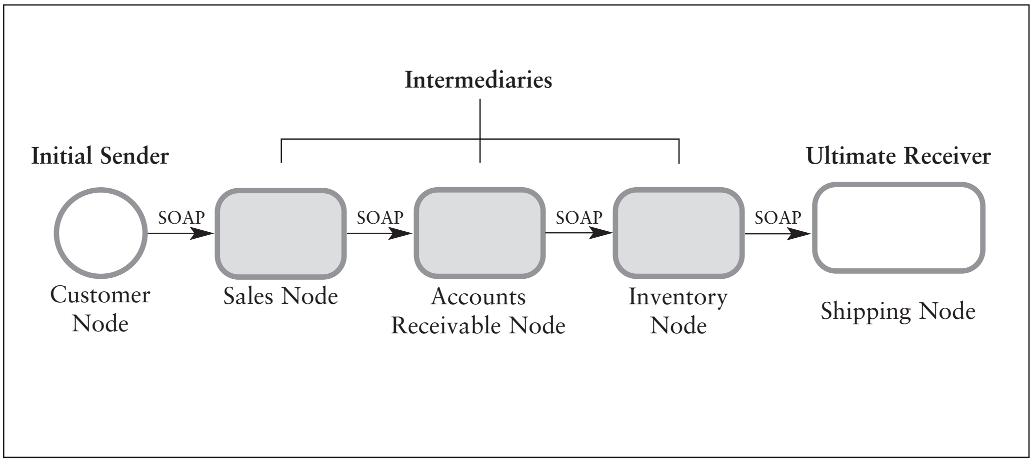

Figure 4–6 The Message Path of the Purchase-Order SOAP Message

processed the message. Listing 4–9 shows a message after each of five applications has added a `node` element to the `processed-by` header block.

Listing 4–9

The `processed-by` *Header Block*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <mi:message-id>11d1def534ea:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
    <proc:processed-by>
      <node>
        <time-in-millis>1013694680000</time-in-millis>
        <identity>http://www.customer.com</identity>
      </node>
      <node>
        <time-in-millis>1013694680010</time-in-millis>
        <identity>http://www.Monson-Haefel.com/sales</identity>
      </node>
      <node>
        <time-in-millis>1013694680020</time-in-millis>
        <identity>http://www.Monson-Haefel.com/AR</identity>
```

```
      </node>
      <node>
        <time-in-millis>1013694680030</time-in-millis>
        <identity>http://www.Monson-Haefel.com/inventory</identity>
      </node>
      <node>
        <time-in-millis>1013694680040</time-in-millis>
        <identity>http://www.Monson-Haefel.com/shipping</identity>
      </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
      <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

When processing a header block, each node reads, acts on, and removes the header block from the SOAP message before sending it along to the next receiver. Any node in a message path may also add a new header block to a SOAP message. But how does a node in the message path know which headers it's supposed to process?

SOAP 1.1 applications use the `actor` attribute to identify the nodes that should process a specific header block. SOAP also employs the `mustUnderstand` attribute to indicate whether a node processing the block needs to recognize the header block and know how to process it.

4.3.1 The `actor` Attribute

The `actor` attribute is defined by the SOAP Note and is a part of the same namespace as the SOAP `Envelope`, `Body`, and `Header` elements; that is, `"http://schemas.xmlsoap.org/soap/envelope/"`.

You use an `actor` attribute to identify a function to be performed by a particular node.

> *Just as a person can perform one or more roles in a stage play, a node can play one or more roles in a SOAP message path. Unfortunately, the designers of SOAP 1.1 confused the words "actor" and "role"; they specified that you must identify the* **roles** *a node will play by declaring an* `actor` *attribute. They've recognized their mistake, and in SOAP 1.2 this attribute has been renamed* `role`. *Because this book focuses on SOAP 1.1, you and I will have to work with the earlier terminology: An* `actor` *attribute specifies a* **role** *a node must play. I'll try to minimize the confusion as much as I can as we go along.*

The actor attribute uses a URI (Uniform Resource Identifier) to identify the role that a node must perform in order to process that header block. When a node receives a SOAP message, it examines each of the header blocks to determine which ones are targeted to roles supported by that node. For example, every SOAP message processed by a Monson-Haefel Books Web service might pass through a **logging intermediary,** a code module that records information about incoming messages in a log, to be used for debugging.

The logging module represents a particular role played by a node. A node may have many modules that operate on a message, and therefore many roles, so every node in a message path may identify itself with several different roles. For example, our company's Sales node (see Figure 4–6) may have a logging module, a security authentication module, and a transaction module. Each of these modules will read and process incoming SOAP messages in some way, and each module may represent a different role played by the Sales node.

The actor attribute is used in combination with the XML namespaces to determine which code module will process a particular header block. Conceptually, the receiving node will first determine whether it plays the role designated by the actor attribute, and then choose the correct code module to process the header block, based on the XML namespace of the header block. Therefore, the receiving node must recognize the role designated by the actor attribute assigned to a header block, as well as the XML namespace associated with the header block.

For example, the actor attribute identifies the logger role with the URL "http://www.Monson-Haefel.com/logger". A node that's intended to perform the logger role will look for header blocks where that URL is the value of the actor attribute. The message-id header block in the purchase-order SOAP message might be assigned the actor attribute value "http://www.Monson-Haefel.com/logger" as shown in Listing 4–10.

Listing 4–10

The actor *Attribute*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <mi:message-id soap:actor="http://www.Monson-Haefel.com/logger" >
      11d1def534ea:b1c5fa:f3bfb4dcd7:-8000
    </mi:message-id>
    <proc:processed-by>
      <node>
        <time-in-millis>1013694680000</time-in-millis>
```

```
        <identity>http://www.customer.com</identity>
      </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
      <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

Only those nodes in the message path that identify themselves with the `actor` value `"http://www.Monson-Haefel.com/logger"` will process the `message-id` header block; all other nodes will ignore it.

In addition to custom URIs like `"http://www.Monson-Haefel.com/logger"`, SOAP identifies two standard roles for the `actor` attribute: `next` and `ultimate receiver`. (These phrases don't actually appear by themselves in SOAP message documents. Nevertheless this chapter will show `next` and `ultimate receiver` in code font when they represent role names, to signal we're not referring to their more general meanings.) These standard roles indicate which nodes should process the header block, and they are relatively self-explanatory.

