Exchanging Messages with SOAP

There are times when less is really more. When asking about SOAP, programmers and programming managers often say things like, “Isn’t SOAP just an alternative to DCOM or CORBA?” The correct answer to that question is, “No, it is much less than that.”

Different application development platforms have various distributed computing mechanisms. Some of these mechanisms are very efficient, but limited in what operating systems and languages they support. In addition to those efficient technologies, all operating systems have a less-efficient way of exchanging data via files full of characters. At the most basic level, SOAP messages are just streams of characters. They are not just randomly created characters, though; they are carefully crafted so that programs on both sides of the transmission can understand exactly what the other side is saying.

SOAP messages are XML documents that are embedded in the transport’s request and response. In this hour, you will learn how SOAP works. You will first learn what SOAP is and where it came from. Next, you will learn the

anatomy of SOAP documents so that you will be able to understand them (at least in rough outline form) when you look at them.

In this hour, you will learn

- About the SOAP language
- The rules for creating a SOAP document
- What the SOAP envelope is for
- The purpose of the SOAP header
- How SOAP handles errors

What SOAP Is

Many of the standard definitions of SOAP sound like buzzwords strung together. One particularly good one is that SOAP is a specification for a ubiquitous XML-based distributed computing infrastructure. If we translate these words, we can get a better feel for what SOAP really is:

- **Specification**—SOAP is not a product that was created and sold by a vendor. Rather, it is a document that describes the characteristics of a piece of software. The basic idea is that if two parties create programs to the same specifications, these programs will be able to interoperate seamlessly.
- **Ubiquitous**—SOAP is defined at a high enough level of abstractions that any operating system and programming language combination could be used to create SOAP-compliant programs.
- **XML-based**—SOAP is built on top of XML, which means that SOAP documents are XML documents constructed to a tighter set of specifications.
- **Infrastructure**—SOAP does not specify what data can be moved or what function calls can take place over it. An analogy could be made to a railroad car. The car is capable of moving any item that will fit in it from point A to point B. In the same way, software products that are constructed to the SOAP specification can move data from computer A to computer B and hand it to another program written to the same specification. The actual real-world meaning of the data is outside the scope of the SOAP specification.

So, a SOAP message is an XML document. Using SOAP can be thought of as a set of layers, as shown in Figure 9.1.

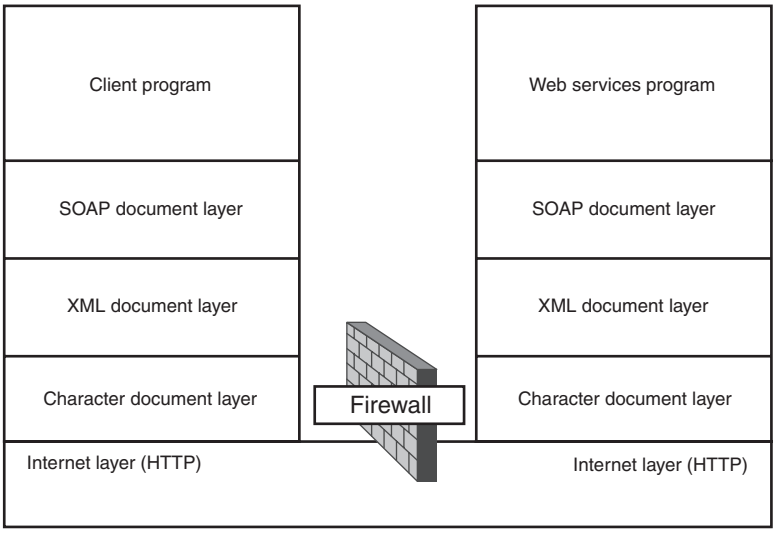

FIGURE 9.1
Looking at the layers makes it easier to understand SOAP's multiple personalities.

From the viewpoint of the Internet layer, computer A is sending an XML document to computer B via HTTP. Computer B's firewall policy states that XML documents are allowed to pass through via HTTP. The Web server on computer B receives the file and hands it to the SOAP processor, which uses an XML parser to read the document.

