Soaping Your Windows

toc: SOAP is an easy way to communicate between Java applets and servers

deck: XML may be a write only language, but SOAP will read and write it for you.

James W. Cooper

Recently, a colleague sent me a document containing specifications for how information needed to be transmitted between various clients and servers as an XML schema. Well… it looked all right, but please don't ask me to be the XML validator: they have programs that do that. In fact, the more I looked at the spec, the more I realized that XML is YAWOL : Yet Another Write-Only Language. It's like APL or even perl—too hard to read for what you can get out of it.

But don't for a minute take this to mean that I don't think XML is really useful and valuable. I'm making more and more use of it all the time. But I'm using Java to read and write this mouthful and give me the parts I want without ever having to digest the whole thing.

This is, of course, exactly what SOAP is for. It provides and XML schema for representing the contents of Java objects, so you can transmit them over a network between programs and platforms without understanding much of how this is done. A couple of years ago, I wrote glowingly of how Java RMI gave you a simple method to transmit objects between Java applets and Java servers. but it turned out not to be a heavily used protocol because it was too hard to use when firewalls were in the way, and because it was *only* between Java programs.

SOAP is different. You can use a Java SOAP serializer to convert an object to an XML data stream and send it off to any program that can reserialize it back into an object. That program could be in Java, C, or even Visual Basic (when Microsoft's Dot Net Studio ships).

Last month I showed you how a Java application could communicate with another using SOAP. This month, I'll show you the rest using an applet.

Let's suppose we want to write a program to send an array of data points from a server to client so we can plot them. Any object we want to send between two programs using SOAP must be serialized. While it isn't impossibly difficult to write a serializer you can use SOAP's BeanSerializer if you can make each object behave like a JavaBean. Bean-like objects must have get and set accessor methods for each private variable they contain. As usual for an object, it doesn't matter how the data are actually stored within the object, or whether additional computations are done in the get and set methods, as long as each variable can be set directly or through computation using these accessor methods. Now, as we noted last month, the java.awt.Point object has X and Y public variables, but since they aren't get and set accessors, you can't use them here. Instead, we create a DataPoint object that has these characteristics:

```
//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 XML representation of an instance of a DataPoint object is just

```
<item xsi:type="ns2:point">
  <x xsi:type="xsd:int">70</x>
  <y xsi:type="xsd:int">90</y>
</item>
```

Now, if we want to get a whole array of these in a single query, we ought to make an XYData class that sets and gets arrays of DataPoint objects.

```
public class XYData {
 //transmits a whole array
 //of DataPoint objects
    private DataPoint[] v;

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

The XML representation of the whole array consists of a header that says it is an array, and an series of <item> entries for each DataPoint in the array.

```
<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>
```

This is what we want to transmit using SOAP, and the BeanSerializer and deserializer create all this XML for us.

Setting Up Your Server

The client and server could in fact be the same machine for this simple test. I found that if my Windows-2000 machine was not connected to my network and had been rebooted since disconnection, that there were some fairly long delays that may have been caused by some TCP/IP timeouts somewhere under the covers. However, I developed the client and server on my laptop and you can too.

The server side needs to have the tomcat JavaServer Pages toolkit installed. Remember that tomcat is the Apache server written by Sun and donated to the Apache project. You will find links to it on both the Sun and Apache pages.

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. You have to restart it any time you change server code: it does not detect and reload updated code as more robust commercial servers do.

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%
    ```

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>
```

Writing Your Server Program

You want to write a program that delivers a populated XYData object on request using the SOAP RPC mechanism. Actually, all you have to do is write the Java class. The SOAP RPC router servlet will launch it and serialize the XYData object. The entire class is

