RC 21979 (98778) 2 March 2001 **Computer Science**

IBM Research Report

Getting Started Using SOAP

James W. Cooper
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Getting Started Using SOAP

James W. Cooper

The Golden Oldies station was playing one more Eighties hangover: "Killing Me Softly..." when I cynically decided the words really should be "Killing Me Soffly With Hi Soap." Or maybe that should be SOAP.

SOAP is an incredible clever way to transmit data over a network using XML and HTTP, and it is just coming to people's attention as they begin to realize its power, especially in Java. Of course, the real power of SOAP, or Simple Object Access Protocol is that it can be cross language as well as cross platform. If it uses Java *and* XML it must be good. That's almost 100% buzz-word compliant

In this article, I'm going to show you how to install and use SOAP for Java to create mote procedure calls between a client and server. While SOAP represents objects using XML, we'll avoid getting bogged down in the details of XML, and instead concentrate on how to use SOAP, which itself uses XML.

The whole point of SOAP is that you can use it to transmit objects over a network in pure XML form. The objects arc assembled into XML on one machine and sent to another, where they arc then reconstructed. This process is called serialization and deserialization, and sounds quite a bit like Java RMI or some other kind of remote procedure call mechanism

SOAP can be used really nicely to send objects back and forth this way. And it has the great advantage that unlike RMI, the transmitting protocol is plain old HTTP. This means it will work between any two machines across any kind of network and through most firewalls without any problem.

The SOAP specification grew out of an initial proposal by Microsoft. However, it quickly became apparent that this initiative had very broad implications and a large number of companies contributed to the development of the ideas in SOAP. The final specification was written by scientists from Microsoft, IBM, Lotus, DevelopMentor and UserLand Software. In addition there were contributions from any number of other workers during the specification process.

One important contribution was the development of IBM SOAP for Java which was initially posted on IBM's AlphaWorks site. This was later further developed and donated to the Apache project, where SOAP is now developed and maintained. It is this Apache version we'll be discussing in this article.

Setting up SOAP on your machine is most of the battle. Once you get everything working, you can write really slick soap modules without much effort. So, in this article, I'll tell you how to get SOAP working on your machine, with an emphasis on Windows 2000/NT machines, although almost everything I say applies to Your Favorite Unix (YFU) as well. We'll end up writing our first SOAP program and you'll be on our way to a bright and bubbly future.

The Software You Need

You will need to download three packages to try out the SOAP system.

1. The Apache Tomcat JavaServer pages / Servlet system. Go to http://java.apache.org and click on Jakarta, then on Tomcat, and then on Download Builds. For Windows, you want Jakarta-tomcat-3.x.x.zip and for YFU, you'll want the tar or tar.gz versions.
2. The SOAP 2.0 libraries. Go to http://xml.apache.org and click on SOAP and Download. You want soap-bin-2.0.zip or later, if available.
3. The Xerces Java XML parser library. Go to http://xml.apache.org and click on Xerces Java and then on Download. You want Xerces-J-bin-1.2.3.zip or later.

Installing the Software

1. Installing Tomcat amounts to just unzipping it into a Tomcat directory. The default is jakarta-tomcat. There is nothing to install after you unzip. To start the JSP engine, you want to start the startup.bat file in the jakarta-tomcat\bin directory. I usually make a desktop shortcut to this batch file.
2. Unzip the SOAP zip file into a Soap directory and the Xerces zip file into a Xerces directory. They unzip into soap-2.0 and \xerces-1_2_3\tools, but I moved or renamed them to \soap and \xerces for simplicity.
3. Edit your CLASSPATH to include \soap\lib\soap.jar. Alternatively, you can just put soap.jar in the \Program Files\Javasoft\JRE\1.3\lib\ext directory and in the \jdk1.3\jre\lib\ext directory. If you want to run the provided examples, you also need to put \soap in your classpath. (Note that in Windows-2000, this whole thing is buried in Settings|Control Panel| System|Advanced|Environment Variables).
4. Edit the tomcat.bat file in your \tomcat\bin directory to add the path to Xerces to the *front* of the CLASSPATH line, so that Xerces is found first. This is necessary because Tomcat comes with an older XML parser that won't work. Line 38 of this file should be changed to:

```
set CLASSPATH=d:\xerces\xerces.jar;%CLASSPATH%;%cp%
```

For YFU, you will need to edit the startup.sh file, and this is described in the tomcat.html file in the soap documents install folder.