From the XML parser's point of view, the document is simply a well-formed and valid XML document. The SOAP processing engine evaluates the file against the rules of the SOAP grammar and an XML schema. It examines the SOAP vocabulary to determine if it is valid. If it is valid, the SOAP processor makes a call to the Web service described in the SOAP document and passes it any parameters that the document might contain.

When the Web service finishes its processing, it creates a response that is formatted in an application-specific way. It wraps this response in a SOAP message format, which is a valid XML document also. It stores this document in a file and hands it to the Web server for delivery back to the client computer, computer A.

On computer A, the HTTP client program receives the response file. It calls the SOAP processor to parse and to validate it. If it is a valid document, the SOAP processor passes the response back to the Web services client program that sent the original request.

A couple of details can change in the preceding scenario, but the basic thrust remains the same. Other transport protocols, such as JMS or SMTP, can be used to actually move the message from computer A to computer B. In addition, the SOAP message might not be a method call and response; it could simply be a single call or even just a document being moved.

The Origins of SOAP

SOAP has evolved from an early attempt to define a way to send method calls and parameters from one computer to another called XML-RPC, which stands for Remote Procedure Call. This early specification was defined by Dave Winer of a company called UserLand. IONA, Microsoft, and IBM became interested in improving the XML-RPC approach. This new approach became the SOAP 1.1 specification. It was submitted to the World Wide Web Consortium (W3C) in 2000. A new specification, SOAP 1.2, is moving toward recommendation status at W3C. In W3C terminology, a recommendation is the highest status for a specification. They dislike using the word "standard" because they are a consortium and not a standards body. Commercial Web services platforms and tools are incorporating parts of the SOAP 1.2 specification as of this writing.

Why SOAP Is Different

You might be wondering how this specification differs from a traditional DCOM or CORBA application. Like a CORBA application, a call is made and a response is returned. At that point, the similarity mostly ends. As we discussed in Hour 4, "Comparing Web Services to Other Technologies," a CORBA client actually makes calls to the object on the server in a tightly coupled way. The SOAP client just formats a text file and transfers it to the other machine.

Another difference is the nonchalance of the client program. After the client sends the file, he can either wait for a response or continue with other work until a response arrives. If it never arrives, the client program is responsible for deciding what to do next. It might retry the call, throw an error message, or just log the problem and go on.

Perhaps the biggest difference between the SOAP and CORBA or DCOM approach comes from the casual nature of the relationship between the two computers. You set up CORBA programs by generating special files and placing some of them on each computer. SOAP, in theory, doesn't need the name of anything. A client could use UDDI to find a service, upload the WSDL, generate the client, and make the call without knowing anything in advance.

This casual relationship between client and server mimics the relationship between a Web surfer and a Web site. Sometimes when we are surfing the Web, we visit sites that we didn't even know existed five minutes before. In theory, a Web service could publish its existence on a special type of directory called a repository. Clients could then discover it and connect to it with no human intervention. If this kind of casual relationship becomes popular, it could usher in a whole new wave of applications that are based on just-in-time peer-to-peer discovery.

The SOAP Grammar

The SOAP grammar is fairly simple to understand. Object access means calling methods. A protocol, generically speaking, is a treaty. The computer science meaning of the word "protocol" is a treaty between parties that want to exchange data between their respective computers. We can describe SOAP as a treaty that describes how to call methods on a different computer than the one that we are running on.

SOAP doesn't involve any new inventions or clever algorithms. In fact, the two strongest features of SOAP are its simplicity and the fact that everyone has agreed to use it. A SOAP message is composed of two mandatory parts—the SOAP envelope and the SOAP body—and one optional part—the SOAP header. In addition, all the XML tags associated with SOAP have the prefix SOAP-ENV. The envelope is SOAP-ENV:Envelope; the header is SOAP-ENV:Header; and the body is SOAP-ENV:Body. Figure 9.2 shows the relationship between the parts of a SOAP message.