```
//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;
    }
 }
```

To tell SOAP that this class is available to be launched and what it will serialize, you need to register it. The easiest way to do this is using the SOAP administration application you can get to at http://localhost:8080/soap/.

And, just as we did in last month's column, you register this server class by filling in the fields to deploy a new service. Click on "Run the admin client" and then on the Deploy icon. Then you fill in the fields as follows:

1. Enter an ID for the service, in this case "urn:soapGetxy." Make the scope Application wide and specify that the Provider Type is Java and the Provider class is XYGetter.

2. We also have to specify each of the classes we want to serialize here.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."

3. 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.

4. 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.

5. Then, click on the Deploy *button* at the bottom of the screen.

You have just deployed the soapGetxy service with two classes, nicknamed "xydata" and "point." Now we can write the client to call them.

The SOAP rpcrouter servlet can only run this program and serialize the classes if they can be found. This means that you need to put them in your class path or in the extensions directory. I usually put them all in a jar file in the c:\Program Files\JavaSoft\jre\1.3\lib\ext directory.

Writing the Client Applet

If we normally avoid a lot of client Java in favor of using JavaServer pages, we might ask when we might actually need Java. One case I can think of is when you want to display some graphics that simply cannot be rendered in HTML. Here Java is wholly appropriate and the only solution. In fact, having JSPs that launch Java applets for these cases is a pretty good technique.

To write your client applet, you need to describe these same objects to the SOAP system

```
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);
```

and then make the call to the SOAP system:

```
Response resp;
    try {
        resp = call.invoke(url, "");
    } catch (SOAPException e) {
        System.out.println("Soap exception: " + e.getMessage());
        return;
    }

    // Check the response.
    if (!resp.generatedFault()) {
        Parameter ret = resp.getReturnValue();
        XYData xydata = (XYData)ret.getValue();
        v = xydata.getData();
        plotData(v);        //plot the results
    } else {
        Fault fault = resp.getFault();
        System.out.println ("Fault: "+ fault.getFaultString());
    }
```

This returns an array of DataPoint objects in the variable v. We can plot them by finding the data scale for plotting:

```
private void plotData(DataPoint[] v) {
 //scales the data for plotting
    xmin = ymin = Integer.MAX_VALUE ;
    xmax = ymax = Integer.MIN_VALUE ;
```

```
        for(int i=0; i < v.length ; i++ ) {
            DataPoint p = v[i];
            int x = p.getX ();
            int y = p.getY ();
        //find maxima and minima
            if(x > xmax) xmax = x;
            if(x < xmin) xmin = x;
            if(y > ymax) ymax = y;
            if(y < ymin) ymin = y;
         }
        //compute scaling factors
        xscale = (xmax - xmin) * 1.1f/ plotPanel.getWidth ();
        yscale = (ymax - ymin) * 1.1f/ plotPanel.getHeight ();
        xoffset = (xmax - xmin) * 0.05f;
        yoffset = (ymax - ymin) * 0.05f;
        dataReady = true;  //ready to draw
        repaint();
}
```

Then our paint routine can draw the lines between the xy data point pairs:

```
public void paint(Graphics g) {
    super.paint(g);
    if (dataReady) {
        for (int i=0; i < v.length -1; i++) {
            DataPoint p = v[i];
            int x = (int)((p.getX ()-xmin) / xscale +xoffset);
            int y = (int)((p.getY ()-ymin) / yscale + yoffset);
            DataPoint p1 = v[i+1];
            int x1 = (int)((p1.getX ()-xmin) / xscale +xoffset);
            int y1 = (int)((p1.getY ()-ymin) / yscale + yoffset);
            g.drawLine (x,y,x1,y1);
        }
    }
}
```
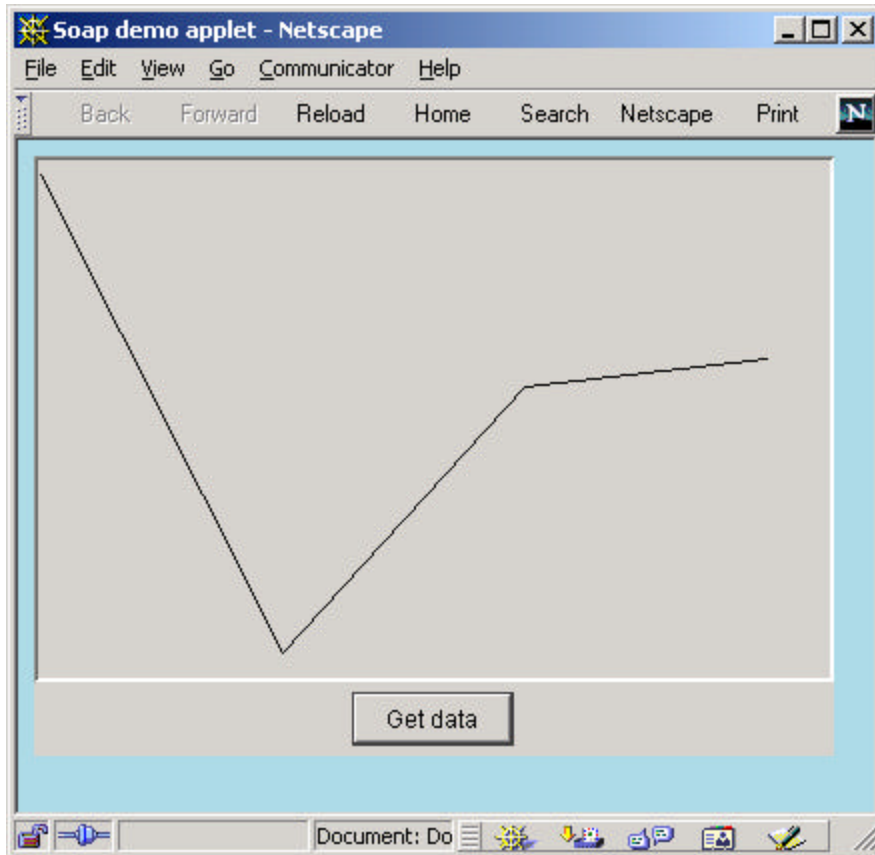The resulting applet is show in Figure 1.

Figure 1- The client applet showing how we plot data point pairs.

Opening the Jars

Now the client applet needs to have access to the applet code and to all the classes you will want to serialize, as well as to the XML parser and the SOAP library classes. Since we are going to be making Java 2 calls here, we will need to use the Java plugin to make sure that any browser (IE or Netscape) can read find the plugin and get the necessary jar files.

I find this all very confusing syntactically, so I write an old-fashioned applet that refers to the archive files in the applet tag:

