CHAPTER 11

Web Service Security

A Web service is a program that has a message-based interface described in a WSDL (**Web Service Description Language**) document. A WSDL description is somewhat similar to a CORBA IDL (**Interface Definition Language**) description, in the sense that it describes the interface of a network service. A WSDL document defines a set of endpoints operating on messages. The operations and messages are described abstractly, and then bound to a concrete network protocol, such as HTTP, and a message packaging format, such as SOAP (**Simple Object Access Protocol**), to define a *binding*. The combination of a binding and a network address makes a concrete endpoint, also known as a *port*. A Web service is simply a collection of ports.

Although it is possible to specify any message format and transport protocol within a WSDL document, the most common choices are SOAP for message format and HTTP/S for transport. Not surprisingly, the term Web service almost always implies exchange of SOAP messages over HTTP/S. As you may already know, SOAP is a XML packaging scheme consisting of a `Header` element and a `Body` element, encapsulated within an `Envelope` element. We will look at examples of SOAP and WSDL documents later in the chapter.

Both SOAP and WSDL have been hailed by the computing industry as revolutionary technologies with the promise to usher us in a new age of program to program communication over the Internet which will change the way businesses interact with each other. This claim needs to be seen in the right historical perspective. The use of widely accepted protocols such as TCP/IP, HTTP and document formats such as HTML has been the primary reason for explosive growth of human-oriented Web applications. With time, it is possible that a number of activities that currently require human interaction will be taken over by program-to-program interaction.

However, for the time being, Web services are mostly being used as integration technology for programs written in different languages and running on different platforms.

Regardless of what problem you are solving with Web services, you need to address the same security issues that other integration or distributed computing technologies need to address. The issues include:

- the client’s need to authenticate itself to the server;
- the server’s need to authenticate the client;
- the need to guarantee integrity and confidentiality of the messages being exchanged; and
- the need to determine the access rights of a client by its verified identity and internal access control policy.

In cases where a Web service invocation is not much different from the synchronous exchange of request and response messages over HTTP, the transport-based security mechanism like SSL may be quite adequate. However, not all Web service-based applications are going to be of this kind. In scenarios where a message needs to be transported to multiple endpoints, one after another, asynchronously, going over multiple transports, and through intermediaries and across corporate firewalls, we need message-based end-to-end security. Recall that the transport-based security works well only when both the communicating endpoints are active at the same time and there is no requirement for an intermediary to examine the message content for routing, validation or any other purpose. Message-based security doesn’t suffer from these limitations.

Having covered transport-based security in the chapter *Securing the Wire* and message-based security in the chapter *Securing the Message*, we are now ready to discuss, develop and deploy secure Web services in this chapter.

But before we get into details let us spend some time talking about Web services standards.

Web Services Standards

We have already mentioned SOAP and WSDL. SOAP describes the format of data that is transmitted over the wire. WSDL describes the interface specifying *what messages* an endpoint may receive and send. These descriptions must be understood and processed by communicating parties for meaningful exchange of messages. For this reason, they are also known as *interoperability standards*. Interoperability standards are critical for programs interacting over the network to work.

Most widely used version of SOAP, SOAP 1.1 is a W3C Note dated May 08, 2000. Since then, W3C has created a working group named XML Protocol Working Group, to work on its standardization. At the time of finalizing this chapter, the current specification, known as *SOAP Version 1.2*, has become a W3C Recommendation stage. There have been some changes from SOAP 1.1 to SOAP Version 1.2 in certain areas, but these changes are not significant for the discussion in this chapter.

Likewise, WSDL 1.1 is a W3C Note dated March 15, 2001. The Web Services Description Working Group is working towards its standardization. The current draft maintained by this group is a working draft and may undergo significant changes before it becomes a recommended specification. Most of the current implementations are based on

WSDL 1.1 and this is what we will use for our discussion. Actually, most of what we talk about is independent of WSDL specifics and hence should be applicable to WSDL Version 1.2 as well.

The Web services standard most relevant to us in this chapter is WS Security (**Web Services Security**) Version 1.0 published by IBM, Microsoft and VeriSign on April 5, 2002 and available online at http://www-106.ibm.com/developerworks/webservices/library/ws-secure/. This document has been augmented with *Web Services Security Addendum Version 1.0*, published on August 18, 2002 and available at http://www-106.ibm.com/developerworks/webservices/library/ws-secureadd.html. Together, these documents describe enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication.

Since the publication of WS Security specification by IBM, Microsoft and VeriSign, an OASIS (**Organization for the Advancement of Structured Information Standards**) Technical Committee, known as Web Services Security TC, has been formed to further develop this specification as a standard. At the time of finalizing this chapter (May 2003), this committee has not yet completed its work. For the purpose of this chapter, we will rely on the two documents mentioned in the previous paragraph.

WEB SERVICES IN JAVA

To develop a Web service in Java, you can either start with a WSDL document describing the Web service interface and supply the Java implementation code or start with an existing Java program and generate the corresponding WSDL document. The choice largely depends on whether you are developing a Web service interface from scratch or you already have a Java-based interface defined.

A Web service implementation can either work with raw SOAP messages or the corresponding Java objects as input. In the later case, a layer of software, commonly referred to as a SOAP engine, converts SOAP document to Java objects. For output, the same engine converts the output objects to response SOAP document.

These conversions require XML data types defined in WSDL documents to be mapped to Java types and vice versa. JAX-RPC (**Java API for XML-Based RPC**) specification defines these mappings. Besides data type mappings, JAX-RPC also defines the programming model for a Web service client. A JAX-RPC client can use stub-based, dynamic proxy or DII (**Dynamic Invocation Interface**) programming models to invoke a Web service endpoint. In stub-based programming model, you generate client side stub classes from WSDL description and place them in the `CLASSPATH` of the client program. A dynamic proxy avoids separate generation of source files and their compilation, still allows the client program to invoke methods on a local instance. The DII model works at much lower level and requires the client program to pass operation and parameter names as `String` and parameter values as an array of Java `Object` instances. Refer to the JAX-RPC specification for more details on these programming models.

JAX-RPC also defines a lifecycle model for a servlet-based Web service endpoint. Keep in mind that a Web service endpoint could be developed as a Servlet or EJB and deployed in a Web container or EJB container.

JAX-RPC also defines a handler mechanism to process SOAP Header elements at both the client and service ends. This mechanism allows one or more handler objects, forming a chain of handlers, to be specified to process request and/or response SOAP messages. As we will see later in the chapter, handlers expect a tree-like object representation of SOAP documents, specified by SAAJ (**SOAP with Attachments API for Java**) specification. In certain respects, SAAJ is similar to W3C DOM API discussed in the chapter *Securing the Message*. However, there are significant differences: SAAJ doesn't support valid XML constructs such as DTDs (**Document Type Declarations**) and PIs (**Processing Instructions**) but supports binary attachments specified by the specification *SOAP Messages with Attachments* (a W3C Note). Most importantly, a SAAJ tree node is not a W3C DOM tree node[1]. This difference does cause a problem if you want to reuse code in a handler that expects W3C DOM structure. We will outline a mechanism to address this problem while writing handlers for WS Security later in the chapter.

The packaging of Web service endpoint implementations and corresponding deployment descriptors are specified in the specification *Web Services for J2EE* (JSR 109). JAX-RPC, SAAJ and *Web Services for J2EE* are all going to be part of J2EE 1.4. However, we will not use any *Web Services for J2EE* features in this chapter.

Invocation of Web Services based on JAX-RPC runtime systems is illustrated in *Figure 11-1*.

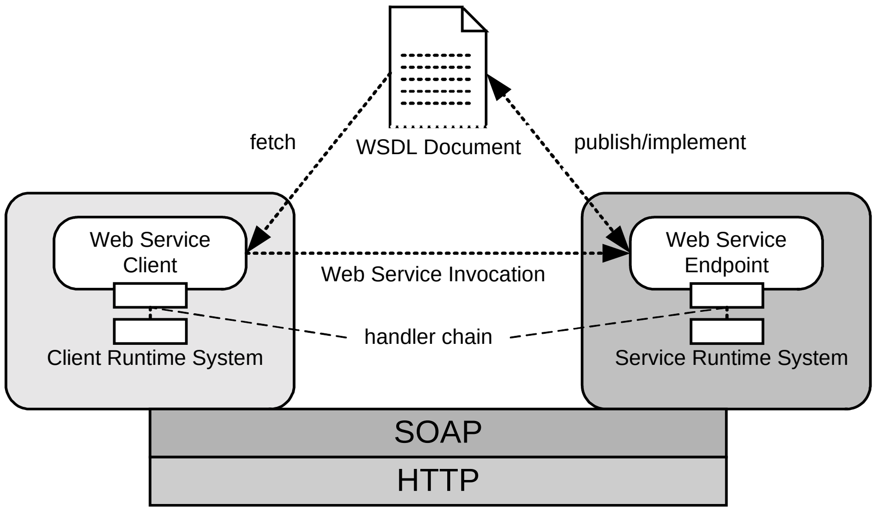


Figure 11-1: JAX-RPC runtime system

[1] This problem has been taken care of in SAAJ 1.2, a maintenance release made available in April 2003.