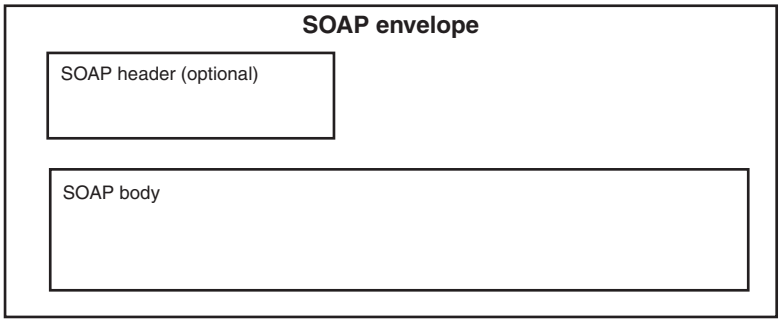

FIGURE 9.2
The SOAP envelope contains both the SOAP header and the SOAP body.

The SOAP envelope is similar to a physical envelope; we can fill it full of data and send it to someone else. The SOAP body is like the contents of the envelope. We can put any information that we want inside the envelope. The SOAP header is like a sticky note that we place inside an envelope when we are sending things to someone else. It contains data that provides special instructions such as "Send your response directly to Bill," or "My password is 12345."

The `SOAP-ENV:Envelope` Tag

A SOAP message is defined as beginning with the tag

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

and ending with the tag

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

Whenever you see the string `<SOAP-ENV:Envelope>`, say in your mind "beginning of the SOAP message;" and when you see `</SOAP-ENV:Envelope>`, you can say "end of the SOAP message." SOAP messages cannot be sent in batches, so you know that you are looking at only one message inside the envelope. There might also be a header section that can contain *n* header elements.

In SOAP 1.1, the `SOAP_ENV:Envelope` tag is normally constructed using the following syntax:

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

The string `xmlns` is a keyword in XML that stands for XML namespace. The namespace is used to uniquely identify all tags in order to avoid tag name conflicts. The `SOAP-ENV` part is the name that SOAP requires to be used as the prefix for all the tag names that SOAP defines. The string `"http://schemas.xmlsoap.org/soap/envelope/"` looks like the address of an ordinary Web site. In reality, it is a unique string that serves the same purpose as a version number would. A client places a string that indicates indirectly which version of SOAP it is using. The Web service that receives the request can look at this string to determine whether it is capable of communicating using that version of SOAP. Namespaces are covered in detail in Hour 7, "Understanding XML."

Two other namespaces that are heavily used in SOAP are xsd and xsi. The xsd namespace specifies that these tags come from the XML schema definition. The xsi namespace indicates that these tags come from the XML schema–instance definition.

The `SOAP-ENV:Body` Tag

The body of the SOAP message begins with the tag

```
<SOAP-ENV:Body>
```

and ends with the tag

```
</SOAP-ENV:Body>
```

Whenever you see the string `<SOAP-ENV:Body>`, say in your mind "beginning of the SOAP body;" and when you see the string `</SOAP-ENV:body>`, you can say "end of the SOAP body." This is where the payload of the SOAP message is placed. Normally, that payload is a method call to a remote computer, complete with parameter values. Sometimes, however, it is simply an XML document that is being transferred.

At other times, it might be a response message containing a bank balance or a picture of the first moonwalk. The format of the body is under the control of whoever is creating a new Web service. A special XML document, called the Web services Description

Language (WSDL) document, is created to describe what a legal method call to that service would look like and what form a valid response can take. The following snippet shows a body that makes a Remote Procedure Call (RPC) to a method called `checkAccountBalance()`:

```
<SOAP-ENV:Body>
          <checkAccountBalance>
           <accountNumber xsi:type="xsd:int">123456780</accountNumber>
       </checkAccountBalance>
</SOAP-ENV:Body>
```


The first line indicates that this is the start of the body and the last line shows the end of the body. The second line,

```
<checkAccountBalance>
```

provides the name of the method to call, `checkAccountBalance`. The first element is called `accountNumber`, and it is a parameter that is being passed in with the `checkAccountBalance` method:

```
<accountNumber xsi:type="xsd:int">123456780</accountNumber>
```

The `xsi:type` is an attribute, and the `xsd:int` means that this value is an integer. `123456780` is the value of the parameter being passed. The net effect of all these characters is a method call that would look something like this in Java:

```
int balance = checkAccountBalance(123456780);
```

The `SOAP-ENV:Header` Tag

The `SOAP-ENV:Header` element is optional in a SOAP message. If a header is present, however, it must be the first child element that appears in the SOAP envelope. The format of the `SOAP-ENV:Header` element is not defined in the specification; therefore, it is available to the clients and services for their own use. Typical use would be to communicate credentials such as username and password.

