

Optimizing the Client/Server Communication for Mobile Applications, Part 2

Version 1.0; May 13, 2003

Java™

NOKIA

Contents

1	Getting Started	4
1.1	Introduction	4
1.2	Architecture and Platforms	4
1.3	Client/Server Applications	5
1.4	CurrencyConverter	5
2	Implementation of Different Protocols	9
2.1	Text Protocol	9
2.2	XML-RPC Protocol	11
2.3	SOAP Protocol	13
3	The Database Layer	15
4	Source Code of the Application	17
4.1	Client	17
4.1.1	CurrencyConverterMIDlet.java	17
4.1.2	RequestCommandFactory.java	20
4.1.3	RequestCommand.java	20
4.1.4	TextRequestCommand.java	21
4.1.5	XMLRPCRequestCommand.java	23
4.1.6	SOAPRequestCommand.java	25
4.1.7	CurrencyConverter.jad	26
4.2	Server	27
4.2.1	CurrencyServletText.java	27
4.2.2	CurrencyServletXmlRpc.java	28
4.2.3	CurrencyWsXmlRpc.java	29
4.2.4	CurrencyWsSoap.java	30
4.2.5	CurrencyDB.java	30
4.2.6	deploy.sh	32
4.2.7	deploy.wsdd	33

Change History

13 May 2003	V1.0	Initial document release
-------------	------	--------------------------

Copyright © 2003 Nokia Corporation. All rights reserved.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Optimizing the Client/Server Communication for Mobile Applications, Part 2

Version 1.0; May 13, 2003

1 Getting Started

1.1 Introduction

The three documents in this series focus on various aspects of a typical client/server environment using mobile devices as the (thin) client. Examples of a typical application that would benefit from this environment might include a networked game or any kind of enterprise application. The goal is to provide a general working framework with a flexible protocol architecture that allows a programmer to choose the perfect mix of program complexity, time to market, and application performance. The implementation of the chosen universal framework could be used to plug in an arbitrary protocol. This will allow the same application to be tested with different protocol techniques to exploit the best solution possible if working in a specific environment (device, operator, transport protocol, etc.).

Part 1 of this series highlighted the critical issues for designing mobile client/server applications and the protocols that could be deployed. Part 2 employs a specific sample program to demonstrate the implementation in practice. Here, emphasis is placed on illustrating how to realize a supplementary abstraction layer, as described in Part 1. In Part 3, due consideration is given to the different transmission times over cellular networks, the size of the MIDlets, and the computation time necessary to process the information by applying performance tests to show the range of differences that exist among the various transport protocols applied in practice.

The sample program is a client/server application that enables actual currency rates to be requested from a database using a cellular phone, thereby permitting conversion of a given sum from one currency into another. The primary purpose of this example is to provide the conceptual development of a client/server architecture and the deployment of various transport protocols for transmitting data over an HTTP connection [HTTPMIDP]. Thus, the application has not been optimized in terms of user friendliness or practical benefits; instead, it has been intentionally kept simple to maintain focus on the client/server architecture itself. For communication with the server on the client end, a framework as described in Part 1 of this series is employed. This framework inserts an additional abstraction layer between the HTTP transport protocol and the application layer.

Three different protocols will be implemented and compared against one another to demonstrate practical deployment of the framework described in Part 1.

1.2 Architecture and Platforms

In addition to a purely text-based protocol, the XML-RPC and SOAP protocols will be implemented for transmitting the client's database requests to the server and returning the results to the client. Several open source products will be set up for the client/server application environment. The open source database MySQL [MYSQL] will be deployed as the server database. Current exchange rates for various currencies will be stored here for conversion use. Another component required on the server is the Tomcat 4.0 Servlet engine [TOMCAT], which is an open source project available from the Apache Software Foundation.

The Servlet engine will be set up to run the various Servlets on the server, each of them acting as server interfaces for the various transport protocols implemented. In order to be able to run the sample application, the plain Servlet engine Tomcat will be extended with additional components from other Apache Software Foundation projects [WSAPACHE]. Among these is the SOAP implementation Axis [AXIS], which provides the framework for Java™ technology-based SOAP Web services, plus Apache's XML-RPC [XMLRPCASF] project for developing XML-RPC-based Web services in Java.