JAX-RPC-compliant SOAP engines include the client and service runtime systems and tools to generate Java stubs from WSDL documents and WSDL documents from Java classes. We will use Apache Axis, an open source JAX-RPC-compliant SOAP engine for our example programs.

Apache Axis

Apache Axis is a JAX-RPC-compliant open source SOAP engine implementation from ASF (**Apache Software Foundation**). It typically runs as a Web application within a Web container like Apache Tomcat. This deployment allows it to benefit from Web container features like resource poolng, multi-threading, HTTP protocol handling, security support etc.

At the time of writing this chapter, the latest binary release of Axis is axis-1.1RC2. This is what we will use, along with Web container software Apache Tomcat 4.1.18, the one we used in the chapter *Web Application Security*. Although these specific releases have been used to develop and test the examples, they should work fine with future releases as well.

Rest of the chapter assumes that you have Tomcat installed on your machine.

Installing and Running Axis

Download the zip file containing the runtime binaries by following the download instructions available at the Axis homepage at http://ws.apache.org/axis. For Axis-1.1RC2, name of the distribution zip file is `axis-1_1rc2.zip`. At the end of the download, you should have this file in your working directory. To unzip it, open a command shell and issue the following command:

```
C:\apache>jar xvf axis-1_1rc2.zip
```

This creates the Axis home directory, `axis-1_1RC2`, in the current directory. Set environment variable `AXIS_HOME` to point to this directory. We will be referring to it for running example programs.

Look inside this directory. Here you will find Axis documentation within the `docs` directory, jar files within the `lib` directory, sample applications within the `samples` directory and the Axis Web application, packaged within a directory tree rooted at `axis`, within the `webapps` directory. If you are new to Axis then spend some time going through the documentation.

To deploy Axis to Tomcat, create file `axis.xml` in the webapps sub-directory of the Tomcat home directory. The content of this file is shown below.

```
<Context path="/axis" docBase="c:\\apache\\axis-1_1RC2\\webapps\\axis"
        debug="0" privileged="false">
</Context>
```

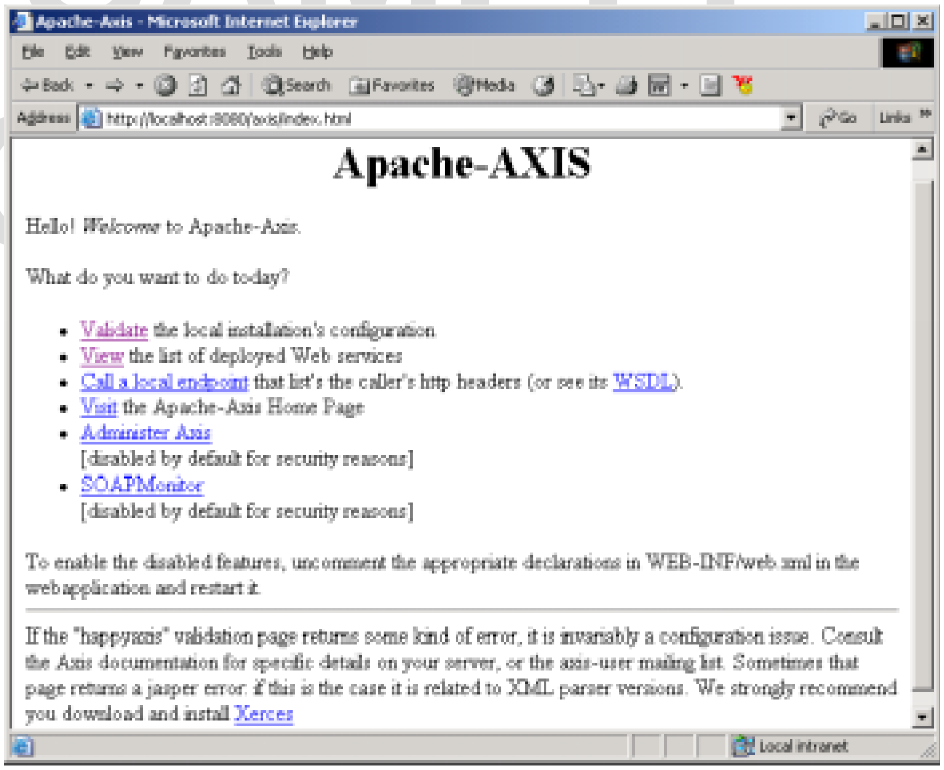
You should change the value of `docBase` to use the pathname of the Axis installation directory on your machine. An alternative deployment mechanism is to copy the Axis Web application directory tree rooted at `axis` to the `webapps` directory of the Tomcat home directory.

Now change your working directory to Tomcat home directory and start Tomcat.

```
C:\apache\jakarta-tomcat-4.1.18-LE-jdk14>bin\startup
```

Once Tomcat is running, you can check for successful deployment of Axis by pointing your Web browser to `http://localhost:8080/axis`. Recall that the Tomcat listens for HTTP connections at port 8080 by default. If your installation is configured for a different port then you should use that port in the URL.

Successful Axis deployment is confirmed by your browser display of the Axis welcome page as shown in *Figure 11-2*.


Apache-Axis - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/axis/index.html
Go Links
Local intranet

Apache-AXIS

Hello! Welcome to Apache-Axis.

What do you want to do today?

- Validate the local installation's configuration
- View the list of deployed Web services
- Call a local endpoint that list's the caller's http headers (or see its WSDL).
- Visit the Apache-Axis Home Page
- Administer Axis
  [disabled by default for security reasons]
- SOAPMonitor
  [disabled by default for security reasons]

To enable the disabled features, uncomment the appropriate declarations in WEB-INF/web.xml in the webapplication and restart it.

If the "happyaxis" validation page returns some kind of error, it is invariably a configuration issue. Consult the Axis documentation for specific details on your server, or the axis-user mailing list. Sometimes that page returns a jasper error: if this is the case it is related to XML parser versions. We strongly recommend you download and install Xerces

Figure 11-2: Axis Welcome Page

Clicking the Validate link brings up a configuration report page with a host of information on jar files in the CLASSPATH, system properties, Web container identification and so on. During development, this page is helpful in identifying configuration problems. However, it reveals too much information about the system and could be a security risk. Hence, it is advised that you disable this particular page in production systems by editing the web.xml file of Axis Web application.

You can view the currently deployed Web services by clicking the View link. The resulting page not only lists the deployed Web services by name but also includes available operations and the URL to retrieve corresponding WSDL documents. Again, this capability should be disabled on production systems.

A Simple Web Service

In this section, we will develop a simple Web service, deploy it, and write a client program to access it. This Web service will accept a String object as input and return the same String as output. The basic motivation is to become comfortable using Axis and build a base example which can be extended to illustrate security concepts later on.

Though there are many different ways to develop a Web service with Axis, we will start with a Java class StringEchoService1. The source file StringEchoService1.java is shown below and its electronic copy, along with other source files and execution scripts, can be found in subdirectory src\jsbook\ch11\ex1 of the JSTK installation directory.

```
// File: src\jsbook\ch11\ex1\StringEchoService1.java
public class StringEchoService1 {
    public String echo(String arg){
        return arg;
    }
}
```

You may wonder: what is special about this class that makes it fit for being a Web service? Nothing! All the magic is in the Axis runtime system that takes this class and provide the necessary plumbing for WSDL generation, conversion of request documents into Java String object, invocation of the method and so on. Of course, this works only for simple classes that follow certain rules. Refer to Axis documentation for more details.

To deploy this Web service, you need to first compile the source file and place the compiled file, or a jar file containing this file, to a location from where Axis Web application can load it. For a compiled class file, the location is axis\WEB-INF\classes directory and for a jar file, the location is axis\WEB-INF\lib directory, within the webapps directory of Tomcat installation directory. Assuming that the environment variable AXIS_HOME is set to the Axis home directory, carry out these steps by issuing commands:

```
C:\ch11\ex1>javac StringEchoService1.java
C:\ch11\ex1>jar cf ex1.jar StringEchoService1.class
C:\ch11\ex1>copy ex1.jar %AXIS_HOME%\webapps\axis\WEB-INF\lib
```

The above sequence of commands compiles the StringEchoService1.java file, archives the compiled class file into the jar file ex1.jar and copies this file to the lib directory of the Axis Web application.

Once this is done, we need to deploy the Web service. This requires running Axis utility org.apache.axis.client.AdminClient with a deployment descriptor file deploy.wsdd as argument. To be able to run this utility, Tomcat should be running with Axis deployed, and you should have all the jar files of directory %AXIS_HOME%\lib in the CLASSPATH. Also, do not forget to restart the Tomcat after copying the ex1.jar to the lib directory of the Axis Web application.