The `next` role indicates that the next node in the message path must process the header. The `next` role has a designated URI, which must be used as the value of the `actor` attribute: `"http://schemas.xmlsoap.org/soap/actor/next"`.

The `ultimate receiver` role indicates that only the ultimate receiver of the message should process the header block. The protocol doesn't specify an explicit URI for this purpose; it's the *absence* of an `actor` attribute in the header block that signals that the role is `ultimate receiver`.

We can use the `next` role in the `processed-by` header block of the purchase-order SOAP message, as shown in Listing 4–11.

Listing 4–11

A Header Block Uses the `actor` *Attribute*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <mi:message-id soap:actor="http://www.Monson-Haefel.com/logger" >
      11d1def534ea:b1c5fa:f3bfb4dcd7:-8000
    </mi:message-id>
    <proc:processed-by
     soap:actor=" http://schemas.xmlsoap.org/soap/actor/next">
```

```
      <node>
        <time-in-millis>1013694680000</time-in-millis>
        <identity>http://www.customer.com</identity>
      </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
      <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

In this case, the next receiver in the message path, no matter what other purpose it may serve, should process the `processed-by` header block. If an intermediary node in the message path supports the logger role, then it should process the `processed-by` header block in addition to the `message-id` header block. In this scenario, the intermediary node fulfills two roles: it's both a logger and the `next` receiver.

When a node processes a header block, it must remove it from the SOAP message. The node may also add new header blocks to the SOAP message. SOAP nodes frequently feign removal of a header block by simply modifying it, which is logically the same as removing it, modifying it, and then adding it back to the SOAP message—a little trick that allows a node to adhere to the SOAP specifications while propagating header blocks without losing any data. For example, the logger node may remove the `message-id` header block, but we don't want it to remove the `processed-by` header block, because we want all the nodes in the message path to add information to it. Therefore, the logger node will simply add its own data to the `processed-by` header block, then pass the SOAP message to the next node in the message path. Listing 4–12 shows the SOAP message after the logger node has processed it. Notice that the `message-id` header block has been removed and the `processed-by` header block has been modified.

Listing 4–12

The SOAP Message After the Header Blocks Are Processed

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <proc:processed-by
     soap:actor="http://schemas.xmlsoap.org/soap/actor/next">
      <node>
        <time-in-millis>1013694680000</time-in-millis>
```

