


# 3

## The SOAP Protocol

**T**HE WEB SERVICES ARCHITECTURE GROUP AT THE W3C has defined a Web service as follows (*italics added*):

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description *using SOAP-messages*, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Although our definition (see Chapter 1, “Web Services Overview and Service-Oriented Architectures”) may be a bit broader, it’s clear that *SOAP*  is at the core of any survey of Web service technology. So just what is SOAP, and why is it often considered the harbinger of a new world of interoperable systems?

The trouble with SOAP is that it’s so simple and so flexible that it can be used in many different ways to fit the needs of different Web service scenarios. This is both a blessing and a curse. It’s a blessing because chances are, SOAP can fit your needs. It’s a curse because you may not know how to make it do what you require. When you’re through with this chapter, you’ll know not only how to use SOAP straight out of the box but also how to extend SOAP to support your diverse and changing needs. You’ll have also followed the development of a meaningful e-commerce Web service for our favorite company, SkatesTown. Last but not least, you’ll be ready to handle the rest of the book and climb higher toward the top of the Web services interoperability stack.

The chapter will cover the following topics:

- The evolution of XML protocols and the history and motivation behind SOAP’s creation
- The SOAP messaging framework: versioning, the extensibility framework, header-based vertical extensibility, intermediary-based horizontal extensibility, error handling, and bindings to multiple transport protocols

- The various mechanisms for packaging information in SOAP messages, including SOAP's own data encoding rules and heuristics for putting just about any kind of data in SOAP messages
- The use of SOAP within multiple distributed system architectures such as RPC- and messaging-based systems in all their flavors
- A quick introduction to building and consuming Web services using the Java-based Apache Axis Web services engine

So, why SOAP? As this chapter will show, SOAP is simple, flexible, and highly extensible. Since it's XML based, SOAP is programming-language, platform, and hardware neutral. What better choice for the XML protocol that's the foundation of Web services? To prove this point, let's start the chapter by looking at some of the earlier work that inspired SOAP.

## SOAP

Microsoft started thinking about XML-based distributed computing in 1997. The goal was to enable applications to communicate via Remote Procedure Calls (RPCs) using a simple network of standard data types on top of XML/HTTP. DevelopMentor (a long-standing Microsoft ally) and Userland (a company that saw the Web as a great publishing platform) joined the discussions. The name SOAP was coined in early 1998.

Things moved forward, but as the group tried to involve wider circles within Microsoft, politics stepped in and the process stalled. The DCOM camp at the company disliked the idea of SOAP and believed that Microsoft should use its dominant position in the market to push the DCOM wire protocol via some form of HTTP tunneling instead of pursuing XML. Some XML-focused folks at Microsoft believed that the SOAP idea was good but had come too early. Perhaps they were looking for some of the advanced facilities that could be provided by XML Schema and Namespaces. Frustrated by the deadlock, Userland went public with a version of the spec published as XML-RPC in the summer of 1998.

In 1999, as Microsoft was working on its version of XML Schema (XML Data) and adding support for namespaces in its XML products, the idea of SOAP gained momentum. It was still an XML-based RPC mechanism, however, which is why it met with resistance from the BizTalk (<http://www.biztalk.org>) team; the BizTalk model was based more on messaging than RPCs. SOAP 0.9 appeared for public review on September 13, 1999. It was submitted to the IETF as an Internet public draft. With few changes, in December 1999, SOAP 1.0 came to life.

Right before the XTech conference in March 2000, the W3C announced that it was looking into starting an activity in the area of XML protocols. At the conference, there was an exciting breakout session in which a number of industry visionaries argued the finer points of what XML protocols should do and where they were going—but this conversation didn't result in one solid vision of the future.






On May 8, 2000 SOAP 1.1 was submitted as a note to the W3C with IBM as a co-author. IBM's support was an unexpected and refreshing change. In addition, the SOAP 1.1 spec was much more modular and extensible, eliminating some concerns that backing SOAP implied backing a Microsoft proprietary technology. This, and the fact that IBM immediately released a Java SOAP implementation that was subsequently donated to the Apache XML Project (<http://xml.apache.org>) for open source development, convinced even the greatest skeptics that SOAP was something to pay attention to. Sun voiced support for SOAP and started work on integrating Web services into the J2EE platform. Not long after, many vendors and open source projects began working on Web service implementations.

In September 2000, the XML Protocol working group at the W3C was formed to design the XML protocol that was to become the core of XML-based distributed computing in the years to come. The group started with SOAP 1.1 as a foundation and produced the first working draft. After many months of changes, improvements, and difficult decisions about what to include, SOAP 1.2 became a W3C recommendation almost two years after that first draft, in June 2003.

## What Is SOAP, Really?

Despite the hype that surrounds it, SOAP is of great importance because it's the industry's best effort to date to standardize on the infrastructure technology for cross-platform XML distributed computing. Above all, SOAP is relatively simple. Historically, simplicity is a key feature of most successful architectures that have achieved mass adoption.

At its heart, SOAP is a specification for a simple yet flexible second-generation XML protocol. Because SOAP is focused on the common aspects of all distributed computing scenarios, it provides the following (covered in greater detail later):

- *A mechanism for defining the unit of communication*—In SOAP, all information is packaged in a clearly identifiable SOAP *message* . This is done via a SOAP *envelope*  that encloses all other information. A message can have a *body*  in which potentially arbitrary XML can be used. It can also have any number of *headers*  that encapsulate information outside the body of the message.
- *A processing model*—This defines a well-known set of rules for dealing with SOAP messages in software. SOAP's processing model is simple; but it's the key to using the protocol successfully, especially when extensions are in play.
- *A mechanism for error handling*—Using SOAP *faults* , you can identify the source and cause of an error and it allows for error diagnostic information to be exchanged between participants of an interaction.
- *An extensibility model*—This uses SOAP headers to implement arbitrary extensions on top of SOAP. Headers contain pieces of extensibility data which travel along with a message and may be *targeted* at particular nodes along the *message path*.

- *A flexible mechanism for data representation*—This mechanism allows for the exchange of data already serialized in some format (text, XML, and so on) as well as a convention for representing abstract data structures such as programming language datatypes in an XML format.
- *A convention for representing Remote Procedure Calls (RPCs) and responses as SOAP messages*—RPCs are a common type of distributed computing interaction, and they map well to procedural programming language constructs.
- *A protocol binding framework*—The framework defines an architecture for building bindings to send and receive SOAP messages over arbitrary underlying transports. This framework is used to supply a binding that moves SOAP messages across HTTP connections, because HTTP is a ubiquitous communication protocol on the Internet.

Before we dive deeper into the SOAP protocol and its specification, let's look at how our example company, SkatesTown, is planning to use SOAP and Web services.

## Doing Business with SkatesTown

When Al Rosen of Silver Bullet Consulting first began his engagement with SkatesTown, he focused on understanding the e-commerce practices of the company and its customers. After a series of conversations with SkatesTown's CTO, Dean Carroll, Al concluded the following:

- SkatesTown's manufacturing, inventory management, and supply chain automation systems are in good order. These systems are easily accessible by SkatesTown's Web-centric applications.
- SkatesTown has a solid consumer-oriented online presence. Product and inventory information is fed into an online catalog that is accessible to both direct consumers and SkatesTown's reseller partners via two different sites.
- Although SkatesTown's order-processing system is sophisticated, it's poorly connected to online applications. This is a pain point for the company because SkatesTown's partners are demanding better integration with their supply chain automation systems.
- SkatesTown's internal purchase order system is solid. It accepts purchase orders in XML format and uses XML Schema-based validation to guarantee their correctness. Purchase order item SKUs and quantities are checked against the inventory management system. If all items are available, an invoice is created. SkatesTown charges a uniform 5% tax on purchases and the higher of 5% of purchases or \$20 for shipping and handling.

Digging deeper into the order-processing part of the business, Al discovered that it uses a low-tech approach that has a high labor cost and isn't suitable for automation. One area that badly needs automation is the process of purchase order submission. Purchase orders

are sent to SkatesTown by email. All emails arrive in a single manager's account in operations. The manager manually distributes the orders to several subordinates. They have to open the email, copy only the XML over to the purchase order system, and enter the order there. The system writes an invoice file in XML format. This file has to be opened, and the XML must be copied and pasted into a reply email message. Simple misspellings of email addresses and cut-and-paste errors are common, and they cost SkatesTown and its partners money and time.

Another area that needs automation is the inventory checking process. SkatesTown's partners used to submit purchase orders without having a clear idea whether all the items were in stock. This often caused problems having to do with delayed order processing. Further, purchasing personnel from the partner companies would engage in long email dialogs with operations people at SkatesTown. To improve the situation, SkatesTown built a simple online application that communicates with the company's inventory management system. Partners can log in, browse SkatesTown's products, and check whether certain items are in stock, all via a standard web browser. This was a good start, but now SkatesTown's partners are demanding the ability to have their purchasing applications directly inquire about order availability.

Looking at the two areas that most needed to be improved, Al chose to focus first on the inventory checking process because the business logic was already present. He just had to enable better automation. To do this, he had to better understand how the application worked.

The logic for interacting with the inventory system is simple. Looking through the JSP pages that made up the online application, Al easily extracted the key business logic operations. Given a SKU and a desired product quantity, an application needs to get an instance of the SkatesTown product database and locate a product with a matching SKU. If such a product is available and if the number of items in stock is greater than or equal to the desired quantity, the inventory check succeeds. Since most of the example in this chapter will talk to the inventory system, let's take a slightly deeper look at its implementation.

### Note

A note of caution: this book's example applications demonstrate uses of Java technology and Web services to solve real business problems while at the same time remaining simple enough to fit in the book's scope and size limitations. To keep the code simple, we do as little data validation and error checking as possible without allowing applications to break. We don't define custom exception types or produce long, readable error messages. Also, to get away from the complexities of external system access, we use simple XML files to store data.

SkatesTown's inventory is represented by a simple XML file stored in `/resources/products.xml`. The inventory database XML format is as follows:

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

```

<product>
  <sku>947-TI</sku>
  <name>Titanium Glider</name>
  <type>skateboard</type>
  <desc>Street-style titanium skateboard.</desc>
  <price>129.00</price>
  <inStock>36</inStock>
</product>
...
</products>

```

By modifying this file, you can change the behavior of the examples. The Java representation of products in SkatesTown's systems is the `com.skatestown.data.Product` class; it's a simple bean that has one property for every element under product.

SkatesTown's inventory system is accessible via the `ProductDB` (for product database) class in package `com.skatestown.backend`. Listing 3.1 shows the key operations it supports. To construct an instance of the class, you pass an XML DOM Document object representation of `products.xml`. After that, you can get a listing of all products or search for a product by its SKU.

---

#### Listing 3.1 SkatesTown's Product Database Class

---

```

public class ProductDB
{
    private Product[] products;

    public ProductDB(Document doc) throws Exception
    {
        // Load product information
    }

    public Product getBySKU(String sku)
    {
        Product[] list = getProducts();
        for ( int i = 0 ; i < list.length ; i++ )
            if ( sku.equals( list[i].getSKU() ) ) return( list[i] );
        return( null );
    }

    public Product[] getProducts()
    {
        return products;
    }
}

```

---

This was all Al Rosen needed to know to move forward with the task of automating the inventory checking process.

## Inventory Check Web Service

SkatesTown's inventory check Web service is simple. The interaction model is that of an RPC. There are two input parameters: the product SKU (a string) and the quantity desired (an integer). The result is a simple Boolean value that's true if more than the desired quantity of the product is in stock and false otherwise.

### Choosing a Web Service Engine

Al decided to host all of SkatesTown's Web services on the Apache Axis Web service engine for a number of reasons:

- The open source implementation guaranteed that SkatesTown won't experience lock-in by a commercial vendor. Further, if any serious problems were discovered, a programmer could look at the code to see what was going on or fix the issue.
- Axis is one of the best Java-based Web services engines. It's better architected and much faster than its Apache SOAP predecessor. The core Axis team includes Web service gurus from companies such as Macromedia, IBM, Computer Associates, and Sonic Software.
- Axis is also one of the most extensible Web service engines. It can be tuned to support new versions of SOAP as well as the many types of extensions that current versions of SOAP allow for.
- Axis can run on top of a simple servlet engine or a full-blown J2EE application server. SkatesTown could keep its current J2EE application server without having to switch.

SkatesTown's CTO, Dean, agreed to have all Web services developed on top of Axis. Al spent some time on <http://ws.apache.org/axis> learning more about the technology and its capabilities.

### Service Provider View

To expose the inventory check Web service, Al had to do two things: implement the service backend and deploy it into the Web service engine. Building the backend for the inventory check Web service was simple because most of the logic was already available in SkatesTown's JSP pages. You can see the service class in Listing 3.2.

Listing 3.2 **Inventory Check Web Service Implementation**

---

```
package com.skatestown.services;

import com.skatestown.data.Product;
import com.skatestown.backend.ProductDB;
import com.skatestown.STConstants;
```

Listing 3.2 Continued

---

```

/**
 * Inventory check Web service
 */
public class InventoryCheck implements STConstants {
    /**
     * Checks inventory availability given a product SKU and
     * a desired product quantity.
     *
     * @param sku          product SKU
     * @param quantity     quantity desired
     * @return             true|false based on product availability
     * @exception Exception most likely a problem accessing the DB
     */
    public static boolean doCheck(String sku, int quantity)
        throws Exception
    {
        // Get the product database, which has been conveniently pre-placed
        // in a well-known place (if you want to see how this works,
        // check out the com.skatestown.GlobalHandler class!).
        ProductDB db = ProductDB.getCurrentDB();

        Product prod = db.getBySKU(sku);
        return (prod != null && prod.getNumInStock() >= quantity);
    }
}

```

---

The backend code for this service relies on the fact that some other piece of code has already made the appropriate `ProductDB` available via a static accessor method on the `ProductDB` class. We'll unearth the provider of `ProductDB` in Chapter 5, "Implementing Web Services with Apache Axis."