The deployment descriptor file deploy.wsdd is shown below in *Listing 11-1*.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

 <service name="StringEchoPort1" provider="java:RPC">
  <parameter name="wsdlTargetNamespace"
    value="http://www.pankaj-k.net/jsbook/examples/"/>
  <parameter name="wsdlServiceElement" value="StringEchoService1"/>
  <parameter name="wsdlServicePort" value="StringEchoPort1"/>
  <parameter name="wsdlPortType" value="StringEcho"/>
  <parameter name="scope" value="session"/>
  <parameter name="className" value="StringEchoService1"/>
  <parameter name="allowedMethods" value="*"/>
 </service>
</deployment>
```

Listing 11-1: Deployment descriptor for example Web service

Note that this deployment descriptor specifies the target namespace URI for WSDL as "http://www.pankaj-k.net/jsbook/examples/", the port name as "StringEchoPort1", the service name as "StringEchoService1" and the implementation class as "StringEchoService1". We will need these values to write a client program for this service. Parameter "allowedMethods" is used to control which methods of the class should be exposed through WSDL. A value of "*" implies that all methods are exposed. Refer to Axis documentation for a detailed description of all parameters.

To deploy the Web service, run the AdminClient class with deployment descriptor file as argument. The class AdminClient is part of the Axis jar files. To be able to run it, you must set the CLASSPATH to include all the jar files in the %AXIS_HOME%\lib directory. This is accomplished by running the following sequence of commands.

```
C:\ch11\ex1>set _CP=.
C:\ch11\ex1>for %f in (%AXIS_HOME%\lib\*) do call cpappend.bat %f
C:\ch11\ex1>set CLASSPATH=%_CP%;%CLASSPATH%
```

Here, the batch file cpappend.bat contains the command "set _CP=%_CP%;%1". The use of this file helps to get all the jar files of the directory %AXIS_HOME%\lib in the CLASSPATH without individually listing each of the files.

This flexibility is useful as the names and the number of jar files sometime change from one release to another.

We are now ready to run the `AdminClient` program.

```
C:\ch11\ex1>java org.apache.axis.client.AdminClient deploy.wsdd
- Processing file deploy.wsdd
- <Admin>Done processing</Admin>
```

You can check for successful deployment by pointing your Web browser to the Axis welcome page and clicking on the View link to list the deployed Web services. You should find `StringEchoPort1` service there. If you get an error, try listing the services after stopping and starting the Tomcat. Once you see the `StringEchoPort1` in the list of deployed services, you know that the deployment has been successful.

A Web service deployment survives Web container stop and start, meaning a restart of the Web container doesn't mean that all the deployed services get undeployed.

To view the WSDL description of the deployed service, point your Web browser to `http://localhost:8080/axis/services/StringEchoPort1?wsdl`. The browser displays the WSDL document generated by Axis. This WSDL file is shown in *Listing 11-2*.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
        targetNamespace="http://www.pankaj-k.net/jsbook/examples/"
        xmlns:impl="http://www.pankaj-k.net/jsbook/examples/"
        xmlns:intf="http://www.pankaj-k.net/jsbook/examples/"
        xmlns:apachesoap="http://xml.apache.org/xml-soap"
        xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:message name="echoResponse">
    <wsdl:part name="echoReturn" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="echoRequest">
    <wsdl:part name="in0" type="xsd:string"/>
  </wsdl:message>
  <wsdl:portType name="StringEcho">
    <wsdl:operation name="echo" parameterOrder="in0">
      <wsdl:input name="echoRequest" message="impl:echoRequest"/>
      <wsdl:output name="echoResponse" message="impl:echoResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="StringEchoPort1SoapBinding"
         type="impl:StringEcho">
    <wsdlsoap:binding style="rpc"
          transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="echo">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="echoRequest">
        <wsdlsoap:body use="encoded"
              encodingStyle=”http://schemas.xmlsoap.org/soap/encoding/”
              namespace="http://www.pankaj-k.net/jsbook/examples/"/>
      </wsdl:input>
      <wsdl:output name="echoResponse">
        <wsdlsoap:body use="encoded"
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="http://www.pankaj-k.net/jsbook/examples/"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="StringEchoService1">
    <wsdl:port name="StringEchoPort1"
          binding="impl:StringEchoPort1SoapBinding">
      <wsdlsoap:address
        location="http://localhost:8080/axis/services/StringEchoPort1"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Listing 11-2: WSDL document of Web service `StringEchoService1`

Can you figure out where all these values came from? Axis got some of them from the deployment descriptor file `deploy.wsdd` and figured out others by introspecting the implementation class `StringEchoService1`.

You can undeploy the Web service by running the `AdminClient` with name of the file with XML text instructing Axis to undeploy the `StringEchoPort1` Web service.

```
C:\...\ex1>java org.apache.axis.client.AdminClient undeploy.wsdd
- Processing file undeploy.wsdd
- <Admin>Done processing</Admin>
```

File `undeploy.wsdd` contains the XML document identifying the service to be undeployed.

```
<undeployment xmlns="http://xml.apache.org/axis/wsdd/">
   <service name="StringEchoPort1"/>
</undeployment>
```

This operation simply instructs Axis to not to process request messages directed to the Web service but it doesn't remove the `ex1.jar` file from the `lib` directory.

A Web Service Client

The next step is to write a client program to invoke the `StringEchoPort1` Web service. We will use JAX-RPC DII (**Dynamic Invocation Interface**) for this purpose. This interface includes low-level APIs to get the WSDL document, form the request message, send it and receive the response message, as shown in *Listing 11-3*.

```
// File: src\jsbook\ch11\ex1\EchoClient.java
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
```

```
public class EchoClient{
  public static void main(String [] args) throws Exception {
    String epAddr =
        "http://localhost:8080/axis/services/StringEchoPort1";
    String wsdlAddr = epAddr + "?wsdl";
    String nameSpaceUri = "http://www.pankaj-k.net/jsbook/examples/";
    String svcName = "StringEchoService1";
    String portName = "StringEchoPort1";

    java.net.URL wsdlUrl = new java.net.URL(wsdlAddr);
    ServiceFactory svcFactory = ServiceFactory.newInstance();
    QName svcQName = new QName(nameSpaceUri, svcName);
    Service svc = svcFactory.createService(wsdlUrl, svcQName);

    Call call = (Call) svc.createCall();

    call.setTargetEndpointAddress(epAddr);
    call.setOperationName( new QName(nameSpaceUri, "echo") );
    call.setPortTypeName( new QName(nameSpaceUri, portName) );

    Object arg = "Hi, How are you?";
    System.out.println("sending: " + arg );
    String res = (String) call.invoke(new Object[] {arg});
    System.out.println("received: " + res );
  }
}
```

Listing 11-3: Source File `EchoClient.java`

Note the use of the service endpoint address, the WSDL location address, the WSDL namespace URI, the service name, the port name and the method name in constructing and initializing an instance of `javax.xml.rpc.Call` object. For actual invocation, you call the method `invoke()` on this object, passing an `Object` array, initialized with the method arguments. For details of JAX-RPC classes and methods used in *Listing 11-3*, refer to the Axis API Javadocs.

The client program is compiled and run like any normal Java program, with the environment variable `CLASSPATH` set so that it includes Axis jar files.

