Web Services using Axis

Barry Cornelius
Computing Services, University of Oxford
Date: 6th May 2003; first created: 17th April 2003
http://users.ox.ac.uk/~barry/papers/
mailto:barry.cornelius@oucs.ox.ac.uk

1 Introduction

A Web Service is the provision of one or more methods that can be invoked by some external site. The external site contacts the webserver of the site providing the Web Service to get a method of the Web Service executed. It is usually a program running on the external site that wants the result of a call of the method. A website might be providing Web Services in several different areas.

Although some people assume that Web Services requires you to use Microsoft's .NET Framework, there are many rival products. In this document, we will look at Axis, a Web Services product from Apache. This product is targetted at Java programmers.

The main aim of this document is to demonstrate how easy it is for any Java programmer:

- to provide Web Services;
- to run Java programs that access Web Services provided on computers around the world.

Besides demonstrating this, the document will also look at some other topics:

- In order to provide Axis Web Services, it is necessary to have a webserver program capable of running servlets. So we will look at the installation and use of a program called `tomcat`.
- In order to communicate with a Web Service, it is necessary to use two XML languages: WSDL and SOAP. We will briefly look at these languages.
- As has already been mentioned, Microsoft's .NET Framework can be used for Web Services. We will look at whether it is easy to interoperate Axis and .NET for Web Services.

Although the instructions given in this document assume you are using Windows, Linux could be used instead. The document also assumes that the Java 2 SDK has been installed.

2 What is a Web Service?

2.1 A definition

A Web Service is the provision of one or more methods that can be invoked by some external site. The external site contacts the webserver of the site providing the Web Service to get a method of the Web Service executed. It is usually a program running on the external site that wants the result of a call of the method. A website might be providing Web Services in several different areas.

2.2 An example

In their book [44] about the .NET Framework, Thai and Lam provide the following answer to the question *How can software be viewed as services?*:

- ‘The example we are about to describe might seem far-fetched; however, it is possible with current technology. Imagine the following. As you grow more attached to the Internet, you might choose to replace your computer at home with something like an Internet Device, specially designed for use with the Internet. Let’s call it an `iDev`. With this device, you can be on the Internet immediately. If you want to do word processing, you can point your `iDev` to a Microsoft Word service somewhere in Redmond and type away without the need to install word processing software. When you are done, the document can be saved at an `iStore` server where you can later retrieve it. Notice that for you to do this, the `iStore` server must host a software service to allow you to store documents. Microsoft would charge you a service fee based on the amount of time your word processor is running and which features you use (such as the grammar and spell checkers). The `iStore` service charges vary based on the size of your document and how long it is stored. Of course, all these charges won’t come in the mail, but rather through an escrow service where the money can be piped from and to your bank account or credit card.’
- ‘All of these things can be done today with Web Services.’

3 Servlets

3.1 Dynamically generated WWW pages

Although the vast majority of WWW pages are static, increasingly these days authors (of WWW pages) want their webserver to be able to generate WWW pages dynamically. It may be that the author has used a WWW form to collect some data from the visitor to the WWW page and the WWW page the author wants to return depends on that data.

For example, the WWW form might contain a textbox in which the visitor types the name of a town and when they click on the form’s *Submit* button somehow or other a database gets queried, and the WWW page that gets returned consists of the information about the town that is contained in the database.

One way of generating information dynamically is to get the webserver to execute some code. The possibilities for doing this depend on the webserver program being used. They include a CGI script, a PHP script, an Active Server Page, some code written in Java (a *servlet*), a JavaServer page, and so on.

3.2 What is a servlet and how is it executed?

A servlet is a WWW page that is written in Java. When a visitor’s browser (such as Netscape Navigator or Internet Explorer) accesses the URL of a servlet, the webserver program arranges for the bytecodes of the servlet to be executed. Usually, instead of starting a new process to execute the bytecodes, the webserver program just executes them in a new thread. Of course, you need to be using a webserver program that can execute bytecodes. One possibility is to use tomcat, a webserver program from the Apache Software Foundation.

The code of the servlet does some work (such as querying a database) and then generates some HTML. The webserver program gets the HTML from the servlet and passes that to the visitor’s browser.

3.3 A digression: differences between servlets and applets

It is important not to confuse servlets with applets. They can easily be confused as both are concerned with the execution of Java bytecodes. Whereas the bytecodes of a servlet are executed by the webserver program running on the webserver at the website, the bytecodes of an applet are executed by the visitor’s browser on his/her computer.

Ever since Java first appeared, it has been possible to write a WWW page that refers to a Java applet. When a visitor visits such a WWW page, the browser downloads the bytecodes of the applet and then it (or a plug-in) executes the bytecodes.

Servlets are a more recent innovation. As previously mentioned, the bytecodes of a servlet are executed by the webserver program running at the website. When the bytecodes are executed, they produce some HTML and this is delivered to the visitor’s browser. The browser has no clue that the HTML received from the webserver was the result of the execution of some bytecodes.

3.4 An example of a servlet

For a simple example of a servlet, suppose we want a WWW page that delivers the current date and time. In Java, we can easily use `java.util.Date` to output the current date and time:

```
Date tDate = new Date();
String tDateString = tDate.toString();
System.out.println(tDateString);
```

How can we convert this code to code that can be executed as a servlet? One way is to produce a class that is derived from the `HttpServlet` class (that is in the `javax.servlet.http` package). If our class overrides `HttpServlet`'s `doGet` method, the webserver program will arrange for our `doGet` method to be executed when someone visits the appropriate WWW page.

Here is the code of a `getDateAndTime` servlet:

```
import java.util.          Date;                      // getDateAndTime.java
import javax.servlet.http. HttpServlet;
import javax.servlet.http. HttpServletRequest;
import javax.servlet.http. HttpServletResponse;
import java.io.            IOException;
import java.io.            PrintWriter;
import javax.servlet.      ServletException;
public class getDateAndTime extends HttpServlet
{
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
         throws IOException, ServletException
   {
      Date tDate = new Date();
      String tDateString = tDate.toString();
      response.setContentType("text/html");
      PrintWriter tResponsePrintWriter = response.getWriter();
      StringBuffer tStringBuffer = new StringBuffer();
      tStringBuffer.append("<html>\n" );
      tStringBuffer.append("<title>Clock</title>\n" );
      tStringBuffer.append("It is now " + tDateString + "\n" );
      tStringBuffer.append("</html>\n" );
      tResponsePrintWriter.println(tStringBuffer);
      tResponsePrintWriter.close();
   }
}
```

When the `doGet` method gets called, it is passed two arguments (`request` and `response`):

- As shown above, the second argument, `response`, provides the means of passing HTML back to the webserver program, i.e., a way of dynamically generating a WWW page.
- As we will see later, the `request` argument enables information to be passed from the person visiting the WWW page to the servlet.

4 Tomcat

4.1 What is tomcat?

In order to get a servlet executed, you need a webserver program that is capable of executing Java bytecodes. One possibility is to use `tomcat`, a webserver program that is provided by the Apache Software Foundation. In a recent survey by Netcraft of IP addresses running servlet engines [33], tomcat, Resin and IBM are equally popular and together have 75% of the market.

Although tomcat is great for running servlets, it is not a very efficient webserver. Another possibility is to use Apache's HTTP Server program (`httpd`) as a webserver program with it configured to run tomcat in order to execute servlets.

However, here we will keep things simple: we will just use tomcat on its own.

4.2 Installing tomcat

Later, we will be installing Axis. The Axis Installation Guide at:

```
http://cvs.apache.org/viewcvs.cgi/~checkout~/xml-axis/java/docs/install.html
```

says that you should get the latest 4.1.x version of tomcat. It also says that you should get the *full distribution* rather than the *LE version* as the LE version omits the Xerces XML parser. The Axis Installation Guide does not cover the installation of tomcat. However, there are installation notes for tomcat at:

```
http://jakarta.apache.org/tomcat/tomcat-4.1-doc/RUNNING.txt
```

These installation notes refer to downloading a binary distribution of tomcat from:

```
http://jakarta.apache.org/builds/jakarta-tomcat-4.0/nightly/
```

However, this directory is empty. So, instead go to:

```
http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/
```

and look for a subdirectory containing the latest 4.1.x release. For example, I downloaded from:

```
http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.1.18/bin/
```

The notes advise you to download a `.zip` file, a file with a name like `jakarta-tomcat-4.1.18.zip`. This file can be unzipped (perhaps using the `jar` command). I think it only writes into the directory that you choose.

