Apache XML-RPC Clients

Apache XML-RPC is an implementation of XML-RPC: Remote procedure calls are transmitted as XML documents and the results are returned in the same way.

The most important feature of XML-RPC is portability: XML-RPC implementations are available for almost any programming language. The protocol is intentionally restricted to some typical data type, including integers, strings, arrays, and maps.

Apache XML-RPC is based on Java reflection: You trade flexibility for type safety and compiler control. But why take the disadvantage? The client generator allows you to have both flexibility and compiler control.

1. How it works

The client generator is roughly similar to the Proxy Generator: Assuming that you have classes A, B, and C on the server, which are being called via XML-RPC, the generator creates classes A', B', and C' with roughly the same set of public methods. The difference is, that the latter classes have every method implemented as an XML-RPC call. However, this XML-RPC call is completely transparent!

More precisely, if you have the following class on the server:

```
public class Calculator {
    public int add(int i1, int i2) {
                return i1 + i2;
    }
    public double multiply(double d1, double d2) {
        return d1 * d2;
    }
}
```

The generator will create a class for you, which is roughly similar to:

```
public class Calculator {
        private XmlRpcCaller caller;
        public Calculator(caller) {
                this.caller = caller;
        }
    public int add(int i1, int i2) {
        Vector v = new Vector();
        v.add(new Integer(i1));
```

```
        v.add(new Integer(i2));
        XmlRpcClient c = new XmlRpcClient(url);
        Object o = caller.xmlRpcCall("Calculator.add", v);
        return ((Integer) o).intValue();
    }
    public double add(double d1, double d2) {
        Vector v = new Vector();
        v.add(new Double(d1));
        v.add(new Double(d2));
        Object o = caller.xmlRpcCall("Calculator.add", v);
        return ((Double) o).doubleValue();
    }
}
```

In particular, note that the generated classes are automatically converting from or to primitive types. A basic implementation for the XmlRpcCaller would be:

```
public class MyCaller implements org.apache.ws.jaxme.js.apps.XmlRpcCaller {
        private final URL url;
        public MyCaller(URL url) {
            this.url = url;
        }
        public Object xmlRpcCall(String name, Vector params) throws Exception {
            return new XmlRpcClient(url).execute(name, params);
        }
}
```

2. The dispatcher

The generator may create one additional class for you, which is dedicated for the server: The dispatcher. In the above example, a dispatcher would be:

```
public class Dispatcher implements org.apache.xmlrpc.XmlRpcHandler {
    public Object execute(String pName, Vector pParams) throws Throwable {
        if ("Calculator-add".equals(pName)) {
                int i1 = ((Integer) params.get(0)).intValue();
                int i2 = ((Integer) params.get(1)).intValue();
                return new Integer(new Calculator().add(i1, i2));
        } else if ("Calculator-multiply".equals(pName)) {
                double d1 = ((Double) params.get(0)).doubleValue();
                double d2 = ((Double) params.get(1)).doubleValue();
                return new Double(new Calculator().multiply(d1, d2));
        }
    }
}
```

Note, that the dispatcher may very well be a static final instance variable. It is quite easy, to embed the dispatcher into the server:

```
        XmlRpcServer xmlrpc = new XmlRpcServer ();
        xmlrpc.addHandler ("$default", new org.apache.xmlrpc.XmlRpcHandler() {
            Dispatcher d = new Dispatcher();
            public Object execute(String pName, Vector pVector) throws Exception {
                try {
                        return d.invoke(pName, pVector);
                } catch (Exception e) {
                    throw e;
                } catch (Throwable t) {
                    throw new UndeclaredThrowableException(t);
                }
            }
        });
        byte[] result = xmlrpc.execute (request.getInputStream ());
        response.setContentType ("text/xml");
        response.setContentLength (result.length());
        OutputStream out = response.getOutputStream();
        out.write (result);
        out.flush ();
```

3. Using the generator

The generator is implemented as an Ant task. A typical invocation will most probably look like this:

```
    <taskdef resource="org/apache/ws/jaxme/js/pattern/ant.properties"
        classpathref="js.test.path"/>
    <xmlRpcGenerator
        targetPackage="com.foo.xmlrpc.client"
                destDir="build/src">
        <dispatcher name="com.foo.xmlrpc.server.Dispatcher"/>
        <serverClasses dir="src" includes="com/foo/xmlrpc/server/XmlRpc*.java"/>
    </xmlRpcGenerator>
```

The Ant task "xmlRpcGenerator" supports the following attributes:

Name	Description	Required Default
classpathRef	Specifies a reference to a classpath. This classpath is used, for loading the server classes, if they are passed in as compiled classes and not Java sources.	No Ant path
destDir	Specifies the directory, where sources are being generated. A package structure is created below the directory: If the target package is "com.foo.xmlrpc.client", and the	No Current directory

	destination directory is "src", then the generated classes will appear in "src/com/foo/xmlrpc/client".	
targetPackage	Specifies the package, in which the generated sources are being placed.	No Root package

The Ant task also supports the following nested elements:

Name	Description	Required Default
classpath	An embedded class path, used as a replacement for the 'classpathref' attribute. Using this nested child element is mutually exclusive with the attribute.	No Ant path
dispatcher	Specifies, that a dispatcher is being generated. The 'name' attribute specifies the dispatchers fully qualified class name. By default, the dispatcher will implement org.apache.xmlrpc.XmlRpcHandler. This may be suppressed by setting the 'implementingXmlRpcHandler' attribute to false. This is mainly usefull for the JaxMeJS test suite.	No
serverClasses	One or more file sets specifying the server side classes, for which clients are being generated. These classes may either be specified as sources (in which case the Java parser is used for scanning them, or as compiled classes, in which case Java reflection is being used.	Yes