Marquée XML-RPC Library

An implementation of the XML- RPC Specification for Java™

Abstract

This document contains a user's guide and implementation walkthrough. It may be useful to both users and developers customising the library or just wishing to increase their knowledge of it.

CONTENTS

1 INTRODUCTION

1.1 Background

The Marquée XML-RPC Library is an implementation of the XML-RPC specification and is based on another implementation for Java™ by Hannes Walnöffer, from which several ideas were borrowed when designing this software. The reason for developing this library was to fill in the blanks, in terms of functionality, where other implementations fell short. Particularly, customizeable serializers was something we could not do without, and we also wanted to optionally make use of features available in JDK 1.3, like the ability to dynamically generate proxy classes. We wanted the design to allow for customization in other areas as well.

1.1.1 Design strategies

When developing the library, we have tried to keep resource consumption to reasonable minimum to make the library slim enough to be used in situations where this approach is preferable or necessary. We have also focused on keeping the code as clean as possible, hoping it will reach production standard and be a natural choice when adding XML-RPC support to various Java™ projects.

We wanted the library to be customizeable in as many areas as possible and will continue to open up the API for further customization. This strategy has been proven useful by Rainer Bischof at Electronic Data Systems, who has added a package that supports transferring of complete object graphs and exceptions in a fashion very similar to that of RMI.

In the spirit of XML-RPC, the source code has been made available for all use in hope that we will receive lots of feedback so that we may tune the library to fit into as many situations as possible. We also welcome code contributions of any kind that will contribute to making this library more stable, efficient, interoperable, or comprehensible.

2 OVERVIEW

2.1 Components

The library consists of four major parts:

Server Can be used in a servlet environment or as a stand-alone server accepting HTTP posts containing XML-RPC messages. The server uses the parser to interpret the XML-RPC messages and the serializer to convert return values to their XML-RPC counterparts. A secure server supporting Secure Sockets Layer (SSL) will be available during the fall 2001, or so.

Client Connects to an XML-RPC server and sends and receives XML-RPC messages and responses. It uses the serializer to convert call arguments and the parser to interpret return values.

Parser Uses any SAX compliant driver to parse XML-RPC messages and extracting the values contained therein.

Serializer Converts Java objects into their XML-RPC counterparts. Custom serializers are used for converting objects not inherently known to the basic serializer.

Users will come in contact with the serializer when specifying which custom serializers to be available during the serialization process. The library contains several custom serializers for serializing Java 2 collections, arrays, and other types of objects. Users will also come in contact with the parser when specifying which SAX driver to use during parsing. All other interaction occurs through the server and the client.

3 SERIALIZATION

3.1 Serializers

When sending XML-RPC messages from an XmlRpcClient to an XML-RPC server and from an XmlRpcServer to an XML-RPC client, Java objects need to be translated into their XML-RPC representations. Servers need to translate return values from their handlers, and clients need to translate arguments supplied in method calls. This is accomplished by using serializers.

All serialization occurs initially through the XmlRpcSerializer class which has support for basic object types like java.lang.String, java.lang.Integer, and so forth (see list below). If the XmlRpcSerializer is passed an object which it does not recognize, it keeps a list of XmlRpcCustomSerializers which it tries to use instead. Each custom serializer reports which kind of Java object it knows how to serialize.

boolean	char
byte	short
int	long
float	double
String	java.util.Date
byte[]	

The XmlRpcSerializer class supplies a static serialize() method through which all Java objects are serialized. This applies when using custom serializers as well. The serialize() method converts the supplied object and appends the XML-RPC representation in the supplied string buffer.

```
public static void serialize(
    Object value,
    StringBuffer output );
```

By using custom serializers, Java objects not inherently supported by the XmlRpcSerializer may be handled as well. Custom serializers are registered and unregistered through the static registerCustomSerializer() and unregisterCustomSerializer() methods.

```
public static void registerCustomSerializer(
    XmlRpcCustomSerializer serializer );

public static void unregisterCustomSerializer(
    XmlRpcCustomSerializer serializer );
```

As the methods in XmlRpcSerializer are static, custom serializers need only be registered with the XmlRpcSerializer once per JVM session and will be available to all classes within the JVM.