Once we have the `ProductDB`, the rest of the service code is trivial; we check if the quantity available for a given product is equal to or greater than the quantity requested, and return true if so.

## Deploying the Service

To deploy this initial service, Al chose to use the instant deployment feature of Axis: Java Web service (JWS) files. In order to do so, he saved the `InventoryCheck.java` file as `InventoryCheck.jws` underneath the Axis webapp, so it's accessible at <http://skatestown.com/axis/InventoryCheck.jws>.

## The Client View

Once the service was deployed, Al wanted some of SkatesTown's partners to test it. To test it himself, he built a simple client using Axis (see Listing 3.3).



---

**Listing 3.3 The InventoryCheck Client Class**

---

```
package ch3.ex2;

import org.apache.axis.AxisEngine;
import org.apache.axis.client.Call;
import org.apache.axis.soap.SOAPConstants;

/*
 * Inventory check Web service client
 */
public class InventoryCheckClient {
    /** Service URL */
    static String url =
        "http://localhost:8080/axis/InventoryCheck.jws";

    /**
     * Invoke the inventory check Web service
     */
    public static boolean doCheck(String sku, int quantity)
        throws Exception {
        // Set up Call object
        Call call = new Call(url);
        // Use SOAP 1.2 (default is SOAP 1.1)
        call.setSOAPVersion(SOAPConstants.SOAP12_CONSTANTS);
        // Set up parameters for invocation
        Object[] params = new Object[] { sku, new Integer(quantity) };
        // Call it!
        Boolean result = (Boolean)call.invoke("", "doCheck", params);
        return result.booleanValue();
    }

    public static void main(String[] args) throws Exception {
        String sku = args[0];
        int quantity = Integer.parseInt(args[1]);
        System.out.println("Making SOAP call...");
        boolean result = doCheck(sku, quantity);
        if (result) {
            System.out.println(
                "Confirmed - the desired quantity is available");
        } else {
            System.out.println(
                "Sorry, the desired quantity is not available.");
        }
    }
}
```

---

The client uses Axis's `call` class, which is the central client-side API. When Al constructs the `call` class, he passes in the URL of his deployed service so that the `call` knows where to send SOAP messages. The actual invocation is simple: He knows he's calling the `doCheck()` method, so he passes the method name and an array of arguments (obtained from the command line) to the `invoke()` method on the `call` object. The results come back as a Boolean object, and when the client is run, it looks like this:

```
% java InventoryCheckClient SKU-56 35
Making SOAP call...
Confirmed - the desired quantity is available.
%
```

## A Closer Look at SOAP

The current SOAP specification is version 1.2, which was released as a W3C recommendation in June 2003. At the time of this writing (early 2004), toolkits are just starting to offer complete support for the new version, and most of them still use SOAP 1.1 as a baseline. Since this chapter is primarily about the SOAP protocol, we'll focus on SOAP 1.2—the standard the industry will be using into the future. The 1.1 version is also critically important, so we'll also explain it and use sidebars to call out differences between the versions as we go. (You can find an exhaustive list of differences between SOAP 1.1 and SOAP 1.2 in the SOAP 1.2 Primer: <http://www.w3.org/TR/2003/REC-soap12-part0-20030624/>.) Most of the other examples in this book use SOAP 1.1, but we want you to be a 1.2-ready developer.

### The Structure of the Spec

The SOAP 1.2 specification is the ultimate reference to the SOAP protocol; the latest version is at <http://www.w3.org/TR/SOAP>. The spec is divided into two parts:

- *Part 1, the Messaging Framework*—Lays out the central foundation of SOAP, consisting of the processing model, the extensibility model, and the message structure.
- *Part 2, Adjuncts*—Important adjuncts to the core spec defined in Part 1. Although they're extensions (and therefore by definition optional), they serve two critical purposes. First, they act as proofs-of-concept for the modular design of SOAP, demonstrating that it isn't limited, for instance, to only being used over HTTP (a common misconception). Second, the core of SOAP in Part 1 isn't enough to build something usable for functional interoperable services. The extensions in part 2, in particular the HTTP binding, provide a baseline for implementers to use, even though the marketplace may define other components beyond those in the spec as well.

## Infosets

The SOAP 1.2 spec has been written in terms of the XML *infoset*, which is an abstract model of all the information in an XML document or document fragment. When the spec talks about “element information items” instead of just elements, it means that what is important is the structure of the information, not necessarily the fact that it’s serialized with angle brackets. As you’ll see later, this becomes important when we talk about bindings. The key thing to remember is that all the information items are really abstract ways of talking about things like elements and attributes that you see in everyday XML. So this XML

```
<elem attr="foo">
  <childE1>text</childE1>
  Other text
</elem>
```

would abstractly look like the structure in Figure 3.1 (rectangles are elements, rounded rectangles attributes, and ovals text).

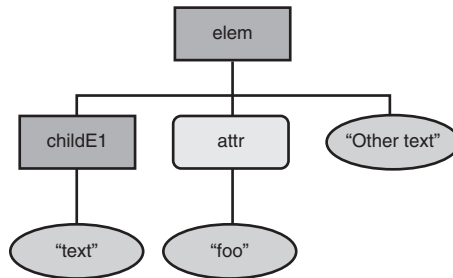


Figure 3.1 A simple XML infoset

## The SOAP Messaging Framework

The first part of the SOAP specification is primarily concerned with defining how SOAP messages are structured and the rules processors must abide by when producing and consuming them. Let’s look at a sample SOAP message, the inventory check request described in our earlier example:

### Note

All the wire examples in this book have been obtained by using the `tcpmon` tool, which is included in the Axis distribution you can obtain with the example package from the Sams Web site. `Tcpmon` (short for *TCP monitor*) allows you to record the traffic to and from a particular TCP port, typically HTTP requests and responses. We’ll go into detail about this utility in Chapter 5.

```

POST /axis/InventoryCheck.jws HTTP/1.0
Content-Type: application/soap+xml; charset=utf-8

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
  <doCheck soapenv:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
    <arg0 xsi:type="soapenc:string"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">947-TI</arg0>
    <arg1 xsi:type="soapenc:int"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">3</arg1>
  </doCheck>
</soapenv:Body>
</soapenv:Envelope>

```

This is clearly an XML document (Chapter 2, “XML Primer,” covered XML in detail), which has been sent via an HTTP POST. We’ve removed a few of the nonrelevant HTTP headers from the trace, but we left the content-type header, which indicates that this POST contains a SOAP message (note that this content-type would be different for SOAP 1.1—see the sidebar for details). We’ll cover the HTTP-specific parts of SOAP interactions further a bit later in the chapter.

The root element is `soapenv:Envelope`, in the `http://www.w3.org/2003/05/soap-envelope` namespace, which surrounds a `soapenv:Body` containing application-specific content that represents the central purpose of the message. In this case we’re asking for an inventory check, so the central purpose is the `doCheck` element. The `Envelope` element has a few useful namespace declarations on it, for the SOAP envelope namespace and the XML Schema data and instance namespaces.

### SOAP 1.1 Difference: Identifying SOAP Content

The SOAP 1.1 envelope namespace is `http://schemas.xmlsoap.org/soap/envelope/`, whereas for SOAP 1.2 it has changed to `http://www.w3.org/2003/05/soap-envelope`. This namespace is used for defining the envelope elements and for versioning, which we will explain in more detail in the “Versioning in SOAP” section.

The content-type used when sending SOAP messages across HTTP connections has changed as well—it was `text/xml` for SOAP 1.1 but is now `application/soap+xml` for SOAP 1.2. This is a great improvement, since `text/xml` is a generic indicator for any type of XML content. The content type was so generic that machines had to use the presence of a custom HTTP header called `SOAPAction`: to tell that XML traffic was, in fact, SOAP (see the section on the HTTP binding for more). Now the standard MIME infrastructure handles this for us.

The `doCheck` element represents the remote procedure call to the inventory check service. We’ll talk more about using SOAP for RPCs in a while; for now, notice that the

name of the method we're invoking is the name of the element directly inside the `soapenv:Body`, and the arguments to the method (in this case, the SKU number and the quantity desired) are encoded inside the method element as `arg0` and `arg1`. The real names for these parameters in Java are `SKU` and `quantity`; but due to the ad-hoc way we're calling this method, the client doesn't have any way of knowing that information, so it uses the generated names `arg0` and `arg1`.

The response to this message, which comes back across in the HTTP response, looks like this:

```
Content-Type: application/soap+xml; charset=utf-8
```

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <doCheckResponse
      soapenv:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
      <rpc:result xmlns:rpc="http://www.w3.org/2003/05/soap-rpc">return</rpc:result>
      <return xsi:type="xsd:boolean">true</return>
    </doCheckResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

The response is also a SOAP envelope, and it contains an encoded representation of the result of the RPC call (in this case, the Boolean value `true`).

What good is having this envelope structure, when we could send our XML formats directly over a transport like HTTP without a wrapper? Good question; as we answer it, we'll examine some more details of the protocol.

## Vertical Extensibility

Let's say you want your purchase order to be extensible. Perhaps you want to include security in the document someday, or you might want to enable a notarization service to associate a token with a particular purchase order, as a third-party guarantee that the PO was sent and contained particular items. How might you make that happen?

You could drop extensibility elements directly into your document before sending it. If we took the purchase order from the last chapter and added a notary token, it might look something like this:

```
<po id="43871" submitted="2004-01-05" customerId="73852">
  <notary:token xmlns:notary="http://notaries-r-us.com">
    XQ34Z-4G5
  </notary:token>
  <billTo>
```

```

    <company>The Skateboard Warehouse</company>
    ...
  </billTo>
  ...
</po>

```

To do things this way, and make it easy for your partners to use, you'd need to do two things. First, your schema would have to be explicitly extensible at any point in the structure where you might want to add functionality later (this can be accomplished in a number of ways, including the `xsd:any/` schema construct); otherwise, documents containing extension elements wouldn't validate. Second, you would need to agree on rules by which those extensibility elements were to be processed—which ones are optional, which ones affect which parts of the document, and so on. Both of these requirements present challenges. Not all schemas have been designed for extensibility, and you may need to extend a document that follows a preexisting standard format that wasn't built that way. Also, processing rules might vary from document type to document type, so it would be challenging to have a uniform model with which to build a common processor. It would be nice to have a standardized framework for implementing arbitrary extensibility in a way that everyone could agree on.

It turns out that the SOAP envelope, in addition to containing a body (which must always be present), may also contain an optional `Header` element—and the SOAP `Header` structure gives us just what we want in an XML extensibility system. It's a convenient and well-defined place in which to put our extensibility elements. Headers are just XML elements that live inside the `soapenv:Header/soapenv:Header` tags in the envelope. The `soapenv:Header` always appears, incidentally, *before* the `soapenv:Body` if it's present. (Note that in the SOAP 1.2 spec, the extensibility elements are known as *header blocks*. However, the industry—and the rest of this book—colloquially refers to them simply as *headers*.)

Let's look at the extensibility example recast as a SOAP message with a header:

```

<soapenv:Envelope
  xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
  <soapenv:Header>
    <notary:token xmlns:notary="http://notaries-r-us.com">
      XQ34Z-4G5
    </notary:token>
  </soapenv:Header>
  <soapenv:Body>
    <PO>
      ...normal purchase order here...
    </PO>
  </soapenv:Body>
</soapenv:Envelope>

```

Since the SOAP envelope wraps around whatever XML content you want to send in the body (the PO, in this example), you can use the `Header` to insert extensions (the

notary:token header) without modifying the central core of the message. This can be compared to a situation in real life where you want to send a document and some auxiliary information, but you don't want to mark up the document—so you put the document inside an envelope and then add another piece of paper or two describing your extra information.

Each individual header represents one piece of extensibility information that travels with your message. A lot of other protocols have this same basic concept—we're all familiar with the email model of headers and body. HTTP also contains headers, and both email and HTTP use the concept of extensible, user-defined headers. However, the headers in protocols like these are simple strings; since SOAP uses XML, you can encode much richer data structures for individual headers. Also, you can use XML's structure to make processing headers much more powerful and flexible than a basic string-based model.

Headers can contain any sort of data imaginable, but typically they're used for two purposes:

- *Extending the messaging infrastructure*—Infrastructure headers are typically processed by middleware. The application doesn't see the headers, just their effects. They could be things like security credentials, correlation IDs for reliable messaging, transaction context identifiers, routing controls, or anything else that provides services to the application.
- *Defining orthogonal data*—The second category of headers is application defined. These contain data that is orthogonal to the body of the message but is still destined for the application on the receiving side. An example might be extra data to accompany nonextensible schemas—if you wanted to add more customer data fields but couldn't change the `billTo` element, for instance.


Using headers to add functionality to messages is known as *vertical* extensibility, because the headers build on top of the message. A little later we'll discuss horizontal extensibility as well.

Now that you know the basics, we'll consider some of the additional framework that SOAP supplies for headers and how to use it. After that, we'll explain the SOAP processing model, which is the key to SOAP's scalability and expressive power.

## The mustUnderstand Flag

Some extensions might use headers to carry data that's nice to know but not critical to the main purpose of the SOAP message. For instance, you might be invoking a “buy book” operation on a store's Web service. You receive a header in the response confirmation message that contains a list of other books the site thinks you might find interesting. If you know how to process that extension, then you might offer a UI to access those books. But if you don't, it doesn't matter—your original request was still processed successfully. On the other hand, suppose the *request* message of that same “buy book” operation contained private information (such as a credit card number). The sender might

want to encrypt the XML in the SOAP body to prevent snooping. To make sure the other side knows what to do with the postencryption data inside the body, the sender inserts a header that describes how to decrypt the message. *That* header is important, and anyone trying to process the message without correctly processing the header and decrypting the body is going to run into trouble.