Two attributes associated with the `SOAP-ENV:Header` element can be used. The first is the `SOAP-ENV:mustUnderstand` attribute. If it is set to `"1"`, an error message will be generated if the Web service is not programmed to handle the fields in this header. The client programmer has to decide whether the processing can take place on a site that can't read the header.

The second attribute is called `SOAP-ENV:actor`. This attribute is used to chain together Web services that this document must visit to be completely processed. Think about how a purchase order could be viewed by the payroll department to calculate commissions, by accounts receivable to create a bill, and by the shipping department to send the physical

merchandise. The chain can be created by adding SOAP-ENV:actor tags along with their URIs to the header.

The following snippet shows a simple SOAP-ENV:Header:

```
<SOAP-ENV:Header>
   <myNS:authentication xmlns:myNS="http://www.stevepotts.com/auth"
                                        SOAP-ENV:mustUnderstand="1">
      <loginID>
         admin
      </loginID>
      <password>
         rover
      </password>
   </myNS:authentication>
<SOAP-ENV:Header>
```

The header contains a made-up element called <myNS:authentication>. That element contains the loginID and password elements. These elements have no standard meaning in SOAP at this time. One criticism of SOAP is that it doesn't support some very needful topics such as sessions, transactions, and the authentication of users. These shortcomings don't keep us from using our own approaches to these problems; it just means that we have to communicate our approaches to our potential clients. Eventually, when these areas are added to the SOAP specification, we will have to replace our proprietary approaches with the approved ones.

Most of the growth in the SOAP standard is expected to take place in the area of headers. Expect to see many more predefined elements and attributes such as SOAP-ENV:actor and SOAP-ENV:mustUnderstand added over time to address the perceived weaknesses in the current SOAP specification.

Reporting Errors to the Client

No technology can be considered production ready until it supports error handling well. Toy systems running in a lab can defer these considerations, but production systems can't afford to be fragile. They must recover from as many errors as possible. In cases in which they can't recover, they must provide the support staff with plenty of information about what went wrong.

The SOAP approach to error handling is based on the proper use of the SOAP-ENV:fault tag.

The SOAP-ENV:Body has one child that is defined by the SOAP specification—the SOAP-ENV:fault tag. This tag is used to communicate that a problem has occurred in the attempted fulfillment of the request sent to the Web service.

This optional element must appear only in response messages, and it can appear only once in that message. The `SOAP-ENV:fault` tag has four optional tags:

- **`SOAP-ENV:faultcode`**—This element is required by the specification. It should contain some code indicating what the problem is.
- **`SOAP-ENV:faultstring`**—This required element is a human-readable version of the faultcode. It should provide details beyond the "error some place" type of message.
- **`SOAP-ENV:faultactor`**—This optional element tells which service generated the fault. This is important when a chain of services was used to process the request.
- **`SOAP-ENV:detail`**—This element should contain as much information as possible about the state of the server at the time of the crash. It often contains the values of variables at the time of the failure.

