XML-RPC

XML-RPC is a protocol that uses XML to describe method calls and method results, and a collection of implementations that allow software running on disparate operating systems, running in different environments (different programming languages) to make method calls to each other over the internet.

RPC stands for Remote Procedure Call.

Other Solutions to this Problem

CORBA

- Common Object Request Broker Architecture
- Designed for interoperability between different languages as well as different machines.

DCOM or COM

- Distributed Component Object Model
- Interoperability between different machines and languages as long as they are Wintel.

RMI

- Remote Method Invocation (Java only).
- Interoperability for any machines that are running the Java virtual machine.

These solutions are complex and opaque, relying on particular platforms or programming languages, or dependent on a complicated methodology for describing data types.

Properties of XML-RPC

- Designed to be as simple as possible, solving only part of the problem, but solving the most important part.
  Nonprogrammers can write XML-RPC calls.

- Encodes messages for calling methods and describing the results of method calls with a standard vocabulary using XML.

- Has a limited vocabulary of XML tags for describing the types of parameters and the type of a return value from a function.

- Uses HTTP as the transport over the internet.

- HTTP is normally used for person-to-person communication between a browser and a server using HTML.

- XML-RPC uses HTTP for computer-to-computer communication with no browser involved.

- Using HTTP allows XML-RPC communication to bypass firewalls.

- Many implementations of XML-RPC servers and XML-RPC clients are available currently.

- See www.xmlrpc.com for more information.

XML-RPC Lifecyle

Client Side

1. Client builds an XML element *methodCall* that names the method to be called and provides the actual parameters for the method.
2. Client sends a POST request whose payload (content) is the XML element just built.

Server Side

3. Server receives the request and uses the HTTP header Content-Length to read the payload XML element.
4. Server parses the XML element, extracts the method name, and retrieves the actual parameters for the method from the XML element.
5. Server searches for the desired method and, if found, invokes it with the given parameters.
6. If the method is called and executes successfully, the server packages its return value in an XML element *methodResponse* with a *params* element and sends it back to the client.

7. If the method is not found or cannot be executed for some reason, the server builds an XML element *methodResponse* whose *value* element contains a *fault* element and sends it back to the client.

8. Client receives the response, parses the XML element returned (a return value or *fault*), and reports the outcome to the client user.

Data Types in XML-RPC

Integers (32-bit values)

Elements: *int* or *i4*

Examples: <int>3928</int>
<i4>-703</i4>

Floating-point Values

Element: *double*

Examples: <double>5.489</double>
<double>-877.23</double>

Boolean Values

Element: *boolean*

Examples: <boolean>0</boolean>
<boolean>1</boolean>

Strings (ascii characters)

Element: *string*

Examples: <string>This is a short string.</string>
<string>May need entity references.</string>

Date and Time

Element: *dateTime.iso8601*

Format: ccyymmddThh:mm:ss

Example:
```
<dateTime.iso8601>
   20061208T15:04:45
</dateTime.iso8601>
```

Binary Data

Element: *base64*

Example:
```
<base64>
   VGhpcyBpcyBhIHNob3J0IHN0cmluZy4K
</base64>
```

Arrays

Element: *array*

Example:
```
<array>
   <data>
      <value>
         <double>43.7</double>
      </value>
      <value>
         <double>91.3</double>
      </value>
      <value>
         <double>18.8</double>
      </value>
      <value>
         <double>33.6</double>
      </value>
   </data>
</array>
```

Structures or Records

Element: *struct*

Format: An unordered list of *member* elements, each of which contains a *name* element and a *value* element.

Example:

```
<struct>
   <member>
      <name>spots1</name>
      <value><int>5</int></value>
   </member>
   <member>
      <name>spots2</name>
      <value><int>9</int></value>
   </member>
   <member>
      <name>faceUp</name>
      <value><boolean>1</boolean></value>
   </member>
</struct>
```

Correspondence with Types in Java

XML-RPC Type	Java Type
int and *i4*	**int**
double	**double**
boolean	**boolean**
string	String
dateTime.iso8601	?
base64	**byte** [] or String
array	array object
struct	Map object

Method Calls

A method call is defined by the *methodCall* element, which contains two elements, *methodName* and *params*.

The *params* element contains zero or more *param* elements that specify the actual parameters to be passed to the method.