5. Make sure that you add the path to xerces.jar to the front of your actual classpath declaration as well.

```
CLASSPATH=d:\xerces\xerces.jar; ..etc.
```

6. In the Tomcat \conf\server.xml file, add the following Context lines near the bottom of the file just above the </ContextManger> line.

```
<Context path="/soap" docBase="d:\soap\webapps\soap"
         debug="1" reloadable="true">
</Context>
```

7. Now start the Tomcat server by running the startup.bat file.

8. Then, point your browser to http://localhost:8080/soap/ and you should see the Apache-SOAP startup screen shown in Figure 1.

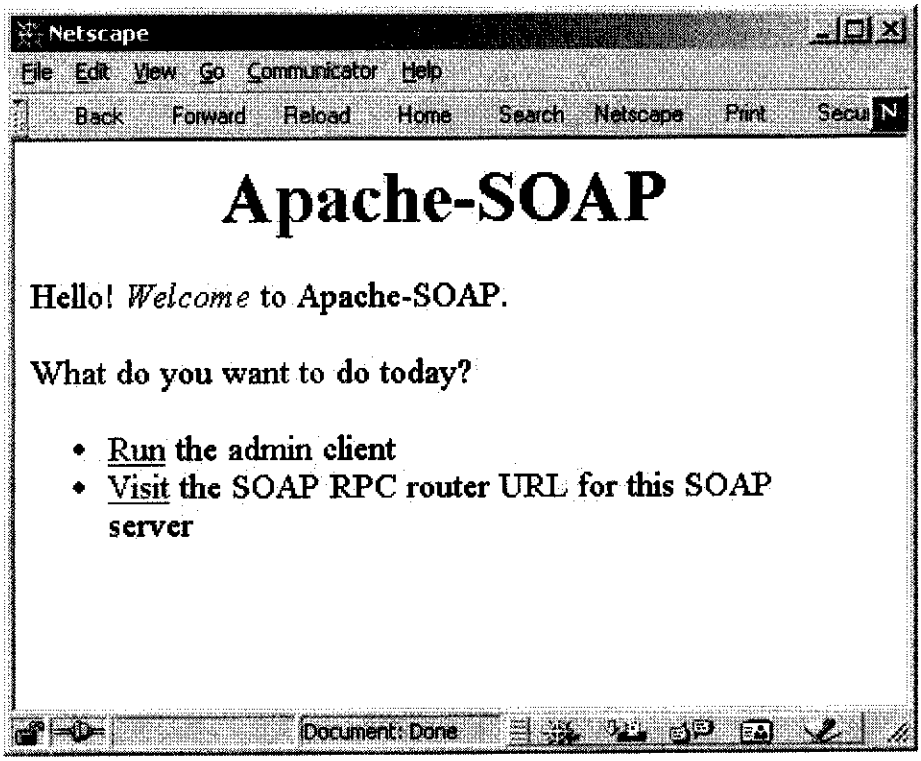

Figure 1 – The Apache-SOAP startup screen.

Running a Sample Application

You might think that you are now ready to run the sample applications, but nooo… First you have to deploy them. This simply means that you have to run a program that registers the names these applications use and puts them in a deployment descriptor file for Tomcat. If you go to the \soap\samples\AddressBook directory, you will find that there is a file called DeploymentDescriptor.xml.

To save typing, copy or rename this file to just dd.xml.

Rather than reading this file and trying to understand it, we can just run a program to deploy this application by typing all one line:

```
java org.apache.soap.server.ServiceManagerClient
      http://localhost:8080/soap/servlet/rpcrouter deploy dd.xml
```

Now, if you go to your browser Apache SOAP client display and click on "Run the admin client," and then on List, you will see that the AddressFetcher SOAP service has been deployed as shown in Figure 2.

Figure 2 – The AddressFetcher service has been deployed.

