Distributed Systems, Laboratory
3. Further RPC

Introduction

We have seen with RMI that RPC gives huge potential for distribution transparency. Other forms of RPC or RPC based protocols such as NFS allow applications to be designed with little change from the non-distributed model.

However a few things hold back RMI from large scale or widespread implementation. RMI is purely Java based. All components in the distributed system must have a layer of Java between application systems and the network. This is not a problem for applications written in Java however for legacy systems or heterogenous distributed systems this could mean either too much work to implement or a design alteration for the whole system. Not many people want to redesign parts of their system.

So we need a way of providing RPC, however we must find a way so every computing system can integrate easily into the distributed systems layer. One transport common to many systems is the http protocol and the provision for plain text transfer. From this we can define representations of common data types for many programming languages and facilitate method calls and reponses. Packaging data into text structures can be acheived with a markup language. XML like HTML is a simple standard for defining documents and formatting.

XML RPC

XML-RPC is a specification for passing method data via a stream of text. We can view the entire specification here: `http://www.xmlrpc.org/spec`. The most notable parts of the specification are the data representation definitions, replicated here.

Scalar type	Representation	Example
four-byte signed integer	<`int`>	-12
0 (false) or 1 (true)	<`boolean`>	1
String	<`string`>	Hello Rob
Double precision signed float	<double>	-12.214
Date	<dateTime.iso8601>	19980717T14:08:55
Binary (base64)	<base65>	eW91IGNhbid0IHJ=

Also in XML mark-up we can represent array data and also structured data. An example of each, taken from the specification are shown below:

```
<array>
   <data>
      <value><i4>12</i4></value>
      <value><string>Egypt</string></value>
      <value><boolean>0</boolean></value>
      <value><i4>-31</i4></value>
      </data>
   </array>


<struct>
   <member>
      <name>lowerBound</name>
      <value><i4>18</i4></value>
      </member>
   <member>
      <name>upperBound</name>
      <value><i4>139</i4></value>
      </member>
   </struct>
```

Request-Response

We can see an entire transaction take place accross an HTTP protocol system. This example makes a request to a remote object for the name of a US state, given its state number.

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181

<?xml version="1.0"?>
<methodCall>
    <methodName>examples.getStateName</methodName>
    <params>
        <param>
            <value>
                <i4>41</i4>
            </value>
        </param>
    </params>
</methodCall>
```

Here we observe the response back to the client.

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT

<?xml version="1.0"?>
<methodResponse>
    <params>
        <param>
            <value>
                <string>South Dakota</string>
            </value>
        </param>
     </params>
</methodResponse>
```

XML-RPC in Java

From these standard data type definitions there is a representation in many common programming language. We can see the mapping from XML-RPC standard data type to Java data type here:

xml-rpc data type	Java data type equiv.
<i4> or <int>	java.lang.Integer
<boolean>	java.lang.Boolean
<string>	java.lang.String
<double>	java.lang.Double
<dateTime.iso8601>	java.util.Date
<struct>	java.util.Hashtable
<array>	java.util.Vector
<base64>	byte[ ]

Like RMI we do not have to start from scratch when implementing an xml-rpc based system. We can use a free API library designed and built by the web services project at the apache software foundation. The homepage for this can be found here: `http://ws.apache.org/xmlrpc/`.

We will work through a simple example, again based on calculator arithmetic. We will start with the Server.

Like the RMI system, xml-rpc has a unified place in which remote objects can be registered in order to be located easier. Unlike RMI however, xml-rpc uses the

HTTP protocol and so may be used through a web server which supports XML-RPC. Our library however comes with a basic web server which we can use for our object's transport. Like the remote object in RMI, the XML-RPC object has a super type which we can extend to provision the basic capabilities of a remote object. This class is the `org.apache.xmlrpc.XmlRpcHandler` Interface. Notice the use of the term *handler* - meaning once again this object just does the workload of the request and is not concerned with location or transport. This interface is very simple and the only method we are required to implement is

```
public Object execute(String method, Vector params)
```

This very broad method is called by the transport provider upon request. Although broad we can take full control of what happens in this object. Notice the `java.lang.Object` return type means we can return *any* java object from the method. The parameters are also packed into a `java.util.Vector`. We are without a common interface in this simple example. It is up to us to decide what to do based on the method name string supplied.
We will define some methods named `MyXmlRpc.add` and `MyXmlRpc.sub`. We must unpack the vector to give us some valid input parameters and perform some calculations. Our resultant handler looks like this:

```
import org.apache.xmlrpc.XmlRpcHandler;
import java.util.Vector;

public class MyXmlRpcHandler implements XmlRpcHandler
{
     Integer Z;

     public Object execute(String method, Vector params)
     {
          Integer X = (Integer)params.elementAt(0);
          Integer Y = (Integer)params.elementAt(1);

          int x = X.intValue();
          int y = Y.intValue();


          if(method.equals("MyXmlRpc.add"))
          {
               Z = new Integer(x+y);
          }
          else if(method.equals("MyXmlRpc.sub"))
          {
               Z = new Integer(x-y);
          }
          else if(method.equals("MyXmlRpc.mult"))
          {
               Z = new Integer(x*y);
          }
          else
          {
               Z = new Integer(0);
          }

          return Z;
     }
}
```

The service is started using another application which, in this case starts a simple web server and creates an instance of our handler object. Our server application look like this:

```
import org.apache.xmlrpc.WebServer;

public class MyXmlRpcServer
{
     public static void main(String[] args)
     {
          MyXmlRpcHandler handler = new MyXmlRpcHandler();

          WebServer ws = new WebServer(Integer.parseInt(args[0]));

          ws.addHandler("MyXmlRpc", handler);

          ws.start();
     }
}
```

The client is also fairly simple, gievn the library does all the work for us. A simple class for us to use is `org.apache.xmlrpc.XmlRpcClientLite`. This class takes the url of the webserver as a parameter and has a public method the same as the Handler interface. So all we need to do is package up some data as parameters for our remote methods and also provide the remote method name as a string. Our client look like this.

```
import org.apache.xmlrpc.XmlRpcClientLite;
import java.util.Vector;

public class MyXmlRpcClient
{
     public static void main(String[] args)throws Exception
     {
          String host = args[0];
          String port = args[1];
          String handler = "MyXmlRpc";

          Integer X = new Integer(Integer.parseInt(args[2]));
          String method = handler+"."+args[3];
          Integer Y = new Integer(Integer.parseInt(args[4]));

          String url = "http://"+host+":"+port+"/RPC2";

          XmlRpcClientLite client = new XmlRpcClientLite(url);

          Vector params = new Vector();
          params.add(X);
          params.add(Y);

          Integer Z = (Integer)client.execute(method, params);

          System.out.println(args[2] + " " +
                             args[3] + " " +
                             args[4] + " = " +
                             Z.intValue());
     }
}
```