```
C:\ch11\ex1>javac EchoClient.java
C:\ch11\ex1>java EchoClient
sending: Hi, How are you?
received: Hi, How are you?
```

Congratulations! You have successfully deployed and accessed your first Web service.

You may have noticed that the client source code is much more complex than the service code. Actually, there are other, simpler ways to write clients using generated stubs and/or dynamic proxies. However, we chose to use the relatively low-level dynamic invocation interface, for only this interface gives us the power we will need to extend this example later in the chapter to perform some of the security related tasks.

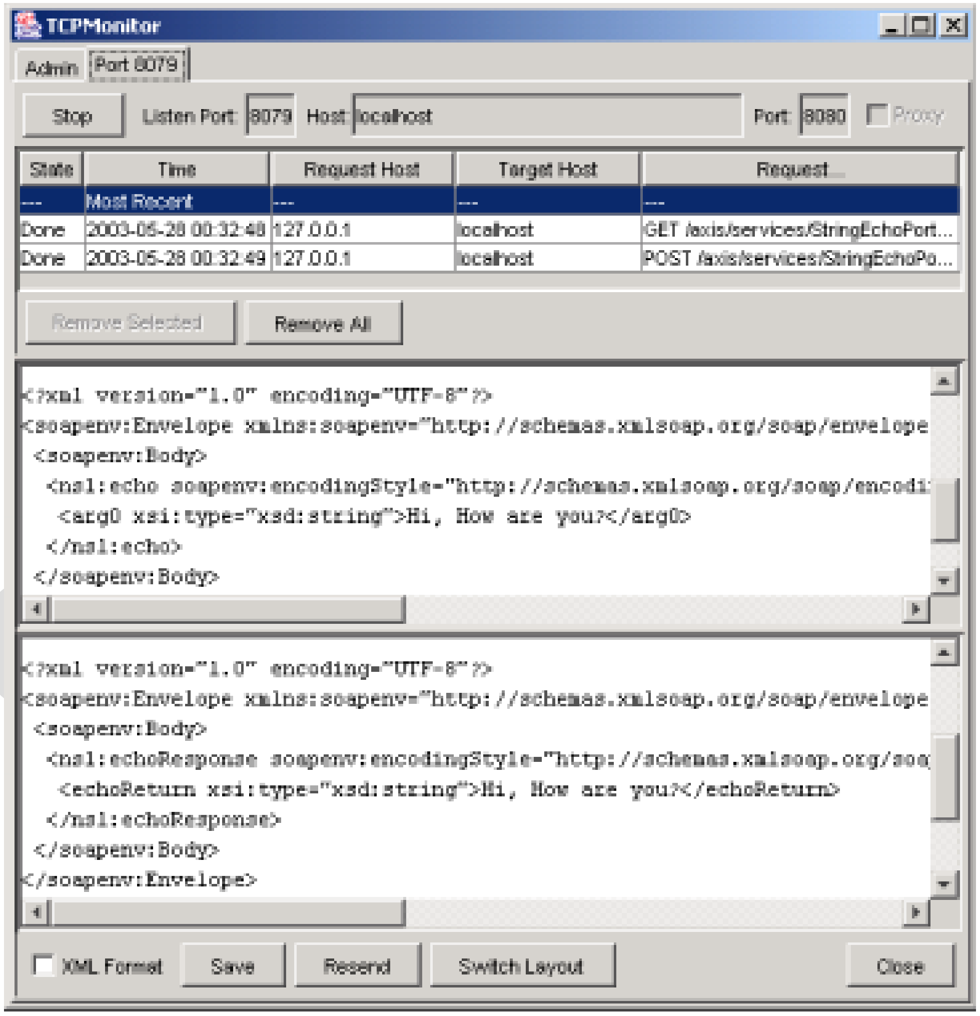
Watching the SOAP Messages

Although our client and service programs are using Java objects as input and output, internally these Java objects are being converted to SOAP messages and being exchanged over an HTTP connection. You could use **`tcpmon`**, a nifty, Swing-based GUI tool packaged with the Axis, to intercept SOAP messages and look at them. The only requirement is that the client program be modified to target all the outbound messages to a port that this tool is configured to listen at. Also, it needs to be configured to forward the message to the address of the Web service.

To run **`tcpmon`** with listen port of 8079 and forward port of 8080 on the local machine, first set the `CLASSPATH` to include all Axis jars and then issue the command

```
C:\ch11\ex1>java org.apache.axis.utils.tcpmon 8079 localhost 8080
```

This brings up a graphical window, ready to display connections and messages that come to port 8079. After display, all these connections and messages will be forwarded to port 8080. The next step is to change the port number in service endpoint address from 8080 to 8079 in `EchoClient.java`, compile it and run it in a separate command window. You should see something like *Figure 11-3*. Although it doesn't sh[illegible]om[illegible]AP request and response, you can figure out that the method argume[illegible] is passe[illegible]s a te[illegible] node within the SOAP `Body` element and so is the return value.



State	Time	Request Host	Target Host	Request...
---	Most Recent	---	---	---
Done	2003-05-28 00:32:48	127.0.0.1	localhost	GET /axis/services/StringEchoPort...
Done	2003-05-28 00:32:49	127.0.0.1	localhost	POST /axis/services/StringEchoPo...

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope
 <soapenv:Body>
  <ns1:echo soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encodi
   <arg0 xsi:type="xsd:string">Hi, How are you?</arg0>
  </ns1:echo>
 </soapenv:Body>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope
 <soapenv:Body>
  <ns1:echoResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soa
   <echoReturn xsi:type="xsd:string">Hi, How are you?</echoReturn>
  </ns1:echoResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Figure 11-3: Watching SOAP Request and Response with **`tcpmon`**

This tool comes quite handy for analyzing what happens under the hood!

Servlet Security For Web Services

Irrespective of what API a Web service client uses, it eventually creates a SOAP message and posts it, using HTTP POST, to the service address URL. This message is picked up by the Tomcat Web container and delivered to the Axis servlet. Axis, after doing its own processing and conversions, invokes the appropriate service implementation code. So, in its guts, interaction between a client program and Web service is not very different from the way a Web browser interacts with a Servlet-based Web application deployed within a Web container.

So you should not surprised to learn that it possible to make use of Servlet security mechanisms, as explained in the chapter *Web Application Security*, to authenticate the client to the server and control access to service address URLs, and hence the Web services themselves. We will look at the specifics of doing so in this section.

As we have seen, service address URLs for Web services deployed within Axis have the format: `http://`*`hostname`*`:`*`port`*`/axis/services/`*`servicename`*. By putting proper declarations in the Web application deployment descriptor for Axis, i.e., file `web.xml` in directory `%TOMCAT_HOME%\webapps\axis\WEB-INF`, we can specify URL patterns that require user login. Shown below are the declarative statements to allow only Tomcat users with the role `“StringEchoPort1UserRole”` to access Web service `StringEchoPort1` of our previous example.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Web service StringEchoPort1</web-resource-name>
    <url-pattern>/services/StringEchoPort1</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>StringEchoPort1UserRole</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Axis Basic Authentication Area</realm-name>
</login-config>

<security-role>
  <role-name>StringEchoPort1UserRole</role-name>
</security-role>
```

You can see that the deployment descriptor specifies HTTP-Basic authentication through the `auth-method` sub-element of the `login-config` element. Although HTTP-Basic is used here, you could use HTTP-Digest as well (provided the Web container supports it). FORM-based authentication, because it relies on showing a login page through the Web browser, is not well suited for a program client and hence is not advised. Client Certificate-based authentication is also a possibility. We will talk about it later in the section *SSL Security for Web Services*.

To apply access control to `StringEchoPort1` Web services, insert these declarations within the `web-app` element of the Axis `web.xml` file at the appropriate location and setup Tomcat user database with a role named `StringEchoPort1UserRole` and assign this role to users who you want to access the service. Refer to the chapter *Web Application Security* details on how to carry out these steps.

After you have modified the Axis `web.xml` and have setup a user with `StringEchoPort1UserRole` role, try to access the service WSDL through a Web browser, either by directly entering the WSDL URL or by clicking the View link on the Axis Welcome Page and following the link for `StringEchoPort1` service WSDL. The Web browser should throw up a login panel and demand a username and password. Once you supply them, the Web browser should display the WSDL document.

With a Web browser, we could simply enter the username and password through a UI element, but how are we going to specify these values from within a client program? The JAX-RPC specification defines two properties, `Call.USERNAME_PROPERTY` and `Call.PASSWORD_PROPERTY`, which can be set in a `Call` object. If these properties are set, JAX-RPC client runtime system then takes care of HTTP protocol-level details to specify proper HTTP headers for authentication. The following code fragment shows how the client program `EchoClient.java` has to be modified to use username and password. The complete working program is in source file `EchoClient.java` within the same directory.

```
String wsdlAddr = "file:test.wsdl";
// ... skip
Call call = (Call) svc.createCall();
// ... skip
// arg: String variable initialized with string to be sent
// username: String variable initialized with username
// password: String variable initialized with password
call.setProperty(Call.USERNAME_PROPERTY, username);
call.setProperty(Call.PASSWORD_PROPERTY, password);

String res = (String) call.invoke(new Object[] {arg});
```

Did you notice that besides setting the `Call` properties, we have also changed the initialization value for `wsdlAddr` variable to a URL pointing to a local file? This is required because the `createService()` method of `ServiceFactory` (refer to source file `EchoClient.java` shown in *Listing 11-3*) attempts to retrieve the WSDL document from the specified URL. If you specify the URL served by the Axis, then the retrieval will fail because there is no way to specify username and password for this access. This appears to be a limitation of using Servlet-based security and the way WSDL URL is created by Axis. As a work-around, you can retrieve the WSDL through some other means and store it in a local file. For example, you can get the WSDL document through your Web browser and save it in a file. This is what we have done.

Can the service program access the username supplied by the client program? We know from our discussion in the chapter *Web Application Security* that class `HttpServletRequest` has the methods `getRemoteUser()` and `getUserPrincipal()` to retrieve user information within a Servlet-based Web application. So, essentially what we need is the ability to access the HttpServletRequest instance within a service class. It turns out that this is possible, at least with Axis. This technique is illustrated in the source code of class DisplayUserInfo.java.

```
// File: src\jsbook\ch11\ex1\DisplayUserInfo.java
import org.apache.axis.MessageContext;
import org.apache.axis.transport.http.HTTPConstants;
import javax.servlet.http.HttpServletRequest;

public class DisplayUserInfo {
  public static void display() {
    MessageContext context = MessageContext.getCurrentContext();
    HttpServletRequest req = (HttpServletRequest)
        context.getProperty(HTTPConstants.MC_HTTP_SERVLETREQUEST);
    System.out.println("remote user = " + req.getRemoteUser());
    System.out.println("remote principal = " + req.getUserPrincipal());
  }
}
```

Note that the method display() relies on the static method getCurrentContext() of org.apache.axis.MessageContext class to get the MessageContext instance. Armed with this, it gets hold of the HttpServletRequest instance corresponding to the service request by getting the property value of the Axis-specific property HTTPConstants.MC_HTTP_SERVLETReQUEST. Once you have the HttpServletRequest instance, getting the remote user name is straightforward. You can invoke the static method display() of DisplayUserInfo within the body of any service class implementation. But keep in mind that this code will work only with Axis.