If you then click on this urn:AddressFetcher link, you will get a whole blizzard of deployment information. The most important parts of this are

```
ID:                 urn:AddressFetcher
Provider Class      samples.addressbook.AddressBook
Methods             getAddressFromName, addEntry,
                    getAllListings, putListings
```

This means that a service called AddressFetcher exists in the Java program AddressBook and that it has four public methods.

Now, let's see how we can call these. Let's try the GetAddress example program. If from the \soap directory we type

```
java samples.addressbook.GetAddress
```

we'll get a message explaining what we actually need to type:

```
Usage:
  java samples.addressbook.GetAddress [-encodingStyleURI] SOAP-router-
URL nameTo Lookup
```

This is also explained in the README file in that directory.

So what we need to type is

```
java samples.addressbook.GetAddress
  http://localhost:8080/soap/servlet/rpcrouter "John B. Good"
```

And if we do, we'll get a response:

```
123 Main Street
Anytown, NY 12345
(123) 456-7890
```

Great. So what's going on here? The program GetAddress is taking our argument "John B. Good" and sending it to the SOAP service AddressFetcher and getting back a response.

The code that does this can be found in the GetAddress.java example file, originally written by Matthew Duftler. A small part of that code is shown below.

```
Vector params = new Vector();
params.addElement(
  new Parameter("nameToLookup",
        String.class,
        nameToLookup, null));
call.setParams(params);
// Invoke the call.
Response resp;
try   {
  resp = call.invoke(url, "");
}
catch (SOAPException e)     {
  System.err.println(e.getMessage());
  return;
}
// Check the response.
if (!resp.generatedFault())      {
  Parameter ret = resp.getReturnValue();
  Object value = ret.getValue();
  System.out.println(value );
}
```

This provides a template for how to call a SOAP service and get a response. Now, with this in mind we could write our own. However, before doing that, it is helpful to see what is going on under the surface of these relatively calm waters.

The TCP Tunnel Tool

We can watch the messages going back and forth between the client and the SOAP service using a convenient tool that watches a pair of TCP/IP ports and shows data sent to them. What we then do, is to run this GetAddress program on a different port, say 8082 and then tell the tunnel tool to connect to that port and pass the data on to port 8080 after monitoring it. You start the tool from the command line by

```
java org.apache.soap.util.net.TcpTunnelGui 8082 localhost 8080
```

Then we start the GetAddress on 8082

```
java samples.addressbook.GetAddress
  http://localhost:8082/soap/servlet/rpcrouter "John B. Good"
```

and see the packets generated in the GUI as shown in Figure 3.

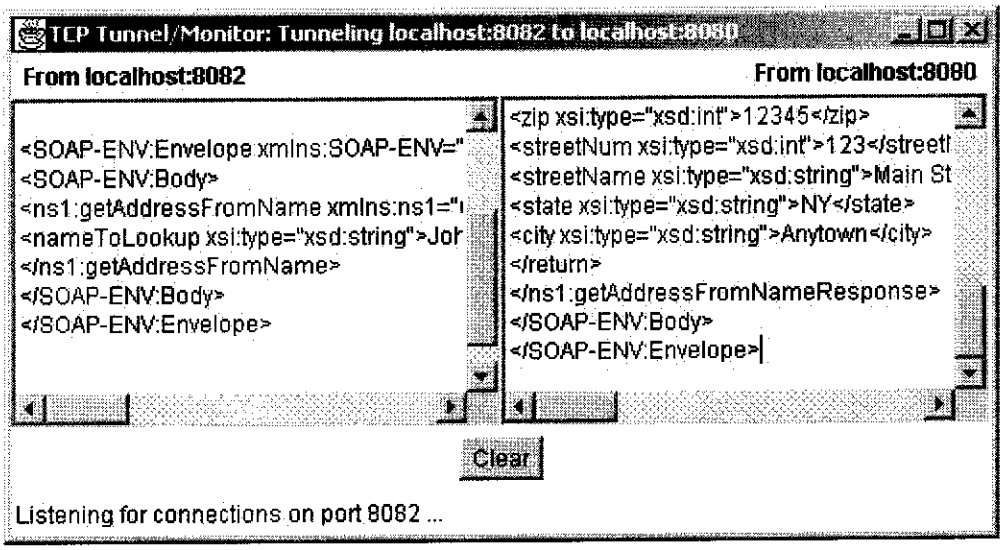

Figure 3 – The TCP Tunnel tool, showing the data from port 8082 and sent from 8080.