3.1.1 XmlRpcCustomSerializer

The following class represents a custom serializer that may handle any kind of Java 2 collection which is serialized into an XML-RPC array. This serializer is included in the Marquée XML-RPC library.

```
public class CollectionSerializer
    implements XmlRpcCustomSerializer
{
    public Class getSupportedClass()
    {
        return Collection.class;
    }

    public void serialize(
        Object Value,
        StringBuffer output )
    {
        Collection c = ( Collection ) value;
        Iterator iter = c.iterator();

        output.append( "<array><data>" );

        while ( iter.hasNext() )
        {
            XmlRpcSerializer.serialize( iter.next() );
        }

        output.append( "</data></array>" );
    }
}
```

This class implements the methods introduced in XmlRpcCustomSerializer and is a complete serializer.

3.1.1.1 getSupportedClass()

The getSupportedClass() method returns the class indicating which types of objects it knows how to serialize. This method is called in two situations by the XmlRpcSerializer. First when the serializer is installed by calling XmlRpcSerializer.registerCustomSerializer(), and second when an actual object has been requested for serialization by calling the XmlRpcSerializer.registerCustomSerializer()

method.

When registering a custom serializer, XmlRpcSerializer investigates the supported class so that it can determine where in the list of custom serializers the serializer belongs. For instance, if the serializer list already contains a serializer that knows how to handle java.util.Vector objects, the generic CollectionSerializer, above, will end up after the java.util.Vector serilizer. That is, the Vector serializer will override the generic CollectionSerializer, as it operates on a more specialized kind of collection.

When an object has been requested for serialization, the XmlRpcSerializer class queries every custom serializer in the list until it finds a serializer that knows how to handle the supplied value. For instance, when passing the XmlRpcSerializer.serialize() method a java.util.ArrayList object, the CollectionSerializer, if registered with XmlRpcSerializer, will catch the object (as it is an instance of java.util.Collection) and convert it into an XML-RPC array.

3.1.1.2 serialize()

Notice how the serialize() method in the CollectionSerializer above reuses the default serialization mechanism introduced in XmlRpcSerializer. The supplied collection may contain any kind of object which the XmlRpcSerializer knows how to handle, or for which a custom serializer has been registered. Theoretically, this may result in a recursive call to the serialize() method above if any of the elements in the collection is another collection.

3.1.1.3 Included custom serializers

The Marquee XML-RPC library supplies a few useful implementations of the XmlRpcCustomSerializer interface for converting generic collections and maps, as well as a few other more specialized types. It also contains a reflective serializer which can serialize any type of object by using Java Reflection.

CollectionSerializer	Serializes Java 2 collections into XML-RPC arrays (requires JDK 1.2 or above).
MapSerializer	Serializes Java 2 maps into XML-RPC structs (requires JDK 1.2 or above).
HashtableSerializer	Serializes hash tables into XML-RPC structs.

VectorSerializer	Serializes vectors into XML-RPC arrays.
IntArraySerializer	Serializes int[] arrays into XML-RPC arrays.
FloatArraySerializer	Serializes float[] arrays into XML-RPC arrays.
DoubleArraySerializer	Serializes double[] arrays into XML-RPC arrays.
BooleanArraySerializer	Serializes boolean[] arrays into XML-RPC arrays.
ObjectArraySerializer	Serializes arrays containing any kind of object into XML-RPC arrays.
ReflectiveSerializer	Serializes any object into XML-RPC structs using reflection.

The generic collection serializer, presented in the example above, requires JDK 1.2, as does the generic map serializer. If you don't support Java 1.2, the VectorSerializer and HashtableSerializer classes may be used instead.

4 THE CLIENT

4.1 Vanilla Flavored XmlRpcClient

There are two ways of invoking methods on an XML-RPC server – through a regular XmlRpcClient or through an XmlRpcProxy. This section describes how an XmlRpcClient is setup and how you use it to invoke methods. You use one XmlRpcClient for each XML-RPC service you are using.

Each client is associated with a particular host, port and path into that host. This is all specified when creating the client and can not be changed once the client is created. Currently, an XmlRpcClient may only be used by one thread at a time. This applies to the XmlRpcProxy as well. This will change with the next release, however, expected sometime this summer. For now, if several threads need to communicate with the same server, a separate XmlRpcClient has to be created for each thread.

The XmlRpcClient uses the XmlRpcSerializer to convert call arguments to their XML-RPC counterparts. Using custom serializers, the arguments may be of just about any type. This is a key issue; if you supply an argument of a type other than those listed in 1.1, a custom serializer must be available to convert the argument. Otherwise an exception will be raised.