Example

Suppose we have a Java method

double compute(**int** num, String str, **boolean** bn)

that we want to call, say *compute(96, "A Java string", **false**)*

XML Equivalent

```
<?xml version="1.0"?>
<methodCall>
   <methodName>compute</methodName>
   <params>
      <param>
         <value>
            <int>96</int>
         </value>
      </param>
      <param>
         <value>
            <string>A Java string</string>
         </value>
      </param>
      <param>
         <value>
            <boolean>0</boolean>
         </value>
      </param>
   </params>
</methodCall>
```


Method Responses

A method returns at most one value.

That value is defined as a single parameter inside a *params* element inside a *methodResponse* element.

Example

Suppose the compute method returns the value 16.25.

XML Code

```
<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value>
            <double>16.25</double>
         </value>
      </param>
   </params>
</methodResponse>
```

If the method cannot be executed for some reason, say it can not be found, the *methodResponse* element will contain a *fault* element with a *value* element that has a *struct* element to describe the mistake.

XML Code

```
<?xml version="1.0"?>
<methodResponse>
   <fault>
      <value>
         <struct>
            <member>
              <name>fault</name>
              <value><int>99</int></value>
            </member>
```

```
            <member>
               <name>faultString</name>
               <value>
                  <string>No such method</string>
               </value>
            </member>
         </struct>
      <value>
   </fault>
</methodResponse>
```

This response indicates that the method called can not be found.

Other faults might describe computations that went awry for some reason, resulting in an exception being thrown by the method.

Another Example

Suppose the method called returns an array of **int** values.

```
<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value>
            <array>
               <data>
                  <value><int>0</int></value>
                  <value><int>1</int></value>
                  <value><int>8</int></value>
                  <value><int>27</int></value>
                  <value><int>64</int></value>
               </data>
            </array>
         </value>
      </param>
   </params>
</methodResponse>
```

DTD for XML-RPC

```
<!ELEMENT methodCall          (methodName, params)>
<!ELEMENT methodName          (#PCDATA)>
<!ELEMENT params              (param*)>
<!ELEMENT param               (value)>
<!ELEMENT value               (i4 | int | string | double | boolean |
                               dateTime.iso8601 | base64 |
                               struct | array)>
<!ELEMENT i4                  (#PCDATA)>
<!ELEMENT int                 (#PCDATA)>
<!ELEMENT double              (#PCDATA)>
<!ELEMENT boolean             (#PCDATA)>
<!ELEMENT string              (#PCDATA)>
<!ELEMENT dateTime.iso8601    (#PCDATA)>
<!ELEMENT base64              (#PCDATA)>
<!ELEMENT array               (data)>
<!ELEMENT data                (value*)>
<!ELEMENT struct              (member+)>
<!ELEMENT member              (name, value)>
<!ELEMENT name                (#PCDATA)>
<!ELEMENT methodResponse      (params | fault)>
<!ELEMENT fault               (value)>
```

XML Schema for XML-RPC

```
<?xml version="1.0"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="methodCall">
    <xs:complexType>
      <xs:all>
        <xs:element name="methodName">
          <xs:simpleType>
            <xs:restriction base="AsciiString">
              <xs:pattern value="([A-Za-z0-9]|/|\.|:|_)+" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>

        <xs:element name="params" minOccurs="0"
                                          maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="param"  type="ParamType"
                    minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:all>
    </xs:complexType>
  </xs:element>
```

```
<xs:element name="methodResponse">
  <xs:complexType>
    <xs:choice>
      <xs:element name="params">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="param" type="ParamType"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>

      <xs:element name="fault">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="value">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="struct">
                    <xs:complexType>
                      <xs:sequence>
                        <xs:element name="member"
                                    type="MemberType">
                        </xs:element>
                        <xs:element name="member"
                                    type="MemberType">
                        </xs:element>
                      </xs:sequence>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
```

```
      </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:complexType name="ParamType">
  <xs:sequence>
    <xs:element name="value" type="ValueType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="ValueType">
  <xs:choice>
    <xs:element name="i4"      type="xs:int"/>
    <xs:element name="int"     type="xs:int"/>
    <xs:element name="string" type="AsciiString"/>
    <xs:element name="double"      type="xs:decimal"/>
    <xs:element name="Base64"
                                type="xs:base64Binary"/>
    <xs:element name="boolean"
                                type="NumericBoolean"/>
    <xs:element name="dateTime.iso8601"
                                     type="xs:dateTime"/>
    <xs:element name="array"  type="ArrayType"/>
    <xs:element name="struct" type="StructType"/>
  </xs:choice>
</xs:complexType>

<xs:complexType name="StructType">
  <xs:sequence>
    <xs:element name="member" type="MemberType"
                                maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