```
      <identity>http://www.customer.com</identity>
    </node>
    <node>
      <time-in-millis>1013694680010</time-in-millis>
      <identity>http://www.Monson-Haefel.com/sales</identity>
    </node>
  </proc:processed-by>
 </soap:Header>
 <soap:Body>
    <!-- Application-specific data goes here -->
 </soap:Body>
</soap:Envelope>
```

4.3.2 The `mustUnderstand` Attribute

The use of standard role types, especially the `next` type, raises some interesting issues. In many cases we may not know the exact message path or the capabilities of all the nodes in a message path, which means we don't always know whether nodes can process header blocks correctly. For example, the `processed-by` header block is targeted at the `next` role, which means the next node to receive it should process it. But what if the next node doesn't recognize that kind of header block?

Header blocks may indicate whether processing is mandatory or not by using the `mustUnderstand` attribute, which is defined by the standard SOAP 1.1 namespace `"http://schemas.xmlsoap.org/soap/envelope/"`. The `mustUnderstand` attribute can have the value of either `"1"` or `"0"`, to represent true and false, respectively.

> ***The SOAP 1.1 XML Schema actually defines the*** `mustUnderstand` ***attribute as an*** `xsd:boolean` ***type, which allows any of* four *lexical literals:*** `"1"`, `"true"`, `"0"`, `or "false"`***. This flexibility has caused interoperability problems in the past, when a receiver expected a value of*** `"1"` ***or*** `"0"`***, but the sender supplied*** `"true"` ***or*** `"false"`***. According to the BP, SOAP applications must set the*** `mustUnderstand` ***attribute to*** `"1"` ***or*** `"0"`**—**`"true"` ***and*** `"false"` ***are* not *allowed.***[BP]

If the `mustUnderstand` attribute is omitted, then its default value is `"0"` (false). Explicitly declaring the `"0"` value is considered a waste of bandwidth.

When a header block has a `mustUnderstand` attribute equal to `"1"`, it's called a **mandatory header block.** SOAP nodes must be able to process any header block that is marked as mandatory *if* they play the role specified by the `actor` attribute of the header block.

The "understand" in `mustUnderstand` means that the node must recognize the header block by its XML structure and namespace, and know how to process it. In other words, if the node plays the role indicated by the `actor` attribute of a header block, but it's not programmed to process that header block, then that header block is *not* understood. This problem can arise very easily if you add an intermediate node but fail to account for all possible header blocks targeted to it, or more likely, fail to consider the `next` role.

If a node doesn't understand a mandatory header block, it must generate a **SOAP fault** (similar to a remote exception in Java) and discard the message; it must not forward the message to the next node in the message path.[BP]

The SOAP 1.1 Note didn't explain what should be done after a SOAP fault is generated. It didn't say whether the message should continue to be processed, which made it hard to predict what a receiver would do after generating a fault. The Basic Profile requires that the receiver discontinue normal processing of the message and generate a fault message.[BP]

In Listing 4–13, the SOAP message declares the `mustUnderstand` attribute in the `processed-by` header to be true.

Listing 4–13

Using the `mustUnderstand` *Attribute to Make Processing of a Header Block Mandatory*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <proc:processed-by
     soap:actor="http://schemas.xmlsoap.org/soap/actor/next"
     soap:mustUnderstand="1" >
      <node>
        <time-in-millis>1013694684723</time-in-millis>
        <identity>http://local/SOAPClient2</identity>
      </node>
      <node>
        <time-in-millis>1013694685023</time-in-millis>
        <identity>http://www.Monson-Haefel.com/logger</identity>
      </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
    <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

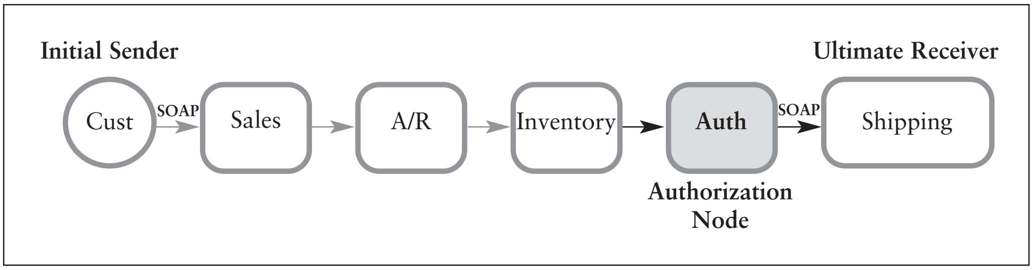

Figure 4–7 The Purchase-Order Message Path with Logger and Authentication-Filter Nodes

Let's say that Monson-Haefel adds a new Authentication node to the purchase-order message path. A SOAP message will be processed by the Authentication node before it's processed by the ultimate receiver, as illustrated in Figure 4–7.

Now suppose that when the Authentication node is added, the programmer neglects to include logic to handle the `processed-by` header block. As a result, the authentication node will not recognize the `processed-by` header block and will have no idea how to process it. Because the header block's `mustUnderstand` attribute has a value of `"1"`, the authentication node will have to discard the SOAP message, generate a SOAP fault, and send it back to the sender.

A SOAP receiver is required to generate a fault with the fault code `MustUnderstand` if it fails to understand a mandatory header block.[BP] This issue is covered in more detail in Section 4.6: SOAP Faults.

Whether or not a fault is sent back to the sender depends on whether the **messaging exchange pattern** (**MEP**) is One-Way or Request/Response. If a SOAP application uses Request/Response messaging, it's required to send a SOAP fault back to the sender; if it uses One-Way messaging, it's not.[BP]

If the `mustUnderstand` attribute is `"0"`, the processing requirements specified by SOAP are very different. If a node performs the role declared by a non-mandatory header block, and an application fails to understand the header (it doesn't recognize the XML structure or the namespace), it must remove the header block. It's not obliged, however, to try and process it, or to discard the message; it's free to remove the header and pass the message on to the next node in the message path.

Receivers should not reject a message simply because a header block targeted at some other node has not been processed (and removed). In other words, receivers should not attempt to determine whether a message was successfully processed by previous nodes in the path based on which header blocks are present. This rule applies especially to the ultimate receiver, which shouldn't reject a message because a header block intended for some unknown role was never processed. If receivers started analyzing and rejecting messages based on the status of header blocks for which they are not targeted, it would be impossible to make changes to the message path without worrying about the ripple effect of those changes downstream. Because nodes are required to "mind their own business," message paths can evolve and are very dynamic. Adding new intermediaries (or removing them) doesn't require adjustments to every other node in a message path.

Although this processing rule is not mentioned in SOAP 1.1 or the BP, it's an explicit requirement in SOAP 1.2 and should be applied when developing receivers for SOAP 1.1.

4.3.3 The WS-I Conformance Header Block

Although the BP doesn't endorse any particular type of header block, it does specify an optional conformance header block that indicates that the SOAP message complies with the BP. Listing 4–14 shows how the conformance header block may appear in a SOAP message.

Listing 4–14

Including a `Claim` *Header Block in a SOAP Message*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  <soap:Header>
    <wsi:Claim conformsTo="http://ws-i.org/profiles/basic/1.0"
     xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/" />
  </soap:Header>
  <soap:Body sec:id="Body">
    <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

The WS-I Basic Profile states that the `Claim` header block is not required. It also states that "absence of a conformance claim in a message must not be construed as inferring that the message does or does not conform to one or more profiles."

A SOAP message can declare a separate `Claim` header for each profile it adheres to. At the time of this writing the WS-I has defined only the Basic Profile 1.0, but it's expected to release other profiles. In the future, it's possible that a SOAP message will conform to both the Basic Profile 1.0 and other, as yet undefined, profiles.

A `Claim` element may be declared only as an immediate child of the `Header` element; it cannot appear in any other part of a SOAP message. In addition, the `Claim` header block is always considered optional, so its `mustUnderstand` attribute must not be `"1"`. You cannot require receivers to process a `Claim` header block.[BP]

4.3.4 Final Words about Headers

SOAP headers are a very powerful way of extending the SOAP protocol. As a construct for meta-data, a SOAP header is far more flexible and easier for developers and vendors to take advantage of than similar mechanisms in other protocols (such as the "service context" in CORBA IIOP). The extensibility of the SOAP headers is another reason why SOAP has become so popular and is likely to succeed where other protocols have not.

The `message-id` and `processed-by` headers are only custom header blocks I created for use in this book. Standards bodies frequently drive the definition of general-purpose SOAP header blocks. These organizations are primarily concerned with header blocks that address **qualities of service**, such as security, transactions, message persistence, and routing. OASIS, for example, is defining the WS-Security SOAP headers used with XML digital signatures—an XML security mechanism. Another example is the ebXML-specific header blocks defined by OASIS for such qualities of service as routing, reliable messaging, and security. Microsoft and IBM are also defining "standard" header blocks for these same qualities of service. The BP does not address any of these potential standards, but WS-I will eventually create more advanced profiles that incorporate many of the proposals evolving at OASIS, W3C, Microsoft, IBM, and other organizations—in fact, at the time of this writing, WS-I has started defining the WS-I Security Profile based on the OASIS WS-Security standard.

4.4 The SOAP Body

Although the `Header` element is optional, all SOAP messages must contain exactly one `Body` element.[BP] The `Body` element contains either the application-specific data or a fault message. Application-specific data is the information that we want to exchange with a Web service. It can be arbitrary XML data or parameters to a procedure call. Either way, the `Body` element contains the application data being exchanged. A fault message is used only when an error occurs. The receiving node that discovers a problem, such as a processing error or a message that's improperly structured, sends it back to the sender just before it in the message path. A SOAP message may carry either application-specific data or a fault, but not both.

Whether the `Body` element contains application-specific data or a fault, most SOAP experts agree that only the ultimate receiver of the SOAP message should process the contents of the `Body`. Intermediary nodes in the message path may view the `Body` element, but they should not alter its contents in any way. This is very different from header blocks, which may be processed by any number of intermediaries along the message path. This is a critical point: Only the ultimate receiver should alter the contents of the `Body` element.

Neither SOAP 1.1 nor the BP explicitly prohibits intermediaries from modifying the contents of the `Body` ***element. As a result, the ultimate***

receiver has no way of knowing if the application-specific data has changed somewhere along the message path. SOAP 1.2 reduces this uncertainty by explicitly prohibiting certain intermediaries, called* forwarding intermediaries, *from changing the contents of the* `Body` *element and recommending that all other intermediaries, called* active intermediaries, *use a header block to document any changes to the* `Body` *element.

4.5 SOAP Messaging Modes

Except in the case of fault messages, SOAP does not specify the contents of the `Body` element (although it does specify the general structure of RPC-type messages). As long as the `Body` contains well-formed XML, the application-specific data can be anything. The `Body` element may contain any XML element or it can be empty.

Although SOAP supports four modes of messaging (RPC/Literal, Document/Literal, RPC/Encoded, and Document/Encoded) the BP permits the use of RPC/Literal or Document/Literal only. The RPC/Encoded and Document/Encoded modes are explicitly prohibited.[BP]

A messaging mode is defined by its messaging style (RPC or Document) and its encoding style. There are two common types of encoding used in SOAP messaging: SOAP encoding as described in Section 5 of the SOAP 1.1 specification, and Literal encoding. SOAP encoding is not supported by WS-I-conformant Web services because it causes significant interoperability problems.[BP] The term "Literal" means that the XML document fragment can be validated against its XML schema.

4.5.1 Document/Literal

In the **Document/Literal mode** of messaging, a SOAP `Body` element contains an **XML document fragment,** a well-formed XML element that contains arbitrary application data (text and other elements) that belongs to an XML schema and namespace separate from the SOAP message's.

For example, a set of XML elements that describes a purchase order, embedded within a SOAP message, is considered an XML document fragment. The purchase-order SOAP message, which is used as an example throughout this chapter, is a Document/Literal message. Listing 4–15 shows the complete purchase-order SOAP message, which contains the `purchaseOrder` XML document fragment.

Listing 4–15

A Document-Style SOAP Message

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
```

```
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <!-- Header blocks go here -->
  </soap:Header>
  <soap:Body>
    <po:purchaseOrder orderDate="2003-09-22"
     xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO">

      <po:accountName>Amazon.com</po:accountName>
      <po:accountNumber>923</po:accountNumber>
      …
      <po:book>
        <po:title>J2EE Web Services</po:title>
        <po:quantity>300</po:quantity>
        <po:wholesale-price>24.99</po:wholesale-price>
      </po:book>
    </po:purchaseOrder>
  </soap:Body>
</soap:Envelope>
```