This is why we have the `mustUnderstand`  attribute, which is always in the SOAP envelope namespace. Here's what our notary header would look like with that attribute:

```
<notary:token xmlns:notary="http://notaries-r-us.com"
  soapenv:mustUnderstand="true">
  XQ34Z-4G5
</notary:token>
```


By marking things `mustUnderstand` (when we refer to headers “marked `mustUnderstand`,” we mean having the `soapenv:mustUnderstand` attribute set to `true`), you're saying that the receiver must agree to all the terms of your extension specification or they can't process the message. If the `mustUnderstand` attribute is set to `false` or is missing, the header is defined as optional—in this case, processors not familiar with the extension can still safely process the message and ignore the optional header.

#### SOAP 1.1 Difference: `mustUnderstand`

In SOAP 1.2, the `mustUnderstand` attribute may have the values `0/false` (false) or `1/true` (true). In SOAP 1.1, despite the fact that XML allows `true` and `false` for Boolean values, the only legal `mustUnderstand` values are `0` and `1`.

The `mustUnderstand` attribute is a key part of the SOAP processing model, since it allows you to build extensions that fundamentally change how a given message is processed in a way that is guaranteed to be interoperable. *Interoperable* here means that you can always know how to gracefully fail in the face of extensions that aren't understood.


## SOAP Modules

When you implement a semantic using SOAP headers, you typically want other parties to use your extension, unless it's purely for internal use. As such, you typically write a specification that details all the constraints, rules, preconditions, and data formats of your extension. These specifications are known as *SOAP modules* . Modules are named with URIs so they can be referenced, versioned, and reasoned about. We'll talk more about module specifications when we get to the SOAP binding framework a bit later.


## SOAP Intermediaries

So far, we've addressed SOAP headers as a means for vertical extensibility within SOAP messages. There is another related notion, however: *horizontal* extensibility. Whereas vertical extensibility is about the ability to introduce new pieces of information within a



SOAP message, horizontal extensibility is about targeting different parts of the same SOAP message to different recipients. Horizontal extensibility is provided by SOAP intermediaries .

## The Need for Intermediaries

SOAP intermediaries are applications that can process parts of a SOAP message as it travels from its origination point to its final destination point. The route taken by a SOAP message, including all intermediaries it passes through, is called the SOAP *message path*  (see Figure 3.2).

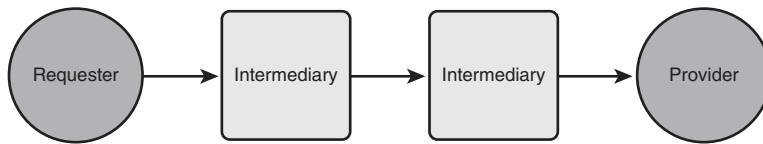


Figure 3.2 The SOAP message path

Intermediaries can both accept and forward SOAP messages, and they usually do some form of message processing as well. Three key use-cases define the need for SOAP intermediaries: crossing trust domains, ensuring scalability, and providing value-added services along the SOAP message path.

Crossing trust domains is a common issue faced while implementing security in distributed systems. Consider the relation between a corporate or departmental network and the Internet. For small organizations, it's likely that the IT department has put most computers on the network within a single trusted security domain. Employees can see their co-workers' computers as well as the IT servers, and they can freely exchange information between them without the need for separate logons. On the other hand, the corporate network probably treats all computers on the Internet as part of a separate security domain that isn't trusted. Before an Internet request reaches the network, it needs to cross from its untrustworthy domain to the trusted domain of the internal network. Corporate firewalls and virtual private network (VPN) gateways guard the network: Their job is to let some requests cross the trust domain boundary and deny access to others.

Another important need for intermediaries arises because of the scalability requirements of distributed systems. A simplistic view of distributed systems could identify two types of entities: those that request work to be done (clients) and those that do the work (servers). Clients send messages directly to the servers they want to communicate with. Servers, in turn, get some work done and respond. In this naïve universe, there is little need for distributed computing infrastructure. However, we can't use this model to build highly scalable distributed systems.

Take email as an example. When `someone@company.com` sends an email message to `myfriend@london.co.uk`, it's not the case that their email client locates the mail server `london.co.uk` and sends the message to it. Instead, the client sends the message to its email server at `company.com`. Based on the priority of the message and how busy the mail server is, the message will leave either by itself or in a batch of other messages. (Messages are often batched to improve performance.) The message will probably make a few hops through different nodes on the Internet before it gets to the mail server in London.

The lesson from this example is that highly scalable distributed systems (such as email) require flexible buffering of messages and routing based both on message parameters such as origin, destination, and priority, and on the state of the system considering factors such as the availability and load of its nodes as well as network traffic information. Intermediaries hidden from the eyes of the originators and final recipients of messages can perform this work behind the scenes.

Finally, we need intermediaries so that we can provide value-added services in a distributed system. The type of services can vary significantly, and some of them involve the message sender being explicitly aware of the intermediary, unlike our previous examples. Here are a couple of common scenarios:

- *Securing message exchanges, particularly through untrustworthy domains*—You could secure SOAP messages by passing them through an intermediary that first encrypts them and then digitally signs them. On the receiving side an intermediary would perform the inverse operations: checking the digital signature and, if it's valid, decrypting the message.
- *Notarization/nonrepudiation*—when the sender or receiver (or both) desires a third party to make a record of an interaction, a *notarizing intermediary* is a likely solution. Instead of sending the message directly to the receiver, the sender sends to the intermediary, who makes a persistent copy of the request and then sends it to the service provider. The response typically comes back via the intermediary as well, and then both parties are usually given a token they can use to reference the transaction record in the future.
- *Providing message tracing facilities*—Tracing allows the message recipient to find out the path the message followed, complete with detailed timings of arrivals and departures to and from intermediaries. This information is indispensable for tasks such as measuring quality of service (QoS), auditing systems, and identifying scalability bottlenecks.

### Transparent and Explicit Intermediaries

Message senders may or may not be aware of intermediaries in the message path. A *transparent intermediary* is one the client knows nothing about—the client believes it's sending

messages to the actual service endpoint, and the fact that an intermediary is doing work in the middle is incidental. An *explicit intermediary*, on the other hand, involves specific knowledge on the part of the client—the client knows the message will travel through an intermediary before continuing to its ultimate destination.

The security intermediaries discussed earlier would likely be transparent; the organization providing the service would publish the outward-facing address of the intermediary as the service endpoint. The notarization service described earlier would be an example of an explicit intermediary—the client would know that a notarization step was going on.

## Intermediaries in SOAP

SOAP is specifically designed with intermediaries in mind. It has simple yet flexible facilities that address the three key aspects of an intermediary-enabled architecture:

- How do you pass information to intermediaries?
- How do you identify who should process what?
- What happens to information that is processed by intermediaries?

All header elements can optionally have the `soapenv:role` attribute. The value of this attribute is a URI that identifies who should handle the header entry. Essentially, that URI is the name of the intermediary. This URI might mean a particular node (for instance “the Solaris machine on John’s desk”), or it might refer to a class of nodes (as in, “any cache manager along the message path”). (This latter case prompted the name change from *actor* to *role* in SOAP 1.2.) Also, a given node can play multiple roles: the Solaris machine on John’s desk might also be a cache manager, for instance, so it would recognize either role URI.

The first step any node takes when processing a SOAP message is to collect all the headers that are targeted at the node—this means headers that have a `role` attribute matching *any* of the roles node is playing. It then looks through these nodes for headers marked `mustUnderstand` and confirms that it recognizes each such header and is able to process it in accordance with the rules associated with that SOAP header. If it finds a `mustUnderstand` header that it doesn’t recognize, it must immediately stop processing.

There are several special values for the `role` attribute:

- `http://www.w3.org/2003/05/soap-envelope/role/next`—Indicates that the header entry’s recipient is the next SOAP node that processes the message. This is useful for hop-by-hop processing required, for example, by message tracing.
- `http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver`—Refers to the final recipient of the SOAP message. Note that omitting the `role` attribute or using an empty value (“”) also implies that the final recipient of the SOAP message should process the header entry. The final recipient of the SOAP message is the same node that processes the body.

- <http://www.w3.org/2003/05/soap-envelope/role/none>—A special role that no SOAP node should ever assume. That means that headers addressed to this role should never be processed; and since no one will ever be in this role, the value of the `mustUnderstand` attribute won't matter for such headers (remember that the first thing a SOAP node does is pick out the headers it can see by virtue of playing the right role, before looking at `mustUnderstand`). Also note that the `relay` attribute (discussed later) never matters on a header addressed to the `none` role, for the same reason. Even though your SOAP node can't act as the `none` role, it can still look at the data inside headers marked as `none`. So, headers marked for the `none` role can still be used to carry data. (We'll give an example in a bit.)

#### SOAP 1.1 Difference: actor versus role

In SOAP 1.1, the attribute used to target headers is called `actor`, not `role`. Also, SOAP 1.1 only specifies a special `next actor` URI (<http://schemas.xmlsoap.org/soap/actor/next>), not an `actor` for `none` or `ultimateRecipient`.

## Forwarding and Active Intermediaries

Some intermediaries, like the notarization example discussed earlier, only do processing related to particular headers in the SOAP envelope before forwarding the message to the next node in the message path. In other words, the work of the intermediary is defined by the contents of the incoming messages. These are known as *forwarding* intermediaries.

Other intermediaries do processing and potentially modify the message in ways *not* defined by the message contents. For instance, an intermediary at a company boundary to the outside world might add a digital signature header to every outbound message to ensure that receivers can check the integrity of all messages. No explicit markers in the messages are used to trigger this behavior; the node simply does it. This type of intermediary is known as an *active* intermediary.

Either type of intermediary may do arbitrary work on the message (including the body) based on its internal rules.

## Rules for Intermediaries and Headers

By default, all headers targeted at a particular intermediary are removed from the message when it's forwarded on to the next node. This is because the specification tells us that the contract implied by a given header is between the sender of that header and the first node satisfying the role at which it's targeted. Headers that aren't targeted at a particular intermediary should, in general, be forwarded through untouched (see Figure 3.3).

An intermediary removes headers targeted at any role it's playing, regardless of whether they're understood. In Figure 3.4, one header is processed and then removed; another isn't understood, but because it's targeted at our intermediary and not marked `mustUnderstand`, it's still removed.

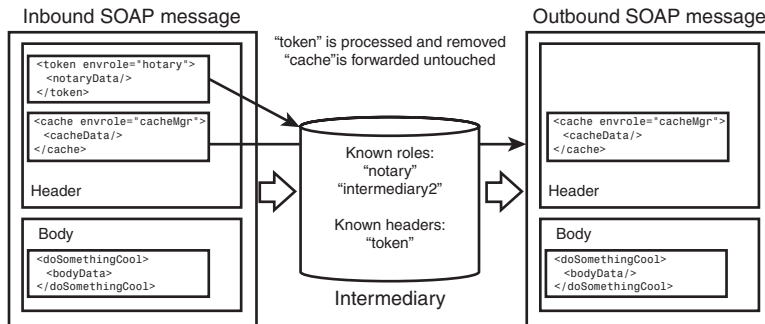


Figure 3.3 Intermediary header removal

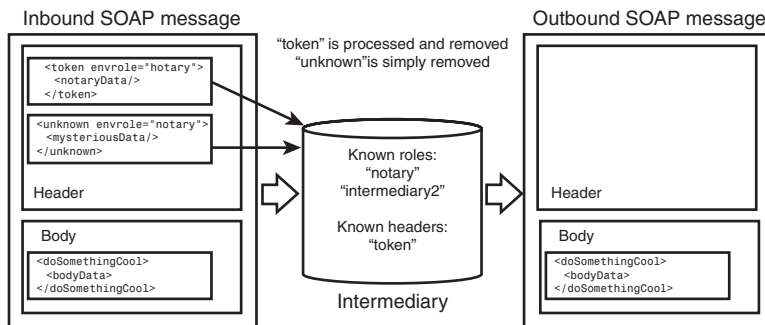


Figure 3.4 Removing optional headers targeted at an intermediary

There are two exceptions to the removal rules. First, the specification for a particular extension may explicitly indicate that an identical copy of a given header from the incoming message is supposed to be placed in the outgoing message. Such headers are known as *reinserted*, and this has the effect of forwarding them through after processing. An example might be a logging extension targeted at a `logManager`. Any log manager receiving it along the message path would make a persistent copy of the message for logging purposes and then reinsert the header so that other log managers later in the chain could do the same.

The second exception is when you want to indicate to intermediaries that extensions targeted at them, but not understood, should still be passed through. SOAP 1.2 introduces the `relay` attribute for this purpose. If the `relay` attribute is present on a header which is targeted at a given intermediary, and it has the value `true`, the intermediary should forward the header regardless of whether it understands it. Figure 3.5 shows an unknown header arriving at our notary intermediary. Since all nodes must recognize the next role, the unknown header is targeted at the intermediary. Despite the fact that the intermediary doesn't understand the header, it's forwarded because the `relay` attribute is `true`.

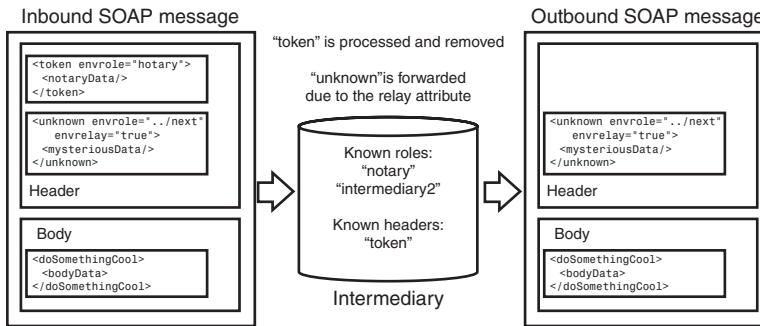


Figure 3.5 Forwarding headers with the `relay` attribute

## The SOAP Body

The SOAP `Body` element immediately surrounds the information that is core to the SOAP message. All immediate children of the `Body` element are body entries (typically referred to as *bodies*). Bodies can contain arbitrary XML. Sometimes, based on the intent of the SOAP message, certain conventions govern the format of the SOAP body (for instance, we discuss the conventions for representing RPCs and communicating error information later).