However, although it is not mentioned in the installation notes, for Windows you can instead download a self-installation binary. This is in a file with a name like `jakarta-tomcat-4.1.18.exe`. This file was 8.45MB. Using a modem on a 33Kbps line, this took about 40 minutes to download.

After this `.exe` file has been downloaded, double click on `jakarta-tomcat-4.1.18.exe` whilst using Windows Explorer. An `Apache Tomcat 4.1.18 Setup` screen appears. It will first look for a Java 2 SDK directory. It found mine in:

```
E:\j2sdk1.4.1_02
```

Click on `OK`. A License Agreement form appears. Read it and click on `I Agree`. An Installation Options screen appears. The default is a `Normal` installation which it said requires 28.8MB of disc space. I took this and clicked on `Next`. It then wants to know where you want the files to be put. For me it defaulted to:

```
E:\Program Files\Apache Group\Tomcat 4.1
```

Elsewhere this is called the `CATALINA_HOME` directory. Click on `Install`. This installs most of the files, installing them into this directory. The only other directory I noticed it write into was:

```
E:\Documents and settings\barry\Start Menu\Programs\Apache Tomcat 4.1
```

Later, it displays a Test Install Setup screen. This defaults to a port of 8080, a username of `admin` and leaves it to you to choose a password. After typing something appropriate into the password box, click on `Next`. This installs a few more files, and then says `Completed`. Click on `Close`.

If you click on `Start | Programs`, you should find a menu item called `Apache Tomcat 4.1`. This menu allows you easily to start and stop tomcat.

4.3 Starting tomcat

So go to `Start | Programs | Apache Tomcat 4.1` and click on `Start Tomcat`. A Command Prompt window will appear, and this will be used to log the activities of the tomcat webserver program. So this window will be present whilst tomcat is running.

An alternative way of starting tomcat is to use a Command Prompt window and execute commands like:

```
0030: set CATALINA_HOME=E:\Program Files\Apache Group\Tomcat 4.1
0031: set JAVA_HOME=e:\j2sdk1.4.1_02
0032: "%CATALINA_HOME%\bin\startup"
```

If your PC needs to use a proxy server to get to the internet, before executing these commands you will also need to execute a command like:

```
0033: set CATALINA_OPTS=-Dhttp.proxyHost=proxy.site.com -Dhttp.proxyPort=8080
```

4.4 Testing tomcat

Start a browser (such as Netscape Navigator or Internet Explorer) and go to:

```
http://localhost:8080/
```

You should get a WWW page announcing that you have successfully setup tomcat.

On the left hand side of this WWW page, you will see a link to `Servlet Examples`. If you click on this link, the page at:

```
http://localhost:8080/examples/servlets/index.html
```

appears. This is a page containing some `Servlet Examples with Code`. If you wish, click on the `Execute` and `Source` links that are on this WWW page. For example, click on the `Execute` link next to `Hello World`. This will take you to:

```
http://localhost:8080/examples/servlet/HelloWorldExample
```

Because of the way in which tomcat is configured, visiting this URL will cause tomcat to execute the `.class` file that has been stored in:

```
%CATALINA_HOME%\webapps\examples\WEB-INF\classes\HelloWorldExample.class
```

5 Using tomcat to execute our own servlets

5.1 Executing the `getDateAndTime` servlet

5.1.1 Creating a directory structure

Suppose we want tomcat to be able to execute the `getDateAndTime` servlet that was given earlier. Although we could add the files for our own servlets to the `examples` directory, suppose we want to store our servlets in a separate directory. Suppose this directory is called `mytomcat`.

We will need to create the `mytomcat` directory as a subdirectory of:

```
%CATALINA_HOME%\webapps
```

We actually have to create a directory tree like this:

```
webapps
   mytomcat
      WEB-INF
         classes
         lib
```

You can do this in Windows Explorer, or execute the following commands in a Command Prompt window:

```
cd %CATALINA_HOME%\webapps
mkdir mytomcat
cd mytomcat
mkdir WEB-INF
cd WEB-INF
mkdir classes lib
```

5.1.2 Storing the source code of the servlet

We now need somewhere to store the servlet's source code. Although tomcat does not require the source code of a servlet to be stored in the `%CATALINA_HOME%` directory tree, one possibility is to create a subdirectory of:

```
%CATALINA_HOME%\webapps\mytomcat
```

called `getDateAndTime` and store the text of the `getDateAndTime` servlet in the file:

```
%CATALINA_HOME%\webapps\mytomcat\getDateAndTime\getDateAndTime.java
```

5.1.3 Compiling the code of the servlet

Then execute the following commands from a Command Prompt window:

```
cd %CATALINA_HOME%\webapps\mytomcat\getDateAndTime
set classpath=..\..\..\common\lib\servlet.jar;.
javac -d ..\WEB-INF\classes getDateAndTime.java
dir /od ..\WEB-INF\classes
```

This will compile the file `getDateAndTime.java`. The `-d` option will arrange for the `.class` file to be put into a directory where tomcat will find it. So this `javac` command creates the file:

```
%CATALINA_HOME%\webapps\mytomcat\WEB-INF\classes\getDateAndTime.class
```

5.1.4 Getting tomcat to recognise the servlet

By convention, a servlet is accessed using a URL like:

```
http://localhost:8080/mytomcat/servlet/getDateAndTime
```

When it gets a URL like this, tomcat knows from the first part of the URL that the files are in:

```
%CATALINA_HOME%\webapps\mytomcat
```

However, we have to tell tomcat that there is a servlet called `getDateAndTime`. We can do this by putting the following text into a file called `web.xml` in the:

```
%CATALINA_HOME%\webapps\mytomcat\WEB-INF
```

directory:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
      PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
      "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
   <display-name>mytomcat</display-name>
   <servlet>
      <servlet-name>getDateAndTime</servlet-name>
      <display-name>getDateAndTime Servlet</display-name>
      <servlet-class>
         getDateAndTime
      </servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>getDateAndTime</servlet-name>
      <url-pattern>/servlet/getDateAndTime</url-pattern>
   </servlet-mapping>
</web-app>
```

5.1.5 Executing the servlet

Having done that, restart tomcat, and then use a browser to go to:

```
http://localhost:8080/mytomcat/servlet/getDateAndTime
```

This will get tomcat to execute the bytecodes of the file:

```
%CATALINA_HOME%\webapps\mytomcat\WEB-INF\classes\getDateAndTime.class
```

5.1.6 Automatically reloading an altered class

If you now alter the text of the servlet `getDateAndTime.java`, and recompile it placing the file `getDateAndTime.class` in the `classes` directory, you will find that this new version will not be executed unless you restart tomcat.

If you wish, you can alter the configuration of tomcat so that it frequently looks at the modification time of `.class` files in all of the `classes` directories and reloads its copy of any `.class` files that have been updated. This can be done by altering the file:

```
%CATALINA_HOME%\conf\server.xml
```

adding the line:

```
            <DefaultContext debug="0" reloadable="true"/>
```

between the line:

```
            </Context>
```

and the line:

```
      </Host>
```

You will have to restart tomcat for this change to `server.xml` to have any effect.

5.2 Getting a visitor to supply input to a servlet

The previous part of this document has given details of how a servlet with a `doGet` method can result in HTML being generated dynamically. We now look at a way in which the visitor to a WWW page can influence the execution of the bytecodes of a servlet. Often the author of a WWW page produces a WWW form if he/she wants the visitor to provide some input.

For example, suppose we want to provide a WWW page that can convert a Centigrade value into Fahrenheit. We could provide a WWW form to obtain the Centigrade value from the visitor and a servlet that does the arithmetic and generates the HTML containing the Fahrenheit value.

Here is a WWW page that contains a WWW form that can be used to obtain a Centigrade value from the visitor:

```
<HTML>
   <BODY>
      <FORM METHOD="POST" ACTION="http://localhost:8080/mytomcat/servlet/toFahrenheit">
         Type in a Centigrade value
         <INPUT TYPE="text" NAME="centigrade">
         <BR>
         <INPUT TYPE="submit" VALUE="Get Fahrenheit">
      </FORM>
   </BODY>
</HTML>
```

Suppose this HTML has been stored in a file called `toFahrenheit.html`. When a person uses a browser to visit this WWW page, the WWW form will appear. The visitor then has to type a value into the textbox and click on the *Get Fahrenheit* button. When the visitor does this, the action given in the `ACTION` attribute of the form will take place. In the above example, this means that the bytecodes of the `toFahrenheit` servlet will get executed.

Here is the source code of an appropriate `toFahrenheit` servlet:

```
0075: import javax.servlet.http. HttpServlet;                // toFahrenheit.java
0076: import javax.servlet.http. HttpServletRequest;
0077: import javax.servlet.http. HttpServletResponse;
0078: import java.io.            IOException;
0079: import java.io.            PrintWriter;
0080: import javax.servlet.      ServletException;
0081: public class toFahrenheit extends HttpServlet
0082: {
0083:    public void doPost(HttpServletRequest request,
0084:                       HttpServletResponse response)
0085:          throws IOException, ServletException
0086:    {
0087:       String tCentigradeString = request.getParameter("centigrade");
0088:       double tCentigrade = Double.parseDouble(tCentigradeString);
0089:       double tFahrenheit = 32 + tCentigrade*9/5;
0090:       response.setContentType("text/html");
0091:       PrintWriter tResponsePrintWriter = response.getWriter();
0092:       StringBuffer tStringBuffer = new StringBuffer();
0093:       tStringBuffer.append("<html>\n" );
0094:       tStringBuffer.append("<head>\n" );
0095:       tStringBuffer.append("<title>Reply</title>\n" );
0096:       tStringBuffer.append("</head>\n" );
0097:       tStringBuffer.append("<body>\n" );
0098:       tStringBuffer.append("<p>\n" );
0099:       tStringBuffer.append("In Fahrenheit, this is " + tFahrenheit + "\n" );
0100:       tStringBuffer.append("</p>\n" );
0101:       tStringBuffer.append("</body>\n" );
0102:       tStringBuffer.append("</html>\n" );
0103:       tResponsePrintWriter.println(tStringBuffer);
0104:       tResponsePrintWriter.close();
0105:    }
0106: }
```

Because a WWW form is being used, an HTTP POST request will be generated. For this reason, the above code overrides `doPost` rather than `doGet`. Like `doGet`, the `doPost` method has two parameters. It is the parameter called `request` that enables the values supplied on the WWW form to be accessed. So, because in the HTML we gave the textbox the name `centigrade`, the servlet can access the text that the visitor entered into the `centigrade` textbox by applying the `getParameter` function to the `request` object in the following way:

```
0087:       String tCentigradeString = request.getParameter("centigrade");
```

5.3 Another digression: JavaServer pages

The code of a servlet is a mix of Java source code doing some work and Java source code that generates HTML instructions. Because a lot of the code often consists of Java statements generating HTML, the code is not easy to read.

A *JavaServer page* is a WWW document that is written in a mix of elements of a markup language (e.g., HTML) and elements that will be dynamically generated by executing some Java code. As with servlets, the webserver program will either itself transform, or instead arrange for another program to transform, the JavaServer page into Java source code (a servlet). Behind the scenes, the servlet is then compiled and executed, and finally its output (HTML) is passed to the webserver program.

Here is a WWW page that has a button that will cause a JavaServer page to be executed:

```
0107: <HTML>
0108:    <BODY>
0109:       <FORM METHOD="POST" ACTION=
0110:          "http://localhost:8080/mytomcat/jtoFahrenheit/toFahrenheit.jsp">
0111:          Type in a Centigrade value
0112:          <INPUT TYPE="text" NAME="centigrade">
0113:          <BR>
0114:          <INPUT TYPE="submit" VALUE="Get Fahrenheit">
0115:       </FORM>
0116:    </BODY>
0117: </HTML>
```

And here is an appropriate JavaServer page (which is stored in the file `toFahrenheit.jsp`):

```
0118: <%@page language="java" %>
0119: <%
0120:    String tCentigradeString = request.getParameter("centigrade");
```

```
   double tCentigrade = Double.parseDouble(tCentigradeString);
   double tFahrenheit = 32 + tCentigrade*9/5;
%>
<html>
<head>
<title>Reply</title>
</head>
<body>
<p>
In Fahrenheit, this is <%= tFahrenheit %>
</p>
</body>
</html>
```

The Netcraft survey points out that ‘over the last year JSP has been the fastest growing scripting technology after ASP.NET. JSP sites are often bigger, more complex, and better funded and run by larger organisations than sites using the more common scripting technologies’ [33]. Some examples of JSP sites are `www.theaa.com`, `www.bt.com`, `www.explore.co.uk` and `www.opodo.co.uk`.

6 Web Services

6.1 Sites providing Web Services

One useful site for Web Services is `www.xmethods.net`: it gives a list of sites providing Web Services. But this list is not ordered in a way that makes it easy to find a Web Service.

UDDI is an initative that aims to make it easier to locate Web Services. UDDI means *Universal Description, Discovery and Integration*. UDDI is targetted at two kinds of customers: those businesses who want to publish what services they offer and those businesses who need to locate a service and access it from some program. More details are available at http://www.uddi.org/.

If you use a browser to go to http://www.xmethods.net/, you will get a WWW page listing the 30 WWW Services that were most recently added to this website. However, half way down this WWW page, there is a link labelled `FULL LIST`. If you click on this link, you will get the full list of Web Services that are known to `www.xmethods.net`.

Towards the bottom of the full list, there is a Web Service described as:

```
Publisher:       dpchiesa
Service Name:    ZipToCityState
Description:     Retrieves valid City+State pairs for a given
                 US Zip Code, or longitude/latitude for a zipcode.
                 Also retrieves zipcodes for city/state pairs
Implementation:  MS .NET
```

If you click on the `ZipToCityState` link, the `www.xmethods.net` site gives you more details about this Web Service. In particular, it says that a WSDL document is available at:

```
http://www.winisp.net/cheeso/zips/ZipService.asmx?WSDL
```

6.2 Finding out what service is being offered (WSDL)

If a site offers a Web Service, it needs to provide some means for an external site to find out the details of the Web Service, i.e., what methods are provided and how they are called.

If you go to:

```
http://www.winisp.net/cheeso/zips/ZipService.asmx?WSDL
```

a WWW page describing the `ZipService` Web Service is produced. Depending on the browser you are using, you may need to click on *View Page Source* in order to see this text.

The page that is returned is written using the notation of an XML language called *WSDL* (*Web Services Description Language*). This is a language originally developed by Ariba, IBM and Microsoft but which is now universally used. It was first released in September 2000.

A WSDL document describes everything that is needed to contact a Web Service. Here is an explanation of its key parts:

1. The `service` element of a WSDL document gives basic information about the Web Service such as the URL needed to contact the Web Service (its *endpoint*), and details about which *ports* (HTTP GET, HTTP POST and/or HTTP POST using SOAP) are supported. Some details about SOAP will be given later.

2. There is a `binding` element for each of the ports. A binding element describes for each method (provided by the Web Service) the way in which a request is coded and the way in which the reply is coded. So it might say the request is in SOAP and the reply is in SOAP. Or it might say the request is url-encoded and the reply is given using some XML coding.

3. There are two `message` elements for each of the ports. One of the message elements is used to give the name and the type of each parameter, and the other message element is used to describe the type of the result. So an *In* message element might say that there is one parameter called `pCentigrade` that is a `string` and the corresponding *Out* message element might say that a `double` is returned.

A fuller description of the purpose of the various parts of a WSDL document is available at [26].

The WSDL document provided by:

```
http://www.winisp.net/cheeso/zips/ZipService.asmx?WSDL
```

will be needed again later. So I used my browser to save the text of this WSDL document in a file called `ZipService.wsdl`.

6.3 Accessing the test WWW pages of a .NET's Web Service

The above URL ends in `.asmx?WSDL`. The `.asmx` is a sure sign that this Web Service has been implemented using Microsoft's .NET Framework. One of the benefits of using .NET for Web Services is that WWW pages that can be used to test a Web Service are dynamically generated. You can go to the first of these WWW pages by using the above URL without the last 5 characters, i.e., by going to:

```
http://www.winisp.net/cheeso/zips/ZipService.asmx
```

You should get a WWW page saying:

```
The following operations are supported. For a formal definition,
please review the Service Description.
    * ZipToLatLong
    * CityToZip
    * ZipToCityAndState
    * CityTo1Zip
    * ZipTo1CityAndState
```

This is a list of the names of methods provided by the Web Service. If you now click on one of these names, e.g., `ZipTo1CityAndState`, another WWW page is dynamically generated. It is the WWW page at:

```
http://www.winisp.net/cheeso/zips/ZipService.asmx?op=ZipTo1CityAndState
```

This WWW page provides a textbox and an *Invoke* button.

If you type a zipcode like `94042` into the textbox and click on the *Invoke* button, the browser will contact the webserver at `www.winisp.net` to get it to execute the `ZipTo1CityAndState` method. After a little time, your browser will give you the following WWW page:

```
<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://dinoch.dyndns.org/webservices/">
MOUNTAIN VIEW CA
</string>
```

So these WWW pages provide a way in which you can test the methods provided by a .NET Web Service.

6.4 Communicating with a Web Service (SOAP)

If an external site wishes to contact a Web Service in order to get some method executed, it needs to indicate to the Web Service the name of the method and the arguments to be passed to the method. And the external site is expecting the Web Service to return a value to the external site.

As a Web Service is being hosted on a webserver, the obvious way of contacting a Web Service is to send an HTTP request to the webserver. In fact, the usual way of contacting a Web Service is to send the webserver an HTTP POST request where the body of the request is coded using an XML language called *SOAP* (*Simple Object Access Protocol*) [50]. The reply from the Web Service is sent as an HTTP reply usually with a body coded in SOAP.

If you return to the WWW page containing the zipcode textbox and the *Invoke* button, i.e. to:

```
http://www.winisp.net/cheeso/zips/ZipService.asmx?op=ZipTo1CityAndState
```

you will see this WWW page also gives an example of the HTTP request that can be used to contact the Web Service and an example of the HTTP reply that comes back from the Web Service.

An example of an HTTP request to this Web Service is:

```
POST /cheeso/zips/ZipService.asmx HTTP/1.1
Host: www.winisp.net
Content-Type: text/xml; charset=utf-8
Content-Length: XXXX
SOAPAction: "http://dinoch.dyndns.org/webservices/ZipTo1CityAndState"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ZipTo1CityAndState xmlns="http://dinoch.dyndns.org/webservices/">
      <zip>94042</zip>
    </ZipTo1CityAndState>
  </soap:Body>
</soap:Envelope>
```

The body of this HTTP POST request is written using the notation of the XML language called SOAP. Hiding in all this detail is the fact that we want to contact a webserver on the computer `www.winisp.net` to visit the page at `/cheeso/zips/ZipService.asmx` to execute the `ZipTo1CityAndState` method with an argument having the name `zip` and value `94042`.

The reply from the webserver (after the appropriate method has been executed) is likely to have a body coded using SOAP. An example of a reply from `www.winisp.net` is:

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

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ZipTo1CityAndStateResponse
          xmlns="http://dinoch.dyndns.org/webservices/">
      <ZipTo1CityAndStateResult>MOUNTAIN VIEW CA</ZipTo1CityAndStateResult>
    </ZipTo1CityAndStateResponse>
  </soap:Body>
</soap:Envelope>
```

Hidden away in this XML is the fact that the result of the call is the value `MOUNTAIN VIEW CA`.

Even though the communication between the external site and the Web Service is coded in SOAP, normally we need not be concerned with the hard work of encoding and decoding the request and the reply: we will find that all of this is done by supporting software.

7 Axis

7.1 What is Axis?

There are currently two products from the Apache Software Foundation for Web Services: there is version 2.3.1 of Apache SOAP and version 1.1 of a newer product called Axis.

The WWW pages for Axis are at

http://ws.apache.org/axis/.

The FAQ says that 'Axis is essentially Apache SOAP 3.0. It is a from-scratch rewrite, designed around a streaming model (using SAX internally rather than DOM). The intention is to create a more modular, more flexible, and higher-performing SOAP implementation (relative to Apache SOAP 2.0).'

7.2 Installing Axis

You can download Axis from

http://ws.apache.org/axis/releases.html. Currently,

the latest release of Axis is Release Candidate 2 of Axis 1.1. So I downloaded a file called `axis-1_1rc2.zip`. This 10.1MB `.zip` file would take about 50 minutes to download through a 33Kbps modem. The file contains one directory called `axis-1_1RC2`. I unzipped this to:

```
e:\
```

so all the files are in the directory:

```
e:\axis-1_1RC2
```

As the `.zip` file is a 10.1MB file, the unzipping takes a long time.

The details about how to get Axis to work are in the Axis Installation Guide. This is available at:

```
http://cvs.apache.org/viewcvs.cgi/~checkout~/xml-axis/java/docs/install.html
```

The first thing to do is to copy the `axis` subdirectory of:

```
e:\axis-1_1RC2\webapps
```

to:

```
%CATALINA_HOME%\webapps
```

The Axis Installation Guide then suggests that you copy some XML parser files. I found that I did not have to do this step. However, if you need to do this, the Axis Installation Guide says that the XML parser files need to be put into the directory:

```
%CATALINA_HOME%\webapps\axis\WEB-INF\lib
```

As the full distribution of tomcat comes with an XML parser, you could copy those files: so you could copy the files `xercesImpl.jar` and `xmlParserAPIs.jar` from:

```
%CATALINA_HOME%\common\endorsed
```

to:

```
%CATALINA_HOME%\webapps\axis\WEB-INF\lib
```

The final step of installing Axis is to restart tomcat.

7.3 Testing Axis

Use a browser to go to the WWW page

http://localhost:8080/axis/.

You should get the WWW page welcoming you to Apache-Axis. The Axis Installation Guide now suggests that you click on the link:

```
Validate the local installation's configuration
```

This will run

http://localhost:8080/axis/happyaxis.jsp.

You should examine the text of the resulting WWW page carefully. The Axis Installation Guide says 'if any of the needed libraries are missing, Axis will not work'.

When I ran `happyaxis.jsp`, it gave me a WWW page with lots of information. However, towards the middle of the page I got:

```
The core axis libraries are present. 1 optional axis library is missing.
```

So that is OK.

7.4 Using the programs provided by Axis

In the following sections of this document, we will be using many of the programs that come with Axis. We will need an appropriate classpath to execute these programs. An appropriate very long classpath will be set up in a Command Prompt window if the following commands are executed:

```
set AXIS_HOME=E:\axis-1_1RC2
set CLASSPATH=.
set CLASSPATH=%AXIS_HOME%\lib\axis.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\commons-discovery.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\commons-logging.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\jaxrpc.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\saaj.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\log4j-1.2.4.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\wsdl4j.jar;%CLASSPATH%
```

8 Using Axis to access a Web Service

8.1 Providing a program that accesses a Web Service

Even though we have not yet produced a Web Service, we can use the Axis that we have just installed in order to access an external Web Service. We will suppose that we want to produce a program that contacts the Web Service (used earlier) that translates a zipcode into a place name.

Earlier, we stored the WSDL document of this Web Service in the file `ZipService.wsdl`. We now execute a program called `WSDL2Java` on this `ZipService.wsdl` file:

```
java org.apache.axis.wsdl.WSDL2Java ZipService.wsdl
```

As this is a program that comes with Axis, in order to execute the above command you will need to set up the very long classpath that was given earlier.

The WSDL2Java program looks at the WSDL document to discover things about the Web Service. It then generates a *proxy class* (aka a *stub class*). For each method of the Web Service, the proxy class will have a method that has the same parameters and the same return type. The idea is that if a client wants to call a method of the Web Service it instead calls the method of the proxy class that has the same name and parameters. Behind the scenes, the call of a method of the proxy class creates the HTTP request (with its complicated SOAP); sends it to the webserver providing the Web Service; waits for the reply; and then decodes the SOAP that is returned by the Web Service.

If, having executed the above `WSDL2Java` command, you then execute the command:

```
dir org\dyndns\dinoch\webservices
```

you will see that the WSDL2Java program has created the files:

```
ArrayOfString.java
Latitude.java
LatLong.java
Longitude.java
ZipcodeLookupService.java
ZipcodeLookupServiceLocator.java
ZipcodeLookupServiceSoap.java
ZipcodeLookupServiceSoapStub.java
```

The file `ZipcodeLookupServiceSoapStub.java` contains the proxy class; the other files are supporting files.

The interface `ZipcodeLookupService` declares headers for `get` methods for each port that is listed in the `service` element of the WSDL document. In our example, it declares a header for a method called `getZipcodeLookupServiceSoap`. The class `ZipcodeLookupServiceLocator` is an implementation of this interface, and so it implements the `getZipcodeLookupServiceSoap` method. This method returns an instance of the proxy class `ZipcodeLookupServiceSoapStub`. This class implements the interface `ZipcodeLookupServiceSoap`: this interface just declares headers for each method that is provided by the Web Service.