Error codes in SOAP are defined in the specification in a way that you might not have predicted. In other languages, integers are used to represent faults. In SOAP, error codes are represented as two-part strings with major and minor error codes separated by a . like this:

```
<SOAP-ENV:faultcode>
    Server.customerCreateFailed
<SOAP-ENV:faultcode>
```

Four types of generic faultcode are defined by the specification:

- **`server`**—An error occurred on the server, but not with the message itself. You should write your client to retry messages that fail with these codes. If the error is with the availability of the service, a subsequent retry would work. Limit the number of retries, however, because the error might be coming from the service itself and would therefore not go away with the passage of time.
- **`client`**—These errors indicate that something is wrong with the message itself, such as a bad message format, incomplete message, and so on.
- **`versionMismatch`**—This error occurs when the versions of the SOAP processors are different between the client and the server. The version is determined by the namespace URI used in the `SOAP-ENV:Envelope` tag.
- **`mustUnderstand`**—This error is generated when an element in the header cannot be processed and that element is marked as required. If an element contains a `mustUnderstand` attribute, it requires that the Web service be able to understand all the contents of the element. If not, you want to receive this error message.

An example of a full-blown fault tag is shown here:

```
<SOAP-ENV:Fault>
   <SOAP-ENV:faultcode>
      Client.Authentication
   </SOAP-ENV:faultcode>
   <SOAP-ENV:faultstring>
       This customer is unknown to our system.
   </SOAP-ENV:faultstring>
   <SOAP-ENV:faultactor>
      http://www.samspublishing.com/authors
   </SOAP-ENV:faultactor>
   <SOAP-ENV:detail>
      <customer custID="12345">
         <name>
            Yogi Bear
         </name>
   </SOAP-ENV:detail>
</SOAP-ENV:Fault>
```

Notice that the `faultcode` has a specific format and that the `faultactor` must contain the URI of a Web service. The other two elements accept any information that you want to provide to your user.

SOAP Data Types

One of the most difficult problems for intercomputer communication concerns the representation of data types. Declaring data to be of a certain type is fundamental to getting a computer program to work correctly.

Data typing provides three primary advantages:

- Strongly typed data is more efficient to store and process than untyped data. Untyped data processing requires additional processing to determine whether a requested action is allowed on this type of data. For typed data, this is a simple task, but for untyped data, many cycles must be consumed to make this determination.
- Typed data can be combined with other data with a higher degree of confidence than nontyped data. There is also less potential for confusion when doing conversions. The processor decides whether an operation can be performed on a piece of data. Because it is inferring the true type of the data, it sometimes behaves differently than the programmer thought it would.
- Typed data makes it easier for the language compiler to recognize and reject nonsensical operations such as multiplying your social security number, which is stored in a string variable number, by the number 3.

SOAP does allow us to pass data without data type information. If we specify no type, the default type of string is used. This default can be overridden by the inclusion of a string that indicates the type of the data along with the string version of the data itself, as shown here:

```
<accountNumber xsi:type="xsd:int">123456780</accountNumber>
```


The simplicity of this scheme is impressive. For example, different computers store integer data in different ways. Some machines store them in 16 bits, others in 32 bits, whereas still others consume 64 bits for each integer. To make matters worse, some computers store numeric data with the higher numbers in the left bytes, whereas others store them in the right bytes. But the last straw is that not all computers use the same bit patterns to represent characters; some use ASCII characters, and some use other types of encoding.

SOAP takes advantage of the one format that all computer brands and models can easily share with each other: text. All software products that can be used as transports for SOAP messages—such as HTTP, JMS, and SMTP—can take a text file that resides on one computer and transfer it to another computer without the loss of data.

SOAP delegates the data type conversion work to the programmers who create the Web service and the client software that accesses it. All Web services must have software (written in a programming language) that implements the business functionality that the service offers. All these programming languages can convert string representations of data into typed variables with accuracy if they know what data type to store it in. By requiring the inclusion of the data type information in the message, SOAP ensures that the conversion from strings to numeric and numeric to strings will be correct. Figure 9.3 graphically shows this process.

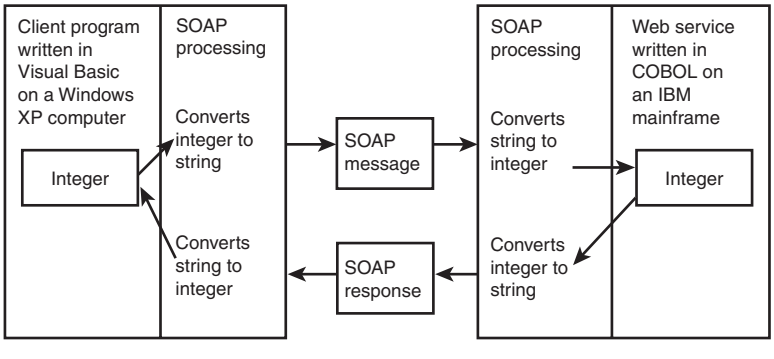

FIGURE 9.3
Data type conversion is the responsibility of the SOAP message creators and consumers.