```
  <xs:complexType name="MemberType">
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:element name="value" type="ValueType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ArrayType">
    <xs:sequence>
      <xs:element name="data">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="value"   type="ValueType"
                  minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="AsciiString">
    <xs:restriction base="xs:string">
      <xs:pattern value="([ -~]|\n|\r|\t)*"/>
    </xs:restriction>         <!-- space to ~, ascii 127 -->
  </xs:simpleType>

  <xs:simpleType name="NumericBoolean">
    <xs:restriction base="xs:boolean">
      <xs:pattern value="0|1"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

Using HTTP for XML-RPC

Both method calls and method responses are sent using HTTP.

Both kinds of message require certain header values to be included.

Method Call

POST / HTTP 1.1
Host: *server host name*
User-Agent: *software making the request*
Content-Type: text/xml
Content-Length: *number of bytes in payload*

payload

Method Response

HTTP 200 OK
Content-Type: text/xml
Content-Length: *number of bytes in payload*

payload

The method response sends the code 200 no matter what happens with the method execution.

The HTTP response indicates success, but the content of the message (the payload) can show method failure by a *fault* element.

Note that XML-RPC has no standard or predefined error codes for the *fault* element in a method response.

Implementing XML-RPC

Many implementations of XML-RPC have been developed, but they are not always easy to install.

To illustrate the basic idea, we provide a relatively simple implementation of an XML-RPC server and an XML-RPC client written in Java.

This system has weak error handling capabilities, but works fine when given correct XML messages.

Server Side

We begin on the server side where four classes provide the functionality of an XML-RPC server.

The first class creates a ServerSocket and waits for clients to connect to the server.

When a connection is made, the Socket object is passed on to a thread that handles the communication between the server and the client.

File: Server.java

```
import java.io.*;
import java.net.*;

public class Server
{
   public static void main(String [] args) throws IOException
   {
      ServerSocket serversocket = new ServerSocket(8000);
      System.out.println(
                        "HTTP Server running on port 8000.");
      System.out.println("Use control-c to stop server.");
```

```
        while (true)
        {
            Socket sock = serversocket.accept();
            String client = sock.getInetAddress().getHostName();
            System.out.println("Connected to client " + client);
            new Handler(sock).start();
        }
    }
}
```

The Handler class does most of the work on the server side.

Parsing the headers in the HTTP request and retrieving the payload is done by the Request class, which comes after the Handler code.

The Handler class parses the XML payload from the request to identify the method name and the actual parameters to the method.

It then uses Java Reflection to find the method requested and invoke it on the actual parameters.

The function call *RpcMethods.class.getMethods()* returns an array of all public methods available to the RpcMethods class.

This array of Method objects is searched for the one we want.

The response to the client is built from the result produced by the method call. In so doing, we use a number of methods for converting Java values, simple and array, into the strings that XML-RPC expects.

The various tasks performed by the server are labeled with the steps for the XML-RPC lifecycle.

These utility methods are described with comments in the Java code.

Sections of the main processing code, found in the *run* method, are labeled to describe the stages of the XML-RPC lifecycle that are being dealt with.

File: Handler.java

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import java.lang.reflect.*;
import java.io.*;
import java.net.*;
import java.util.*;

public class Handler extends Thread
{
   private Socket sock;

   Handler(Socket sk)
   {  sock = sk;   }

   public void run()
   {
/* 3. Read request (method call) from client */
      try
      {
         BufferedInputStream bis =
                        new BufferedInputStream(
                                 sock.getInputStream());
         byte [] buffer = new byte [2000];
         int num = bis.read(buffer);
         System.out.println("Handler got " + num + " bytes.");
         byte [] request = new byte [num];
         System.arraycopy(buffer, 0, request, 0, num);
```

```
        Request req = new Request(request);
        byte [] xmlrpcRequest = req.getContent();

/* 4. Parse method call */
        byte [] response;
        DocumentBuilderFactory factory =
                        DocumentBuilderFactory.newInstance();
        DocumentBuilder domParser =
                        factory.newDocumentBuilder();

        Document dom = null;
        try
        {  dom =  domParser.parse(new InutSource(
                                new StringReader(
                                        new String(xmlrpcRequest))));
        }
        catch (SAXException e)
        {  response = e.getMessage().getBytes(); }