4.5.2 RPC/Literal

The **RPC/Literal mode** of messaging enables SOAP messages to model calls to procedures or method calls with parameters and return values. In RPC/Literal messaging, the contents of the `Body` are always formatted as a `struct`. An RPC request message contains the method name and the input parameters of the call. An RPC response message contains the return value and any output parameters (or a fault). In many cases, RPC/Literal messaging is used to expose traditional components as Web services. A traditional component might be a servlet, stateless session bean, Java RMI object, CORBA object, or DCOM component. These components do not explicitly exchange XML data; rather, they have methods with parameters and return values.

For example, Monson-Haefel Books has a JAX-RPC service endpoint (a J2EE Web Service endpoint) called BookQuote that Monson-Haefel's sales force uses. The remote interface to the BookQuote looks like this:

```
package com.jwsbook.soap;
import java.rmi.RemoteException;

public interface BookQuote extends java.rmi.Remote {
    // Get the wholesale price of a book
    public float getBookPrice(String ISBN)
```

```
      throws RemoteException, InvalidISBNException;
}
```

The `getBookPrice()` method declares a parameter in the form of an ISBN (International Standard Book Number), a unique string of characters assigned to every retail book. When you invoke this method with a proper ISBN, the Web service will return the wholesale price of the book identified.

This JAX-RPC service endpoint can use the RPC/Literal mode of messaging. The Web service uses two SOAP messages: a request message and a reply message. The request message is sent from an initial sender to the Web service and contains the method name, `getBookPrice`, and the ISBN string parameter. The reply message is sent back to the initial sender and contains the price of the book as a `float` value. Listing 4–16 shows the SOAP request message for the BookQuote Web service.

Listing 4–16

An RPC/Literal SOAP Request Message

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Body>
      <mh:getBookPrice>
          <isbn>0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

Listing 4–17 shows the corresponding response.