To setup an XmlRpcClient you perform the following steps:

```
// 1) Register the serializers we need
XmlRpcSerializer.registerCustomSerializer(
    new HashtableSerializer() );

// 2) Create the client.
XmlRpcClient client = new XmlRpcClient(
    "www.stuffeddog.com", 80, "/speller/speller-rpc.cgi" );
```

In step 1, we make sure that all arguments we will send to the client will be serializable to XML-RPC. One of the methods we will call

later on this client will receive a Hashtable which is not inherently supported by the XmlRpcSerializer. Therefor, we register a custom serializer (supplied with the Marquée XML-RPC Library) that knows how to serialize Hashtables.

In step 2, we create the actual XmlRpcClient and specify which server the client should send invocations to. It is not important that step 1 comes before step 2; it is not until we actually invoke the client that serialization occurs.

To invoke methods on the server, the invoke() method is used. This comes in two flavors, both accepting the name of the method to in-

voke and either a java.util.Vector or an object array containing the arguments to use for the call:

```
// 3) Invoke a method.
Vector response = ( Vector ) client.invoke(
    "speller.spellChecking",
    new Object[] {
        "To be or not to be, that is the qwestion",
        new Hashtable() } );
```

In this step, several things happen. First, we choose to call the object array version of invoke(). The arguments are a string with a typo, and an empty Hashtable. The client will serialize these arguments into XML-RPC and open a socket towards www.stuffeddog.com, using port 80. The server will hopefully respond with an XML-RPC response. The response is parsed and the contained return value will be constructed and returned to the caller. In this case, we expect to receive a vector with suggestions for all spelling mis takes made in the text supplied in the call. Sending the response vector to the console will show a suggestion for the "qwestion" typo.

When using an XmlRpcClient we are responsible for converting the return value to what we expect the server to respond with. This will always be of one of the types listed in 1.1, or a java.util.Vector or a java.util.Hashtable. Whenever the response is an XML-RPC array, the return value will be a java.util.Vector, and whenever it is an XML-RPC struct it will be a java.util.Hashtable. Note that this has nothing t o do with the serializer. That is, even if we did not register the HashtableSerializer, we would still receive a Hashtable if the response was an XML-RPC struct. The serializer is only involved when sending arguments to the server.

This feature requires JDK 1.3.

4.2 Double Chocolate Chip XmlRpcProxy

An alternative way of calling server procedures is to use a dynamic XmlRpcProxy. When creating an instance of the XmlRpcProxy you specify the URL of the server which should be proxied for, and a list of interfaces the proxy should implement. The proxy may be typecast to and called through any of these interfaces, which will convert calls to XML-RPC messages that are sent to the server using the "<handler>.<method>" naming convention. The names of the interfaces and their methods should, in other words, correspond to services and procedures available on the server.

4.2.1 Example

When developing an application using XML-RPC servers located

somewhere on the Internet, you start by expressing the services available on the servers in Java interfaces.

```
interface mailToTheFuture
{
    /**
     * Adds a message to username's queue.
     *
     * @param username The email address of a
     * registered user.
     *
     * @param password must be that user's password.
     *
     * @param message A hashtable containing the
     *                following elements; dateTime,
     *                messageBody, receiverMailAd-
     *                dress, and subject.
     *
     * @return The number of messages in username's
     *         queue.
     */

    int addMessage(
        String username,
        String password,
        Hashtable message )
        throws Throwable;


    /**
     * Deletes a message from username's queue.
     *
     * @param username The email address of a
     * registered user.
     *
     * @param password must be that user's password.
     *
     * @param ordinal The number of the message to
     *                remove.
     *
     * @return An empty string.
     */

    String deleteMessage(
        String username,
        String password,
        int ordinal )
        throws Throwable;

    /** Rest of the methods are omitted for brevity */
}
```

Every service offered by a server that is to be used by the application is defined in its own interface. The list of interfaces are supplied to XmlRpcProxy.createProxy() along with the URL of the server, to receive an object implementing the supplied interfaces.

```
Object o = XmlRpcProxy.createProxy(
    "www.mailtothefuture.com",
    "/RPC2",
    80,
    new Class[] { mailToTheFuture.class } );
```

The proxy object may be typecast to any of the interfaces and used as such.

```
mailToTheFuture mttf = (mailToTheFuture) o;

System.out.println(
    mttf.addMessage(
        “usr@mailtothefuture.com”,
        “secret”,
        aMsgTable ) );
```

The call to mttf.addMessage() will result in an XML-RPC message being sent to t he www.mailtothefuture.com host with "mailToThe-Future. addMessage" as the included method call. The compiler may check that the arguments are of correct type and allow IDE's to perform code completion and such.