Now we can produce a Java program that is going to be a client of this Web Service:

```
import java.rmi.                      RemoteException;  // MyTestClient.java
import javax.xml.rpc.                 ServiceException;
import org.dyndns.dinoch.webservices. ZipcodeLookupService;
import org.dyndns.dinoch.webservices. ZipcodeLookupServiceLocator;
import org.dyndns.dinoch.webservices. ZipcodeLookupServiceSoap;
public class MyTestClient
{
   public static void main(String[] pArgs) throws RemoteException, ServiceException
   {
      String tZipcodeString = pArgs[0];
      ZipcodeLookupService tZipcodeLookupService =
            new ZipcodeLookupServiceLocator();
      ZipcodeLookupServiceSoap tZipcodeLookupServiceSoap =
            tZipcodeLookupService.getZipcodeLookupServiceSoap();
      String tCityAndStateString =
            tZipcodeLookupServiceSoap.zipTo1CityAndState(tZipcodeString);
      System.out.println(tCityAndStateString);
   }
}
```

`MyTestClient.java` can be compiled and run using a particular zipcode by the commands:

```
javac MyTestClient.java
java MyTestClient 94042
```

The program should contact the Web Service and then output `MOUNTAIN VIEW CA`. Note: if your PC needs to use a proxy server to get to the internet, you will need a `java` command like:

```
java -Dhttp.proxyHost=proxy.site.com -Dhttp.proxyPort=8080 MyTestClient 94042
```

8.2 Using tcpmon to monitor the request and the response

We can have a clearer idea of what information is being transferred if we run the `tcpmon` program (that comes with Axis). Instead of the client program contacting the Web Service directly we arrange for it to go through the `tcpmon` program. So the client program contacts `tcpmon` and `tcpmon` contacts the Web Service.

We need to make a few changes for this to occur. First, modify the WSDL file so that the URL of the Web Service refers to a port on the local computer. For example, the three occurrences of:

```
<http:address location="http://www.winisp.net/cheeso/zips/ZipService.asmx" />
```

in the file `ZipService.wsdl` could be changed to:

```
<http:address location="http://localhost:8081/cheeso/zips/ZipService.asmx" />
```

where 8081 is an arbitrarily chosen port. Then run the WSDL2Java program again in order to generate new proxy classes. Now, when you want to run `MyTestClient`, first type the following command in a Command Prompt window:

```
start java org.apache.axis.utils.tcpmon 8081 www.winisp.net 80
```

As `tcpmon` is a part of Axis, it requires the very long classpath given earlier. Then, after the tcpmon window appears, type the following command in a Command Prompt window:

```
java MyTestClient 94042
```

The tcpmon window should show you the HTTP request that goes to `www.winisp.net` and the HTTP response that comes back from that site.

8.3 The security and reliability of Web Services

There are many issues that need to be considered if you wish to secure Web Services. These are examined by the papers at [13], [18], [20] and [46].

Of course, writing a program in terms of a Web Service means that the program is dependent on a connection to the internet, the availability of the computer and the webserver of the Web Service, whether the Web Service still maintains the same contract as when the program was written, ... Some of these reliability issues are addressed by the papers at [19] and [22].

9 Using Axis to provide a Web Service

9.1 The easiest way of providing an Axis Web Service

We first look at the easiest way in Axis of providing a Web Service.

First, produce a Java class declaration containing the methods that you wish to be made available. For example, we could produce:

```
public class Convert {
   public double toFahrenheit(double pCentigrade) {
      return 32 + pCentigrade*9/5;
   }
}
```

Store this text in a file with a `.jws` extension. In the above example, the text would be stored in a file called `Convert.jws`.

Put this file in the directory:

```
%CATALINA_HOME%\webapps\axis
```

And that's it! You have a Web Service available at:

```
http://localhost:8080/axis/Convert.jws
```

If you use a browser to go to this URL, you will get a WWW page that says:

```
There is a Web Service here
Click to see the WSDL
```

If you do click on `Click to see the WSDL`, your browser will go to:

```
http://localhost:8080/axis/Convert.jws?wsdl
```

You will then be able to see the WSDL of this Web Service. (As mentioned earlier, depending on the browser you are using, you may need to click on *View Page Source* to see this XML.)

Having provided a Web Service, you can now follow the steps given earlier if you want to provide a client that accesses this Web Service.

The Axis User's Guide says 'JWS web services are intended for simple web services. You cannot use packages in the pages, and as the code is compiled at run time you can not find out about errors until after deployment'. The Axis User's Guide gives details of alternative ways of *deploying* Web Services which it says should be used for 'production quality web services'.

9.2 A better way: six steps to providing an Axis Web Service

In my view, the best way of deploying an Axis Web Service is to use the following six steps. It is one of the ways described in the Axis User's Guide.

9.2.1 Step 1: Provide a Java interface or class and compile it

The first step is to provide and then compile an interface declaration or a class declaration giving details of the methods that you wish to make available as a Web Service.

So we could provide the following interface declaration:

```
package Pack;                                                // Convert.java
public interface Convert {
   public double toFahrenheit(double pCentigrade);
}
```

storing it in the file `Convert.java` in some subdirectory called `Pack`. We could then compile it:

```
javac Pack\Convert.java
```

This produces the file `Convert.class` in the subdirectory `Pack`.

However, if we want our choice of the names of the parameters of methods to prevail, it is better to provide a class declaration and compile it with `javac`'s debug option (`-g`). The class declaration could consist of method declarations with nearly empty bodies. For example, we could store:

```
package Pack;                                                // Convert.java
public class Convert {
   public double toFahrenheit(double pCentigrade) {
      return 42;
   }
}
```

in the file `Convert.java` in the subdirectory `Pack` and then compile it using:

```
javac -g Pack\Convert.java
```

to produce the file `Convert.class` in the subdirectory `Pack`.

9.2.2 Step 2: Use the Java2WSDL program to create a WSDL file

Having produced a `.class` file, the Java2WSDL program can be used to create a WSDL file. Once again, this program is only accessible if the very long classpath that was given earlier has been set up.

As well as indicating the name of the class (`Pack.Convert`) and the name of the WSDL file (`Convert.wsdl`), the command line that is used to run the Java2WSDL program also needs to mention the namespace to be used in the WSDL file and the URL to be used as the endpoint of the Web Service. Here is a typical use of the program:

```
java org.apache.axis.wsdl.Java2WSDL -o Convert.wsdl ^
      -l"http://localhost:8080/axis/services/Convert" ^
      -n "urn:Convert" -p"Pack" "urn:Convert" Pack.Convert
```

The Java2WSDL program will ensure that the appropriate message, portType, binding and service elements are included in the WSDL file. If the Java interface/class refers to other interfaces/classes (that we have written), the WSDL file will also contain XML that can be used to represent those types. The Axis User's Guide points out that the Java2WSDL program supports bean classes, arrays and holder classes.

9.2.3 Step 3: Use the WSDL2Java program to create the files for a Web Service

In this step, the WSDL2Java program is used again. Earlier we used it to create proxy files (aka stub files) for a client. It can also be used to create files needed to support the code of a Web Service.

Here is a typical use of the WSDL2Java program for this purpose:

```
java org.apache.axis.wsdl.WSDL2Java -o . -s -S true -Nurn:Convert Pack Convert.wsdl
```

When the WSDL2Java program is used with these options, it will examine the WSDL file (that is named as the final parameter) and generate eight or more files. In our example, these files will be created in the `Pack` subdirectory. There are three kinds of files:

- files needed to provide a Web Service:

  ```
  Convert.java
  ConvertSoapBindingImpl.java
  ConvertSoapBindingSkeleton.java
  ```

- files needed to deploy/undeploy the Web Service:

  ```
  deploy.wsdd
  undeploy.wsdd
  ```

- files needed to provide a client of the Web Service:

  ```
  ConvertService.java
  ConvertServiceLocator.java
  ConvertSoapBindingStub.java
  ```

As we are currently only interested in providing a Web Service, the last three files will be ignored.