Though we have illustrated the use of Servlet-based security for client authentication to a Web service with the Axis and Tomcat, this technique is fairly general and will apply to all Web services that are deployed within a Web container and are HTTP accessible.

SSL SECURITY FOR WEB SERVICES

JAX-RPC doesn't mandate the support for HTTPS. However it is possible to configure the Tomcat to accept only HTTPS connections in the same way as for a Web application. It is also possible to configure mandatory client authentication through the client certificate, resulting in mutual authentication. We have already described the required configuration details in the chapter *Web Application Security* and hence will not repeat them here. Instead, we will go through the steps in configuring and running the previous example to use HTTPS.

Web service client programs can use HTTPS by simply using address URLs with scheme https in place of http, to access the service and setting appropriate system properties. The relevant system properties for Sun's implementation of J2SE v1.4.x are described in the chapter *Securing the Wire*.

Let us go through the steps in running the example service `StringEchoPort1` and the client so that SOAP messages are exchanged over an HTTPS connection with mutual authentication. For this purpose, we will create self-signed certificates for both the client program and the Tomcat server. These certificates and the corresponding private keys will be stored in respective keystore files. Then we will populate the client's truststore with the server's certificate and server's truststore with the client's certificate. As the main ideas behind these steps have already been covered in previous chapters, we will skip the explanations and simply show the steps with the relevant commands and configuration changes.

Step 1: Create keystore and truststore for service and client with self-signed certificates. This step is required only to make the example self-contained. In practice, you will be using existing certificates and keystore and truststore files.

The commands to create self-signed certificates with a Windows script file are shown below.

```
set SERVER_DN="CN=localhost, OU=X, O=Y, L=Z, S=XY, C=YZ"
set CLIENT_DN="CN=Client, OU=X, O=Y, L=Z, S=XY, C=YZ"
set KSDEFAULTS=-storepass changeit -storetype JCEKS
set KEYINFO=-keyalg RSA

keytool -genkey -dname %SERVER_DN% %KSDEFAULTS% -keystore server.ks\
%KEYINFO% -keypass changeit
keytool -export -file temp$.cer %KSDEFAULTS% -keystore server.ks
keytool -import -file temp$.cer %KSDEFAULTS% -keystore client.ts –alias\
serverkey -noprompt

keytool -genkey -dname %CLIENT_DN% %KSDEFAULTS% -keystore client.ks\
%KEYINFO% -keypass changeit
keytool -export -file temp$.cer %KSDEFAULTS% -keystore client.ks
keytool -import -file temp$.cer %KSDEFAULTS% -keystore server.ts\
-alias clientkey -noprompt
```

The complete script is in the `setup.bat` file under `src\jsbook\ch11\ex1` directory. After running this script, you have the server private key and certificate in the server's keystore `server.ks`, the client private key and certificate in the client's keystore `client.ks`, the server certificate in the client's truststore `client.ts` and the client certificate in the server's truststore `server.ts`.

Note that we have used JCEKS (**Java Cryptographic Extension Key Store**) as the type of the keystore. This must be specified as the keystore type whenever we access these keystore files.

Step 2: Copy the server keystore and truststore files in the Tomcat home directory. Strictly speaking, the keystores need not be in Tomcat home directory but then you will have to specify the exact path in the configuration described in the next two steps.

Step 3: Modify the Tomcat configuration file `server.xml` as shown below. This file can be found in `%TOMCAT_HOME%\conf` directory.

```
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
    port="8443" minProcessors="5" maxProcessors="75"
    enableLookups="true"
    acceptCount="100" debug="0" scheme="https" secure="true"
    useURIValidationHack="false" disableUploadTimeout="true">
  <Factory
      className="org.apache.coyote.tomcat4.CoyoteServerSocketFactory"
      protocol="TLS"
      clientAuth="true"
      keystoreFile="server.ks" keystoreType="JCEKS"
      truststoreFile="server.ts" truststoreType="JCEKS"
      keystorePass="changeit"
  />
</Connector>
```

Step 4: Run Tomcat with system properties set for server truststore. To do this, go to the Tomcat home directory and issue the following commands.

```
C:\...-jdk14>set TS_PROP=-Djavax.net.ssl.trustStore=server.ts
C:\...-jdk14>set TSTYPE_PROP=-Djavax.net.ssl.trustStoreType=JCEKS
C:\...-jdk14>set CATALINA_OPTS=%TS_PROP% %TSTYPE_PROP%
C:\...-jdk14>bin\startup
```

The prompt has been shortened to fit each command within a line.

Step 5: Modify the client program `EchoClient.java` to use `https://` URL and compile it.

```
//String epAddr = "http://localhost:8080/axis/services/StringEchoPort1";
String epAddr = "https://localhost:8443/axis/services/StringEchoPort1";
String wsdlAddr = epAddr + "?wsdl";
```

The modified source code is available in `EchoClient2.java` file under the example source directory.

Step 6: Run the client program. This involves specifying the system properties for SSL-specific parameters.

```
C:\ch11\ex1>java -Djavax.net.ssl.keyStore=client.ks \
-Djavax.net.ssl.keyStoreType=JCEKS \
-Djavax.net.ssl.keyStorePassword=changeit \
-Djavax.net.ssl.trustStore=client.ts \
-Djavax.net.ssl.trustStoreType=JCEKS EchoClient
```

A point worth noting is that we resorted to changing the URL in the client program. For Web applications, one could simply rely on making the appropriate changes in deployment descriptor file `web.xml` and the Web container would redirect requests for SSL-protected URLs to corresponding HTTPS URLs. One could do this for Web services as well and the Web container will faithfully issue HTTP redirect messages. However, the client library of Axis-1.1RC2 implementing HTTP is not capable of handling HTTP redirects and fails.

This makes it hard to protect only certain services within a Web container with HTTPS and let others be accessed with plain HTTP. You must have all services deployed within a particular Web container accepting HTTPS connection or none. It is also not possible to have separate Web service-specific server certificates.

WS SECURITY

As mentioned earlier, the WS Security specification and an addendum to it were initially published by IBM, Microsoft and VeriSign and have now been submitted to OASIS for further development as a standard. At the time of finalizing this chapter (April 2003), the OASIS Technical Committee has not published the final specification. So our discussion here is going to be based on the original proposed specification and the addendum to it. It goes without saying that there are no standard Java APIs for it.

Why bother covering WS Security if it is yet not a standard and there are no standard Java APIs for it? The reason has to do with its significance to Web services security. Transport-level security is just not adequate for a number of Web services-based applications and there is no other credible alternative standard for message-level security. A number of implementation of the current WS Security specification already are in existence and early adopters are either piloting or even using WS Security in production systems as it is.

The aim of WS Security specification is to specify SOAP `Header` elements for quality of protection of a single SOAP message or certain parts of it through message integrity, confidentiality and authentication. Towards this, it allows various types of *security tokens*, or collection of client-made statements, to be embedded within a SOAP message. A security token could be signed, such as X.509 certificate or a Kerberos ticket. It also allows SOAP message elements to be signed, encrypted or signed and encrypted using XML Signature and XML Encryption standards. This is how the goal of message integrity, confidentiality and authentication are met.

WS Security itself is not a security protocol involving exchange of multiple messages between two communicating parties, although it does allow development of such protocols. Being message-oriented, it enables end-to-end security, even in the presence of intermediate gateways that might do transport protocol conversion and/or need to examine certain portions of the message.

We will not present a detailed discussion of the WS Security specification but instead focus on using a WS Security library built on top of VeriSign's TSIK, the same toolkit that we used for XML-Signature and XML-Encryption in the chapter *Securing The Message*. Executing the programs written using this library would give us an opportunity to examine the output and understand the structure of messages protected with WS Security.

VeriSign's WS Security library can be downloaded from the same location where we got TSIK, i.e., http://www.xmltrustcenter.org. The download consists of a single jar file, `wssecurity.jar` and the corresponding source file `WSSecurity.java`.

If you are keen on knowing more about WS Security, you should read the specification documents and go through the source file `WSSecurity.java`. As you do so, you will find that `WSSecurity.java` doesn't implement all options of the WS Security specification. In line with TSIK philosophy, WS Security implementation is designed for ease-of-use than the completeness of features.

Our objective in this section is to learn about WS Security specification and the VeriSign's API to perform apply it on SOAP messages. We will do so by looking at signature creation and verification programs written using VeriSign’s `WSSecurity` class. These programs operate on SOAP messages store in a file.