When a node that identifies itself as the ultimate recipient (the service provider in the case of requests, or the client in the case of responses) receives a message, it's required to process the contents of the body and perform whatever actions are appropriate. The body carries the core of the SOAP message.

## The SOAP Processing Model

Now we're ready to finish describing the SOAP 1.2 processing model. Here are the steps a processor must perform when it receives a SOAP message, as described in the spec:

1. Determine the set of roles in which the node is to act. The contents of the SOAP envelope, including any SOAP header blocks and the SOAP body, *may* be inspected in making such determination.
2. Identify all header blocks targeted at the node that are mandatory.
3. If one or more of the SOAP header blocks identified in step 2 aren't understood by the node, then generate a single SOAP fault with the value of `Code` set to `env:mustUnderstand`. If such a fault is generated, any further processing *must not* be done. Faults related to processing the contents of the SOAP body *must not* be generated in this step.

4. Process all mandatory SOAP header blocks targeted at the node and, in the case of an ultimate SOAP receiver, the SOAP body. A SOAP node *may* also choose to process nonmandatory SOAP header blocks targeted at it.
5. In the case of a SOAP intermediary, and where the SOAP message exchange pattern and results of processing (for example, no fault generated) require that the SOAP message be sent further along the SOAP message path, relay the message.

The processing model has been designed to let you use `mustUnderstand` headers to do anything you want. We could imagine a `mustUnderstand` header, for instance, that tells the processor at the next hop to process all headers and ignore the `role` attribute.

## Versioning in SOAP

One interesting note about SOAP is that the `Envelope` element doesn't expose any explicit protocol version in the style of other protocols such as HTTP (HTTP/1.0 versus HTTP/1.1) or even XML (`?xml version="1.0"?`). The designers of SOAP explicitly made this choice because experience had shown simple number-based versioning to be fragile. Further, across protocols, there were no consistent rules for determining what changes in major versus minor version numbers mean.

Instead of going this way, SOAP leverages the capabilities of XML namespaces and defines the protocol version to be the URI of the SOAP envelope namespace. As a result, the only meaningful statement you can make about SOAP versions is that they are the same or different. It's no longer possible to talk about compatible versus incompatible changes to the protocol.

This approach gives Web service engines a choice of how to treat SOAP messages that have a version other than the one the engine is best suited for processing. Because an engine supporting a later version of SOAP will know all previous versions of the specification, it has options based on the namespace of the incoming SOAP message:

- If the message version is the same as any version the engine knows how to process, it can process the message.
- If the message version is recognized as older than any version the engine knows how to process, or older than the preferred version, it should generate a `VersionMismatch` fault and attempt to negotiate the protocol version with the client by sending information regarding the versions it can accept. SOAP 1.1 didn't specify how such information might be encoded, but SOAP 1.2 introduces the `soapenv:Upgrade` header for this purpose. (We'll describe it in detail when we cover faults.)
- If the message version is newer than any version the engine knows how to process (in other words, completely unrecognized), it must generate a `VersionMismatch` fault.

The simple versioning based on the namespace URI results in fairly flexible and accommodating behavior of Web service engines.

## Processing Headers and Bodies

The SOAP spec has a specific meaning for the word *process*. Essentially, it means to fulfill the contract indicated by a particular piece of a SOAP message (a header or body). Processing a header means following the rules of that extension, and processing the body means performing whatever operation is defined by the service.

SOAP says you don't have to process an element in order to look at it as a part of other processing. So even though an intermediary might, for instance, encrypt the body as a message passes through it, we don't consider this processing in the SOAP sense, because encrypting the body isn't the same as doing what the body requests.

This gets back to the question of why you might use the `none` role. Imagine that SkatesTown wants to extend its purchase order schema by adding additional customer information. The company didn't design the schema for explicit extensibility, so adding elements in the middle will cause any older systems receiving the new XML to fail validation. SkatesTown can continue to use the old schema in the body but add arbitrary additional information in a SOAP header. That way, newer systems will notice the extensions and use them, but older ones won't be confused. This header would be purely data, without an associated SOAP module specification and processing rules, so it would make sense for SkatesTown to target the header at the `none` role to make sure no one tries to process it.

## Faults: Error Handling in SOAP

When something goes wrong in Java, we expect someone to throw an exception; the exception mechanism gives us a common framework with which to deal with problems. The same is true in the SOAP world. When a problem occurs, the SOAP spec provides a well-known way to indicate what has happened: the SOAP fault. Let's look at an example fault message:

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
  xmlns:st="http://www.skatestown.com/ws">
  <env:Header>
    <st:PublicServiceAnnouncement>
      Skatestown's Web services will be unavailable after 5PM today
      for a two hour maintenance window.
    </st:PublicServiceAnnouncement>
  </env:Header>
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Sender</env:Value>
        <env:Subcode>
          <env:Value>st:InvalidPurchaseOrder</env:Value>
        </env:Subcode>
      </env:Code>
    </env:Fault>
  </env:Body>
</env:Envelope>
```



```
<env:Reason>
  <env:Text xml:lang="en-US">
    Your purchase order did not validate!
  </env:Text>
</env:Reason>
<env:Detail>
  <st:LineNumber>9</st:LineNumber>
  <st:ColumnNumber>24</st:ColumnNumber>
</env:Detail>
</env:Fault>
</env:Body>
</env:Envelope>
```

## Structure of a Fault

A SOAP fault message is a normal SOAP message with a single, well-known element inside the body: `soapenv:Fault`. The presence of that element acts as a signal to processors to indicate something has gone wrong. Of course, just knowing something is wrong is rarely useful enough; you need a structure to help determine what happened so you can either try again with a better idea of what might work or let the user know the problem. SOAP faults have several components to help in this regard.

### Fault Code

The fault code is the first place to look, since it tells you in a general sense what the problem was. Fault codes are QNames, and SOAP defines the set of legal codes as follows (each item is the local part of the QName—the namespace is always the SOAP envelope namespace):

- **Sender**—The problem was caused by incorrect or missing data from the sender. For instance, if a service required a security header in order to do its work and it was called without one, it would generate a `Sender` fault. You typically have to make a change to your message before resending it if you hope to be successful.
- **Receiver**—Something went wrong on the receiver while processing the message, but it wasn't directly attributable to the message contents. For example, a necessary resource like a database was down, a thread wasn't available, and so on. A message causing a `Receiver` fault might succeed if resent at a later time.
- **mustUnderstand**—This fault code indicates that a header was received that was targeted at the receiving node, marked `mustUnderstand="true"`, and not understood.
- **VersionMismatch**—The `VersionMismatch` code is generated when the namespace on the SOAP envelope that was received isn't compatible with the SOAP version on the receiver. This is the way SOAP handles protocol versioning; we'll talk about it in more detail later.

The fault code resides inside the `Code` element in the fault, in a subelement called `Value`. In the example code, you can see the `Sender` code, meaning something must have been wrong with the request that caused this fault. We have the `Value` element instead of putting the code `qname` directly inside the `Code` element so that we can extend the expressive space of possible fault codes by adding more data inside another element, `Subcode`.

### Subcodes

SOAP 1.2 lets you specify an arbitrary hierarchy of fault subcodes, which provide further detail about what went wrong. The syntax is a little verbose, but it works. Here's an example:

```
<env:Code>
  <env:Value>env:Sender</env:Value>
  <env:Subcode>
    <env:Value>st:InvalidPurchaseOrder</env:Value>
  </env:Subcode>
</env:Code>
```

The `Code` element contains an optional `Subcode` element. Just as `Code` contains a mandatory `Value`, so too does each `Subcode`—and each `Subcode` may contain another `Subcode`, to whatever level of nesting is desired. Generally the hierarchy won't go more than about three levels deep. In our example, the subcode tells us that the problem was an invalid purchase order.

### Reason

The `Reason` element, also required, contains one or more human-readable descriptions of the fault condition. Typically, the reason text might appear in a dialog box that alerts the user of a problem, or it might be written into a log file. The `Text` element contains the text and there can be one or more such messages. Why would you have more than one? In the increasingly international environment of the Web, you might wish to send the fault description in several languages, as in this example from the SOAP primer:

```
<env:Reason>
  <env:Text xml:lang="en-US">Processing error</env:Text>
  <env:Text xml:lang="cs">Chyba zpracování</env:Text>
</env:Reason>
```

The spec states that if you have multiple `Text` elements, you should have a different value for `xml:lang` in each one—otherwise you might confuse the software that's trying to print out a single coherent message in a given language.

### Node and Role

The optional `Node` element, not shown in our example, tells us which SOAP node (the sender, an intermediary, or the ultimate destination) was processing the message at the time the fault occurred. It contains a URI.

The `Role` element tells which role the faulting node was playing when the fault occurred. It contains a URI that has exactly the same semantics, and the same values, as the `role` attribute we described when we were talking about headers. Note the difference between this element and `Node`—`Node` tells you *which* SOAP node generated the fault, and `Role` tells *what* part that node was playing when it happened. The `Role` element is also optional.

### Fault Details

We have a custom fault code and a fault message, both of which can tell a user or software something about the problem; but in many cases, we would also like to pass back some more complex machine-readable data. For example, you might want to include a stack trace while you're developing services to aid with debugging (though you likely wouldn't do this in a production application, since stack traces can sometimes give away information that might be useful to someone trying to compromise your system).

You can place anything you want inside the SOAP fault's `Detail` element. In our example at the beginning of the section, the line number and column number where the validation error occurred are expressed, so that automated tools might be able to help the user or developer to fix the structure of the transmitted message.

#### SOAP 1.1 Difference: Handling Faults

Faults in SOAP 1.2 got an overhaul from SOAP 1.1's version. All the subelements of the SOAP `Fault` element in SOAP 1.1 are unqualified (in no namespace). The `Fault` subelements in SOAP 1.2 are in the `envelope` namespace.

In SOAP 1.1, there is no `Subcode`, only a single `faultcode` element. The SOAP 1.1 fault code is a QName, but its hierarchy is achieved through dots rather than explicit structure—in other words, whereas in SOAP 1.1 you might have seen

```
<faultcode>env:Sender.Authorization.BadPassword</faultcode>
```

in SOAP 1.2 you see something like:

```
<env:Code>
  <env:Value>env:Sender</env:Value>
  <env:Subcode>
    <env:Value>myNS:Authorization</env:Value>
    <env:Subcode>
      <env:Value>myNS:BadPassword</env:Value>
    </env:Subcode>
  </env:Subcode>
</env:Code>
```

The `env:Reason` element in SOAP 1.2 is called `faultstring` in SOAP 1.1. Also, 1.1 only allows a single string inside `faultstring`, whereas 1.2 allows different `env:Text` elements inside `env:Reason` to account for different languages.

The `Client` fault code from 1.1 is now `Sender`, which is less prone to interpretation. Similarly, 1.1's `Server` fault code is now `Receiver`.

In SOAP 1.1, the `detail` element is used only for information pertaining to faults generated when processing the SOAP body. If a fault is generated when processing a header, any machine-readable information about the fault must travel in headers on the fault message. The reasoning for this went something like this: Headers exist so that SOAP can support orthogonal extensibility; that means you want a given message to be able to carry several extensions that might not have been designed by the same people and might have no knowledge of each other. If problems occurred that caused each of these extensions to want to pass back data, they might have to fight for the `detail` element. The problem with this logic is that the `detail` element isn't a contended resource, in the same way the `soapenv:Header` isn't a contended resource. If multiple extensions want to drop their own elements into `detail`, that works just as well as putting their own headers into the envelope. So this restriction was dropped in SOAP 1.2, and `env:Detail` can contain anything your application desires—but the rule still must be followed for SOAP 1.1.

SOAP 1.2 introduces the `NotUnderstood` header and the `Upgrade` header, both of which exist in order to clarify what went wrong with particular faults (`mustUnderstand` and `VersionMismatch`) in a standard way.

## Using Headers in Faults

Since a fault is also a SOAP message, it can carry SOAP headers as well as the fault structure. In our example at the beginning of this section, you can see that `SkatesTown` has included a public service announcement header. This optional information lets anyone who cares know that the Web services will be down for maintenance; and since it isn't marked `mustUnderstand`, it doesn't affect the processing of the fault message in any way. SOAP defines some headers specifically for use in faults.

### The NotUnderstood Header

You'll recall that SOAP processors are forced to fault if they encounter a `mustUnderstand` header that they should process but don't understand. It's great to know something wasn't understood, but it's more useful if you have an indication of *which* header was the cause of the problem. That way you might be able to try again with a different message if the situation warrants. For example, let's say a message was sent with a routing header marked `mustUnderstand="true"`. The purpose of the routing header is to let the service know that after it finishes processing the message, it's supposed to send a copy to an endpoint whose address is in the contents of the header (probably for logging purposes). If the receiver doesn't understand the header, it sends back a `mustUnderstand` fault. The sender might then, for instance, ask the user if they would still like to send the message, but without the carbon-copy functionality. If the routing header is the only one in the envelope, then it's easy to know which header the `mustUnderstand` fault refers to. But what if there are multiple `mustUnderstand` headers?

SOAP 1.2 introduced a `NotUnderstood` header to deal with this issue. When sending back a `mustUnderstand` fault, SOAP endpoints should include a `NotUnderstood` header for each header in the original message that was not understood. The `NotUnderstood` header (in the SOAP envelope namespace) has a `qname` attribute containing the QName of the header that wasn't understood. For example:

```
<env:Envelope xmlns:env='http://www.w3.org/2003/05/soap-envelope'>
  <env:Header>
    <abc:Extension1
      xmlns:abc='http://example.org/2001/06/ext'
      env:mustUnderstand='true' />
    <def:Extension2
      xmlns:def='http://example.com/stuff'
      env:mustUnderstand='true' />
  </env:Header>
  <env:Body>
    . . .
  </env:Body>
</env:Envelope>
```

If a processor received this message and didn't understand `Extension1` but did understand `Extension2`, it would return a fault like this:

```
<env:Envelope
  xmlns:env='http://www.w3.org/2003/05/soap-envelope'
  xmlns:xml='http://www.w3.org/XML/1998/namespace'>
  <env:Header>
    <env:NotUnderstood qname='abc:Extension1'
      xmlns:abc='http://example.org/2001/06/ext' />
  </env:Header>
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:mustUnderstand</env:Value>
      </env:Code>
      <env:Reason>
        <env:Text xml:lang='en'>One or more mandatory
          SOAP header blocks not understood
        </env:Text>
      </env:Reason>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

This information is handy when you're trying to use the SOAP extensibility mechanism to negotiate QoS or policy agreements between communicating parties.