9.2.4 Step 4: Alter the `ConvertSoapBindingImpl.java` file

One of the files created by the WSDL2Java program is a file containing an interface declaration that declares the methods of the Web Service. For example, given the above command line it would produce a file called `Convert.java` that contains:

```
package Pack;                                              // Convert.java
public interface Convert extends java.rmi.Remote {
   public double toFahrenheit(double pCentigrade) throws java.rmi.RemoteException;
}
```

It also produces a stab at an implementation of the Web Service in the file `ConvertSoapBindingImpl.java`:

```
package Pack;                               // ConvertSoapBindingImpl.java
public class ConvertSoapBindingImpl implements Pack.Convert {
   public double toFahrenheit(double pCentigrade) throws java.rmi.RemoteException {
      return -3;
   }
}
```

This file needs to be altered. In our example, the:

```
      return -3;
```

needs to be replaced by:

```
      return 32 + pCentigrade*9/5;
```

9.2.5 Step 5: Compile the files of the Web Service

The files of the Web Service need to be compiled and put into their correct places. I do this using commands like:

```
cd Pack
javac *.java
mkdir "%CATALINA_HOME%\webapps\axis\WEB-INF\classes\Pack"
copy Convert.class                    "%CATALINA_HOME%\webapps\axis\WEB-INF\classes\Pack"
copy ConvertSoapBindingImpl.class     "%CATALINA_HOME%\webapps\axis\WEB-INF\classes\Pack"
copy ConvertSoapBindingSkeleton.class "%CATALINA_HOME%\webapps\axis\WEB-INF\classes\Pack"
cd ..
dir "%CATALINA_HOME%\webapps\axis\WEB-INF\classes\Pack"
```

9.2.6 Step 6: Deploy the Web Service

One of the files resulting from running the WSDL2Java program is a file called `deploy.wsdd`. In our example, the WSDL2Java program produces a `deploy.wsdd` file containing:

```
<deployment
      xmlns="http://xml.apache.org/axis/wsdd/"
      xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
   <service name="Convert" provider="java:RPC" style="rpc" use="encoded">
      <parameter name="wsdlTargetNamespace" value="urn:Convert"/>
      <parameter name="wsdlServiceElement" value="ConvertService"/>
      <parameter name="wsdlServicePort" value="Convert"/>
      <parameter name="className" value="Pack.ConvertSoapBindingSkeleton"/>
      <parameter name="wsdlPortType" value="Convert"/>
      <parameter name="allowedMethods" value="*"/>
   </service>
</deployment>
```

Essentially this *WSDD* (*Web Services Deployment Descriptor*) file is describing:

- The name of the service, e.g., `Convert`.
- The class that handles the service, e.g., `Pack.ConvertSoapBindingSkeleton`.
- The methods that are to be made available to clients of the Web Service. Here the `*` means all of the public methods of the class declaration. If we wish to restrict which methods are available, we could instead provide a space-separated or a comma-separated list of the names of the methods that are to be made available.

Once this file has been created (by the WSDL2Java program), we can use the AdminClient program to get the Web Service deployed.
Here is a typical use of the AdminClient program:

```
java org.apache.axis.client.AdminClient Pack\deploy.wsdd
```

Note: you can check what Web Services are deployed either by executing the following command in a Command Prompt window:

```
java org.apache.axis.client.AdminClient list
```

or by visiting the WWW page at:

```
http://localhost:8080/axis/servlet/AxisServlet
```

Note: the following command can be used to undeploy a Web Service:

```
java org.apache.axis.client.AdminClient Pack\undeploy.wsdd
```

9.2.7 Accessing the Web Service

Once the AdminClient program has been used to deploy a Web Service, the Web Service can be accessed by client programs in the usual way.

10 Session handling and Interoperability with .NET

10.1 Introduction

Microsoft's .NET Framework can also be used to produce Web Services and to access them. We now look at an example that illustrates how easy it is to interoperate Axis and .NET for Web Services. So we want the answers to the following questions:

- Is it easy to access an Axis Web Service from a .NET client?
- Is it easy to access a .NET Web Service from an Axis client?

The example we will look at provides a Web Service that accumulates the sum of a series of values. The Web Service is to provide a method called `inc` that has one parameter, and, each time a client calls `inc`, the value of the argument is added to a variable stored somewhere in the Web Service. So this Web Service is different from those we have looked at so far: it requires the Web Service to retain a value between successive calls.

10.2 ATotal: an Axis Web Service to store the sum of a series of values

Here is the code of the implementation of an Axis Web Service (called `ATotal`) that can be used to accumulate the sum of a series of values:

```
package Pack;                                  // ATotalSoapBindingImpl.java
public class ATotalSoapBindingImpl implements Pack.ATotal{
   private double iTotal = 0.0;
   public double inc(double pIncrement) throws java.rmi.RemoteException {
      iTotal = iTotal + pIncrement;
      return iTotal;
   }
}
```

If we proceed to use the six steps (described earlier) to produce an Axis Web Service, then each call of `inc` will use a different `ATotalSoapBindingImpl` object. What we want is for all calls of `inc` to access a specific `ATotalSoapBindingImpl` object. Besides getting a different object (called *Request*), Axis permits two other possibilities: either every call gets the same object (called *Application*) or a call can give the *name* of the object it wishes to use (called *Session*).

Step 3 of the six steps uses the WSDL2Java program. When executing this program, a `-d` option can be used to indicate which kind of Web Service is to be generated. So if a:

```
-d Session
```

option is supplied, as in:

```
java org.apache.axis.wsdl.WSDL2Java -o . -s -S true -d Session ^
   -Nurn:ATotal Pack ATotal.wsdl
```

the line:

```
<parameter name="scope" value="Session"/>
```

will appear in the file `deploy.wsdd`.

If this is done, the HTTP response from the first call of a method of the Web Service will include a header line like:

```
Set-Cookie: JSESSIONID=595B7DAE454938844EEB49EFFE67C6B3; Path=/axis
```

If a client wishes to access the object used in this call in later calls, it needs to capture this value and provide it in the header of the HTTP requests of subsequent calls to the Web Service, e.g.:

```
Cookie: JSESSIONID=595B7DAE454938844EEB49EFFE67C6B3
```

10.3 Accessing ATotal from an Axis client

Here is an Axis client of the `ATotal` Web Service:

```
0258: import ATotal_pkg.    ATotal;                            // MyTestClient.java
0259: import ATotal_pkg.    ATotalService;
0260: import ATotal_pkg.    ATotalServiceLocator;
0261: import ATotal_pkg.    ATotalSoapBindingStub;
0262: import java.io.       BufferedReader;
0263: import java.io.       InputStreamReader;
0264: import java.io.       IOException;
0265: import javax.xml.rpc. ServiceException;
0266: public class MyTestClient
0267: {
0268:    public static void main(String[] pArgs) throws IOException, ServiceException
0269:    {
0270:       ATotalService tATotalService = new ATotalServiceLocator();
0271:       ATotal tATotal = tATotalService.getATotal();
0272:       ATotalSoapBindingStub tATotalSoapBindingStub = (ATotalSoapBindingStub)tATotal;
0273:       tATotalSoapBindingStub.setMaintainSession(true);
0274:       BufferedReader tKeyboard = new BufferedReader(new InputStreamReader(System.in));
0275:       while (true)
0276:       {
0277:          System.out.print("Increment: ");
0278:          String tLine = tKeyboard.readLine();
0279:          if ( tLine.equals("") ) break;
0280:          double tIncrement = Double.parseDouble(tLine);
0281:          double tNewTotal = tATotal.inc(tIncrement);
0282:          System.out.println("NewTotal is now: " + tNewTotal);
0283:       }
0284:    }
0285: }
```

The client sits there waiting for the user to type some value. It thens calls `inc` which will add the value to the total being stored by the Web Service. The client then waits for another value to be typed. An empty line is used to terminate the execution of the client.

The code of this client is similar to the code of the Axis Web Service client given earlier. The only new idea is contained in the statements:

```
0272:       ATotalSoapBindingStub tATotalSoapBindingStub = (ATotalSoapBindingStub)tATotal;
0273:       tATotalSoapBindingStub.setMaintainSession(true);
```

The call of `setMaintainSession` ensures that the session id is collected from the first HTTP response and is used in the HTTP requests of any subsequent calls of methods of the Web Service.