        Element el =
           (Element)dom.getElementsByTagName(
                                        "methodName").item(0);
        String methodName =
                        el.getFirstChild().getNodeValue();
        NodeList parms =
                        dom.getElementsByTagName("param");
        List<Object> pList = new ArrayList<Object>();
        for (int k=0; k<parms.getLength(); k++)
        {
           Node parm = parms.item(k);
           Node value = parm.getFirstChild();
           Node type = value.getFirstChild();
           Node actVal = type.getFirstChild();
           Object ob =
              mapArgToType(type.getNodeName(), actVal);
           pList.add(ob);
        }
```

```
/* 5. Search for method and call it */
        Object [] args = new Object[pList.size()];
        Class [] argTypes = new Class[pList.size()];
        for (int k=0; k<pList.size(); k++)
        {
           args[k] = pList.get(k);
           argTypes[k] = pList.get(k).getClass();
        }
        String methodResponse = "";
        try
        {
           Method [] meths = RpcMethods.class.getMethods();
           Method m = null;
           int k = 0;
           boolean found = false;
           while (k < meths.length && !found)
           {
              m = meths[k];
              Class [] paramTypes = m.getParameterTypes();
              if (m.getName().equals(methodName) &&
                 match(argTypes, paramTypes))
                    found = true;
              k++;
           }
           if (found)
           {
              Object value = m.invoke(null, args);
/* 6. Build method response for a successful call */
              StringBuffer returnValue = new StringBuffer();
              if (value.getClass().isArray())
              {                          // Need to build array
                 Class cT =
                    value.getClass().getComponentType();
```

```
                if (cT.isPrimitive())
                    value = wrap(value);
                Object [] array = (Object [])value;
                returnValue.append("<array><data>");
                for (int k=0; k<array.length; k++)
                    returnValue.append("<value>" +
                        mapValueFromType(array[k])+"</value>");
                returnValue.append("</data></array>");
            }
            else
                returnValue.append(
                            mapValueFromType(value));
            methodResponse =
                "<?xml version=\"1.0\"?>" +
                "<methodResponse><params><param>" +
                "<value>" + returnValue + "</value>" +
                "</param></params></methodResponse>";
        }
        else    // not found
/* 7. Build method response for a failed call */
            methodResponse =
                "<?xml version=\"1.0\"?>" +
                "<methodResponse><fault><value>" +
                "<struct><member><name>fault</name>" +
                "<value><int>88</int></value></member>" +
                "<member><name>faultString</name>" +
                "<value><string>No such method" +
                "</string></value></member></struct>" +
                "</value></fault></methodResponse>";
        }
        catch (Exception e)
        {
            methodResponse =
                "<?xml version=\"1.0\"?>" +
                "<methodResponse><fault><value>" +
```

```
                "<struct><member><name>fault</name>" +
                "<value><int>99</int></value></member>" +
                "<member><name>faultString</name>" +
                "<value><string> " + e.toString() +
                "</string></value></member></struct>" +
                "</value></fault></methodResponse>";
        }
        response = methodResponse.getBytes();

/* 6. and 7. Return response to client */
        BufferedOutputStream client =
                new BufferedOutputStream(
                        sock.getOutputStream());
        String headers =
                "HTTP 200 OK\r\n" +
                "Content-Type:  text/xml \r\n" +
                "Content-Length: " + response.length +
                "\r\n\r\n";
        client.write(headers.getBytes());
        client.write(response);     // write data of payload
        client.close();
      }
      catch (Exception e)
      {  System.out.println(e);  }
   }

/* mapArgToType takes an XML-RPC type and a Node value
   and returns that value as a Java object of the correct type. */

   private Object mapArgToType(String type, Node value)
   {
      if (type.equals("i4") || type.equals("int"))
         return new Integer(value.getNodeValue());
      else if (type.equals("boolean"))
         return
              new Boolean("1".equals(value.getNodeValue()));
```

```
    else if (type.equals("double"))
      return new Double(value.getNodeValue());
    else if (type.equals("string"))
      return new String(value.getNodeValue());
    else if (type.equals("array"))
      return mkArray(value);
    else return null;
  }