In addition to the Mobile Information Device Profile that acts as the foundation for the client platform, the given sample uses KXML-RPC [KXMLRPC] and kSOAP [KSOAP], two well-known open source projects, each of which is based on the XML parser KXML [KXML]. This parser has been optimized for mobile applications. All three projects are hosted as part of the EnhydraME Project [ENHYDME].

1.3 Client/Server Applications

In Part 2 we will examine the sample application that allows the client software to execute a remote database request, after selecting the desired transport protocol, a value, a source, and target currency. Part 3 will then introduce a short benchmark program, which in a subsequent step applies the various framework classes implemented in the first program to measure the performance of various protocols in respect to the database request. The goal is to determine the time required for calling with each of the three transport protocols—Text, XML-RPC, and SOAP—up until the result reaches the mobile device. Accordingly, the service will be called several times, and the mean value of the time required for the call determined thereafter. This will be performed in succession with all three protocols, so that the times can be compared directly.

Subsequently, it would be interesting to incorporate the latency periods experienced in practice when using different transmission technologies. Whereas test runs of the client on an emulator located on the same computer or in the same network as the server exhibit scarcely any differences among the protocols, such differences become truly noticeable with a GSM or even a GPRS link. A Nokia 3650 device is selected as the reference device, allowing testing with both the slower GSM connection and the generally (but not always) faster GPRS connection.

The following section describes the client application, followed by detailed descriptions of the different transport protocol variants with their respective servers.

1.4 CurrencyConverter

As previously stated, the MIDlet CurrencyConverter is a mobile client application for currency conversions based on the Mobile Information Device Profile 1.0. This program is comprised primarily of the *CurrencyConverterMIDlet* class derived from the *MIDlet* class that implements a *CommandListener* for event handling.



Figure 1: Different transport protocols supported by the application

The necessary user interface elements are initially defined here. These include the form *mainForm* for storing the remaining elements, four text fields for the source and target currency, the output value, and the result. They are supplemented by the *ChoiceGroup transportChoice* employed by the user to select the transport protocol to be deployed. Two other commands, *callCommand* and *exitCommand*, are also defined for calling the calculation and terminating the application. In the constructor of the *CurrencyConverterMIDlet* class, each of the user interface elements is then added to *mainForm*, once the *transportChoice* element has been populated with the values for the three different protocols to be implemented later.

It is also important to note that a *CommandListener* needs to be set up for the form with the parameter *this*, which means that this listener will process the events from the *CurrencyConverterMIDlet* class. The display for the *MIDlet* class is established to ensure that the form *mainForm* will be shown when the application is started using the method *startApp()*. The method *commandAction* is the event handler of this class that serves two tasks—terminating the application when the Exit button is operated and releasing the request to the server when the Calc button is pressed.

We have now reached the interesting part of the application. If this segment of the application is run, it creates a new instance of a class, which implements the interface *RequestCommand*. As described in Part 1, this is achieved by applying the static method of the *RequestCommandFactory* class:

```
RequestCommand rc =
RequestCommandFactory.getRequestCommand(transportChoice.getSelectedIndex())
;
```

First, the method *getSelectedIndex()* of the *transportChoice* object is used to determine the desired transport protocol. The integer value defines which protocol should be deployed. In this case, the number of the index of the *ChoiceGroup* is directly related to the value of the constant for each protocol. This makes it necessary for the order of the entries in the *ChoiceGroup* to be equal to the numeration of the constants defined in the class *RequestCommandFactory*. Although not the most elegant, it is the easiest solution for this sample application. A production release of this framework should use a more elegant solution.

The static method *getRequestCommand()* then returns a specific implementation of the interface *RequestCommand*. This could be one of the three classes *TextRequestCommand*,

XMLRPCRequestCommand, or *SOAPRequestCommand* (described individually in more detail later), each of which implements the command design pattern. After the implementation of the desired protocol is returned through the Factory, the next step is to prepare the procedure call with the help of the relevant methods defined in the *RequestCommand* interface.