Listing 4–17

An RPC/Literal SOAP Response Message

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
   <soap:Body>
      <mh:getBookPriceResponse>
          <result>24.99</result>
      </mh:getBookPriceResponse>
   </soap:Body>
</soap:Envelope>
```

Unlike Document/Literal messaging, which makes no assumptions about the type and structure of elements contained in the `Body` of the message—except that the document fragment adheres to some XML schema—RPC/Literal messages carry a simple set of arguments. RPC-style messaging is a common idiom in distributed technologies, including EJB, CORBA, DCOM, and others, so SOAP defines a standard XML format for RPC-style messaging, called RPC/Literal. The RPC/Literal mode of messaging specifies how methods and their arguments (parameters and return values) are represented within the `Body` element of a SOAP message.

It's important to understand that RPC/Literal and Document/Literal may be indistinguishable from the perspective of a developer using tools like JAX-RPC, because JAX-RPC can present procedure-call semantics for both RPC/Literal and Document/Literal. A few people question the usefulness of RPC/Literal in the first place. Why use it when you can use Document/Literal, which is arguably simpler to implement in some respects, and can exploit XML schema validation? This book covers both models without taking sides on this issue.

4.5.3 Messaging Modes versus Messaging Exchange Patterns

It's easy to confuse Document/Literal and RPC/Literal modes of messaging with the One-Way and Request/Response message exchange patterns (MEPs), but the concepts are distinctly different. When you say a messaging mode is Document/Literal or RPC/Literal, you are usually describing the *payload* of the SOAP message: an XML document fragment or an XML representation of the parameters and return values associated with a remote procedure call. In contrast, One-Way and Request/Response MEPs refer to the *flow* of messages, not their contents. One-Way messaging is unidirectional, Request/Response is bi-directional. You can use the

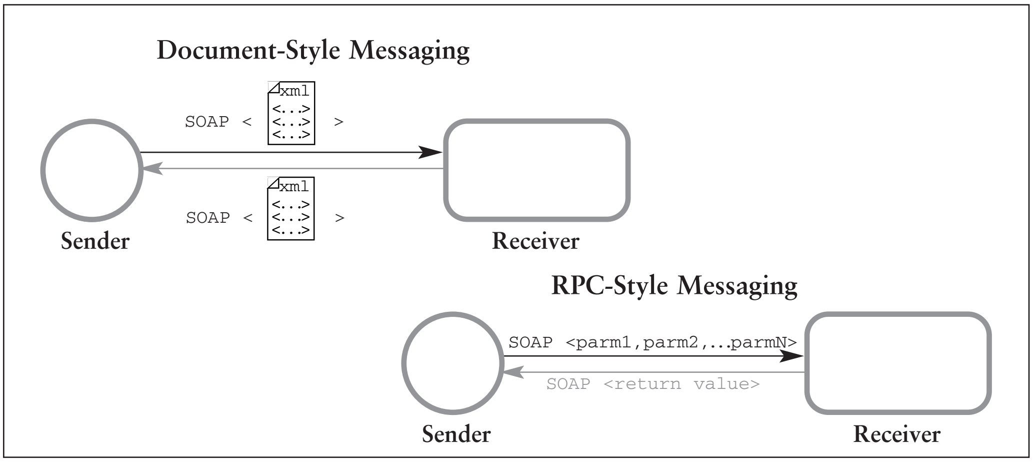

Figure 4–8 Using Document/Literal and RPC/Literal with One-Way and Request/Response Messaging

Document/Literal mode of messaging with either One-Way or Request/Response messaging. The RPC/Literal mode of messaging can also be used with either MEP, although it's usually used with Request/Response messaging.

Figure 4–8 shows the response message flows (the gray arrows) as optional in both document-style and RPC-style messaging.

4.5.4 Other Messaging Modes

Two other modes of messaging can be used with SOAP, **RPC/Encoded** and **Document/Encoded,** but the BP frowns on them, for two reasons: XML schema makes them obsolete, and they introduce a number of difficult interoperability problems.[BP]

RPC/Encoded actually receives more attention from SOAP than any other messaging mode. It attempts to define a mapping between common RPC semantics and programmatic types on one hand and XML on the other. An entire section of the SOAP 1.1 Note (the infamous Section 5) is devoted to explaining SOAP encoding. RPC/Encoded relies on built-in XML schema types, but it is designed to represent a graph of objects. XML schema organizes data into a tree, which is not nearly as flexible as an object graph. Although RPC/Encoded messaging was popular at first, its overall complexity and the interoperability problems it has caused have convinced most SOAP specialists to avoid it. You can accomplish pretty much the same results using the RPC/Literal or Document/Literal modes of messaging, with the added bonuses of better interoperability and conformance with the XML schema. It's likely, however, that you will encounter legacy Web service implementations that continue to use RPC/Encoded messaging despite its lack of support from the WS-I. To help you understand and work with these services, Appendix D: SOAP RPC/Encoded provides detailed coverage of this messaging mode.

Document/Encoded messaging applies SOAP encoding as defined in Section 5 of the SOAP 1.1 Note to document-style messaging. This mode of messaging is rarely, if ever, used in practice because Document/Literal messaging is much simpler, and interoperable. Document/Encoded messaging is not supported in J2EE Web services, so it's given no more consideration in this book.

4.6 SOAP Faults

SOAP fault messages are the mechanism by which SOAP applications report errors "upstream," to nodes earlier in the message path. It's the mission of this section to provide a full and detailed explanation of SOAP faults so that you can handle them appropriately in your own Web services.

SOAP faults are generated by receivers, either an intermediary or the ultimate receiver of a message. The receiver is required to send a SOAP fault back to the sender only if the Request/Response messaging mode is used. In One-Way mode, the receiver should generate a fault and may store it somewhere, but it must not attempt to transmit it to the sender.

SOAP faults are returned to the receiver's immediate sender. For example, if the third node in a message path generates a fault, that fault message is sent to the second node in the message path and nowhere else. In other words, you don't send the fault to the original sender unless it's also the immediate sender. When that sender receives the fault message, it may take some action, such as undoing operations, and may send another fault further upstream to the next sender if there is one.

Most developers see error handling as a pretty dull subject, so it's often ignored or poorly implemented. The tendency to ignore error handling is natural, but it's not wise. As the saying goes, "Stuff happens": Things can, and often do, go wrong; it's inevitable that errors will occur in the normal course of events. Because errors are fairly common, it's logical that some time should be dedicated to error handling. The SOAP Note recognizes the importance of error handling and dedicates a considerable amount of verbiage to addressing the issue. Even so, SOAP is not strict enough to avoid interoperability problems, so the BP provides a lot more guidance on the generation and processing of SOAP fault messages.

A SOAP message that contains a `Fault` element in the `Body` is called a **fault message.** A fault message is analogous to a Java exception; it's generated when an error occurs. Fault messages are used in Request/Response messaging. Nodes in the message path generate them when processing a request message. When an error occurs, the receiving node sends a fault message back to the sender just upstream, instead of the anticipated reply message. Faults are caused by improper message formatting, version mismatches, trouble processing a header, and application-specific errors.

When a fault message is generated, the `Body` of the SOAP message must contain only a single `Fault` element and nothing else. The `Fault` element itself must contain a `faultcode` element and a `faultstring` element, and optionally `faultactor` and `detail` elements. Listing 4–18 is an example of a SOAP fault message.

Listing 4–18

A SOAP Fault Message

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Client</faultcode>
      <faultstring>
        The ISBN value contains invalid characters
      </faultstring>
      <faultactor>http://www.xyzcorp.com</faultactor>
```

```
      <detail>
        <mh:InvalidIsbnFaultDetail>
          <offending-value>19318224-D</offending-value>
          <conformance-rules>
            The first nine characters must be digits. The last
            character may be a digit or the letter 'X'. Case is