### The Upgrade Header

Back in the section on versioning, we mentioned the `Upgrade` header, which SOAP 1.2 defines as a standard mechanism for indicating which versions of SOAP are supported by a node generating a `VersionMismatch` fault. This section fully defines this header.

An `Upgrade` header (which actually is a misnomer—it doesn't always imply an upgrade in terms of using a more recent version of the protocol) looks like this in context:

```
<?xml version="1.0" ?>
<env:Envelope
  xmlns:env="http://www.w3.org/2003/05/soap-envelope"
  xmlns:xml="http://www.w3.org/XML/1998/namespace">
  <env:Header>
    <env:Upgrade>
      <env:SupportedEnvelope qname="ns1:Envelope"
        xmlns:ns1="http://www.w3.org/2003/05/soap-envelope"/>
      <env:SupportedEnvelope qname="ns2:Envelope"
        xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope"/>
    </env:Upgrade>
  </env:Header>
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:VersionMismatch</env:Value>
      </env:Code>
      <env:Reason>
        <env:Text xml:lang="en">Version Mismatch</env:Text>
      </env:Reason>
    </env:Fault>
  </env:Body>
</env:Envelope>
```


This fault would be generated by a node that supports both SOAP 1.1 and SOAP 1.2, in response to some envelope in another namespace. The `Upgrade` header, in the SOAP envelope namespace, contains one or more `SupportedEnvelope` elements, each of which indicates the QName of a supported envelope element. The `SupportedEnvelope` elements are ordered by preference, from most preferred to least. Therefore, the previous fault indicates that although this node supports both SOAP 1.1 and 1.2, 1.2 is preferred.

All the `VersionMismatch` faults we've shown so far use SOAP 1.2. However, if a SOAP 1.1 node doesn't understand SOAP 1.2, it won't be able to parse a SOAP 1.2 fault. As such, SOAP 1.2 specifies rules for responding to SOAP 1.1 messages from a node that only supports SOAP 1.2. It's suggested that such nodes recognize the SOAP 1.1 namespace and respond with a SOAP 1.1 version mismatch fault containing an `Upgrade` header as specified earlier. That way, nodes that have the capability to switch to SOAP 1.2 will know to do so, and nodes that can't do so will still be able to understand the fault as a versioning problem.

## Objects in XML: The SOAP Data Model

As you saw in Chapter 2, XML has an extremely rich structure—and the possible contents of an XML data model, which include mixed content, substitution groups, and many other concepts, are a lot more complex than the data/objects in most modern programming languages. This means that there isn't always an easy way to map any given XML Schema into familiar structures such as classes in Java. The SOAP authors recognized this problem, so (knowing that programmers would like to send Java/C++/VB objects in SOAP envelopes) they introduced two concepts: the *SOAP data model* and the *SOAP encoding*. The data model is an abstract representation of data structures such as you might find in Java or C#, and the encoding is a set of rules to map that data model into XML so you can send it in SOAP messages.

### Object Graphs

The SOAP data model  is about representing *graphs* of nodes, each of which may be connected via directional *edges* to other nodes. The nodes are values, and the edges are labels. Figure 3.6 shows a simple example: the data model for a `Product` in SkatesTown's database, which you saw earlier.

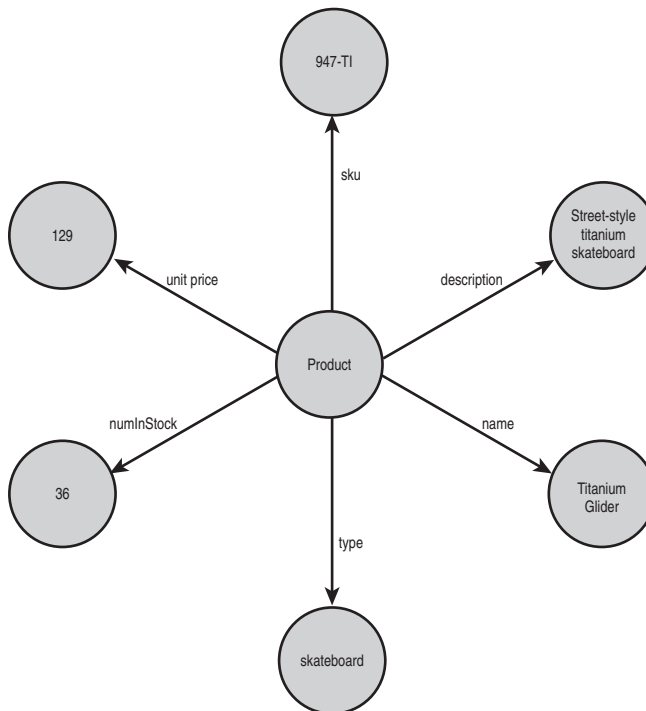


Figure 3.6 An example SOAP data model


In Java, the object representing this structure might look like this:

```
class Product {
    String description;
    String sku;
    double unitPrice;
    String name;
    String type;
    int numInStock;
}
```

Nodes may have outgoing edges, in which case they're known as *compound* values, or only incoming edges, in which case they're *simple* values. All the nodes around the edge of the example are simple values. The one in the middle is a compound value.

When the edges coming out of a compound value node have names, we say the node represents a *structure*. The edge names (also known as *accessors*) are the equivalent of field names in Java, each one pointing to another node which contains the value of the field. The node in the middle is our `Product` reference, and it has an outgoing edge for each field of the structure.

When a node has outgoing edges that are only distinguished by position (the first edge, the second edge, and so on), the node represents an *array*. A given compound value node may represent either a structure or an array, but not both.

Sometimes it's important for a data model to refer to the same value more than once—in that case, you'll see a node with more than one incoming edge (see Figure 3.7). These values are called *multireference* values, or *multirefs* .

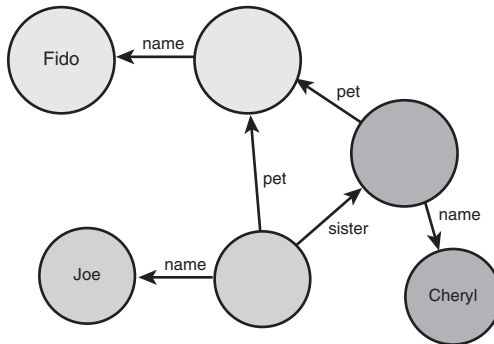



Figure 3.7 Multireference values

The model in this example shows that someone named Joe has a sister named Cheryl, and they both share a pet named Fido. Because the two pet edges both point at the same node, we know it's exactly the same dog, not two different dogs who happen to share the name Fido.



With this simple set of concepts, you can represent most common programming language constructs in languages like C#, JavaScript, Perl, or Java. Of course, the data model isn't very useful until you can read and write it in SOAP messages.

## The SOAP Encoding

When you want to take a SOAP data model and write it out as XML (typically in a SOAP message), you use the SOAP *encoding* . Like most things in the Web services world, the SOAP encoding has a URI to identify it, which for SOAP 1.2 is <http://www.w3.org/2003/05/soap-encoding>. When serializing XML using the encoding rules, it's strongly recommended that processors use the special `encodingStyle` attribute (in the SOAP envelope namespace) to indicate that SOAP encoding is in use, by using this URI as the value for the attribute. This attribute can appear on headers or their children, bodies or their children, and any child of the `Detail` element in a fault. When a processor sees this attribute on an element, it knows that the element and all its children follow the encoding rules.

### SOAP 1.1 Difference: `encodingStyle`

In SOAP 1.1, the `encodingStyle` attribute could appear anywhere in the message, including on the SOAP envelope elements (`Body`, `Header`, `Envelope`). In SOAP 1.2, it may only appear in the three places mentioned in the text.

The encoding is straightforward: it says when writing out a data model, each outgoing edge becomes an XML element, which contains either a text value (if the edge points to a terminal node) or further subelements (if the edge points to a node which itself has outgoing edges). The earlier product example would look something like this:

```
<product soapenv:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
  <sku>947-TI</sku>
  <name>Titanium Glider</name>
  <type>skateboard</type>
  <desc>Street-style titanium skateboard.</desc>
  <price>129.00</price>
  <inStock>36</inStock>
</product>
```

If you want to encode a graph of objects that might contain multirefs, you can't write the data in the straightforward way we've been using, since you'll have one of two problems: Either you'll lose the information that two or more encoded nodes are identical, or (in the case of circular references) you'll get into an infinite regress. Here's an example: If the structure from Figure 3.7 included an edge called `owner` back from the `pet` to the `person`, we might see a structure like the one in Figure 3.8.

If we tried to encode this with a naïve system that simply followed edges and turned them into elements, we might get something like this:

```

<person soapenv:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
  <name>Joe</name>
  <pet>
    <name>Fido</name>
    <owner>
      <name>Joe</name>
    </pet>
    --uh oh! stack overflow on the way!--
  
```

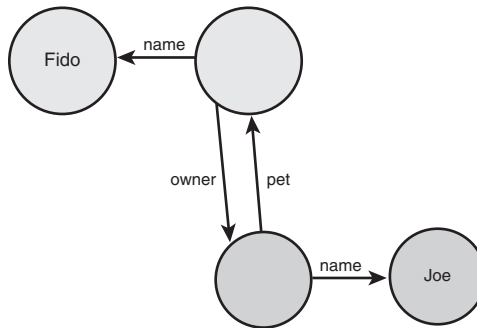


Figure 3.8 An object graph with a loop

Luckily the SOAP encoding has a way to deal with this situation: *multiref encoding*. When you encode an object that you want to refer to elsewhere, you use an `id` attribute to give it an anchor. Then, instead of directly encoding the data for a second reference to that object, you can encode a reference to the already-serialized object using the `ref` attribute. Here's the previous example using multirefs:

```

<person id="1" soapenv:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
  <name>Joe</name>
  <pet id="2">
    <name>Fido</name>
    <owner ref="#1"/> <!-- refer to the person -->
  </pet>
</person>

```

Much nicer. Notice that in this example you see an `id` of 2 on Fido, even though nothing in this serialization refers to him. This is a common pattern that saves time on processors while they serialize object graphs. If they only put IDs on objects that were referred to multiple times, they would need to walk the entire graph of objects before writing any XML in order to figure that out. Instead, many serializers always put an ID on any object (any nonsimple value) that might potentially be referenced later. If there is no further reference, then you've serialized an extra few bytes—no big deal. If there is, you can notice that the object has been written before and write out a `ref` attribute instead of reserializing it.

### SOAP 1.1 Differences: Multirefs

The `href` attribute that was used to point to the data in SOAP 1.1 has changed to `ref` in SOAP 1.2.

Multirefs in SOAP 1.1 must be serialized as *independent elements*, which means as immediate children of the `SOAP:Body` element. This means that when you receive a SOAP body, it may have multiref serializations either before or after the real `body` element (the one you care about). Here's an example:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding">
  <soap:Body>
    <!-- Here is the multiref -->
    <multiRef id="obj0" soapenc:root="0" xsi:type="myNS:Part"
soapenv:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
      <sku>SJ-47</sku>
    </multiRef>
    <!-- Here is the method element -->
    <myMultirefMethod soapenc:root="1"
      soapenv:encodingStyle=
        "http://www.w3.org/2003/05/soap-encoding">
      <arg href="#obj0"/>
    </myMultirefMethod>
    <!-- The multiref could also have appeared here -->
  </soap:Body>
</soap:Envelope>
```

This is the reason for the SOAP 1.1 `root` attribute (which you can see in the example). Multiref serializations typically have the `root` attribute set to 0; the real `body` element has a `root="1"` attribute, meaning it's the root of the serialization tree of the SOAP data model. When serializing a SOAP message 1.1, most processors place the multiref serializations *after* the main `body` element; this makes it much easier for the serialization code to do its work. Each time they encounter a new object to serialize, they automatically encode a forward reference instead (keeping track of which IDs go with which objects), just in case the object was referred to again later in the serialization. Then, after the end of the main `body` element, they write out all the object serializations in a row. This means that *all* objects are written as multirefs whenever multirefs are enabled, which can be expensive (especially if there aren't many multiple references). SOAP 1.2 fixes this problem by allowing *inline multirefs*. When serializing a data model, a SOAP 1.2 engine is allowed to put an ID attribute on an inline serialization, like this:

```
<SOAP:Body>
  <method>
    <arg1 id="1" xsi:type="xsd:string">Foo</arg1>
    <arg2 href="#1"/>
  </method>
</SOAP:Body>
```

Now, making a serialized object available for multireferencing is as easy as dropping an `id` attribute on it. Also, this approach removes the need for the `root` attribute, which is no longer present in SOAP 1.2.

## Encoding Arrays

The XML encoding for an array in the SOAP object model looks like this:

```
<myArray soapenc:itemType="xsd:string"
  soapenc:arraySize="3">
  <item>Huey</item>
  <item>Duey</item>
  <item>Louie</item>
</myArray>
```

This represents an array of three strings. The `itemType` attribute on the array element tells us what kind of things are inside, and the `arraySize` attribute tells us how many of them to expect. The name of the elements inside the array (`item` in this example) doesn't matter to SOAP processors, since the items in an array are only distinguishable by position. This means that the ordering of items in the XML encoding *is* important.

The `arraySize` attribute defaults to “\*,” a special value indicating an unbounded array (just like [] in Java—an `int []` is an unbounded array of ints).

Multidimensional arrays are supported by listing each dimension in the `arraySize` attribute, separated by spaces. So, a 2x2 array has an `arraySize` of “2 x 2.” You can use the special “\*” value to make one dimension of a multidimensional array unbounded, but it may only be the first dimension. In other words, `arraySize="* 3 4"` is OK, but `arraySize="3 * 4"` isn't.

Multidimensional arrays are serialized as a single list of items, in row-major order (across each row and then down). For this two-dimensional array of size 2x2

0	1
Northwest	Northeast
Southwest	Southeast

the serialization would look like this:

```
<myArray soapenc:itemType="xsd:string"
  soapenc:arraySize="2 2">
  <item>Northwest</item>
  <item>Northeast</item>
  <item>Southwest</item>
  <item>Southeast</item>
</myArray>
```

### SOAP 1.1 Differences: Arrays

One big difference between the SOAP 1.1 and SOAP 1.2 array encodings is that in SOAP 1.1, the dimensionality and the type of the array are conflated into a single value (`arrayType`), which the processor needs to parse into component pieces. Here are some 1.1 examples:

arrayType Value	Description
xsd:int [5]	An array of five integers
xsd:int [] [5]	An array of five integer arrays
xsd:int [, ] [5]	An array of five two-dimensional arrays of integers
p:Person [5]	An array of five people
xsd:string [2, 3]	A 2x3, two-dimensional array of strings

In SOAP 1.2, the `itemType` attribute contains only the types of the array elements. The dimensions are now in a separate `arraySize` attribute, and multidimensionality has been simplified.

SOAP 1.1 also supports *sparse arrays* (arrays with missing values, mostly used for certain kinds of database updates) and *partially transmitted arrays* (arrays that are encoded starting at an offset from the beginning of the array). To support sparse arrays, each item within an array encoding can optionally have a `position` attribute, which indicates the item's position in the array, counting from zero. Here's an example:

```
<myArray soapenc:arrayType="xsd:string[3]">
  <item soapenc:position="1">I'm the second element</item>
</myArray>
```

This would represent an array that has no first value, the passed string as the second element, and no third element. The same value can be encoded as a partially transmitted array by using the `offset` attribute, which indicates the index at which the encoded array begins:

```
<myArray soapenc:arrayType="xsd:string[3]" soapenc:offset="1">
  <item>I'm the second element</item>
</myArray>
```

Due to several factors, including not much uptake in usage and interoperability problems when they were used, these complex array encodings were removed from the SOAP 1.2 version.

## Encoding-Specific Faults

SOAP 1.2 defines some fault codes specifically for encoding problems. If you use the encoding (which you probably will if you use the RPC conventions, described in the next section), you might run into the faults described in the following list. These all are subcodes to the code `env:Sender`, since they all relate to problems with the sender's data serialization. These faults aren't guaranteed to be sent—they're recommended, rather than mandated. Since these faults typically indicate problems with the encoding system in a SOAP toolkit, rather than with user code, you likely won't need to deal with them directly unless you're building a SOAP implementation yourself:

- **MissingID**—Generated when a `ref` attribute in the received message doesn't correspond to any of the `id` attributes in the message
- **DuplicateID**—Generated when more than one element in the message has the same `id` attribute value
- **UntypedValue**—Optional; indicates that the type of node in the received message couldn't be determined by the receiver

## The SOAP RPC Conventions

Once you have the SOAP data model, it's quite natural to map remote procedure call (RPC) interactions to SOAP by using a struct to represent a call. The best way to describe this is with an example.

Let's say we have this method in Java:

```
public int addFive(int arg);
```

A request message representing a call to this method in SOAP would look something like this:

```
<env:Envelope>
  <env:Body>
    <myNS:addFive xmlns:myNS="http://my-domain.com/"
      enc:encodingStyle="http://">
      <arg xsi:type="xsd:int">33</arg>
    </myNS:addFive>
  </env:Body>
</env:Envelope>
```

Notice that the method name has been translated into XML (the rules by which a name in a language like Java gets turned into an XML name, and vice versa, can be found in part 2 of the SOAP 1.2 spec), and we've put it in a namespace that is specific to our service (this is common practice, but not strictly necessary). The method invocation, as we mentioned, is an encoded struct with one accessor for each argument, so the `arg` element is inside the method element, and the argument contains the value we're passing: 33.

If we pass this message to a service, the response looks something like this:


```
<env:Envelope>
  <env:Body>
    <myNS:addFiveResponse xmlns:myNS="http://my-domain.com/"
      xmlns:rpc="http://www.w3.org/2003/05/soap-rpc"
      enc:encodingStyle="http://">
      <rpc:result>ret</rpc:result>
      <ret xsi:type="xsd:int">38</ret>
    </myNS:addFive>
  </env:Body>
</env:Envelope>
```

The RPC response is also modeled as a struct, and by convention the name of the response struct is the name of the method with *Response* appended to the end. The struct contains accessors for all *inout* and *out* parameters in the method call (see the next section) as well as the return value.


The first accessor in the struct is interesting, and it brings to light another difference between SOAP 1.1 and 1.2: In SOAP 1.1's RPC style, there was no way to tell which accessor in the struct was the return value of the method and which were the *out* parameters. This was a problem unless you had good meta-data, and even then the situation could be confusing. SOAP 1.2 resolves this issue by specifying that an RPC

response structure containing a return value must contain an accessor named `result` in the SOAP RPC namespace. The value of this field is a QName that names the accessor containing the return value for the invocation.

## out and inout Parameters

In some environments, programmers use *out parameters*  to enable returning multiple values from a given RPC call. For instance, if we wanted to return not only a Boolean yes/no value from our inventory check service but also the actual number of units in stock, we might change our signature to something like this (using pseudocode):

```
boolean doCheck(String SKU,
                int quantity,
                out int numInStock)
```

The idea is that the `numInStock` value is filled in by the service response as well as returning a Boolean true/false. *Inout parameters*  are similar, except that they also get passed in—so we could use an *inout* like this:

```
boolean doCheck(String SKU,
                inout int quantity)
```

In this situation, we'd pass a quantity value in, and then we expect the value of the `quantity` variable to have been updated to the actual quantity available by the service.

Java developers aren't used to the concept of *inout* or *out* parameters because, typically, in Java all objects are automatically passed by reference. When you're using RMI, simple objects may be passed by value, but other objects are still passed by reference. In this sense, any mutable objects (whose state can be modified) are automatically treated as *inout* parameters. If a method changes them, the changes are seen automatically by anyone else.

In Web services, the situation is different: All parameters are passed by value. SOAP has no notion of passing parameters by reference. This design decision was made in order to keep SOAP and its data encoding simple. Passing values by reference in a distributed system requires distributed garbage collection and (potentially) a lot of network round-trips. This not only complicates the design of the system but also imposes restrictions on some possible system architectures and interaction patterns. For example, how can you do distributed garbage collection when the requestor and the provider of a service can both be offline at the same time?

Therefore, for Web services, the notion of *inout* and *out* parameters doesn't involve passing objects by reference and letting the target backend modify their state; instead, copies of the data are exchanged. It's then up to the service client code to create the perception that the state of the object that has been passed in to the client method has been modified. We'll show you what we mean. Let's take the modified `doCheck()` you saw earlier:

```
boolean doCheck(String SKU,
                inout int quantity)
```

When this method is called, the request message looks like this on the wire:

```
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <doCheck soapenv:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
      <SKU>318-BP</SKU>
      <quantity xsi:type="xsd:int">3</quantity>
    </doCheck>
  </soapenv:Body>
</soapenv:Envelope>
```

And here's the response message:

```
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <doCheckResponse soapenv:encodingStyle=
      "http://www.w3.org/2003/05/soap-encoding">
      <rpc:result xmlns:rpc="http://www.w3.org/2003/05/soap-rpc">return</rpc:result>
      <return>true</return>
      <quantity xsi:type="xsd:int">72</quantity>
    </doCheckResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

This is a request to see if 3 items are available, and the response indicates that not only are there 3 (the true response), but there are in fact 72 (the new quantity value). The endpoint receiving this response should then update the appropriate programming-language construct for the quantity parameter with the new value.

Finally, here's the example that adds an extra out parameter containing the number of items in stock to our doCheck() method:

```
boolean doCheck(in sku, in quantity, out numInStock)
```

If we called this new method, the request would look identical to the one you saw earlier in this chapter, but the response would now look like this:

```
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <doCheckResponse soapenv:encodingStyle=
      "http://www.w3.org/2003/05/soap-encoding">
      <rpc:result xmlns:rpc="http://www.w3.org/2003/05/soap-rpc">return</rpc:result>
      <return>true</return>
      <numInStock>72</numInStock>
```



```
</doCheckResponse>  
</soapenv:Body>  
</soapenv:Envelope>
```

Despite the fact that Java doesn't have a native concept of `inout` and `out` parameters, we can still use them with toolkits like Axis—we'll explore how to do this in Chapter 5.

## XML, Straight Up: Document-Style SOAP

Although the RPC pattern is a common use case for SOAP, there are no restrictions on the contents of the SOAP body. Typically, sending nonencoded XML content in the body is known as *document-style SOAP*, since it centers around the message as an XML document rather than an abstract data model that happens to be encoded into XML.

Keep in mind that even if you don't explicitly use the SOAP RPC conventions as described, you can still map your document-style XML to and from procedure calls (or use whatever programming paradigm you like, since SOAP is about the structure of messages on the wire). Various toolkits, including .NET and Axis, have been doing this for some time now. When you publish a Visual Basic or C# class as a .NET Web service, the default behavior is to use document-style SOAP to expose the methods. The messages still look similar to the RPC style, but they don't use the SOAP encoding. The only problem with this approach is that there are no standard encodings for things like arrays or structures, and you lose the referential integrity of objects that are referenced multiple times.

## When to Use Which Style

As a Java programmer, it might not always be clear which style of SOAP to use. Here are a few rules of thumb you can use when thinking about it:

- If you're starting from code in Java, C#, or another procedural/OO language, and you're trying to expose methods as Web services, the RPC style is a natural fit.
- If you have preexisting XML formats (schemas and so on) that you want to support, using the SOAP encoding would just get in the way. So, document style is preferable.
- If you need to transmit object graphs that must maintain referential integrity (circular lists, for instance, or complex graphs), RPC provides an interoperable way to do it.
- Validating document-style messages is straightforward, because they typically have XML schemas. You can't easily write an XML schema that will correctly validate an encoded SOAP data model, since the model allows for a nondeterministic set of valid serializations—for instance, `multiref` objects can be serialized in a variety of places (either in-place at any of the references, or as an independent element at the top level), and arrays are still valid regardless of the XML element name of the items. So, if schema validation using standard tools is important, document style is the way to go.


As you'll see in Chapter 13, "Web Services Interoperability," the Web Service Interoperability organization (WS-I) has come down hard against SOAP encoding, and has in fact banned its use in their Basic Profile of SOAP 1.1 (we can only assume they will continue to dislike it for SOAP 1.2).

The jury is definitely still out with respect to the value of the SOAP encoding. A lot of companies seem to be moving away from it; but, on the other hand, it's a developer-friendly technology for people trying to expose preexisting classes and methods as Web services. Despite WS-I's claims, there has been a lot of interoperability work to make sure RPC style works between most implementations of SOAP 1.1 and SOAP 1.2.

We'll talk a lot more about these styles in the next couple of chapters, when we describe them in relationship to WSDL and Axis.

## The Transport Binding Framework

The SOAP processing model talks about what a node should do when it processes a SOAP message. As we've discussed, messages are described abstractly in terms of the XML infoset. Now it's time to look more closely at how those infosets are moved from place to place.

A *SOAP protocol binding*  is a set of rules that describes a method of getting a SOAP infoset from one node to another. For instance, you already know that HTTP is a common way to transport SOAP messages. The purpose of the SOAP HTTP binding (which you'll find in part 2 of the spec) is to describe how to take a SOAP infoset at one node and serialize it across an HTTP connection to another node.

Many bindings, including the standard HTTP one, specify that the XML 1.0 serialization for the infoset should be used—that means if you look at the messages on the wire, you'll see angle brackets as in any regular XML document. But the binding's job is really to move the infoset from node to node, and the way the infoset is represented on the wire is up to the binding author. This means bindings have the freedom to specify custom serializations; they might do this to increase efficiency with compression or a binary protocol, or to add security via encryption, among other reasons.

Since bindings determine the concrete structure of the bits on the wire and how they're processed into SOAP messages, it's critical that communicating parties agree on what binding to use—that agreement is just as important as the agreement of what data to send. This is why bindings, like SOAP modules, are named with URIs. The standard SOAP 1.2 HTTP binding, for instance, is identified by the URI `http://www.w3.org/2003/05/soap/bindings/HTTP/`. Note that there can be many different SOAP bindings for a given underlying protocol—agreeing on a binding means not only agreeing to use HTTP to send SOAP messages, but also to follow all the rules of the given binding specification when doing so.

Bindings can be simple or complex, and they can provide a variety of different levels of functionality. A raw UDP binding, for instance, might not provide any guarantee that the messages sent arrive at the other side. On the other hand, a binding to a reliable message queuing system such as the Java Message Service (JMS) might provide


guaranteed (eventual) arrival, even in the face of crashes or network failures. We can also imagine bindings to underlying protocols that provide security, such as HTTPS, or with the ability to broadcast messages to a variety of recipients at once, like multicast IP.