The procedure that is called on the server is first selected with the help of the method *setOperation()*. In the sample program, there is basically just one procedure designated as “convert.” Under this designation, the respective procedure call is defined clearly in all implementations of the interface *RequestCommand*. The selection of the operation hence appears as follows:

```
rc.setOperation("convert");
```

Now the parameters for the procedure call are set with the help of the *setParameter()* method. In this case, the three parameters designated as *val*, *src*, and *dst* are passed to the remote procedure. Here, *val* contains the particular currency value, while *src* and *dst* hold the codes for the source and target currencies. The parameters are thus set through the following instructions:

```
rc.setParameter("val", numberField.getString());  
rc.setParameter("src", currField1.getString());  
rc.setParameter("dst", currField2.getString());
```

For each call of the method *setParameter()*, the parameter’s key word is passed and then the parameter’s value. Both parameters of the method are string objects.

Looking at the program code of *CurrencyConverterMIDlet*, it is apparent that thereafter the current system time is saved. This helps to measure the actual run time for the subsequent procedure call, which occurs when the respective method of the object *rc* is called:

```
rc.execute();
```

This calls up the procedure on the server using the chosen transport protocol. Any resulting return value of the remote function is not yet returned by this method. The method *execute()* first returns a value, which confirms whether or not the call was successful. If the value returned is 0, then everything is OK. If a -1 value is returned, then the procedure call was not successful and the program responds with an appropriate error message. If the method call is successful, the system time is looked up, and the difference from the previous value calculated. This is the run time in milliseconds indicated to the user via a later alert.

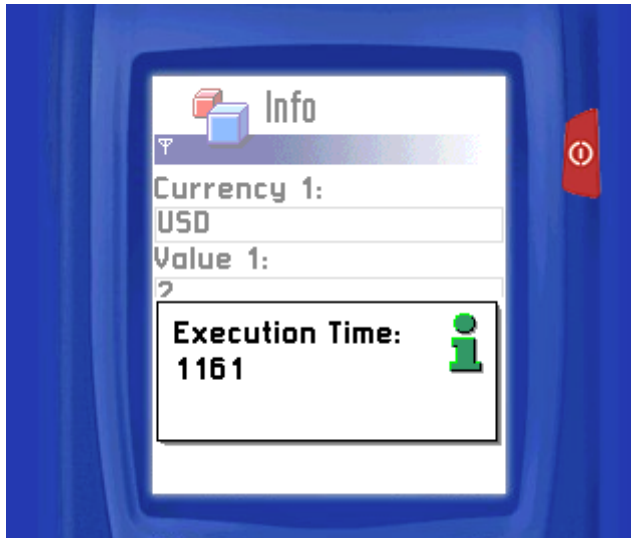


Figure 2: Execution time of the procedure call displayed by an alert

The next important step is to determine the actual result of the remote function call:

```
String number = new String(rc.getResult());
```

Because it was defined in the first part that the method *getResult()* is to return a byte array, the result of the currency conversion is now returned in this form, even though it actually represents a double value. Since the MIDP does not support any double data types, the result here is transformed into a string that can be shown directly in the form's appropriate *TextField resultField*. Since the conversion should only have two digits after the decimal, but may contain many, all subsequent ones are simply truncated.

```
resultField.setString(number.substring(0, number.indexOf(".") + 3));
```

Programmers interested in using the application in practice, should implement at least another rounding function. For the sake of simplicity, it has not been incorporated here.

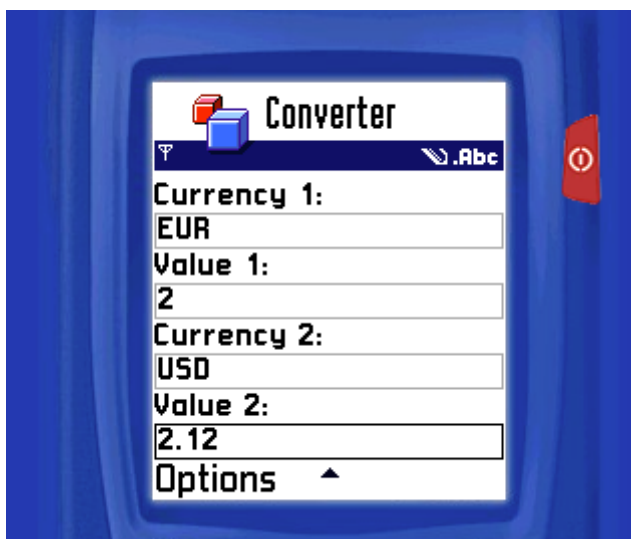


Figure 3: Result of the currency conversion

In this example, it is clearly evident that the entire communication with the server is handled uniformly, regardless of the underlying protocol. The only difference occurs when calling the *getRequestCommand()* method. In that case, one of three constants is passed, as defined in the *RequestCommandFactory* class:

```
static final int REQUEST_PROTOCOL_TEXT = 0;
static final int REQUEST_PROTOCOL_XMLRPC = 1;
static final int REQUEST_PROTOCOL_SOAP = 2;
```