/* mkArray takes the Node value and builds a Java array of
   primitive components or strings from the data in the Node
   value.                                                    */

  private Object mkArray(Node value)
  {
    NodeList values = value.getChildNodes();
    String compType =
          values.item(0).getFirstChild().getNodeName();
    if (compType.equals("int"))
    {
      int [] ia = new int [values.getLength()];
      for (int k=0; k<values.getLength(); k++)
      {
        String ival =
            values.item(k).getFirstChild().getNodeValue();
        ia[k] = Integer.parseInt(ival);
      }
      return ia;
    }
    else if (compType.equals("double"))
    {
      double [] da = new double [values.getLength()];
      for (int k=0; k<values.getLength(); k++)
      {
        String dval = values.item(k).getFirstChild().
                        getFirstChild().getNodeValue();
        da[k] = Double.parseDouble(dval);
      }
```

```
            return da;
        }
        else if (compType.equals("boolean"))
        {
            boolean [] ba = new boolean [values.getLength()];
            for (int k=0; k<values.getLength(); k++)
            {
                String bval = values.item(k).getFirstChild().
                                    getFirstChild().getNodeValue();
                ba[k] = bval.equals("1");
            }
            return ba;
        }
        else if (compType.equals("string"))
        {
            String [] sa = new String [values.getLength()];
            for (int k=0; k<values.getLength(); k++)
                sa[k] = values.item(k).getFirstChild().
                                    getFirstChild().getNodeValue();
            return sa;
        }
        return null;
    }                   // end of run method

/* mapValueFromType takes a Java object and creates a
   corresponding XML-RPC element for its value.                  */

    private String mapValueFromType(Object value)
    {
        Class type = value.getClass();
        if (type.equals(Integer.class))
            return "<int>" + value + "</int>";
        else if (type.equals(Boolean.class))
            return "<boolean>" +
                    (value.equals(new Boolean(true)):"1":"0") +
                    "</boolean>";
        else if (type.equals(String.class))
            return "<string>" + value + "</string>";
        else if (type.equals(Double.class))
```

```
            return "<double>" + value + "</double>";
        else return "";
    }

/* wrap takes a Java array of primitive values and returns a
   corresponding Java array of wrapper values.                  */

    private Object wrap(Object value)
    {
        Class cT = value.getClass().getComponentType();
        if (cT.equals(int.class))
        {
            int [] ia = (int [])value;
            Integer [] iA = new Integer [ia.length];
            for (int k=0; k<ia.length; k++)
                iA[k] = new Integer(ia[k]);
            return iA;
        }
        else if (cT.equals(double.class))
        {
            double [] da = (double [])value;
            Double [] dA = new Double [da.length];
            for (int k=0; k<da.length; k++)
                dA[k] = new Double(da[k]);
            return dA;
        }
        else if (cT.equals(boolean.class))
        {
            boolean [] ba = (boolean [])value;
            Boolean [] bA = new Boolean [ba.length];
            for (int k=0; k<ba.length; k++)
                bA[k] = new Boolean(ba[k]);
            return bA;
        }
        return value;
    }
```

```
/* match takes Class arrays representing the types of the
   actual parameters and the formal parameters and sees
   if they are the same.                                    */

   private boolean match(Class [] at, Class [] pt)
   {
      if (at.length != pt.length)
         return false;
      for (int k=0; k<at.length; k++)
         if (!at[k].equals(adjust(pt[k])))
            return false;
      return true;
   }


/* adjust converts a primitive type into its corresponding
 object (wrapper) type. This method is used by match.      */

   private Class adjust(Class paramType)
   {
      Class c = paramType;
      if (c.equals(int.class))
         return Integer.class;
      else if (c.equals(double.class))
         return Double.class;
      else if (c.equals(boolean.class))
         return Boolean.class;
      else
         return c;
   }
}
```

The Request class processes a request, placing the header information into a Map object by tokenizing the header information, and extracts the payload in the request.

It is used to process both the method call request and the method response request.

File: Request.java

```
import java.util.*;
import java.io.*;

public class Request
{
   private Map<String,String> headers =
                                 new HashMap<String,String>();
   private byte [] content = null;
   private String requestURI = "";

   Request(byte [] rawData)
   {  parseHeaders(rawData);  }

   private void parseHeaders(byte [] rawData)
   {
/* 3. and 8. Read request (method call and method response) */
      int eoh = findEOH(rawData);     // Find end of Headers
      String heads = new String(rawData, 0, eoh);

      StringTokenizer st = new StringTokenizer(heads, "\r\n");