What's Going On?

Now we can see what is going on in a top-down sort of way. I didn't start by subjecting you to an XML tutorial, nor did I discuss the format of SOAP messages. Instead, we tried one and can just look at the results. On the left side of Figure 3, we see that the method getAddressFromName seems to have been executed. The packet sent out is

```
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<SOAP-ENV:Body>
<ns1:getAddressFromName xmlns:ns1="urn:AddressFetcher" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<nameToLookup xsi:type="xsd:string">John B. Good</nameToLookup>
</ns1:getAddressFromName>

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

We also see that the argument "John B. Good" appears under "nameToLookup. In the right-hand pane of Figure 3, we see the response. Part of it is excerpted below:

```
<SOAP-ENV:Body>
<ns1:getAddressFromNameResponse xmlns:ns1="urn:AddressFetcher" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xmlns:ns2="urn:xml-soap-address-demo" xsi:type="ns2:address">
<phoneNumber xsi:type="ns2:phone">
 <exchange xsi:type="xsd:string">456</exchange>
 <areaCode xsi:type="xsd:int">123</areaCode>
 <number xsi:type="xsd:string">7890</number>
</phoneNumber>
```

```
<zip xsi:type="xsd:int">12345</zip>
<streetNum xsi:type="xsd:int">123</streetNum>
<streetName xsi:type="xsd:string">Main Street</streetName>
<state xsi:type="xsd:string">NY</state>
<city xsi:type="xsd:string">Anytown</city>
</return>
```

In other words, the phone number, address, street name, state and town are passed back as elements in an XML structure which we don't actually have to understand to use.

Writing our own SOAP Client and Service

Well the examples are all very nice, but we don't own it if we can't write it ourselves. So, let's write a program to get some data object from a server and learn the other half of this process.

The way SOAP works is that objects are converted to XML using a process called *serialization* and converted back to objects using *deserialization.* The SOAP package has a number of serializer classes for various purposes, but we will concentrate here exclusively on the BeanSerializer class to do these conversions. In order for the BeanSerializer to work every parameter has to have a get and set method, just as a Bean does in other contexts. In fact, under the covers, deserialization consists of creating a new instance of the class on the client side and calling its *set* methods to put the variables back into it. Thus, there really has to be one set method for each variable you want to carry into the client. In addition, all such classes have to have a constructor without arguments, since the deserializer creates an instance calling this constructor and then calls the set methods.

Let's consider a simple class to hold an x-y data point pair.

```
//Represents one x-y data point pair
public class DataPoint {
    private int x, y;
    //must have an empty constructor
    //for the Bean deserializer to work
    public DataPoint() {
        x=0;
        y=0;
    }
    //use this constructor when we first create it
    public DataPoint(int x_, int y_) {
        x = x_;
        y = y_;
    }
    //get and set for x and y values
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    public void setX(int x_ ) {
        x = x_;
```

```
    }
    public void setY(int y_) {
        y = y_;
    }
}
```

The only reason we don't use the java.awt.Point class is that it doesn't have the appropriate get and set methods.

Now if all we're going to do is transfer a single point pair, this is probably overkill. So, let's create another class which holds an array of these point pairs. It too will have bean-like set and get methods.

```
public class XYData {

    private DataPoint[] v;
    public XYData() {
    }
    //get the array out of the class
    public DataPoint[] getData() {
        return v;
    }
    //set some data back into the class
    public void setData(DataPoint[] vc) {
        v = vc;
    }
}
```

So now, we have a class XYData with a getData method that returns an array of DataPoint objects. We want to have the SOAP service call this some object on the server and return an instance of that data. We call that class the XYGetter class. All it does is create and return the data on request: Note that we create a fixed-size array of DataPoint objects here. While in theory you could also create a Vector here, the current SOAP implementation does not deserialize that class correctly, although it does handle arrays.