Another way to communicate the type of an element is to use an XML schema. The following is a XML schema example that contains data type declarations for the elements:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLschema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <xsd:complexType name=customer content="mixed">
      <xsd:element type="custID"></element>
      <xsd:element type="lastName"></element>
      <xsd:element type="firstName"></element>
   </xsd:complexType>
   <xsd:simpleType name="custID"
      xsi:type=" xsd:integer">
   </xsd:simpleType>
   <xsd:simpleType name="lastName"
      xsi:type="xsd:string">
   </xsd:simpleType>
   <xsd:simpleType name="firstName"
       xsi:type="xsd:string">
   </xsd:simpleType>
</xsd:schema>
```

Two different categories of data types are present in this example. One category is the `complexType`. This type is made up of multiple `simpleTypes`. Each `simpleType` has a specific datatype, such as `xsd:integer` or `xsd:string`. The `xsd:` prefix indicates that this tag is a well-known member of the XML schema namespace. If you stick to the data types in this namespace, you can be certain that the Web service on the other end will be able to handle the data that you send. The data types defined by this namespace are `string`, `normalizedString`, `token`, `byte`, `unsignedByte`, `integer`, `positiveInteger`, `negativeInteger`, `short`, `decimal`, `float`, `double`, `boolean`, `time`, `datetime`, `anyURI`, `language`, and a number of other esoteric types.

You can obtain a full list of the valid XML data types at `http://www.w3.org/TR/xmlschema-0/#SimpleTypeFacets`.

If you include a schema definition in your SOAP document, you can skip the explicit declaration of the data types inside the document. The WSDL document normally includes this type of schema data. You will learn about the WSDL document in Hour 10, "Describing a Web Service with the Web Services Description Language (WSDL)."

Summary

This hour has introduced you to the basic concepts behind SOAP. Early in the hour, we defined what SOAP is and why it is needed. Following that, we covered how SOAP messages are used to exchange messages between Web services and their clients.

Next, we covered the SOAP grammar. You learned different parts of the SOAP document and why they exist. We also discussed how SOAP reports errors back to the client.

Finally, you saw how to specify the types of data that is being transferred between computers.

Q&A

Q **What is the purpose of SOAP?**

A SOAP is a specification that describes how to move data from one computer to another. Originally, SOAP was specified to make it easy to make method calls on another computer and return the result. Now, however, whole documents are commonly sent using SOAP.

Q **What role does XML play in SOAP?**

A SOAP is written using XML-style tags. A SOAP document is an XML document that follows a more stringent set of rules.

Workshop

The Workshop is designed to help you review what you've learned and begin learning how to put your knowledge into practice.

Quiz

1. What role does SOAP play in the creation of a Web service transaction?
2. What types of problems are well suited to this technology?
3. Why is it important to be able to specify data types in a SOAP document?

Quiz Answers

1. SOAP format is for the actual message that gets sent from the client to the service and from the service back to the client. The syntax of the SOAP grammar allows for instructions to be added to the header with the actual method call or XML document to be added to the body.

2. SOAP excels at allowing a client to make a method call on a Web service. It also does a good job of supporting the transfer of XML files from clients to servers and vice versa.
3. Values are sent in SOAP documents as strings. At times, it can be difficult for programs to determine the type of the data that is being sent unless it is provided with a hint.

Activities

1. Go to the SOAP tutorial's appendix on data types and see how many different types are supported.
2. Use a SOAP monitor to examine the messages that one of your Web services is exchanging with clients. Use one of the SOAP monitors that are provided in the commercial products that we cover in Hours 13–19 of this book.