10.4 Accessing ATotal from a .NET program

We could use Microsoft's Visual Studio.NET to create a client of the ATotal Web Service [12, 25]. For example, we could use the wizard that creates a client that is a Windows Form Application. The form could have a textbox to obtain the increment, a button to submit the query, and a label that can be used to report back the latest value of the total. By default, the wizard would create a class called `Form` with private variables called `textbox1`, `button1` and `label1`. The wizard also generates a skeleton for a method called `button1_Click` and we could supply the following body for this method. (This code is given in C#.) This method would be executed whenever there is a click of the button:

```
0286: private void button1_Click(object sender, System.EventArgs e)
0287: {
0288:    double tIncrement = double.Parse(textBox1.Text);
0289:    double tNewTotal = iATotalService.inc(tIncrement);
0290:    label1.Text = tNewTotal.ToString();
0291: }
```

This code assumes that `iATotalService` is declared as a private variable of the `Form` class:

```
private ATotalService iATotalService;
```

and that it is pointing to an object of the proxy class for the Web Service. This object can be created using the statement:

```
iATotalService = new ATotalService();
```

The client needs to store the cookie it receives from the first call of `inc` and pass it in the headers of the HTTP requests of subsequent calls. This can be done using the statements:

```
CookieContainer tCookieContainer = new CookieContainer();
iATotalService.CookieContainer = tCookieContainer;
```

where `CookieContainer` is a class declared in the `System.Net` namespace. The last three statements need to be executed once. So they can go in the constructor of the `Form` class.

The above code assumes the existence of a proxy class called `ATotalService`. Visual Studio.NET has a wizard (called `Add Web Reference`) that can be used to generate a proxy class. It needs you to supply the URL of a WSDL document, e.g.:

```
http://localhost:8080/axis/services/ATotal?WSDL
```

If you supply a URL (like this one) that includes a port number, you may find that Visual Studio.NET ignores the port number when generating the proxy class. If this is the case, you will need to edit the constructor of the proxy class to include the port number:

```
public ATotalService()
{
   this.Url = "http://localhost:8080/axis/services/ATotal";
}
```

10.5 MTotal: a .NET Web Service to store the sum of a series of values

We now suppose that, instead, the .NET Framework is being used to provide the Web Service [12, 24]. Visual Studio.NET has a wizard that makes it easy to create a Web Service. It generates most of the code for you in a class that is derived from a class called `WebService` (which is declared in the `System.Web.Services` namespace). All you need to do is to include in this class the code of the methods that you wish to make available. You will have to declare each of these methods with a `WebMethod` attribute.

You will need to do two things to get a value retained from one call of a method to the next:

- Include `EnableSession=true` in `inc`'s `WebMethod` attribute.
- Store the value away so that it can be retrieved the next time. This can be done through an indexer of the `WebService` class called `Session`. (In the same way that a *property* provides access to a single value, an *indexer* provides access to an array of values. Here we only need one value and the index `"MTotal"` is being used to access this value.)

Here is the full code (in C#) for the web method `inc`:

```
[WebMethod(EnableSession=true)]
public double inc(double pIncrement)
{
   double tNewTotal;
   object tObject = Session["MTotal"];
   if ( tObject == null )
      tNewTotal = pIncrement;
   else
      tNewTotal = (double)tObject + pIncrement;
   Session["MTotal"] = tNewTotal;
   return tNewTotal;
}
```

The HTTP response from the first call of a method of this Web Service will include a header line like:

```
Set-Cookie: ASP.NET_SessionId=50jlsti3jqa5gg45321sbkia; path=/
```

A client needs to ensure that it captures this value and uses it in the header of the HTTP requests of subsequent calls to the Web Service, e.g.:

```
Cookie: ASP.NET_SessionId=50jlsti3jqa5gg45321sbkia
```

10.6 Accessing MTotal from Axis clients and .NET clients

We can produce an Axis client or a .NET client as before. Because the code of the client is driven by the contents of the WSDL document, it does not really matter whether the Web Service is an Axis Web Service or a .NET Web Service.

11 Accessing the Web Service provided by Amazon

11.1 Accessing and using Amazon's WSDL document

A WSDL document for accessing the Web Service provided by `www.amazon.com/www.amazon.co.uk` is at:

```
http://soap.amazon.com/schemas2/AmazonWebServices.wsdl
```

If you run the WSDL2Java program on this document, it will produce a lot of `.java` files in the directory:

```
com\amazon\soap
```

I compiled these files, produced a jar file of the resulting `.class` files, and then moved this jar file to the directory:

```
%CATALINA_HOME%\webapps\mytomcat\WEB-INF\lib
```

I ensured that this directory also included all the jar files needed to use Axis.

11.2 Providing a JavaServer page that is a client of Amazon's Web Services

I then produced the following JavaServer page. You need to register with Amazon to use these facilities. For this reason, in the following code, my registration id has been replaced by `XXXXXXXXXXXXXX`. Setting the `locale` to `uk` is meant to get it to yield information from `amazon.co.uk` rather than `amazon.com` but this does not seem to be the case for some pieces of information. It seems it is part of the design of Amazon's Web Service to return `null` if a value is not available in time: it seems that their view is that it is better to be fast in providing some of the information rather than hanging around for all of it to become available.

```
<html>
<body>
<%@page language="java" import="com.amazon.soap.*" %>
<%
   try
   {
      String tIsbn = "0201711079";
      AmazonSearchService tAmazonSearchService = new AmazonSearchServiceLocator();
      AmazonSearchPort tAmazonSearchPort = tAmazonSearchService.getAmazonSearchPort();
      AsinRequest tAsinRequest = new AsinRequest();
      tAsinRequest.setTag("webservices-20");
      tAsinRequest.setDevtag("XXXXXXXXXXXXXX");
      tAsinRequest.setAsin(tIsbn);
      tAsinRequest.setLocale("uk");
      tAsinRequest.setType("heavy");
      ProductInfo tProductInfo = tAmazonSearchPort.asinSearchRequest(tAsinRequest);
      Details[] tDetailsArray = tProductInfo.getDetails();
      String tProductName = tDetailsArray[0].getProductName();
      String tReleaseDate = tDetailsArray[0].getReleaseDate();
      String tOurPrice = tDetailsArray[0].getOurPrice();
      String tListPrice = tDetailsArray[0].getListPrice();
      String tUsedPrice = tDetailsArray[0].getUsedPrice();
      String tSalesRank = tDetailsArray[0].getSalesRank();
      String tAvailability = tDetailsArray[0].getAvailability();
      String tImageUrlSmall = tDetailsArray[0].getImageUrlSmall();
      String tImageUrlMedium = tDetailsArray[0].getImageUrlMedium();
      String tImageUrlLarge = tDetailsArray[0].getImageUrlLarge();
%>
      <table><tr><td><img src="<%= tImageUrlMedium %>"></td><td>
      My book <em><%= tProductName %></em> is on sale at
      <a href="http://www.amazon.co.uk/exec/obidos/ASIN/0201711079/">www.amazon.co.uk</a>
      for <%= tOurPrice %>.
<%
      if ( ! tOurPrice.equals(tListPrice) )
      {
%>
         <br>Its list price is <%= tListPrice %>.
<%
      }
%>
      <br>Grimly, it's also available there with a used price of <%= tUsedPrice %>!
      <br>Currently, its sales rank is <%= tSalesRank %>.
      <br>It was published on <%= tReleaseDate %>.
      </td></tr></table>
<%
   }
   catch (Exception pException)
   {
      pException.printStackTrace();
   }
%>
</body>
</html>
```

12 Other products for Web Services

12.1 Using Microsoft's .NET Framework for Web Services

Axis is not the only product for Web Services: there are many others that can be used.

As has already been mentioned, it is possible to provide a Web Service and to access Web Services using Microsoft's .NET Framework. For more details, see [12].

12.2 Using Sun's Java Web Services Developer Pack

Sun provide a *Java Web Services Developer Pack* (*Java WSDP*) [43]. This assumes you have no existing specialist software (other than the Java 2 SDK), and it allows you to get started in a number of different areas besides Web Services.

Sun's WWW page says that the Java WSDP 'is a free integrated toolkit that allows Java developers to build, test and deploy XML applications, Web services, and Web applications. The Java WSDP provides Java standard implementations of existing key Web services standards including WSDL, SOAP, ebXML, and UDDI as well as important Java standard implementations for Web application development such as JavaServer Pages and the JSP Standard Tag Library. These Java standard implementations allow developers to send and receive SOAP messages, browse and retrieve information in UDDI and ebXML registries, and quickly build and deploy Web applications based on the latest Java standard implementations'.