The signature creation program, `WSSSign.java`, is shown in *Listing 11-4*, that takes a file with SOAP message as input and produces another file with WS Security compliant SOAP message as output. The output SOAP message will have the SOAP body signed and the signature element and the verification key information placed in the SOAP header. We will also write a verification program, `WSSVerify.java`, to verify the signed SOAP message. These source files can be found in `src\jsbook\ch11\wss` directory.

```
// File: src\jsbook\ch11\wss\WSSSign.java
public class WSSSign {
  public static void main(String[] args) throws Exception {
    String datafile = args[0];
    outfile = args[1];

    String keystore = "my.keystore";
    String storepass = "changeit";
    String kstype = "JCEKS";
    String alias = "mykey";

    System.out.println("Signing XML data in file \"" + datafile + "\"");
    System.out.println("Using private key in keystore \"" +
            keystore + "\" ...");

    java.io.FileInputStream fis = new java.io.FileInputStream(keystore);
    java.security.KeyStore ks =
            java.security.KeyStore.getInstance(kstype);
    ks.load(fis, storepass.toCharArray());
    java.security.PrivateKey key = (java.security.PrivateKey)
            ks.getKey(alias, storepass.toCharArray());
    java.security.cert.X509Certificate cert =
          (java.security.cert.X509Certificate)ks.getCertificate(alias);

    org.w3c.dom.Document doc = XmlUtility.readXML(datafile);
    com.verisign.xmlsig.SigningKey sk =
            com.verisign.xmlsig.SigningKeyFactory.makeSigningKey(key);
    com.verisign.xmlsig.KeyInfo ki = new com.verisign.xmlsig.KeyInfo();
    ki.setCertificate(cert);

    com.verisign.messaging.WSSecurity wss =
            new com.verisign.messaging.WSSecurity();
    wss.sign(doc, sk, ki);

    XmlUtility.writeXML(doc, new java.io.FileOutputStream(outfile));
    System.out.println("... Wrote the output to file: \"" +
            outfile + "\"");
  }
}
```

Lising 11-4: Source File `WSSSign.java`

A bit of explanation is required for this program. First of all, note the use of `XmlUtility` class introduced in the chapter *Securing the Message* for reading and writing XML documents. Another point worth noting is the use of a Java keystore to retrieve a private key and the corresponding certificate. As you know, you need a private key for signing. The certificate is needed for embedding the verification key information in the signature through the `KeyInfo` object. The actual signing is done by the `sign()` method of the `WSSecurity` class. This method signs the SOAP body, and inserts the signature and other relevant information in SOAP Header. We will see a sample WS Security-compliant signed SOAP document shortly.

Compiling and executing this program requires both `tsik.jar` and `wssecurity.jar` to be in `CLASSPATH`. Once the program is compiled, let us run it with the following `soap.xml` input file:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:echo
   soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:ns1="http://www.pankaj-k.net/jsbook/examples/">
    <arg0 xsi:type="xsd:string">Hi, How are you?</arg0>
  </ns1:echo>
 </soapenv:Body>
</soapenv:Envelope>
```

Assuming that the `CLASSPATH` has `tsik.jar` and `wssecurity.jar`, and the keystore `my.keystore` is properly initialized with a private key and certificate, issue the following command to run `WSSSign` program.

```
C:\ch11\wss>java WSSSign soap.xml signed.xml
Signing XML data in file "soap.xml"
Using private key in keystore "my.keystore" ...
... Wrote the output to file: "signed.xml"
```

The signed SOAP message is saved in the `signed.xml` file. This file is reproduced in *Listing 11-5*. In this listing, text in different font-weight is used to group the logically related information.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Header>
   <wsse:Security
       xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext"
       soapenv:mustUnderstand="1">
     <wsse:BinarySecurityToken ValueType="wsse:X509v3"
         EncodingType="wsse:Base64Binary"
         Id="wsse-04554370-7798-11d7-9a53-3d469a48eb3e">
         ... base64 encoded binary data ...
     </wsse:BinarySecurityToken>
     <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
       <ds:SignedInfo>
         <ds:CanonicalizationMethod
```

```
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
            <ds:SignatureMethod
                Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
            <ds:Reference URI="#wsse-0433b1b0-7798-11d7-9a53-3d469a48eb3e">
              <ds:Transforms>
                <ds:Transform
                    Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
              </ds:Transforms>
              <ds:DigestMethod
                    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
              <ds:DigestValue>GvYITJtrR35QAvfi08qiTmWmP9A=</ds:DigestValue>
            </ds:Reference>
            <ds:Reference URI="#wsse-042d9730-7798-11d7-9a53-3d469a48eb3e">
              <ds:Transforms>
                <ds:Transform
                    Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
              </ds:Transforms>
              <ds:DigestMethod
                  Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
              <ds:DigestValue>WNZ9INu1/7kBC4alIDjcl7RBZ7Q=</ds:DigestValue>
            </ds:Reference>
          </ds:SignedInfo>
         <ds:SignatureValue>base64 encoded binary data</ds:SignatureValue>
          <ds:KeyInfo>
            <wsse:SecurityTokenReference>
              <wsse:Reference
                  URI="#wsse-04554370-7798-11d7-9a53-3d469a48eb3e"/>
            </wsse:SecurityTokenReference>
          </ds:KeyInfo>
        </ds:Signature>
      </wsse:Security>
      <wsu:Timestamp
          xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">
        <wsu:Created
            wsu:Id="wsse-042d9730-7798-11d7-9a53-3d469a48eb3e">
          2003-04-26T03:34:41Z
        </wsu:Created>
      </wsu:Timestamp>
    </soapenv:Header>
    <soapenv:Body
        wsu:Id="wsse-0433b1b0-7798-11d7-9a53-3d469a48eb3e"
        xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">
      <ns1:echo
          soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:ns1="http://www.pankaj-k.net/jsbook/examples/">
        <arg0 xsi:type="xsd:string">Hi, How are you?</arg0>
      </ns1:echo>
    </soapenv:Body>
  </soapenv:Envelope>
```

Listing 11-5: WS Security-compliant signed SOAP message

WS Security signing has introduced two SOAP Header elements: `wsse:Security` and `wsu:Timestamp`. The best way to visualize these elements, their children and their relationship is look at *Figure 11-4*.

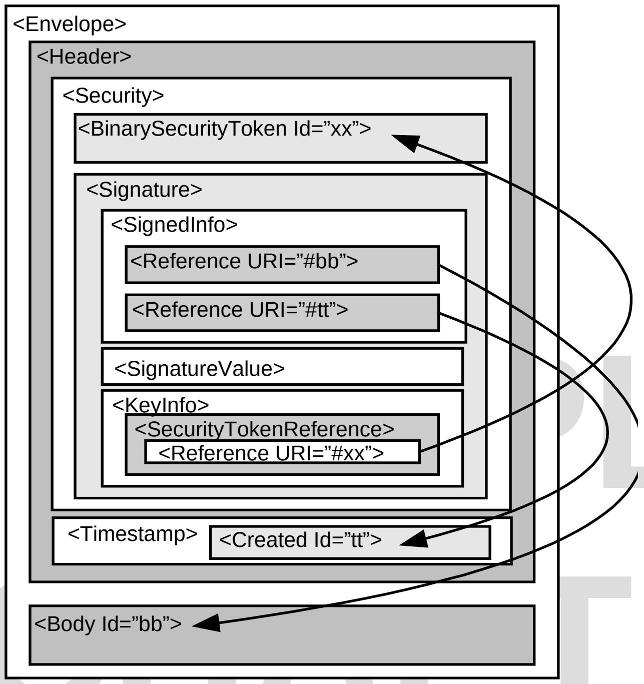


Figure 11-4: WS Security elements and their relationships

From this diagram, it is quite clear that the signature has been computed over two elements: the SOAP `Body` element and the `Timestamp` Header element inserted by WS Security. The tamper-evident Timestamp provides a limited amount of protection against replay attacks as the recipient can detect duplicates. Another point to note is that the verification key information is not part of `KeyInfo` element of XML Signature, but rather points to the WS Security element `BinarySecurityToken`.

The program to verify a WS Security signed SOAP message is in Java source file `WSSVerify.java`, as shown in *Listing 11-6*.

```
// File: src\jsbook\ch11\wss\WSSVerify.java
public class WSSVerify {
  public static void main(String[] args) throws Exception {
    String datafile = args[0];

    String keystore = "my.keystore";
    String storepass = "changeit";
    String kstype = "JCEKS";

    System.out.println("Verifying SOAP data in file \"" + datafile +
          "\" using trusted certs");
    System.out.println("in keystore \"" + keystore + "\" ...");
```

```
        java.io.FileInputStream fis = new java.io.FileInputStream(keystore);
        java.security.KeyStore ks =
                java.security.KeyStore.getInstance(kstype);
        ks.load(fis, storepass.toCharArray());

        org.w3c.dom.Document doc = XmlUtility.readXML(datafile);
        org.xmltrustcenter.verifier.TrustVerifier verifier =
                new org.xmltrustcenter.verifier.X509TrustVerifier(ks);

        com.verisign.messaging.WSSecurity wss =
                new com.verisign.messaging.WSSecurity();
        com.verisign.messaging.MessageValidity[] resa =
                wss.verify(doc, verifier, null);

        for (int i = 0; i < resa.length; i++){
          System.out.println("result[" + i + "] = " + resa[i].isValid());
        }
    }
}
```

Listing 11-6: Source File `WSSVerify.java`

This program uses the public key of the signer from the keystore for signature verification. As the certificate of the signer is included in the `wsse:Security` element, one could use the public key from this certificate as well. This is what happens if you specify `null` in place of `verifier` in the `verify()` method of `WSSecurity` class. However, if you want to trust only those certificates which are in your truststore or are signed by CAs whose certificates are in your truststore, then you should specify an appropriate `verifier`.