      String firstLine = st.nextToken();

      while (st.hasMoreTokens())     // Process the headers
      {
          String requestLine = st.nextToken();
          int separator = requestLine.indexOf(": ");
          String header = requestLine.substring(0, separator);
          String value = requestLine.substring(separator+1);
          headers.put(header.trim(), value.trim());
      }
      int length = Integer.parseInt(getHeader("Content-Length"));
      content = new byte [length];
```

```
      System.arraycopy(rawData, eoh+4, content, 0, length);
      System.out.println(new String(rawData));  // display payload
   }

   public String getHeader(String name)
   {  return headers.get(name);  }

   public byte [] getContent()
   {  return content;  }

   public int findEOH(byte [] headers)
   {
      String heads = new String(headers);
      return heads.indexOf("\r\n\r\n");
   }
}
```

The RpcMethods class contains the methods that can be called remotely.

Each method is a public class method.

The methods are designed to illustrate each type of parameter that the system allows: int, double, boolean, String, Integer, Double, Boolean, and one-dimensional arrays of these types.

File: RpcMethods.java

```
import java.util.*;
import java.lang.reflect.*;
import java.math.BigInteger;
```

```
public class RpcMethods
{
   public static String reverse(String s)
   {
      StringBuffer sb = new StringBuffer(s);
      return sb.reverse().toString();
   }

   public static String concat(String [] sa)
   {
      StringBuffer sb = new StringBuffer();
      for (int k=0; k<sa.length; k++)
        sb.append(sa[k]);
      return sb.toString();
   }

   public static int add(int num1, int num2)
   {
      return num1 + num2;
   }

   public static String [] listMethods()
   {
      Class c = Rpc0Methods.class;
      Method [] methods = c.getMethods();
      List<String> meths = new ArrayList<String>();
      for (int k=0; k<methods.length; k++)
      {
        Method m = methods[k];
        int mods = m.getModifiers();
        if (Modifier.isStatic(mods))
            meths.add(m.getName());
      }
      return (String [])meths.toArray(new String [0]);
   }
```

```
public static Double sum(Double [] da)
{
      double sm = 0.0;
      for (int k=0; k<da.length; k++)
          sm = sm + da[k];
      return sm;                  // note autoboxing
}

public static boolean and(boolean [] ba)
{
   boolean result = true;
   for (int k=0; k<ba.length; k++)
     result = result && ba[k];
   return result;
}

public static Double product(Double x, Double y)
{
   return x*y;              // note autoboxing
}

public static Integer [] mkArray(int size)
{
   Integer [] ia = new Integer [size];
   for (int k=0; k<ia.length; k++)
     ia[k] = new Integer(k*k*k);
   return ia;
}


public static double [] mkArray()
{
   double [] da = new double [5];
   for (int k=1; k<=da.length; k++)
     da[k-1] = k*k*k*k/100.0;
   return da;
}
```

```
    public static String [] fibo(int m, int n, int num)
    {
        String [] fibs = new String [num];
        BigInteger low  = BigInteger.ONE;
        BigInteger high = BigInteger.ONE;
        low = BigInteger.valueOf(m);
        high = BigInteger.valueOf(n);
        for (int k=1; k<=num; k++)
        {
              fibs[k-1] = low.toString();
              BigInteger temp = high;
              high = high.add(low);
              low = temp;
        }
        return fibs;
    }
}
```

Client Side

The client side has a class Test that creates an RpcClient object that will connect with the RPC server and then calls each of the remote methods using an instance method *execute* for the RpcClient object.

Parameters are collected in a List object and are passed to the method *execute* with the name of the method to be called.

The value returned by the *execute* method is displayed by the Test class.

Note the use of generic containers in this code.

File: Test.java

```
import java.util.*;

public class Test
{
    public static void main(String [] args) throws Exception
    {
        RpcClient rpcClient =
                new RpcClient("r-lnx233.cs.uiowa.edu", 8000);
        System.out.println("----------------------------------------");
        List<Object> parms = new ArrayList<Object>();
        parms.add("XML-RPC Hello");
        Object result = rpcClient.execute("reverse", parms);
        System.out.println("\nResult: " + result);
        System.out.println("----------------------------------------");
        parms = new ArrayList<Object>();
        List<String> sa = new ArrayList<String>();
        sa.add("abc");  sa.add("DEFG");  sa.add("highlmnop");
        parms.add(sa);
        result = rpcClient.execute("concat", parms);
        System.out.println("\nResult: " + result);
        System.out.println("----------------------------------------");
        parms = new ArrayList<Object>();
        parms.add(new Integer(368));
        parms.add(new Integer(927));
        result = rpcClient.execute("add", parms);
        System.out.println("\nResult: " + result);
        System.out.println("----------------------------------------");
        parms = new ArrayList<Object>();
        List methods =
                (List)rpcClient.execute("listMethods", parms);
        System.out.println("\nResult: ");
        for (Iterator it = methods.iterator(); it.hasNext(); )
            System.out.println(it.next());
```