```
//called by SOAP service
public class XYGetter {
    private XYData xydata ;
    private DataPoint[] v;
//--------------------
   public XYGetter() {
       xydata = new XYData();  //create data class
       //fill it with data
       v = new DataPoint[4];
       v[0] = new DataPoint(10,20);
       v[1] = new DataPoint(30,200);
       v[2] = new DataPoint(50,100);
       v[3] = new DataPoint(70,90);
       xydata.setData (v);      //put in class
     }
  //--------------------
   //return data to soap service
    public XYData getXY() {
        return xydata;
    }
 }
```

Writing Our SOAP Client

Now we finally have to deal with writing the client that requests these data. Remember, we said that the transport mechanism for SOAP is most commonly HTTP. So we actually connect to our SOAP service using what looks like an ordinary URL. However, the definition of these identifiers has been expanded to be more than just an HTTP address, but an identifier in the SOAP deployment registry, so we refer to it as a URI for Uniform Resource Identifier.

While we would normally connect to the Tomcat JSP engine through port 8080, we make this a variable so that we can use another port and watch things happen using the TCP tunnel program if we want to. So we start out be creating an instance of the URL class to connect to the URI for the RPC router.

```
public class sTest {
    private String port;
    public sTest(String portNum) {
        URL url = null;
        port = portNum;
        //create the URI to connect to
        String encodingStyleURI =  Constants.NS_URI_SOAP_ENC;
        try {
            url = new URL("http://localhost:"+ port+
                  "/soap/servlet/rpcrouter");
        } catch (MalformedURLException e) {
            System.out.println("Bad url");
        }
```

Then, for each object we want to transmit other than the simple, base Java objects, we have to tell the serializer how to handle them:

```
        //map the XYData and DataPoint objects
        SOAPMappingRegistry smr = new SOAPMappingRegistry();
        BeanSerializer beanSer = new BeanSerializer();

        smr.mapTypes(Constants.NS_URI_SOAP_ENC,
                     new QName("urn:xy-demo", "xydata"),
                     XYData.class, beanSer, beanSer);
        smr.mapTypes(Constants.NS_URI_SOAP_ENC,
                     new QName("urn:xy-demo", "point"),
                     DataPoint.class, beanSer, beanSer);
```

In the above code, we create a namespace for our program called "xy-demo," and indicate that this is a URN or Uniform Resource Name. This is the namespace for this program. We also create names for the two objects in this namespace, calling one "xydata" and the other "poi nt." We map these types as shown above and indicate that they are serialized using the BeanSerializer.

Then we create the call to the SOAP service and invoke it. The specific call we want to invoke has the name "urn:soapGetxy." We'll see below how we give the service code this name when we deploy it.

```
        // Build the call.
        Call call = new Call();

        call.setSOAPMappingRegistry(smr);
```

```
//this is the object we are calling
call.setTargetObjectURI("urn:soapGetxy");
//and this is the method we will call
call.setMethodName("getXY");
call.setEncodingStyleURI(encodingStyleURI);

Response resp;

try {
    //invoke the call
    resp = call.invoke(url, "");
} catch (SOAPException e) {
    System.err.println(e.getMessage());
    return;
}

// Check the response.
if (!resp.generatedFault()) {
    Parameter ret = resp.getReturnValue();
    //get the data and print out the result
    XYData xydata = (XYData)ret.getValue();
    DataPoint[] v = xydata.getData();
    System.out.println("size="+v.length);