            not important.
          </conformance-rules>
        </mh:InvalidIsbnFaultDetail>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Note that the `Fault` element and its children are part of the SOAP namespace, just as the SOAP `Envelope` and `Body` elements are.

Did you notice in Listing 4–18 that the children of the `Fault` element weren't qualified with the `soap` prefix? The children of the `Fault` element may be unqualified.[BP] In other words, they need *not* be prefixed with the SOAP 1.1 namespace. Note as well that it's forbidden for the `Fault` element to contain any immediate child elements other than `faultcode`, `faultstring`, `faultactor`, and `detail`.[BP]

4.6.1 The `faultcode` Element

The `faultcode` element may use any of four standard SOAP fault codes to identify an error.

SOAP Standard Fault Codes

`Client`
`Server`
`VersionMismatch`
`MustUnderstand`

Although you're allowed to use arbitrary fault codes, you should use only the four standard codes listed.[BP]

The `faultcode` element should contain one of the standard codes listed above, with the appropriate SOAP namespace prefix. Prefixing the code, as in **`soap:`**`Client`, allows for easy versioning of standard fault codes. As SOAP evolves, it's possible that new fault codes will be added. New fault codes can easily be distinguished from legacy fault codes by their namespace prefix. The meaning of a fault code will always correlate to both the code (the local name) and the namespace (the prefix).

The SOAP Note recommends the use of the dot separator between names to discriminate general standard fault codes from specific application subcodes. This convention is not used in J2EE Web services, which prefers the use of XML namespace-based prefixes for SOAP fault codes. If you use one of the standard SOAP fault codes, the namespace prefix must map to the SOAP namespace `"http://schemas.xmlsoap.org/soap/envelope/"`.[BP]

4.6.1.1 *The* `Client` *Fault*

The `Client` fault code signifies that the node that sent the SOAP message caused the error. Basically, if the receiver cannot process the SOAP message because there is something wrong with the message or its data, it's considered the fault of the client, the sender. The receiving node generates a `Client` fault if the message is not well formed, or contains invalid data, or lacks information that was expected, like a specific header. For example, in Listing 4–19, the SOAP fault indicates that the sender provided invalid information.

Listing 4–19

An Example of a SOAP Fault with a `Client` *Fault Code*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Client</faultcode>
      <faultstring>The ISBN contains invalid characters</faultstring>
      <detail/>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

When a node receives a fault message with a `Client` code, it should not attempt to resend the same message. It should take some action to correct the problem or abort completely.

4.6.1.2 *The* `Server` *Fault*

The `Server` fault code indicates that the node that received the SOAP message malfunctioned or was otherwise unable to process the SOAP message. This fault is a reflection of an error by the receiving node (either an intermediary or the ultimate receiver) and doesn't point to any problems with the SOAP message itself. In this case the sender can assume the SOAP message to be correct, and can redeliver it after pausing some period of time to give the receiver time to recover.

If, for example, the receiving node is unable to connect to a resource such as a database while processing a SOAP message, it might generate a `Server` fault. The following is an example of a `Server` fault, generated when the BookPrice Web service could not access the database to retrieve price information in response to a SOAP message.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring> Database is unavailable.</faultstring>
      <detail/>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

4.6.1.3 The `VersionMismatch` *Fault*

A receiving node generates a `VersionMismatch` fault when it doesn't recognize the namespace of a SOAP message's `Envelope` element. For example, a SOAP 1.1 node will generate a fault with a `VersionMismatch` code if it receives a SOAP 1.2 message, because it finds an unexpected namespace in the `Envelope`. This scenario is illustrated by the fault message in Listing 4–20.

Listing 4–20

An Example of a `VersionMismatch` *Fault*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:VersionMismatch</faultcode>
      <faultstring>Message was not SOAP 1.1-conformant</faultstring>
      <detail/>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

The `VersionMismatch` fault applies only to the namespace assigned to the `Envelope`, `Header`, `Body`, and `Fault` elements. It does not apply to other parts of the SOAP message, like the header blocks, XML document version, or application-specific elements in the `Body`.

The `VersionMismatch` fault is also used in the unlikely event that the root element of a message is not `Envelope`, but something else. Sending a `Version Mismatch` fault message back to the sender in this case may not be helpful, however: The sender may be designed to handle a different protocol and doesn't understand SOAP faults.

4.6.1.4 The `MustUnderstand` *Fault*

When a node receives a SOAP message, it must examine the `Header` element to determine which header blocks, if any, are targeted at that node. If a header block is targeted at the current node (via the `actor` attribute) and sets the `mustUnderstand` attribute equal to `"1"`, then the node is required to know how to process the header block. If the node doesn't recognize the header block, it must generate a fault with the `MustUnderstand` code. Listing 4–21 shows an example.

Listing 4–21

A `MustUnderstand` *Fault*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:MustUnderstand</faultcode>
      <faultstring>Mandatory header block not understood.</faultstring>
      <detail/>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

4.6.1.5 Non-standard SOAP Fault Codes

It is also possible to use non-standard SOAP fault codes that are prescribed by other organizations and belong to a separate namespace. For example, Listing 4–22 uses a fault code specified by the WS-Security specification.

Listing 4–22

Using Non-standard Fault Codes

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/06/secext">
  <soap:Body>
    <soap:Fault>
```

```
      <faultcode>wsse:InvalidSecurityToken</faultcode>
      <faultstring>An invalid security token was provided</faultstring>
      <detail/>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

4.6.2 The `faultstring` Element

The `faultstring` element is mandatory. It should provide a human-readable description of the fault. Although the `faultstring` element is required, the text used to describe the fault is not standardized.

Optionally, the `faultstring` element can indicate the language of the text message using a special attribute, `xml:lang`.[BP] The set of valid codes is defined by IETF RFC 1766.[4] For example, a `Client` fault could be generated with a Spanish-language text as shown in Listing 4–23.

Listing 4–23

Using the `xml:lang` *Attribute in the* `faultstring` *Element*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Client</faultcode>
      <faultstring xml:lang="es" >
        El ISBN tiene letras invalidas
      </faultstring>
     <detail/>
   </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Although it's not specified, it's assumed that, in the absence of the `xml:lang` attribute, the default is English (`xml:lang="en"`). The `xml:lang` attribute is part of the XML 1.0 namespace, which does not need to be declared in an XML document.