```
System.out.println("---------------------------------------");
parms = new ArrayList<Object>();
parms.add(new Double(3.68));
parms.add(new Double(9.27));
result = rpcClient.execute("product", parms);
System.out.println("\nResult: " + result);

System.out.println("---------------------------------------");
parms = new ArrayList<Object>();
List<Boolean> ba = new ArrayList<Boolean>();
ba.add(new Boolean(true));
ba.add(new Boolean(false));
ba.add(new Boolean(true));
parms.add(ba);
result = rpcClient.execute("and", parms);
System.out.println("Result: " + result);

System.out.println("---------------------------------------");
parms = new ArrayList<Object>();
Object obj = rpcClient.execute("unknown", parms);
System.out.println("\nResult: " + obj);

System.out.println("---------------------------------------");
parms = new ArrayList<Object>();
nums = (List)rpcClient.execute("mkArray", parms);
System.out.println("\nResult: ");
for (Iterator it = nums.iterator(); it.hasNext(); )
    System.out.println(it.next());

System.out.println("---------------------------------------");
parms = new ArrayList<Object>();
parms.add(new Integer(12));
nums = (List)rpcClient.execute("mkArray", parms);
System.out.println("\nResult: ");
for (Iterator it = nums.iterator(); it.hasNext(); )
    System.out.println(it.next());
```

```
        System.out.println("---------------------------------------");
        parms = new ArrayList<Object>();
        List<Double> da = new ArrayList<Double>();
        da.add(new Double(43.7)); da.add(new Double(91.3));
        da.add(new Double(18.8)); da.add(new Double(33.6));
        da.add(new Double(83.5)); da.add(new Double(76.1));
        parms.add(da);
        result = rpcClient.execute("sum", parms);
        System.out.println("Result: " + result);

        System.out.println("---------------------------------------");
        parms = new ArrayList<Object>();
        parms.add(new Integer(3));
        parms.add(new Integer(5));
        parms.add(new Integer(22));
        nums = (List)rpcClient.execute("fibo", parms);
        System.out.println("\nResult: ");
        for (Iterator it = nums.iterator(); it.hasNext(); )
            System.out.println(it.next());
        System.out.println("---------------------------------------");
    }
}
```

The RpcClient class builds the XML payload for the method call and sends it to the server.

When the response returns, it parses the XML to extract the value produced by the method or the nature of the fault if the method failed for some reason.

The various tasks performed by the server are labeled with the steps from the XML-RPC lifecycle.

File: RpcClient.java

```
import org.xml.sax.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import java.net.*;
import java.util.*;
import java.io.*;

public class RpcClient
{
   private String server = "";
   private int srvPort;

   public RpcClient(String host, int port)
   {
      server = host;
      srvPort = port;
   }

   public Object execute(String method, List parms)
                                                throws Exception
   {
      Socket sock = new Socket(server, srvPort);
      DataOutputStream out =
            new DataOutputStream(sock.getOutputStream());

/* 1. Construct XML for method call */
      String methodCall =
                  "<?xml version=\"1.0\"?>" +
                  "<methodCall><methodName>" +
                  method + "</methodName>" +
                  "<params>" + buildParams("param", parms) +
                  "</params></methodCall>";
```

```
/* 2. Send HTTP request for method call */
    String request =
                "POST / HTTP 1.1\r\n" +
                "Host: " + server + ":" + srvPort + "\r\n" +
                "User-Agent: RPCClient\r\n" +
                "Content-Type: text/xml\r\n" +
                "Content-Length: " + methodCall.length() +
                "\r\n\r\n";
    String bothParts = request + methodCall;
    out.write(bothParts.getBytes());

/* 8. Read response from method call */

    BufferedReader reader = new BufferedReader(
       new InputStreamReader(sock.getInputStream()));
    ByteArrayOutputStream reqBA =
                          new ByteArrayOutputStream();
    int ch = reader.read();
    while (ch > -1)
    {
       reqBA.write(ch);
       ch = reader.read();
    }
    reader.close(); out.close(); sock.close();
    Request req = new Request(reqBA.toByteArray());
    int length = Integer.parseInt(
                          req.getHeader("Content-Length"));
    byte [] response = new byte [length];
    int eoh = req.findEOH(reqBA.toByteArray());
    System.arraycopy(reqBA.toByteArray(), eoh+4,
                                    response, 0, length);
```