Even when the underlying protocol doesn't provide desired functionality directly, it's still possible to write a binding that does fancy things. For instance, you could design a custom HTTP binding specifying that, for security reasons, the XML forming each message should be encrypted before being sent (and decrypted on the receiving side). Such custom bindings are possible, but it's often a better idea to use the extensibility built into the envelope to implement this kind of functionality. Also, bindings only serve to pass messages between adjacent nodes along a SOAP message path. If you need end-to-end functionality in a situation where intermediaries are involved, it's usually better to use in-message extensibility with SOAP headers.

## Features and Properties

Bindings can do a huge variety of different things for us—compare this variety to what the SOAP header mechanism can do, and you might notice that the set of possible semantics you can achieve is similar (almost anything, in fact). The designers of SOAP 1.2 noticed the similarity as well, and wanted to write a framework that might help future extension authors describe their semantics in ways that were easy to reason about, compare with each other, and refer to in machine-readable ways.

What was needed was a way to specify that a given semantic could be implemented either by a binding or by using SOAP headers. Nodes might use this information to decide whether to send particular headers—if you're running on top a secure binding, for instance, you might not need to add security headers to your messages, which can save your application effort and time. Also, since a variety of modules might be available to an engine at any given time, it would also be nice to be able to unambiguously indicate that a particular module performed a certain set of desired semantics. To achieve these goals, the first job was to have a standard way to name the semantics in question.

The SOAP authors came up with an extensibility framework that is described in section 3 of part 1 of the SOAP 1.2 spec. The framework revolves around the notion of an abstract *feature* , which is essentially a semantic that has a name (a URI) and a specification explaining what it means. Features can be things like reliability, security, transactions, or anything you might imagine. A feature is typically specified by describing the behavior of each node involved in the interaction, any data that needs to be known before the feature does its thing, and any data that is transferred from node to node as a result. It's generally convenient if the specification of the feature's behavior can be found at the URI naming the feature—so if a user sees a reference to it somewhere, they can easily locate a description of what it does.

The state relevant to a feature is generally described in terms of named *properties* . Properties, in the SOAP 1.2 sense, are pieces of state, named with URIs, which affect the operation of features. For instance, you might describe a “favorite color” feature that involves transferring a hexadecimal color value from one node to another. You would


pick a URI for the feature, describe the rules of operation, and name the color property with another URI. Because a feature definition is abstract, you don't say anything about how this color would move across the wire.

Features are *expressed* (turned into reality via some mechanism) by *bindings* and *modules*. We've already described both of these—recall that a binding is a means for performing functions below the SOAP processing model, and a module is a means for performing functions using the SOAP processing model, via headers.

Here's a simple example of how a feature might be expressed by a binding in one case and a module in another. Imagine a “secure channel” feature. The specification for this feature indicates that it has the URI `http://skatestown.com/secureChannel`, and the abstract feature describes a message traveling from node to node in an unspoofable fashion (to some reasonable level of security). We might then imagine a SOAP binding to the HTTPS protocol, which would specify that it implements the `http://skatestown.com/secureChannel` feature. Since HTTPS meets the security requirements of the abstract feature, the feature would be satisfied by the binding with no extra work, and the binding specification would indicate that it natively supports this feature. We could also imagine a SOAP module (something like WS-Security, which we discuss in Chapter 9, “Securing Web Services”) that provides encryption and signing of a SOAP message across any binding. The module specification would also state that it implements the `secureChannel` feature. With this information in hand, it would be possible, in a situation that required a secure channel, to decide to engage our SOAP module in some situations (when using the HTTP binding, for instance) and to not bother in other situations where the HTTPS binding is in use.

Another example of making features concrete could be the color feature discussed earlier. The feature spec describes moving a color value from the sender to the receiver in the abstract. A custom binding over email might define a special SMTP header to carry this color information outside the SOAP envelope. Writing a SOAP module to satisfy the feature would involve defining a SOAP header that carried the value in the envelope.

### Message Exchange Patterns

One common type of feature is a *Message Exchange Pattern (MEP)* . An MEP specifies how many messages move around in a given interaction, where they originate, and where they end up. Each binding can (and must) support one or more message exchange patterns. The SOAP 1.2 spec defines two standard MEPs: Request-Response and SOAP Response. Without going into too much detail (you can find the full specifications for these MEPs in the SOAP 1.2 spec, part 2), we'll give you a flavor of what these MEPs are about.

The Request-Response MEP involves a requesting node and a responding node. As you might expect, it has the following semantics: First, the requesting node sends a SOAP message to the responding node. Then the responding node replies with a SOAP message that returns to the requesting node. See Figure 3.9.

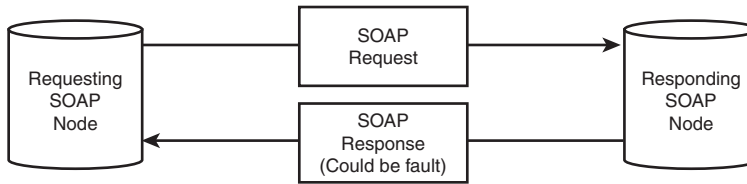


Figure 3.9 The Request-Response MEP

You should note a couple of important things here. First, the response message is correlated to the request message—in other words, the requesting node must be able to figure out which response goes with which request. Since the MEP is abstract, though, it doesn't specify *how* this correlation is achieved but leaves that up to implementations. Second, if a fault is generated at the responding node, the fault is delivered in the response message.

The SOAP Response MEP is similar to the Request-Response MEP, except for the fact that the request message is explicitly *not* a SOAP message. In other words, the request doesn't trigger the execution of the SOAP processing model on the receiving node. The receiving node responds to the request with a SOAP message, as in the Request-Response MEP (see Figure 3.10).

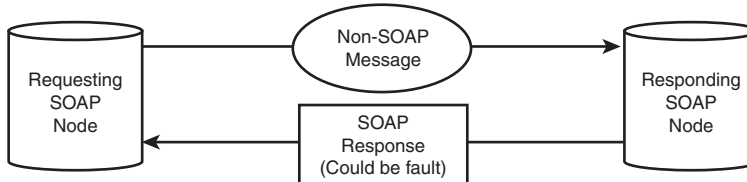


Figure 3.10 The SOAP Response MEP

The primary reason for introducing this strange-seeming MEP is to support REST-style interactions with SOAP (see the sidebar “REST-Style versus Tunneled Web Services”). The SOAP Response MEP allows a request to be something as simple as an HTTP GET; and since the responding node doesn't have to implement the SOAP processing model, it can return a SOAP message in any way it deems appropriate. It might, for instance, be an HTTP server with a variety of SOAP messages in its filesystem.

As you'll see in a moment, the HTTP binding natively supports both of these MEPs. Of course, MEPs, like other abstract features, can also be implemented via SOAP modules. Here's an example of how the Request-Response MEP might be implemented using SOAP headers across a transport that only allows one-way messages:

```

<soapenv:Envelope
  xmlns:soapenv="http://www.w3.org/2003/05/soap/envelope">
  <soapenv:Header>

```

```


<!--This header specifies the return address-->
<reqresp:ReplyTo soapenv:mustUnderstand="true"
  xmlns:reqresp="http://skateestown.com/requestResponse">
  <destination>udp://me.com:6666</destination>
</reqresp:ReplyTo>
<!--This header specifies a correlation ID-->
<reqresp:correlationID soapenv:mustUnderstand="true"
  xmlns:reqresp="http://skateestown.com/requestResponse">
  1234
</reqresp:correlationID>
</soapenv:Header>
<soapenv:Body>
  <doSomethingCool/>
</soapenv:Body>
</soapenv:Envelope>

```

This message might have been sent in a UDP datagram, which is a one-way interaction. The receiver would have to understand these headers (since they're marked `mustUnderstand`) in order to process the message; when it generated a reply, it would follow the rules of the `ReplyTo` header and send the reply via UDP to port 6666 of host `me.com`. The reply would contain the same `correlationID` sent in the request, so that the receiver on `me.com` would be able to match the response to the pending request. The WS-Addressing spec includes a mechanism a lot like this, which we'll cover in Chapter 8, "Web Services and Stateful Resources."

If your interaction requires the request-response MEP, you might choose to use the HTTP binding (or any binding that implements that MEP natively), or you might use a SOAP module that specifies how to implement the MEP across other bindings. The important goal of the framework is to enable *abstractly* defining what functionality you need—then you're free to select appropriate implementations without tying yourself to a particular way of doing things.

### REST-Style versus Tunneled Web Services

*REpresentational State Transfer (REST)*  refers to an architectural style of building software that mirrors what HTTP is built to do: transfer representations of the state of named resources from place to place with a small set of methods (GET, POST, PUT, DELETE). The methods have various semantics associated with them—for instance, a GET is a request for a resource's representation, and by definition GETs should be safe operations. "Safe" in this context means they should have no side effects. So, you do a GET on my bank account URL in order to obtain your current balance, you can repeat that operation many times with no ill effects. However if the GET withdrew funds from your account, that would clearly be a serious side effect (and therefore illegal in REST).

Another interesting thing to note about GET is that the results are often *cacheable*: If you do a GET on a weather report URL, the results probably won't change much if you do it again in 10 minutes. As a result, the infrastructure can cache the result, and if someone asks for the same GET again before the cache entry expires, you can return the cached data instead of going out over the Net to fetch the same weather again.

The HTTP infrastructure uses this caching style to great effect; most large ISPs or companies have shared caching proxies at the edge of the network that vastly reduce the network bandwidth outside the organization—each time anyone asks for `http://cnn.com`, the cache checks if a local, fresh copy exists before going out across the network.

We bring up REST because it shines light on an interesting controversy. REST advocates (sometimes known as RESTifarians) noticed that SOAP 1.1's HTTP binding mandated using the POST method. Since POSTs are *not* safe or cacheable in the same way GETs are, even simple SOAP requests to obtain data like stock quotes or weather reports had to use a mechanism that prevented the HTTP infrastructure from doing its job. Even though these operations might have been safe, SOAP was ignoring the HTTP semantic for safe operations and losing out on valuable caching behavior as a consequence.

This issue has since been fixed in the SOAP 1.2 HTTP binding, which supports the WebMethod feature. In other words, SOAP 1.2 is a lot more REST-friendly than SOAP 1.1 was, since it includes a specific technique for utilizing HTTP GET, including all that goes along with it in terms of caching support and operation safety. HTTP semantics are respected, and HTTP isn't always used as a tunnel for SOAP messages.

We believe that SOAP's flexible messaging and extensibility model (which supports both semantics provided by the underlying protocol and also those provided by higher-level extensions in the SOAP envelope) offer the right framework for supporting a large variety of architectural approaches. We hope this framework will allow the protocol both good uptake and longevity in the development community.

## Features and Properties Redux

The features and properties mechanism in SOAP 1.2 is a powerful and flexible extensibility framework. By naming semantics, as well as the variables used to implement those semantics, we enable referencing those concepts in an unambiguous way. Doing so is useful for a number of reasons—as you'll see in Chapter 4, “Describing Web Services,” naming semantics lets you express capabilities and requirements of your services in machine-readable form. This allows software to *reason* about whether a given service is compatible with particular requirements. In addition to the machine-readability aspect, naming features and properties allows multiple specifications to refer to the same concepts when describing functionality. This encourages good composition between extensions, and also allows future extensions to be written that implement identical (but perhaps faster, or more flexible) semantics.

Now that you understand the framework and its components, we'll go into more detail about the SOAP HTTP binding.

## The HTTP Binding

You already know the HTTP binding is identified with the URI `http://www.w3.org/2003/05/soap/bindings/HTTP/` and that it supports both the Request-Response and the SOAP Response MEPs. Before we examine the other interesting facets of this binding, let's discuss how those MEPs are realized.

Since HTTP is natively a request/response protocol, when using the Request-Response MEP with the HTTP binding, the request/response semantics map directly to the equivalent HTTP messages. In other words, the SOAP request message travels in the HTTP request, and the SOAP response message is in the HTTP response. This is a great example of utilizing the native capabilities of an underlying protocol to implement an abstract feature.

The SOAP Response MEP is also implemented natively by the HTTP binding and is typically used with the GET Web method to retrieve representations of Web resources expressed as SOAP messages, or to make idempotent (safe) queries. When using this MEP this way, the non-SOAP request is the HTTP GET request, and the SOAP response message (or fault) comes back, as in the Request/Response case, in the HTTP response.

The HTTP binding also specifies how faults during the processing of a message map to particular HTTP status codes and how the semantics of HTTP status codes map to Web service invocations.

#### SOAP 1.1 Differences: Status Codes

In SOAP 1.1, the HTTP binding specified that if a fault was returned in an HTTP response, the status code had to be 500 (server error). This worked, but it didn't allow systems to use the inherent meaning in the HTTP status codes—400 means a problem with the sender, 500 means a problem with the receiver, and so on. SOAP 1.2 resolves this issue by specifying a richer set of fault codes; for example, if a `soapenv:Sender` fault is generated, indicating a problem with the request message, the engine should use the 400 HTTP status code when returning the fault. Other faults generate a 500, as in SOAP 1.1.

The HTTP binding also implements two abstract features, which are described next.

### The SOAP Action Feature

In the SOAP 1.1 HTTP binding, you were required to send a custom HTTP header called `SOAPAction` along with SOAP messages. The purpose of this header was twofold: to let any system receiving the message know that the contents were SOAP and to convey the intent of the message via a URI. In SOAP 1.1, the first purpose was necessary because the media type was `text/xml`; since that media type could carry any XML document, processors needed to look inside the XML (which might involve an expensive parse) to check if it was a SOAP envelope. The `SOAPAction` header allowed them to realize this was a SOAP message and perhaps make decisions about routing or logging based on that knowledge. The presence of the header was enough to indicate that it was SOAP, but the URI could also convey more specific meaning, abstractly describing the intent of the message. Many implementations use this URI for dispatching to a particular piece of code on the backend, especially when using document-style SOAP. For example, you might send a purchase order as the body contents to two different methods—one called `submitPO()` and the other called `validatePO()`. Since the XML is the same in both cases, you could use the value of the `SOAPAction` URI to differentiate.