[4] Internet Engineering Task Force, *RFC 1766: Tags for the Identification of Languages* (1995). Available at http://www.ietf.org/rfc/rfc1766.txt. The codes themselves are derived from ISO 639, and can be found at http://www.w3.org/WAI/ER/IG/ert/iso639.htm.

4.6.3 The `faultactor` Element

The `faultactor` element indicates which node encountered the error and generated the fault (the **faulting node**). This element is required if the faulting node is an intermediary, but optional if it's the ultimate receiver. For example, let's assume that an intermediary node in the message path, the authentication node, did not recognize the mandatory (`mustUnderstand="1"`) `processed-by` header block, so it generated a `MustUnderstand` fault. In this case the authentication node must identify itself using the `faultactor` element, as in Listing 4–24.

Listing 4–24

Locating the Source of the Fault Using the `faultactor` *Element*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:MustUnderstand</faultcode>
      <faultstring>Mandatory header block not understood. </faultstring>
      <faultactor>
        http://www.Monson-Haefel.com/jwsbook/authenticator
      </faultactor>
      <detail/>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

The `faultactor` element may contain any URI, but is usually the Internet address of the faulting node, or the URI used by the `actor` attribute if a header block was the source of the error.

SOAP 1.1 doesn't recognize the concept of a *role* as distinct from a *node*. In fact, it lumps these two concepts together into the single concept *actor.* Thus you can see the `faultactor` as identifying both the node that generated the fault and the role that it was manifesting when it generated the fault.

4.6.4 The `detail` Element

The `detail` element of a fault message must be included if the fault was caused by the contents of the `Body` element, but it must *not* be included if the error occurred while processing a header block. The SOAP message in Listing 4–25 provides further details about the invalid ISBN reported in the `faultstring` element.

Listing 4–25

A SOAP Fault `detail` *Element*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Client</faultcode>
      <faultstring>
        The ISBN value contains invalid characters
      </faultstring>
      <detail>
        <mh:InvalidIsbnFaultDetail>
          <offending-value>19318224-D</offending-value>
          <conformance-rules>
            The first nine characters must be digits. The last
            character may be a digit or the letter 'X'. Case is
            not important.
          </conformance-rules>
        </mh:InvalidIsbnFaultDetail>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

> ***The*** `detail` ***element may contain any number of application-specific elements, which may be qualified or unqualified, according to their XML schema. In addition, the*** `detail` ***element itself may contain any number of qualified attributes, as long as they do not belong to the SOAP 1.1 namespace,*** `"http://schemas.xmlsoap.org/soap/envelope"`.[BP]

It's perfectly legal to use an empty `detail` element, but you must *not* omit the `detail` element entirely if the fault resulted while processing the contents of the original message's `Body` element.

4.6.4.1 Processing Header Faults: Omitting the `detail` *Element*

SOAP provides little guidance on how details about header faults should be provided. It says only that detailed information must be included in the `Header` element. Some SOAP toolkits place a SOAP `Fault` element inside the `Header` element, or nested within a header block, while other toolkits may use a different strategy.

4.6.5 Final Words about Faults

As a developer, it's your responsibility to be aware of the various circumstances under which faults must be generated, and to ensure that your code properly implements the processing of those faults.

This is probably a good time to recap. Faults result from one of several conditions:

1. The message received by the receiver is improperly structured or contains invalid data.
2. The incoming message is properly structured, but it uses elements and namespaces in the `Body` element that the receiver doesn't recognize.
3. The incoming message contains a mandatory header block that the receiver doesn't recognize.
4. The incoming message specifies an XML namespace for the SOAP `Envelope` and its children (`Body`, `Fault`, `Header`) that is not the SOAP 1.1 namespace.
5. The SOAP receiver has encountered an abnormal condition that prevents it from processing an otherwise valid SOAP message.

The first two conditions generate what are considered `Client` faults, faults that relate to the contents of the message: The client has sent an invalid or unfamiliar SOAP message to the receiver. The third condition results in a `MustUnderstand` fault, and the fourth results in a `VersionMismatch` fault. The fifth condition is considered a `Server` fault, which means the error was unrelated to the contents of the SOAP message. A server fault is generated when the receiver cannot process a SOAP message because of an abnormal condition.

4.7 SOAP over HTTP

The vast majority of all Internet traffic today is data transferred using HTTP (HyperText Transfer Protocol), mostly by people browsing the World Wide Web. HTTP is ubiquitous because it is supported by an extensive, long-established infrastructure of servers and browsers. The inventors of SOAP took note of this infrastructure and shrewdly designed SOAP so that every message can be carried as the payload of an HTTP message. This "tunneling" has been fundamental to SOAP's rapid adoption and unprecedented success.

It's possible to deliver SOAP messages using other protocols, such as SMTP and FTP as well, but details of these non-HTTP bindings are not specified by SOAP and are not supported by the BP, so this book discusses SOAP over HTTP only.

SOAP messages sent over HTTP are placed in the payload of an HTTP request or response, an area that is normally occupied by form data and HTML. HTTP is a Request/Response protocol, which means that the sender expects a response (either an error code or data) from the receiver. HTTP requests are typified by the messages that your browser sends to a Web server to request a Web page or submit a form. A request for a Web page is usually made in an HTTP GET message, while submission of a form is done with an HTTP POST message.

There is nothing intrinsic to HTTP that limits it to requesting Web pages, but that's been its primary occupation for the past decade. Most HTTP traffic is composed of HTTP GET requests and HTTP replies. The HTTP GET request identifies the Web page requested and may include some parameters. An HTTP reply message returns the Web page to the requester as its payload.

While the HTTP GET request is perfectly suited for requesting Web pages, it doesn't have a payload area and therefore cannot be used to carry SOAP messages. The HTTP POST request, on the other hand, does have a payload area and is perfectly suited to carrying a SOAP message. HTTP reply messages, whether they are replies to GET or POST messages, follow the same format and carry a payload. Web services that use SOAP 1.1 with HTTP always use HTTP POST and not HTTP GET messages.

4.7.1 Transmitting SOAP with HTTP POST Messages

Sending a SOAP message as the payload of an HTTP POST message is very simple. Listing 4–26 shows the BookQuote SOAP message embedded in an HTTP POST message.

Listing 4–26

A SOAP Request over HTTP