5 THE SERVER

5.1 Overview

When setting up an XML-RPC server you supply a set of objects that will receive the method calls parsed by the server dispatchers. These objects must implement the XmlRpcInvocationHandler interface which can be achieved by extending the ReflectiveInvocationHandler class, wrapping the object in a ReflectiveInvocationHandler instance, or implementing it from scratch. The methods that are to be invoked through XML-RPC must only use parameters of the types listed in Serializers (3.1).

> *An exception to this rule is when using invocation processors that modify the list of call arguments according to some set of rules. For instance, an invocation processor may add an additional transaction object for methods with names starting with "tx_". These arguments are not transported using XML-RPC and may be of any type.*

Return values may be of any type supported by the built-in serializer or any of the registered custom serializers. That is, if the CollectionSerializer supplied with the XML-RPC library is registered with the server, invocation handlers may return any type of object implementing the Java 2 Collection interface. Custom serializers are ordered by specialization. That is, the reflective serializer will always be placed last and the MapSerializer will always be placed before the CollectionSerializer, and so on. This ensures that the most appropriate serializer will be used for an object during serialization – for instance, a HashMap will not be serialized using the generic Collection serializer if a MapSerializer is available.

5.1.1 Setting up an XML-RPC server

This example shows how to set up a server by registering invocation handlers and invocation processors. It also shows how to specify which SAX driver and custom serializers to use.

```
import marquee.xmlrpc.*;
import marquee.xmlrpc.serializers.*;

class SampleHandler extends ReflectiveInvocationHandler
{
    String getNameOfMonth( int month )
        throws IllegalArgumentException
    {
        if ( month > 0 && month < 13 )
        {
            return months[ month ];
        }

        throw new IllegalArgumentException(
                    "Invalid month." );
    }

    String[] getAllMonths()
    {
        return months;
    }

    private final static String[] months =
    {
        "January", "February", "March", "April",
        "May", "June", "July", "August", "September",
        "October", "November", "December"
    }
}

public class SampleServer
{
    public staic void main( String[] args )
    {
        XmlRpcParser.setDriver(
            "com.sun.xml.parser.Parser" );

        XmlRpcSerializer.registerCustomSerializer(
            new ObjectArraySerializer() );

        XmlRpcServer server = new XmlRpcServer();

        server.registerInvocationHandler(
            new SampleHandler() );

        server.runAsService( 80 );
    }
}
```

The ObjectArraySerializer added in main() makes sure that methods returning arrays of objects (including arrays of Strings) are interpretable by the serializer.

5.1.2 Invocation Handlers

XML-RPC messages are parsed by the XmlRpcServer and dispatched to an XmlRpcInvocationHandler corresponding to the handler name contained in the element. The server creates an XML-

RPC response based on the return value of the handler, or based on an exception thrown by the handler. The XmlRpcInvocationHandler interface contains a single method that invocation handlers must implement. The ReflectiveInvocationHandler class supplies a default implementation of this interface that you'll use most of the times.

```
public Object invoke(
    String methodName,
    Vector arguments )
    throws Throwable;
```

5.1.2.1 ReflectiveInvocationHandler

The easiest way to create an invocation handler is to inherit the XmlRpcReflectiveInvocationHandler class which implements the XmlRpcInvocationHandler using Java Reflection to find the method targeted by the call. The XmlRpcReflectiveInvocationHandler class may also be used to wrap a Java object in a reflective handler, if your class is already inheriting from another class. If you look at the SampleServer above, the SampleHandler represents a complete invocation handler that extends the reflective invocation handler. This is the most common way of creating invocation handlers.