SOAP 1.2 uses the `application/soap+xml` media type, and the `SOAPAction` header is no longer needed to differentiate SOAP traffic from other XML documents moving across HTTP connections. But a lot of people still want a standard way to indicate a message's purpose outside of the SOAP envelope. The features and properties mechanism described earlier seemed like a perfect fit for an abstract feature.

The `SOAPAction` feature can be made available for any binding that uses the `application/soap+xml` media type to send its messages. The definition of that media type (which you can find in the glossary) specifies an optional `action` parameter, which is used in SOAP 1.2 instead of an HTTP-specific header to carry the `SOAPAction` URI. The feature has a single property, `http://www.w3.org/2003/05/soap/features/action/Action`. When this property is given a URI value, the spec indicates that the binding that implements the feature must place the value into the `action` parameter. So, if the property had the value `http://example.com/myAction` at the sender, the message would start like this (assuming the HTTP binding):

```
POST /axis/SomeService.jws
Content-Type: application/soap+xml; charset=utf-8;
➡action="http://example.com/myAction"
...rest of message...
```

On the receiving side, a node receiving a message with an `action` parameter over a binding that supports the SOAP Action feature is required to make the value of the `action` parameter available in the `http://www.w3.org/2003/05/soap/features/action/Action` property.

## The Web Method Feature

Normal SOAP exchanges across HTTP use the `POST` HTTP verb. However, sometimes other HTTP methods are more appropriate for a given situation. For instance, when you use the SOAP Response MEP, you're often making a state query, which in HTTP is modeled with the `GET` verb. If you only allowed `POST`, you would be forcing HTTP into a particular limited set of uses (purely as a transport). Therefore, in an effort to allow developers to better integrate the semantics of SOAP with the semantics of HTTP, the Web Method Feature (defined in part 2 of the spec) was born.

This feature defines a single property, `http://www.w3.org/2003/05/soap/features/web-method/Method`, which contains one of the words `GET`, `POST`, `PUT`, or `DELETE` (although other values may be supported later). When sending a message, the HTTP binding will use the verb specified in this property instead of the default `POST`. This is most often used in concert with the SOAP Response MEP to implement REST semantics.

Right now this feature is only relevant to HTTP; but if other bindings are developed to underlying protocols with REST-like semantics, it would be available for them as well.

## Using SOAP to Send Binary Data


Our example messages to date have been fairly small, but we can easily imagine wanting to use SOAP to send large binary blobs of data. For example, consider an automated insurance claim registry—remote agents might use SOAP-enabled software to submit new claims to a central server, and part of the data associated with a claim might be digital images recording damages or the environment around an accident. Since XML can't directly encode true 8-bit binary data at present, a simple way to do this kind of thing might be to use the XML Schema type `base64binary` and encode your images as `base64` text inside the XML:

```
<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soap:Body>
  <submitClaim>
    <accountNumber>5XJ45-3B2</accountNumber>
    <eventType>accident</eventType>
    <image imageType="jpg" xsi:type="base64binary">
      4f3e9b0...(rest of encoded image)
    </image>
  </submitClaim>
</soap:Body>
</soap:Envelope>
```

This technique works, but it's not particularly efficient in terms of bandwidth, and it takes processing time to encode and decode bytes to and from `base64`. Email has been using the Multipurpose Internet Mail Extensions (MIME) standard for some time now to do this job, and MIME allows the encoding of 8-bit binary. MIME is also the basis for some of the data encoding used in HTTP; since HTTP software can usually deal with MIME, it might be nice if there were a way to integrate the SOAP protocol with this standard and a more efficient way of sending binary data.

## SOAP with Attachments and DIME

In late 2000, HP and Microsoft released a specification called “SOAP Messages with Attachments.” The spec describes a simple way to use the `multiref` encoding in SOAP 1.1 to reference MIME-encoded attachment parts. We won't go into much detail here; if you want to read the spec, you can find it at <http://www.w3.org/TR/2000/NOTE-SOAP-attachments-20001211>.

The basic idea behind *SOAP with Attachments (SwA)*  is that you use the same HREF trick you saw in the section “Object Graphs” to insert a reference to the data in the SOAP message instead of directly encoding it. In the SwA case, however, you use the content-id (`cid`) of the MIME part containing the data you're interested in as the reference instead of the ID of some XML. So, the message encoded earlier would look something like this:

```

MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary;
type=application/soap+xml;start="<claim@insurance.com>"

--MIME_boundary
Content-Type: application/soap+xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim@insurance.com>


<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <submitClaim>
      <accountNumber>5XJ45-3B2</accountNumber>
      <eventType>accident</eventType>
      <image href="cid:image@insurance.com"/>
    </submitClaim>
  </soap:Body>
</soap:Envelope>

--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <image@insurance.com>

...binary JPG image...

--MIME_boundary--


```



Another technology called *Direct Internet Message Encapsulation (DIME)* , from Microsoft and IBM, used a similar technique, except that the on-the-wire encoding was smaller and more efficient than MIME. DIME was submitted to the IETF in 2002 but has since lost the support of even its original authors.

SwA and DIME are great technologies, and they get the job done, but there are a few problems. The main issue is that both SwA and DIME introduce a data structure that is explicitly outside the realm of the XML data model. In other words, if an intermediary received the earlier MIME message and wanted to digitally sign or encrypt the SOAP body, it would need rules that told it how the content in the MIME attachment was related to the SOAP envelope. Those rules weren't formalized for SwA/DIME. Therefore, tools and software that work with the XML data model need to be modified in order to understand the SwA/DIME packaging structure and have a way to access the data embedded in the MIME attachments.

Various XML and Web service visionaries began discussing the general issue of merging binary content with the XML data model in earnest. As a result, several proposals are now evolving to solve this problem in an architecturally cleaner fashion.

## PASWA, MTOM, and XOP

In April 2003, the “*Proposed Infoset Addendum to SOAP With Attachments*” (PASWA)  document was released by several companies including Microsoft, AT&T, and SAP. PASWA introduced a logical model for including binary content directly into a SOAP infoset. Physically, the messages that PASWA deals with look almost identical to our two earlier examples (the image encoded first as base64 inline with the XML and then as a MIME attachment)—the difference is in how we think about the attachments. Instead of thinking of the MIME-encoded image as a separate entity that is explicitly referred to in the SOAP envelope, we logically think of it as if it were still inline with the XML. In other words, the MIME packaging is an optimization, and implementations need to ensure that processors looking at the SOAP data model for purposes of encryption or signing still see the actual data as if it were base64-encoded in the XML.

In July 2003, after a long series of conversations between the XML Protocol Group and the PASWA supporters, the *Message Transmission Optimization Mechanism (MTOM)*  was born, owned by the XMLP group. It reframed the ideas in PASWA into an abstract feature to better sync with the SOAP 1.2 extensibility model, and then offered an implementation of that feature over HTTP. The serialization mechanism is called *XML-Binary Optimized Packaging (XOP)* ; it was factored into a separate spec so that it could also be used in non-SOAP contexts.

As an example, we slightly modified the earlier insurance claim by augmenting the XML with a content-type attribute (from the XOP spec) that tells us what MIME content type to use when serializing this infoset using XOP. Here’s the new version:

```
<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xop-mime="http://www.w3.org/2003/12/xop/mime">
  <soap:Body>
    <submitClaim>
      <accountNumber>5XJ45-3B2</accountNumber>
      <eventType>accident</eventType>
      <image xop-mime:content-type="image/jpeg"
        xsi:type="base64binary">
        4f3e9b0...(rest of encoded image)
      </image>
    </submitClaim>
  </soap:Body>
</soap:Envelope>
```

An MTOM/XOP version of our modified insurance claim looks like this:

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary;
type=application/soap+xml;start="<claim@insurance.com">
```

```

--MIME_boundary
Content-Type: application/soap+xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim@insurance.com>

<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:xop="http://www.w3.org/2003/12/xop/include"
  xmlns:xop-mime="http://www.w3.org/2003/12/xop/mime">
  <soap:Body>
    <submitClaim>
      <accountNumber>5XJ45-3B2</accountNumber>
      <eventType>accident</eventType>
      <image xop-mime:content-type='image/jpeg' >
        <xop:Include href="cid:image@insurance.com"/>
      </image>
    </submitClaim>
  </soap:Body>
</soap:Envelope>

--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <image@insurance.com>

...binary JPG image...

--MIME_boundary--

```

Essentially, it's the same on the wire as the SwA version, but it uses the `xop:Include` element instead of just the `href` attribute. The real difference is architectural, since we imagine tools and APIs will manipulate this message exactly as if it were an XML data model.

MTOM and XOP are on their way to being released by the XML Protocol Working Group some time in 2004, and it remains to be seen how well they will be accepted by the broader user community. Early feedback has been very positive, however, and the authors of this book are behind the idea of a unified data model for XML and binary content.

## Small-Scale SOAP, Big-Time SOAP

To finish our coverage of the SOAP protocol in this chapter, we'll briefly examine how the simple rules of SOAP's design allow a lot of flexibility for implementations. Let's consider three different styles of SOAP processors and look at how this one baseline protocol can scale from the very small to the very large.

Imagine that a maker of scientific equipment has just released a new digital thermometer for the home. Being a forward-looking company, it decides to add a SOAP-over-HTTP interface so that other devices plugged into the home network can query the temperature. This thermometer has exactly one method, `getTemperature()`, which returns the current temperature in degrees Celsius. Since it's so simple, the vendor has built a single-purpose SOAP engine into the hardware. This engine doesn't include a full XML processor but instead uses simple pattern-matching with regular expressions to do three simple things:

- Check that the envelope looks right, and return a `VersionMismatch` fault if necessary.
- If any header blocks are targeted for the ultimate destination that are marked `mustUnderstand=true`, return a `mustUnderstand` fault.
- Confirm the `soap:Body` contains exactly one element with the `getTemperature` QName.

Despite its simplicity and extremely limited processing capability, our thermometer is perfectly SOAP 1.2 compliant. That means any other software that speaks SOAP can talk to it.

Our second example is a single-purpose application built to talk to a particular kind of Web service. SkatesTown might have built something like this in order to automate finding the lowest price on parts from its various suppliers (all of whom implement a standard RPC-style price checking service) each day. This sort of system typically has real XML parsing and uses shared libraries to manage the SOAP processing model, so that the developer doesn't have to rewrite the same `mustUnderstand`-checking logic again and again. It might also support one or two built-in extensions, such as a commonly supported security module. However, the capabilities of this system are determined when it's constructed, and it can't be dynamically extended.

Finally, let's consider a general-purpose middleware SOAP solution as you might implement it with a toolkit like Axis. Axis was designed to be a flexible framework that provides as much generic processing capability as possible while letting you build applications and extensions that easily plug in to the framework. It uses full-scale XML parsing, supports schema validation, and has a mechanism for building *handlers*—pieces of extension code that process SOAP messages in order to implement either local functionality (logging, management) or extension semantics (security, correlation).

The system in this example might be for a Grid computing system (see Chapter 8 for more) that needs to be able to talk to an unbounded variety of other services. Our SOAP engine has a variety of extensions for popular security, transactionality, reliability, and management protocols on top of SOAP. When conversations begin, an initial negotiation phase uses SOAP headers to determine which encryption protocols to use, whether the parties are legally allowed to transact business across the network, and what kind of notarization service will be used. Essentially, we use `mustUnderstand` headers to ask for the maximum level of functionality; then, if we receive errors, we back off to

lower levels if appropriate. Once the business-level messages start to flow, they will be safely encrypted and contain SOAP headers to manage transactional context, handle routing through intermediaries, and ensure nonrepudiation. The system is also extensible by dropping in new JAR files and tweaking a few deployment descriptors.

So we've gone from a completely static, dumb system like a thermometer all the way to a richly functional B2B fabric—and SOAP's model of simple, well-layered abstractions has been the foundation throughout. Although it's true that the secure Grid client won't be able to make the thermometer talk at its level, the inherent capabilities of the SOAP processing model make it possible for the more functional client to find that out and to scale back (if appropriate) in order to talk to the simpler service.

Of course, wouldn't it be nice if there were standard ways for communicating parties to automatically find out about each other's capabilities *before* any messages even flow over the wire? There is such a way: It's called WSDL, and it's the subject of the next chapter.

## Summary

We hope this chapter has demystified what SOAP is about and given you a good grounding in the essentials of the protocol and its behavior. You've seen how vertical (SOAP headers) and horizontal (intermediaries) extensibility, plus the binding framework, can be used to stretch the capabilities of the core in controlled ways. We'll return again and again to these extensibility concepts as we move through the examples in the rest of the book.

Of course, unless you plan to implement a SOAP engine on your own, you'll be using a toolkit to perform SOAP invocations. As such, we've also talked about Apache Axis, the open source SOAP engine that SkatesTown and SBC are using, and you've seen how the HTTP-based purchase order system can be converted into a SOAP Web service. You'll learn a lot more about Axis in Chapter 5, but before that we'll look at the other major pillar that makes up the foundation of Web services: the Web Service Description Language, or WSDL.

## Resources

- *XML Protocol Working Group* (home of all SOAP 1.2 information)—<http://www.w3.org/2000/xp/Group>
- *SOAP 1.2 spec* (latest version)—<http://www.w3.org/TR/SOAP>
- *SOAP 1.2 Primer*—<http://www.w3.org/TR/2003/REC-soap12-part0-20030624/>
- *SOAP 1.1 spec*—<http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>
- *SOAP with Attachments*—<http://www.w3.org/TR/SOAP-attachments>
- *MTOM*—<http://www.w3.org/TR/soap12-mtom/>
- *XOP*—<http://www.w3.org/TR/xop10/>