```
POST /jwsbook/BookQuote HTTP/1.1
Host: www.Monson-Haefel.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 295
SOAPAction=""

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Body>
      <mh:getBookPrice>
          <isbn>0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

The HTTP POST message must contain a `SOAPAction` header field, but the value of this header field is not specified. The `SOAPAction` header field can improve throughput by providing routing information outside the SOAP payload. A node can then do some of the routing work using the `SOAPAction`, rather than having to parse the SOAP XML payload.

While the `SOAPAction` header field can improve efficiency, it's also the source of a lot of debate in the Web services industry. SOAP purists don't like the use of the `SOAPAction` HTTP header field because it expands the SOAP processing model to include the carrier protocol (in this case HTTP). They believe that all of the routing and payload should be contained in the SOAP document, so that SOAP messages are not dependent on the protocol over which they are delivered. This is a creditable argument, so the `SOAPAction` header field may contain an empty string, as indicated by an empty pair of double quotes. The decision to use a value for the `SOAPAction` header field is up to the person who develops the Web service. SOAP 1.2 will replace the `SOAPAction` header with the protocol-independent **action media type** (a parameter to the `"application/soap+xml"` MIME type) , so dependency on this feature may result in forward-compatibility problems. The BP requires that the `SOAPAction` header field be present and that its value be a quoted string that matches the value of the `soapAction` attribute declared by the corresponding WSDL document. If that document declares no `soapAction` attribute, the `SOAPAction` header field can be an empty string. Details are provided in Chapter 5: WSDL.

You may have noticed that the `Content-Type` is `text/xml`, which indicates that the payload is an XML document. The WS-I Basic Profile 1.0 prefers that the `text/xml` `Content-Type` be used with SOAP over HTTP. It's possible to use others (for example, SOAP with Attachments would specify `multipart/related`) but it's not recommended.

The reply to the SOAP message is placed in an HTTP reply message that is similar in structure to the request message, but contains no `SOAPAction` header. Listing 4–27 illustrates.

Listing 4–27

A SOAP Reply over HTTP

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset='utf-8'
Content-Length: 311

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
   <soap:Body>
      <mh:getBookPriceResponse>
          <result>24.99</result>
      </mh:getBookPriceResponse>
```

```
  </soap:Body>
</soap:Envelope>
```

4.7.2 HTTP Response Codes

Although SOAP faults provide an error-handling system in the SOAP context, you must also understand HTTP response codes, which indicate the success or failure of an HTTP request. In Listing 4–27 you'll notice that the first line of text is `HTTP/1.1 200 OK`. The `HTTP/1.1` portion indicates the version of HTTP used. Although HTTP 1.1 is the preferred protocol, you may also use HTTP 1.0.[BP] The rest of the line, `200 OK`, is the HTTP response code.

HTTP defines a number of success and failure codes that can be included in an HTTP reply message, but the BP takes special care to specify exactly which codes can be used by conformant SOAP applications. The types of response codes used depend on the success or failure of the SOAP request and the type of messaging exchange pattern used, Request/Response or One-Way.

4.7.2.1 Success Codes

The 200-level HTTP success codes are used to indicate that a SOAP request was received or successfully processed. The `200 OK` and `202 Accepted` HTTP success codes are used in Web services.

`200 OK` When a SOAP operation generates a response SOAP message, the HTTP response code for successful processing is `200 OK`. This response code indicates that the reply message is not a fault, that it does contain a normal SOAP response message.

`202 Accepted` This response code means that the request was processed successfully but that there is no SOAP response data. This type of SOAP operation is similar to a Java method that has a return type of `void`.

Although a One-Way SOAP message is conceptually unidirectional, when it's sent over HTTP some type of HTTP reply will be transmitted back to the receiver. One-Way SOAP messages do not return SOAP faults or results of any kind, so the HTTP `202 Accepted` response code indicates only that the message made it to the receiver—it doesn't indicate whether the message was successfully processed.[BP]

4.7.2.2 Error Codes

In general, HTTP uses the 400-level response codes to indicate that the client made some kind of error when transmitting the message. For example, you have undoubtedly encountered the infamous `404 Resource Not Found` error when using a Web browser. The `404` error code signifies that the client attempted to access a Web page or some other resource that doesn't exist. Web services uses a specific set of

400-level codes when the error is related to the contents of the SOAP message itself, rather than the HTTP request. HTTP also uses the 500-level response codes to indicate that the server suffered some type of failure that is not the client's fault.

`400 Bad Request` This error code is used to indicate that either the HTTP request or the XML in the SOAP message was not well formed.

`405 Method Not Allowed` If a Web service receives a SOAP message via any HTTP method other than HTTP POST, the service should return a `405 Method Not Allowed` error to the sender.

`415 Unsupported Media Type` HTTP POST messages must include a `Content-Type` header with a value of `text/xml`. If it's any other value, the server must return a `415 Unsupported Media Type` error.

`500 Internal Server Error` This code must be used when the response message in a Request/Response MEP is a SOAP fault.

4.7.3 Final Words about HTTP

HTTP provides a solid bedrock on which to base SOAP messaging. HTTP is ubiquitous, well understood, and widely supported. That said, HTTP has its detractors. For example, Don Box has characterized HTTP as the "cockroach of the Internet," to convey his view that it's an undesirable protocol that can't easily be done away with. The fact that modern firewalls do not restrict HTTP traffic on port 80 makes HTTP convenient for accessing servers and clients behind firewalls—which are a major impediment to distributed computing. Of course this introduces security issues because we are effectively circumventing the firewalls that help keep organizations safe from malicious hackers. It seems likely that firewall vendors will not permit "tunneling" to go on forever. Eventually they will feel compelled to enhance firewall products so that they will filter for, and block, HTTP communications that carry SOAP messages.

Blocking SOAP messages at the firewall is not necessary, however. Because SOAP is a transparent protocol (it's simple text rather than opaque data), a firewall can easily inspect the contents and route the message to a SOAP-specific security processor.

HTTP is not the only protocol over which you can send SOAP messages. You can also use SMTP (e-mail) and raw TCP/IP. The WS-I may one day extend the BP to include these other protocols—but for now HTTP is the only protocol endorsed by the WS-I.

4.8 Wrapping Up

SOAP 1.1, the focus of this chapter, is the XML protocol used in J2EE 1.4 Web Services because it's well supported and fairly well understood. The BP has done a lot to clear up ambiguities in SOAP 1.1, and the SOAP 1.2 protocol also includes these clarifications. It seems likely that SOAP 1.2 will supplant SOAP 1.1, but I wouldn't expect that development to occur for a while yet. Usually it takes a new version of a protocol a couple of years to replace the earlier version—in some cases longer.

I'm pretty sure that the WS-I will have updated the BP to support SOAP 1.2 by the time the next version of J2EE, tentatively labeled J2EE 1.5, is released, and thus that J2EE 1.5 Web Services will support SOAP 1.2. Until that day, though, jumping on the SOAP 1.2 bandwagon is a risk. Interoperability depends on common understanding of the protocol and a lack of ambiguity. It will be a while before we know where SOAP 1.2's bugs lie and have a BP to address them. For now, save yourself some headaches and stick with SOAP 1.1 and the BP.