If, for some reason, you do not wish to inherit XmlRpcReflectiveInvocationHandler, you may wrap any object in a new ReflectiveInvocationHandler. This gives the overhead of an additional object being created (although with a single reference data member);

```
ReflectiveInvocationHandler handler =
    new ReflectiveInvocationHandler( myObject );
```

This approach is useful when exposing legacy code with XML-RPC or if your object class is already inheriting from another class. The ReflectiveInvocationHandler constructor has an optional second parameter in which you may supply a list of methods that should be available in the wrapped object.

```
ReflectiveInvocationHandler handler =
    new ReflectiveInvocationHandler(
        myObject,
        new String[] { “myFunc1”, “myFunc5” } );
```

The string array indicates that only methods named “myFunc1” or “myFunc5” should be exposed for remote invocation, regardless of how many methods are available in myObject. You may update this

list later on through a call to setEntryPoints() which is also available for classes extending ReflectiveInvocationHandler. Supplying null will make all methods available.

```
handler.setEntryPoints( new String[] { “myFunc1” } );
```

5.1.2.2 Writing your own handler from scratch

If you do not wish to use the reflective handler wich uses Java Reflection to identify which method to call, you may write your own invocation handler. This may increase performance slightly if you have a single method or only a few methods in your class.

```
public class MyInvocationHandler
    implements XmlRpcInvocationHandler
{
    public Object invoke(
        String methodName,
        Vector arguments )
        throws Throwable
    {
        if ( methodName.equals( "someMethod" ) )
        {
            return someMethod();
        }

        throw new Exception( "Method not found in handler." );
    }

    public String someFunction()
    {
        return "All done!";
    }
}
```

5.1.3 Invocation Processors

The XML-RPC library contains a powerful mechanism for attaching functionality that should be executed on every invocation regardless of the intended invocation handler. Through invocation processors, developers may perform various steps before and after handlers are invoked, and when exceptions during the invocation occur.

For instance, writing a custom event logger that loggs all interaction on an XML-RPC server to a file or over a socket is just a matter of implementing the XmlRpcInvocationProcessor interface, and attaching the processor to the XmlRpcServer. Its preProcess(), postProcess() and onException() methods will be called on every call sent to the server (onException() being called only on exceptions). Invocation processors may be used in several other, more

useful scenarios as well.

The preProcess() method receives a lot of useful information from the server that can be used to achieve some functionality; the callId is a sequence number that may be used to match a corresponding call to postProcess() at a later time; the callerIp contains the IP address of the client performing the call which may be very useful in logging, filtering, or session management situations; the handler and method names indicate which method will ultimately be called after the preProcess() method has had its say. This may be useful when logging or filtering out clients from specific handlers or methods. Finally, the arguments array contains the arguments that will be used when invoking the invocation handler. This parameter is particularly useful as you may add or remove elements from this list.

```
boolean preProcess(
    int callId,
    String callerIp,
    String handler,
    String method,
    Vector arguments );
```

Returning false from preProcess() means that the invocation should be aborted. Currently, there is no way of indicating to the user the reason for aborting.

5.1.3.1 Scenarios

The following list shows a few scenarios where invocation handlers could be used.

5.1.3.1.1 Filtering invocations

Implementing a filter is not very difficult. The preProcess() method of the filter just returns false when the IP address of the caller matches an IP address of an internal list. This may be extended to include handlers and methods as well. The library contains an exemple of a filtering processor that accepts IP addresses with wild cards, among other things.

5.1.3.1.2 Managing transactions

In this case, preProcess() could examine the name of the method (or handler) and create some Transaction object if the name starts with "tx_" for instance. The transaction object could be inserted first in the arguments list, and would be commited in postProcess(). The invocation handlers with names starting with "tx_" would naturally

have to be prepared to accept this additional argument. Similarly, other types of objects may be sent to invocation handlers that are not serialized over XML-RPC. Using naming conventions is one way of solving this problem.

5.1.3.1.3 Managing sessions

The library contains an additional processor example which, during the invocation, associates the client with a Session object. The Session object extends java.util.Hashtable and may be used by invocation handlers to store client state between invocations. The preProcess() method of that processor associates the handling thread with the IP address of the caller. At any time, the invocation handler (or any code called from the handler) may acquire the session object associated with the client. If a client performs five concurrent calls to the server, these calls (assuming threads are available) will be handled by five worker threads in the server, simultaneously. The processor associates all five threads with the same IP address, so all threads will receive the same session object when asked for.

5.1.3.1.4 Encryption and Compression

Pre-processors may, based on some criteria, decrypt or decompress particular arguments before they end up in the invocation handlers. Correspondingly, post-processors may encrypt or compress return values before sending them back to the client.

5.1.3.1.5 Authorization

A server could force all invocations to include the username and password of the caller which are extracted from the argument list before calling the handlers. This could be combined with the session processor to allow signing in once during a session. Security, in general, is not addressed in this library and there are several things to wish for. Currently, the library does not even support basic HTTP authentication. Optional support for SSL though JSSE is in planning though, although no release date is set yet. Contributions are very welcome.