```
<HTML>
<TITLE >Soap demo applet</TITLE>
</HEAD>
<BODY bgcolor="LIGHTBLUE" >
<APPLET CODE = "sTest.class"
  WIDTH = 400 HEIGHT = 300
  archive="xerces.jar, soap.jar, myobjects.jar">
<PARAM NAME = "port" VALUE ="8082">
</APPLET>
</BODY></HTML>
```

Then I just run the Java HTMLConverter program to get it in the right form for the plugin. This gives us:

```
<HTML>
<TITLE >Soap demo applet</TITLE>
</HEAD>
<BODY bgcolor="LIGHTBLUE" >
<OBJECT classid="clsid:131ABCDE-3110-11d4-991C-005004D3B3DB"
  WIDTH = 400 HEIGHT = 300
  codebase="http://java.sun.com/products/plugin/1.3.1/jinstall-131-
win32.cab#Version=1,3,1,0">
<PARAM NAME = CODE VALUE = "sTest.class" >
<PARAM NAME = ARCHIVE
  VALUE = "xerces.jar, soap.jar, myobjects.jar" >

<PARAM NAME="type" VALUE="application/x-java-applet;jpi-version=1.3.1">
<PARAM NAME="scriptable" VALUE="false">
<PARAM NAME = "port" VALUE  ="8082">
<COMMENT>
<EMBED type="application/x-java-applet;jpi-version=1.3.1"
CODE = "sTest.class" ARCHIVE = "xerces.jar, soap.jar, myobjects.jar"
WIDTH = 400 HEIGHT = 300 port = "8082"
scriptable=false
pluginspage="http://java.sun.com/products/plugin/1.3.1/plugin-
install.html">
<NOEMBED>

</NOEMBED>
</EMBED>
</COMMENT>
</OBJECT>

</BODY></HTML>
```

Note that by making the TCP port a parameter, we can use the TCP TunnelGUI to watch the 8080 port, just as we did last month.

You have to be a little careful in setting this all up. If you put the client and server on the same machine, the SOAP applet will run even without having the archive tag include th xerces.jar and soap.jar. However, it won't work across a network, so you need to try it that way to make sure.

Remember, the class path is used for the server code and the archive tags for the client code. The file myobjects.jar includes the sTest.class, and the DataPoint, XYGetter and XYData classes.

Wiping Up the SOAP

So now we've seen how to use SOAP in two client-server environments using the BeanSerializer. It's very easy to use in this environment and provides a light-weight fireall-pproof way to transmit objects. There's a lot more you *can* do, and we'll write more another time.

For a future column, I'd be interested in hearing about some of the *worst* Java code you've ever seen. Maybe we can learn from it.