If you look into directory `src\jsbook\ch11\wss`, you will find programs to do WS Security-based encryption, decryption, signing and encryption and validation processing; where validation processing refers to signature verification for a signed document, decryption for an encrypted document, and decryption and signature verification for a signed and encrypted document. Look at their source code and play with these programs, analyzing their output the same way we analyzed the signed SOAP message. For running encryption or decryption programs, you will need a JCE provider that supports RSA encryption. We have already talked about one such provider, *Bouncy Castle JCE Provider*, and have used it in the chapter *Securing the Message*.

WS SECURITY WITH APACHE AXIS

How can we use WS Security with Apache Axis? As we saw, VeriSign's WS Security API works on SOAP messages whereas you usually don't work with SOAP messages while using Axis client library or writing a service. If you look at the `EchoClient.java` file, what you find is a Java `Object` as argument and a Java `Object` as return value. Likewise, at the service end, you work with the Java objects. We know that Axis libraries convert the

Java objects into SOAP messages at the transmitting end and SOAP messages into Java objects at the receiving end. As WS Security protects SOAP messages, we must have some way of accessing and modifying a SOAP message after the conversion at the transmitting end and before the conversion at the receiving end.

The JAX-RPC handler mechanism provides a solution. One or more handlers, forming a chain of handlers, can be specified to process outgoing and/or incoming messages at the client or the service. At the client, a handler chain must be specified programmatically by making appropriate API calls. At the service end, a handler chain can be specified through the deployment descriptor, at the time of deployment. We will talk more about both these forms of handler specification later, in the subsection *WS Security Example*.

The next subsection outlines how to write JAX-RPC-compliant handlers for *WS Security* using VeriSign's implementation. The subsequent subsection will use these handlers to augment our example client program `EchoClient` and the service `StringEchoPort1` with WS Security-based message protection. The complete source code of WS Security handlers is in the directory `src\jsbook\ch11\wss4axis` and the augmented example is in the directory `src\jsbook\ch11\ex2`. To make it possible to deploy the original as well as modified services simultaneously, we will call the modified service `StringEchoPort2` and the corresponding source file `StringEchoService2.java`. Different service names are required to keep them unique within a single instance of the Axis engine.

WS Security Handlers

The implementation class of a JAX-RPC handler must implement the `Handler` interface defined in the package `javax.xml.rpc.handler`. This interface has the methods `handleRequest()` which gets invoked for outgoing messages, `handleResponse()` which gets invoked for incoming messages and `handleFault()` which gets invoked when a SOAP Fault occurs. All of these methods take a `MessageContext` object as argument and can retrieve the `SOAPMessage` from it. Besides these methods, it also has the lifecycle methods `init()` to initialize the handler instance and `destroy()` to perform the cleanup.

This brief description gives us sufficient background to understand the source code in `WSServiceHandler.java`, the file defining the service side handler for WS Security processing. The source code for this handler is shown in *Listing 11-7*. As you can see, the handler assumes that it is configured with details of a keystore and truststore. The keystore has a key entry with the service's private key and certificate and the truststore has certificate entry with the client's certificate. The handler retrieves the configured parameters in its `init()` method, which gets invoked by Axis engine at the time of initializing the handler, and stores them in private member fields. The mechanism to specify these parameters and their values are different for client and service and illustrated in the subsection *WS Security Example*.

```
// File: wss4axis\src\org\jstk\wss4axis\WSServiceHandler.java
package org.jstk.wss4axis;
```

```
import javax.xml.rpc.handler.Handler;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.soap.SOAPMessage;
import org.w3c.dom.Document;
import java.util.Map;

public class WSSServiceHandler implements Handler {
   private String keyStoreFile, keyStoreType, keyStorePassword ,
        keyEntryAlias, keyEntryPassword, trustStoreFile,
       trustStoreType, trustStorePassword, certEntryAlias;

  public boolean handleRequest(MessageContext context) {
    try {
      SOAPMessageContext soapCtx = (SOAPMessageContext)context;
      SOAPMessage soapMsg = soapCtx.getMessage();
      Document doc = SOAPUtility.toDocument(soapMsg);

      WSSUtility.decrypt(doc, keyStoreFile, keyStoreType,
            keyStorePassword, keyEntryAlias, keyEntryPassword);
      WSSUtility.verify(doc, trustStoreFile, trustStoreType,
            trustStorePassword);
      WSSUtility.cleanup(doc);

      soapMsg = SOAPUtility.toSOAPMessage(doc);
      soapCtx.setMessage(soapMsg);
    } catch (Exception e){
        System.err.println("handleRequest -- Exception: " + e);
        return false;
    }
    return true;
  }

  public boolean handleResponse(MessageContext context) {
    try {
      SOAPMessageContext soapCtx = (SOAPMessageContext)context;
      SOAPMessage soapMsg = soapCtx.getMessage();
      Document doc = SOAPUtility.toDocument(soapMsg);

      WSSUtility.sign(doc, keyStoreFile, keyStoreType,
          keyStorePassword, keyEntryAlias, keyEntryPassword);
      WSSUtility.encrypt(doc, trustStoreFile, trustStoreType,
          trustStorePassword, certEntryAlias);

      soapMsg = SOAPUtility.toSOAPMessage(doc);
      soapCtx.setMessage(soapMsg);
    } catch (Exception e){
      System.err.println("handleResponse -- Exception: " + e);
      return false;
    }
    return true;
  }

  public boolean handleFault(MessageContext context) {
    return true;
  }
```

```
    public void init(HandlerInfo config) {
      Map configProps = config.getHandlerConfig();
      keyStoreFile = (String)configProps.get("keyStoreFile");
      keyStoreType = (String) configProps.get("keyStoreType");
      keyStorePassword = (String) configProps.get("keyStorePassword");
      keyEntryAlias = (String) configProps.get("keyEntryAlias");
      keyEntryPassword = (String) configProps.get("keyEntryPassword");
      trustStoreFile = (String)configProps.get("trustStoreFile");
      trustStoreType = (String) configProps.get("trustStoreType");
      trustStorePassword = (String) configProps.get("trustStorePassword");
      certEntryAlias = (String) configProps.get("certEntryAlias");
    }
  }
```

Listing 11-7: JAX-RPC handler to process request and response at a Web service

This handler decrypts, verifies and cleans up (i.e., removes the Header elements) the incoming request SOAP message in the `handleRequest()` method and encrypts and signs the outgoing response SOAP message in the `handleResponse()` method making use of the utility class `WSSUtility.` This utility class is a simple wrapper over VeriSign's WSSecurity library. The code for each operation in this class, such as `sign()`, `encrypt()`, `verify()`, `decrypt()` etc., is very similar to the one we saw in `WSSSign.java` file, shown in *Listing 11-4*, and hence is not shown here. You can find source file `WSSUtility.java` along with other source files in the `src\jsbook\ch11\wss4axis` subdirectory.

There is one more aspect of this program that needs some discussion. As you must have noticed, what you get in a handler method is a `javax.xml.soap.SOAPMessage` object and not an `org.w3c.dom.Document` object. However, WSSecurity library expects a W3C DOM `Document` object as input. Although both classes represent an XML document (a SOAP message *is* an XML document), they have their own internal structure and cannot be simply converted from one to another by a typecast. We have delegated this task of conversion to utility class `SOAPUtility`. The source code of this utility class is not shown here but can be found in the JSTK distribution along with other files referenced in the chapter. This class achieves conversion by serializing the input object into an in-memory byte stream and recreating the desired output object. This way of doing the conversion is quite expensive and can have significant performance impact, especially for large documents.

Moreover, there seems to be no easy way to avoid this performance hit. Essentially, there is an impedance mismatch between what JAX-RPC API provides and what WSSecurity library expects. A epecially written WS Security library that works efficiently for `SOAPMessage` class could be an option[2].

The client side handler class `WSSClientHandler` is very similar, performing the signing and encryption in `handleRequest()` and decryption, verification and SOAP header cleanup in `handleResponse()`.

[2] This problem has been solved by SAAJ 1.2, a maintenance release of SAAJ made available in April 2003, by adding `org.w3c.dom.Node` as one of the base interfaces to `javax.xml.soap.SOAPMessage`.

The `wss4axis` directory, within `src\jsbook\ch11` directory, includes scripts to compile the source files and create a jar file – `wss4axis.jar,` with all the handler and supporting class files. We will use this jar in our next example.

WS Security Example

To make use of WS Security in our previous example, we need to do three things:

1. Generate keys and certificates for client and service and store them in respective keystore and truststore files.
2. Modify the client program to setup the client handler and initialize it with client keystore and truststore details.
3. Modify the service deployment descriptor to specify the service handler and initialize it with service keystore and truststore details.