5.1.3.1.6 Profiling

Measuring the time between preProcess() and postProcess() invocations for a call allows you to do minimal profiling like average call times and such.

6 XML-RPC AND SERVLETS

If your application is already using a web server you may want to use the XmlRpcServer through a servlet wrapper instead of running the XmlRpcServer as a service.

6.1 Example

A servlet example is available in the marquee.xmlrpc.testing package. We'll just take a look at the highlights. This sample builds on code supplied by David Watson. Thanks, David.

6.1.1 The init() method

The servlet init() method makes sure the XmlRpcServer is properly set up and that there are custom serialzers available to convert all Java objects used by the code.

```
public void init(
    ServletConfig config )
    throws ServletException
{
    XmlRpcParser.setDriver(
        "com.sun.xml.parser.Parser" );

    XmlRpcSerializer.registerCustomSerializer(
        new VectorSerializer() );

    XmlRpcSerializer.registerCustomSerializer(
        new HashtableSerializer() );

    server = new XmlRpcServer();

    server.registerInvocationHandler(
        “Echo”, new EchoInvocationHandler() );

    server.registerInvocationHandler(
        “Speller”, new XmlRpcClient(
            “www.stuffeddog.com”,
            80,
            “/speller/speller-rpc.cgi” ) );
}
```

The example uses two invocation handlers; an Echo handler and an XmlRpcClient hooked up to the spellchecking service at www.stuffeddog.com. This shows a nice feature introduced by Hannes Walnöffer in the Helma XML-RPC Library; XmlRpcClients may also serve as InvocationHandlers. How is this? XmlRpcClients have an invoke() method that you call when you want the client object to invoke a method on the server the client is connected to. Invocation handlers also have an invoke() method that the XmlRpcServer calls when an inbound call to that handler is received. By chance the two invoke() methods have the same signature. By registering an XmlRpcClient as an invocation handler,

when the server receives a call to that handler, it calls its invoke() method as it does with all handlers. The invoke() method of the client, as I explained, in turn calls the server for which it was created. We have created a kind of relay that is very useful when an applet is performing XML-RPC invocations on the server it was loaded from.

Applets may only communicate with the server it originated from, but if that server has registered an XmlRpcClient as an invocation handler, that client will act as a relay to another XML-RPC service. The sample servlet registers an XmlRpcClient under the name "Speller", which is hooked up to the spelling service at stuffeddog.com. Invoking the " Speller.speller.spellCheck" method on this servlet, will make the client automatically relay the call to the stuffeddog server and back to the caller.

6.1.2 The doPost() method

The doPost() method for servlets using the XML-RPC library will look pretty much the same.

```
public void doPost(
    HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, IOException
{
    try
     {
        byte[] result = server.execute(
            new ServerInputStream(
               req.getInputStream(),
               req.getContentLength() ) );

        res.setContentType( "text/xml" );
        res.setContentLength( result.length );

        OutputStream output = res.getOutputStream();
        output.write( result );
        output.flush();
    }
    catch ( java.lang.Throwable e ) { /* do whatever */}
}
```

Currently, when calling the execute() method of the XmlRpcServer, you may have to wrap the input stream containing the XML-RPC message in a marquee.xmlrpc.util.ServerInputStream if the XmlRpcServer is unable to identify the end of the stream. The ServerInputStream class will make sure that when the supplied amount of bytes are read, and end of stream is reported. So the server is not responding to calls, using ServerInputStream will fix it.

7 OBJECTCOMM

Rainer Bischof of Electronic Data Systems has created an addon to the library that may be used to transparently transfer complex objects via the XmlRpc protocol. It is located in the marquee.xmlrpc.objectcomm. This chapter contains a description of the package, written by Rainer himself.

This feature requires JDK 1.3.

7.1 The Objectcomm Package

Using the marquee.xmlrpc.objectcomm package you can call methods on the server that expect parameters of any type (objects and primitives) and return any type (object, primitive or void). That means it behaves similar to RMI except with the limitation that any returned object is passed by value, not as a remote object reference as is possible with RMI. That means it simulates only objects retrieved via the RMI naming service. This allows the developer to use RMI or XML-RPC transparently and switch between both even at runtime. So you can use RMI within intranets and XML-RPC for Internet connections. And that's really cool!