    for (int i=0; i < v.length ; i++) {
        DataPoint p = v[i];
        System.out.println(p.getX ()+" "+p.getY ());
    }
}
```

You might think that we can just compile and run this program, but noooo.. We have two more things we have to do.

Deploying Our Service

Just as we had to deploy the sample Addressbook code above, we have to deploy this service, telling the SOAP deployment system that there is a SOAP service named "urn:soapGetxy," and what Java class it has to use and what method it has to call. We could do this using an XML file for the deployment descriptor, except that these things are hard to write. Fundamentally, XML is ASCII machine readable text, it really isn't that human readable, and it can be hellish to type in without errors. So instead, the Apache SOAP system has conveniently provided us with a screen where we can type in the values to deploy a new service.

If you go to the same Apache-SOAP panel we had in Figure 1 and click on the Administrative Client and then on the Deploy button, you will get a screen to fill in like that shown in Figure 4. Enter the ID for the service to match the name in the program we just wrote, in this case "urn:soapGetxy." Make the scope Application wide and specify that the Provider Type is Java and the Provider class is XYGetter.

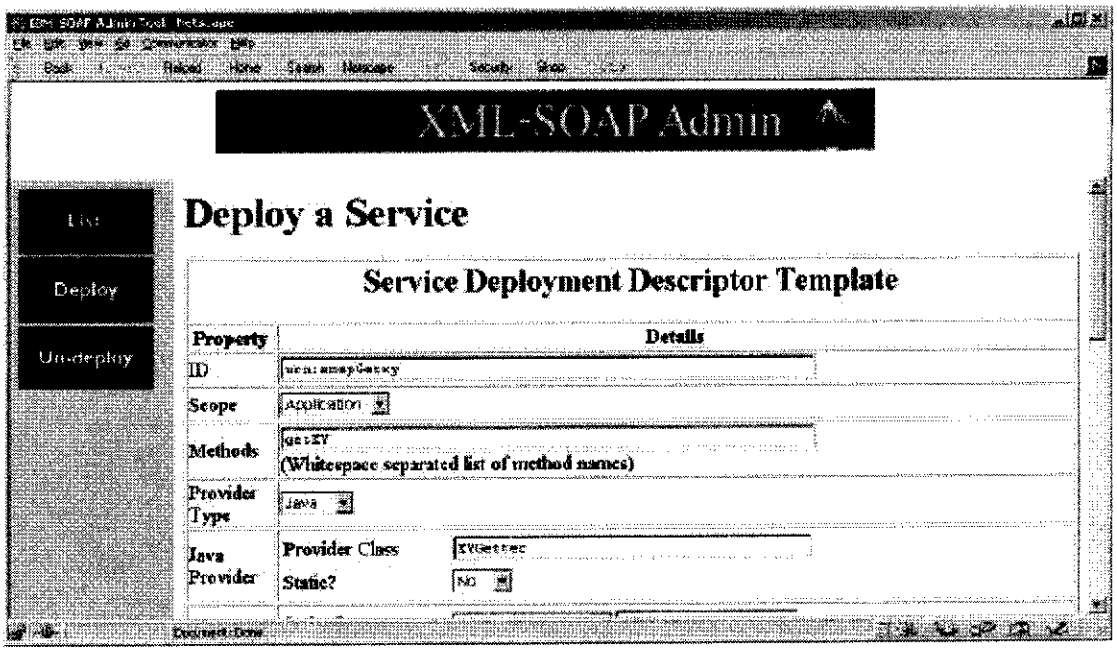

Figure 4 – Deploying a new SOAP service

We also have to specify each of the classes we want to serialize here just as we did in the program. The bottom of the deployment screen is shown in Figure 5.

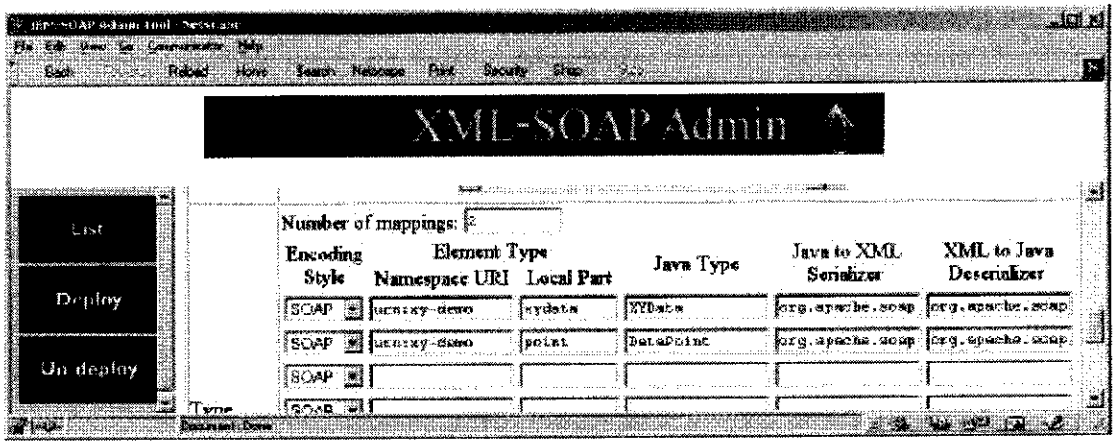

Figure 5 – Registering mappings of two classes

The Namespace URI is urn:xy-demo, since this is the name we used in declaring our classes in the Java program above. The Local Part is the names we gave these classes, here "xydata" and "point." The Java Type fields are the names of the actual classes. And finally, all 4 of the serializer fields are filled with