For the first step, we will use the keystore and truststore files `client.ks`, `client.ts`, `server.ks` and `server.ts`, generated in the section *SSL Security for Web Services*.

For the second step, let us modify `EchoClient.java` as shown below. The bold statements indicate additions to the original `EchoClient.java` program of *Listing 11-3*.

```
Service svc = svcFactory.createService(wsdlUrl, svcQName);

Java.util.HashMap cfg = new java.util.HashMap();
cfg.put("keyStoreFile", "client.ks");
cfg.put("trustStoreFile", "client.ts");
cfg.put("certEntryAlias", "serverkey");

Class hdlrClass = org.jstk.wss4axis.WSSClientHandler.class;
java.util.List list = svc.getHandlerRegistry().
                 getHandlerChain(new QName(nameSpaceUri, portName));
list.add(new javax.xml.rpc.handler.HandlerInfo(hdlrClass, cfg, null));

Call call = (Call) svc.createCall();
```

The new statements initialize a `HashMap` with name value pairs, get the handler chain associated with the `Service` object, create a `HandlerInfo` initialized with `WSSClientHandler` class and the `HashMap` object and add this `HandlerInfo` to the handler chain. The Axis library will create a `WSSClientHandler` object and invoke `init()` with `HandlerInfo` as argument, letting the handler initialize itself.

The third step is to modify the deployment descriptor for the service. Let us look at the modified deployment descriptor file `deploy.wsdd`, with new declarations shown in bold.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
```

```
      <service name="StringEchoPort2" provider="java:RPC">
       <parameter    name="wsdlTargetNamespace"    value="http://www.pankaj-
k.net/jsbook/examples/"/>
       <parameter name="wsdlServiceElement" value="StringEchoService2"/>
       <parameter name="wsdlServicePort" value="StringEchoPort2"/>
       <parameter name="wsdlPortType" value="StringEcho"/>
       <parameter name="scope" value="session"/>
       <parameter name="className" value="StringEchoService2"/>
       <parameter name="allowedMethods" value="*"/>
       <requestFlow>
        <handler type="java:org.apache.axis.handlers.JAXRPCHandler">
         <parameter name="scope" value="session"/>
         <parameter name="className"
                 value="org.jstk.wss4axis.WSSServiceHandler"/>
         <parameter name="keyStoreFile"
                 value="c:\\jstk\\src\\jsbook\\ch11\\ex2\\server.ks"/>
         <parameter name="trustStoreFile"
                 value="c:\\jstk\\src\\jsbook\\ch11\\ex2\\server.ts"/>
         <parameter name="certEntryAlias" value="clientkey"/>
        </handler>
       </requestFlow>
       <responseFlow>
        <handler type="java:org.apache.axis.handlers.JAXRPCHandler">
         <parameter name="scope" value="session"/>
         <parameter name="className"
                 value="org.jstk.wss4axis.WSSServiceHandler"/>
         <parameter name="keyStoreFile"
                 value="c:\\jstk\\src\\jsbook\\ch11\\ex2\\server.ks"/>
         <parameter name="trustStoreFile"
                 value="c:\\jstk\\src\\jsbook\\ch11\\ex2\\server.ts"/>
         <parameter name="certEntryAlias" value="clientkey"/>
        </handler>
       </responseFlow>
      </service>

     </deployment>
```

You may find it a bit odd that the same parameter names and values need to be specified twice within the deployment descriptor. This is so because Axis allows separate handlers for request and response path. The original Axis handler mechanism, with separate handler classes for request and response, was designed and implemented before JAX-RPC specification was developed. Later on, the JAX-RPC API was added to the existing design.

To deploy the service and run the client program, follow the same sequence of steps as in the previous example. One thing to remember is that before you run the client program, you must copy `tsik.jar`, `wssecurity.jar` and `wss4axis.jar` to the `lib` directory of Axis deployment and make sure that a JCE Provider with RSA encryption is properly installed in your J2SE setup.

Figure 11-5 shows different components of this example.

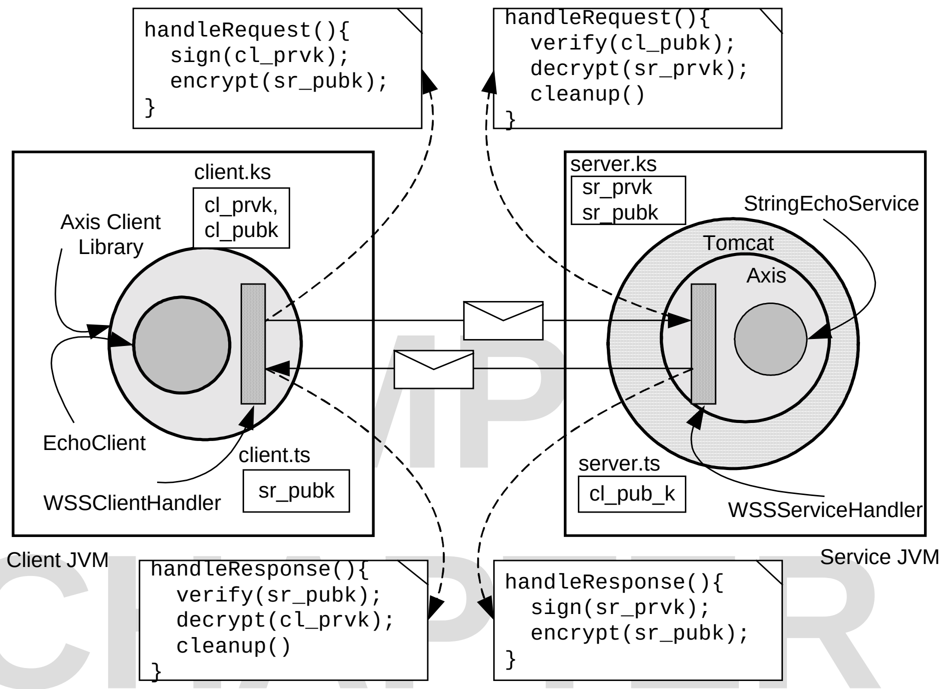


Figure 11-5: WS Security example setup

At a high level, we have essentially defined a simple application level message based protocol where the client sends a SOAP message with signed and encrypted Body. The service decrypts and verifies the message, performs the processing and sends back the response SOAP message with signed and encrypted Body. The client decrypts the messages and verifies it. Both client and service have their own private keys that they use for signing and decryption. They also have each other's public key that they use for encryption.

Note that the handlers retrieve the public key of the recipient from the truststore for encryption based on static configuration. This means that these handlers won't work if an endpoint wants to communicate with more than one party with message signing and encryption using different keys. Also, we have used the same private key for signing as well as decryption, something not recommended for high security systems.

One advantage of Web services is that all the interaction takes place by exchanging well-defined XML messages and it is possible to intercept and process these messages en-route. Security-related processing is an ideal candidate for such interception at enterprise perimeter and centralized processing, and there are commercial products that perform this kind of processing. VeriSign's XML Trust Gateway, based on TSIK library, is an example of such a commercial product.

SUMMARY

A Web service describes its interface, or collection of operations, in a WSDL document and interacts with other entities by exchanging SOAP messages over HTTP/S. Web services utilize the simplicity, extensibility and flexibility of XML and Web protocols for program to program communication over the Internet. Security requirements of Web services could be more complex than simple transport security commonly used for Web applications.

Fundamental security issues for Web services are the same as any other distributed programming technology – authentication, authorizations, confidentiality and message integrity. An important distinction is that Web services can be invoked either synchronously, using request-response paradigm or asynchronously, using document exchange paradigm.

Transport-level security can be used for a certain class of Web services. This is most appropriate when Web services are used as interoperable, platform-independent RPC infrastructure where both the client and the service communicate over a transport level connection. HTTPS provides good security in these cases and can be setup in the same way as for Web applications.

WS Security specification defines elements to provide message-level authentication, confidentiality and integrity for SOAP messages. Message-based security is more appropriate than the transport-based security in a number of Web service use scenarios. The JAX-RPC handler mechanism provides a convenient mechanism to incorporate WS Security-based message security without changing the client or service code significantly.

FURTHER READING

Though a number of books have come out explaining SOAP and WSDL, the authoritative source continues to be the official specification documents themselves. You can find the specification documents for SOAP 1.1 at http://www.w3.org/TR/SOAP/ and WSDL 1.1 at http://www.w3.org/TR/wsdl. It is quite likely that SOAP Version 1.2 will become a W3C recommended standard by the time this book reaches you. In this version, the specification is broken into three documents – Primer, Messaging Framework and Adjuncts. The Primer gives a nice overview of SOAP and is worth reading. WSDL Version 1.2 is also likely to reach candidate recommendation status soon.

The definitive source of information for JAX-RPC is the official specification and Javadoc documentation. You can find both of these by following the appropriate links from http://java.sun.com/xml/jaxrpc/. Sun’s Java site has a tutorial on Java Web Services with a chapter on JAX-RPC. This tutorial has good information, though it uses Sun’s JAX-RPC implementation, and not Axis, for examples. For Axis specific information, refer to the Axis documentation.