```
/* 8. Parse response from method call */
      DocumentBuilderFactory factory =
                        DocumentBuilderFactory.newInstance();
      DocumentBuilder domParser =
                        factory.newDocumentBuilder();
      Document dom = null;
      try
      {
         dom = domParser.parse(new InputSource(
                     new StringReader(new String(response))));
      }
      catch (Exception e)
      {  System.out.println(e);  }

      NodeList pList = dom.getElementsByTagName("fault");
      if (pList.getLength() > 0)
      {
         Text text =
                  (Text)pList.item(0).getFirstChild().
                    getFirstChild().getLastChild().getLastChild().
                    getFirstChild().getFirstChild();
         String errorMessage = text.getData();
         return errorMessage;
      }
      pList = dom.getElementsByTagName("param");
      Object actualValue = "";
      Node parm = pList.item(0);
      Node value = parm.getFirstChild();
      Node type = value.getFirstChild();
      if (type.getNodeName().equals("array"))
         actualValue = buildList((Element)type);
      else
         actualValue = type.getFirstChild().getNodeValue();
      return actualValue;
   }
```

```
/* buildList extracts the values from the array and returns
   them as a list object.                                    */

    private List buildList(Element element)
    {
       List<Object> returnValues = new ArrayList<Object>();
       NodeList values =
                   element.getElementsByTagName("value");
       for (int k=0; k<values.getLength(); k++)
       {
          Node value = values.item(k);
          Node type = value.getFirstChild();
          String actualValue =
                      type.getFirstChild().getNodeValue();
          String actualType = type.getNodeName();
          returnValues.add(mapArgToObject(actualType,
                                                 actualValue));
       }
       return returnValues;
    }

/* buildParams builds an XML element representing an array
   of values to be sent as a parameter to the method.        */

    private String buildParams(String tag, List parms)
    {
       String start = "", end = "";
       if (!tag.equals(""))
       {
          start = "<" + tag + ">";
          end = "</" + tag + ">";
       }
       StringBuffer returnedParms = new StringBuffer();
       Iterator type = parms.iterator();
       while (type.hasNext())
       {
          Object aType = type.next();
```

```
        if (aType instanceof Integer)
        {
           returnedParms.append(start + "<value><int>");
           returnedParms.append(((Integer)aType).toString());
           returnedParms.append("</int></value>" + end);
        }
        else if (aType instanceof Boolean)
        {
           returnedParms.append(start + "<value><boolean>");
           boolean bval = ((Boolean)aType).booleanValue();
           returnedParms.append(bval?"1":"0");
           returnedParms.append("</boolean></value>" + end);
        }
        else if (aType instanceof String)
        {
           returnedParms.append(start + "<value><string>");
           returnedParms.append(new String((String)aType));
           returnedParms.append("</string></value>" + end);
        }
        else if (aType instanceof Double)
        {
           returnedParms.append(start + "<value><double>");
           returnedParms.append(((Double)aType).toString());
           returnedParms.append("</double></value>" + end);
        }
        else if (aType instanceof List)
        {
           List subList = (List)aType;
           returnedParms.append(start +
                                    "<value><array><data>");
           returnedParms.append(buildParams("", subList));
           returnedParms.append("</data></array></value>"
                                                   + end);
        }
     }
     return returnedParms.toString();
  }
```

```
/* mapArgToObject takes an XML-RPC type and a value as
   a string and returns that value as a Java object of the
   correct type.                                          */

   private Object mapArgToObject(String type, String value)
   {
      if (type.equals("i4") || type.equals("int"))
         return new Integer(value);
      else if (type.equals("string"))
         return new String(value);
      else if (type.equals("double"))
         return new Double(value);
      else if (type.equals("boolean"))
         return new Boolean("1".equals(value));
      else
         return null;
   }
}
```

Limitations of XML-RPC

- Limited choice of data types.
- No provision for passing objects.
- Little or no security since firewalls are bypassed.
- No type checking of array values; mixed type not forbidden.
- No check that a *struct* has no duplicate names.
- Strings allow only ascii.
- No representation of NaN for *double*.