The objects transferred are encoded into XML-RPC structs that adhere to some objectcomm specific conventions. These conventions allow to express the object's class name, property names, property values including null support, property class names and indicators for circular references. This protocol is implemented on top of XML-RPC and is not explained in detail here. Maybe later when I have some time, for now please look into the code or the XML documents being transferred if you are curious.

That brings us to the Disclaimer:

> *The XML documents transferred between client and server are completely in line with the XmlRpc specification. Nevertheless the method calls, parameters and return values are encoded as structs in a specific format that is quite some effort to express in other implementations of XmlRpc. Therefore this package should only be used in environemts where both client and server use this package.*

7.1.1 Using the package

First I would suggest that you install and try out the Marquée XML-RPC Library without the objectcomm extension as this will get you started with less hassle. Have a special look at how to use the XmlRpcProxy. This stuff itself is really cool and very useful when

doing non-object oriented RPC via XML. The objectcomm package relies on the proxy functionality provided by the default XML-RPC implementation. As any dynamic proxy implementation this requires at least JDK 1.3.

The examples that are explained below are located in the package marquee.xmlrpc.objectcomm.example. These examples assume that you have the required XML-RPC libraries in the classpath. There are two examples:

1. A simple XmlRpc service that transfers objects between a client and a server.
2. The same example that uses not only XML-RPC but also RMI to communicate with the same service.

7.1.1.1 Example 1

The first example uses of the following classes:

Employee	A simple object to be transferred.
EmployeeServiceInterface	An interface defining a service that holds some employees and allows the client to retrieve and update an employee.
XmlEmployeeService	A service implementing the EmployeeServiceInterface for use with XML-RPC.
XmlServer	A small clas that starts the XmlRpc server and registers the XmlEmployeeService with it.
XmlClient	A client that uses the XmlEmployeeService to retrieve and updates employees.

How to invoke it:

1. Start the marquee.xmlrpc.objectcomm.example.XmlServer.
2. Start the marquee.xmlrpc.objectcomm.example.XmlClient.

All code is pretty straight forward and I will just explain the specifics for objectcomm in each class. Employee implements java.io.Serializeable. This is a tagging interface required by the Serializer used in objectcomm to indicate that this object may be transferred over the wire. Additionally a default constructor is required by objectcomm since we can't reconstruct an object from

scratch without it.

The EmployeeServiceInterface extends the tagging interface java.rmi.Remote and all methods declare to throw java.rmi.RemoteException. This is required by the objectcomm Proxy. This ensures that the Proxy may legally throw the RemoteException if something happens during the call. And lot's of things can happen during a network call: server down, server side out of memory error, broken connecton, etc. Any server side exception that is not declared in the interface definition is wrapped into a java.rmi.RemoteException. Declared exceptions and RuntimeExceptions are re-thrown at the client side with the same message.

XmlEmployeeService is a simple class that provides the service to retrieve and store employees. XmlServer uses the objectcomm server instead of the default XmlRpcServer. And finally, the XmlClient first uses the XmlEmployeeService directly in a non-networked mode to show what the output should be and afterwards uses the objectcomm transport to talk to the remote service.

7.1.1.2 Example 2

The second example uses these classes:

Employee	Same as in the first example.
EmployeeServiceInterface	Same as in the first example.
XmlRmiEmployeeService	A simple class that provides the service to retrieve and store employees.
XmlRmiServer	A small wrapper that starts the server.
XmlRmiClient	A client that uses the XmlEmployeeService to retrieve and updates employees via XML-RPC and RMI.

How to invoke it:

1. Create the RMI stub and skeleton for XmlRmiEmployeeService using RMIC (required by the UnicastRemoteObject).
2. Start the marquee.xmlrpc.objectcomm.example.XmlRmiServer.
3. Start the marquee.xmlrpc.objectcomm.example.XmlRmiClient.

Again I will just explain the specifics for objectcomm in each class. The XmlRmiEmployeeService class is the same as above except that it ext ends java.rmi.server.UnicastRemoteObject so that it can be used not only with XML-RPC but also with RMI.

The XmlRmiServer starts the XML-RPC server and the RMI registry and registers the same XmlRmiEmployeeService with both of them.

The XmlClient is the same as above but aditionally the operations are performed using RMI as well. As you can see the output for the RMI based query includes the third employee that was stored in the server when doing the XML-RPC based operations before.

That's all you need to know to build applic ations that transparently use one of three modes; local, remote using XML-RPC, and remote using RMI.