```
org.apache.soap.encoding.soapenc.BeanSerializer
```

which is the complete package name to the BeanSerializer class.

We also must fill in the Number of Mappings as 2 in this interface so that these two lines are copied into the deployment descriptor.

Finally, to deploy this, it is *very important* that you scroll to the bottom of this window and click on the Deploy *Button*.

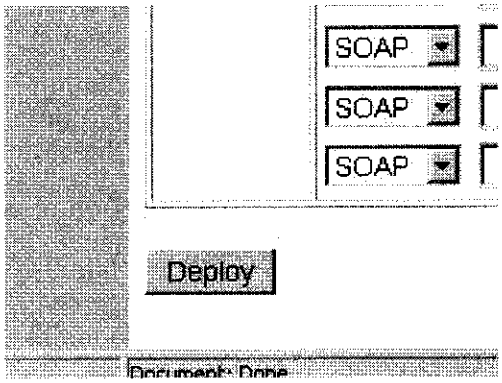

Figure 6 – The Deploy button you are to click on to deploy this SOAP service

DO NOT click on the pretty Deploy icon to the left, as this will erase your work and give you a new screen to fill in from scratch.

Making the Classes Available

Finally, before all this will work, you have to make sure the classes we have created are all in your CLASSPATH so that can be found by the SOAP system and by the deserializer. I think the easiest way to do this is to package them all up in a jar file by going to the project folder \soapdemo and typing on the command line

```
jar -c0f myobjects.jar *.class
```

This will put all the class files in your \soapdemo project directory into the class file. I found that I changed the base sTest.java file a lot after I got the other classes done, so I opened the jar file with WinZip and deleted the sTest.class file.

Then you either have to put the path to this jar file in the CLASSPATH declaration in the Control Panel, or you need to copy the jar file to the two \jre\lib\ext directories.

Running Our Program

Now we're finally ready to try out this program. It's been glued, screwed, tattooed, deployed and annoyed. We can start it by typing

```
java sTest 8082
```

and watching the drama unfold in the windows of the TcpTunnel program.

The calling program writes out

```
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<SOAP-ENV:Body>
<ns1:getXY xmlns:ns1="urn:soapGetxy" SOAP-
```

```
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
</ns1:getXY>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

and the sending program writes back, in part,

```
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<SOAP-ENV:Body>

<ns1:getXYResponse xmlns:ns1="urn:soapGetxy" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xmlns:ns2="urn:xy-demo" xsi:type="ns2:xydata">
<data xmlns:ns3="http://schemas.xmlsoap.org/soap/encoding/"
xsi:type="ns3:Array" ns3:arrayType="ns2:point[4]">
<item xsi:type="ns2:point">
<x xsi:type="xsd:int">10</x>
<y xsi:type="xsd:int">20</y>
</item>

<item xsi:type="ns2:point">
<x xsi:type="xsd:int">30</x>
<y xsi:type="xsd:int">200</y>
</item>
```

The actual result printed to the console is

```
java sTest 8082
size=4
10 20
30 200
50 100
70 90
```

so *voila!* it works. Or maybe *viola*, since we really had to orchestrate all this.

Conclusions

It certainly takes a bit of doing to set up a SOAP service the first time, as you've seen, but after that they're really quite easy to add to. SOAP provides a really light-weight, portable, cross-platform way of exchanging objects. It can replace the heavier weight RMI in many cases and will work across networks and through firewalls. We'll look at some further uses of SOAP in months to come.