There is a lot of software (70MB) in the Java WSDP including Java Architecture for XML Binding (JAXB), Java API for XML Messaging (JAXM), Java API for XML Processing (JAXP), Java API for XML Registries (JAXR), Java API for XML-based RPC (JAX-RPC), SOAP with Attachments API for Java (SAAJ), JavaServer Pages Standard Tag Library (JSTL), Java WSDP Registry Server, Ant Build Tool and Apache Tomcat. The size of the file containing the download of the Java WSDP is 34.5Mb.

13 References

1. Dion Almaer, 'Creating Web Services with Apache Axis',
   http://www.onjava.com/pub/a/onjava/2002/06/05/axis.html

2. Amazon, 'Web Services Developer's Kit',
   http://associates.amazon.com/exec/panama/associates/join/developer/kit.html

3. Amazon, 'WSDL description of Amazon.com's Web Services APIs',
   http://soap.amazon.com/schemas2/AmazonWebServices.wsdl

4. Amazon, 'Amazon Web Services Version 2.1 API and Integration Guide'.

5. Apache Axis, 'Axis User's Guide',
   http://cvs.apache.org/viewcvs.cgi/~checkout~/xml-axis/java/docs/user-guide.html

6. Apache Axis, 'Installing and deploying web applications using xml-axis',
   http://cvs.apache.org/viewcvs.cgi/~checkout~/xml-axis/java/docs/install.html

7. Apache Axis, 'Axis Reference Guide',
   http://cvs.apache.org/viewcvs.cgi/~checkout~/xml-axis/java/docs/reference.html

8. Apache Tomcat, 'Running the Tomcat 4.0 Servlet/JSP Container',
   http://jakarta.apache.org/tomcat/tomcat-4.1-doc/RUNNING.txt

9. Mike Clark, Peter Fletcher, J. Jeffrey Hanson, Romin Irani, Mark Waterhouse, Jorgen Theli,
   'Web Services Business Strategies and Architectures',
   Expert Press, 2002, 1904284132.

10. Barry Cornelius, 'A Taste of C#',
    http://www.dur.ac.uk/barry.cornelius/papers/a.taste.of.csharp/

11. Barry Cornelius, 'Comparing .NET with Java',
    http://www.dur.ac.uk/barry.cornelius/papers/comparing.dotnet.with.java/

12. Barry Cornelius, 'Web Services using .NET',
    http://www.dur.ac.uk/barry.cornelius/papers/web.services.using.dotnet/

13. Ray Djajadinata, 'Yes, you can secure your Web services documents',
    http://www.javaworld.com/javaworld/jw-08-2002/jw-0823-securexml.html
    http://www.javaworld.com/javaworld/jw-10-2002/jw-1011-securexml.html

14. Chrisina Draganova, 'Web Services and Java',
    http://www.ics.ltsn.ac.uk/pub/jicc7/draganova2.doc

15. Steve Graham, Simeon Simeonov, Toufic Boubez, Glen Daniels, Doug Davis, *et al*, ‘Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI’, Sams, 2001, 0672321815.

16. Gil Hansen, ‘Gil Hansen’s XML & WebServices URLs’, http://www.javamug.org/mainpages/XML.html#WebServices

17. Erik Hatcher and Steve Loughan, ‘Java Development with Ant’, Manning, 2002, 1930110588. ‘Chapter 15: Working with web services’, http://www.manning.com/hatcher/chap15.pdf

18. Pankaj Kumar, ‘Web Services Over SSL: HOW TO’, http://www.pankaj-k.net/WSOverSSL/WSOverSSL-HOWTO.html

19. IBM and Microsoft, ‘Reliable Message Delivery in a Web Services World’, ftp://www6.software.ibm.com/software/developer/library/ws-rm-exec-summary.pdf

20. IBM and Microsoft, ‘Security in a Web Services World: A Proposed Architecture and Roadmap’, http://msdn.microsoft.com/ws-security/

21. Jonathan Lurie and R. Jason Belanger, ‘The great debate: .Net vs. J2EE’, http://www.javaworld.com/javaworld/jw-03-2002/jw-0308-j2eenet.html

22. Anbazhagan Mani and Arun Nagarajan, ‘Understanding quality of service for Web services’, http://www-106.ibm.com/developerworks/library/ws-quality.html

23. Microsoft, ‘Visual Studio.NET and .NET Framework Reviewers Guide’, http://download.microsoft.com/download/VisualStudioNET/Utility/7.0/W9X2K/EN-US/frameworkevalguide.doc

24. Microsoft, ‘Walkthrough: Creating a Web Service Using Visual Basic or C#’, Help system with Visual Studio.NET.

25. Microsoft, ‘Walkthrough: Accessing a Web Service Using Visual Basic or C#’, Help system with Visual Studio.NET.

26. Microsoft, ‘Web Services Description Language (WSDL) Explained’, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/wsdlexplained.asp

27. Microsoft, ‘HOW TO: Integrate an Apache SOAP 2.2 Client with a .NET XML Web Service’, http://support.microsoft.com/directory/article.asp?id=q308466&sd=msdn

28. Microsoft, ‘Web Services Enhancements for Microsoft .NET’, http://msdn.microsoft.com/webservices/building/wse/default.aspx

29. Microsoft, ‘Web Services Enhancements 1.0 and Java Interoperability’, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/wsejavainterop.asp

30. The Mind Electric, ‘Glue’, http://www.themindelectric.com/

31. Tarak Modi, ‘Axis: The next generation of Apache SOAP’, http://www.javaworld.com/javaworld/jw-01-2002/jw-0125-axis.html

32. Ramesh Nagappan, Robert Skoczylas, Rima Patel Sriganesh, ‘Developing Java Web Services: Architecting and Developing Secure Web Services Using Java’, John Wiley, 2002, 0-471-23640-3.

33. Netcraft, ‘Java Servlet Engines’, http://news.netcraft.com/archives/2003/04/10/java_servlet_engines.html

34. Uche Ogbuji, ‘The Past, Present and Future of Web Services’, http://www.webservices.org/index.php/article/articleview/663/1/61/ http://www.webservices.org/index.php/article/articleview/679/1/61/

35. Chris Peiris, ‘Creating a .NET Web Service’, http://www.15seconds.com/issue/010430.htm

36. Matt Powell, ‘Using ASP.NET Session State in a Web Service’, http://msdn.microsoft.com/library/en-us/dnservice/html/service08062002.asp

37. Aaron Skonnard, ‘Publishing/Discovering Web Services via DISCO/UDDI’, http://www.develop.com/conferences/conferencedotnet/materials/W4.pdf

38. Frank Sommers, ‘Web services take float with JAXR’,
http://www.javaworld.com/javaworld/jw-05-2002/jw-0517-webservices.html

39. Frank Sommers, ‘Supplement: The adventures of JWSDP’,
http://www.javaworld.com/javaworld/jw-05-2002/jw-0517-jwsdp.html

40. Frank Sommers, ‘I like your type: Describe and invoke Web services based on service type’,
http://www.javaworld.com/javaworld/jw-09-2002/jw-0920-webservices.html

41. Frank Sommers, ‘Publish and find UDDI tModels with JAXR and WSDL’,
http://www.javaworld.com/javaworld/jw-12-2002/jw-1213-webservices.html

42. Martin Streicher, ‘Creating Web Services with AXIS’,
http://www.linux-mag.com/2002-08/axis_01.html

43. Sun Microsystems, ‘Java Web Services Developer Pack 1.1’,
http://java.sun.com/webservices/webservicespack.html

44. Thuan Thai and Hoang Q. Lam, ‘.NET Framework Essentials (2nd edition)’,
O’Reilly, 2002, 0-596-00302-1.

45. Doug Tidwell, James Snell, Pavel Kulchenko, ‘Programming Web Services with SOAP’,
O’Reilly, 2001, 0-596-00095-2.

46. Paul Townend, ‘Web Service Security’,
http://www.dur.ac.uk/p.m.townend/webServiceSecurity.ppt

47. ‘uddi.org’,
http://www.uddi.org/

48. Venu Vasudevan, ‘A Web Services Primer’,
http://webservices.xml.com/pub/a/ws/2001/04/04/webservices/

49. Mark Volkmann, ‘Axis: an open source web service toolkit for Java’,
http://www.ociweb.com/javasig/knowledgebase/2002Sep/

50. W3C, ‘Simple Object Access Protocol (SOAP)’,
http://www.w3.org/2000/xp/Group/

51. W3C, ‘Web Services Activity’,
http://www.w3.org/2002/ws/

52. ‘WebServices.org’,
http://www.webservices.org/

53. ‘XMethods’,
http://www.xmethods.net/