As such, each protocol is assigned a definitive integer value. This is evaluated in the switch instruction when the *getRequestCommand()* is called, and a new instance of the respective implementation of *RequestCommand* is instantiated and returned as a result – as described in Part 1.

2 Implementation of Different Protocols

The following chapter delves into the implementation of each of the different protocols for implementing the *RequestCommand* interface to set up the various alternative implementations that could be returned by the *getRequestCommand()* method.

2.1 Text Protocol

The Text protocol implementation is comprised of a class named *TextRequestCommand*, which implements the *RequestCommand* interface and remotely runs a Servlet called *CurrencyServletText* on the Tomcat Servlet engine (both of these ends are addressed in greater detail below). In this case, the client/server communication should occur through a simple text-based protocol, with the call going through an HTTP-GET method that passes all text parameters in the form of URL parameters. The server then returns the result as a separate octet-stream. This procedure was described thoroughly in Part 1 of this series.

Within the *TextRequestCommand* class, a few variables are first defined for storing certain critical values. The string *URL* stores the Servlet's URL running on the server, and the string *URLParam* takes the parameters that should be passed to the Servlet. The string was previously initialized with a "?", since the URL parameters are introduced with this symbol and separated with an "&". The variable *paramCount* stores the number of parameters already passed, whereas the string *result* is intended for holding the outcome.

The next step is to address the method *setOperation()*. A check is made to determine whether the name given for the call was "convert"; the string *URL* is then accordingly assigned the Servlet's address responsible for the operation "convert".

```
if(operation.equals ("convert")) URL =
"http://127.0.0.1:8080/currency/servlet/CurrencyServletText";
```

When deploying the Servlet on a server in the Internet, the URL must be modified accordingly. The address given above merely refers to the development environment where the server runs locally on the same computer as the MIDP emulator itself. If there are additional remote functions on the server, each would then be assigned its own key word. In its stead, the respective URL could be assigned, regardless of whether it is hosted on another Servlet or even on another server.

Each of the remote procedure parameters is then passed in the form of key/value pairs through a sequence of calls of the method *setParameter()*. These are appended to the URL by enhancing each *URLParam* string with the key/value pair separated by an "=" sign.

```
URLParam = URLParam + key + "=" + value;
```

Attention should be paid to make sure that an “&” symbol is entered before each key/value pair, with the exception of the first pair. For this reason, the number of parameters previously passed is also counted, and the symbol is entered only if the *paramCount* does not equal 0. At the end of each call, the variable *paramCount* is incremented by one.

Now we reach the most critical part – calling the Servlet. The method *execute()* calls the static method *open()* of the Connector class, and thereby passes on the URL and its parameters in a continuous string:

```
HttpConnection conn = (HttpConnection) Connector.open (URL + URLParam);
```

If the result of this call is null then the error code -1 is returned; otherwise, the parameters for the connection are set. First the connection method is set to HTTP GET by the method *setRequestMethod()*, as discussed in Part 1:

```
conn.setRequestMethod (HttpConnection.GET);
```

In the next step, various attributes are set, which are transmitted during the request inside the HTTP header:

```
conn.setRequestProperty ( "User-Agent", "Profile/MIDP-1.0  
Configuration/CLDC-1.0");  
conn.setRequestProperty ( "Accept", "application/octet-stream");  
conn.setRequestProperty ( "Connection", "close");
```

This header defines the type of user agent that is accessing the server as well as the MIME type of the response accepted by the client. The property “Connection” signals that the active connection must be closed by the server after returning the response to the client.

The Servlet is called by executing the method *openInputStream()* of the class Connection, and a *DataInputStream* containing the result of the procedure call is created based on the returned *InputStream*. Since the result is of type String, it is read with the help of the method *readUTF()* of the *DataInputStream* and assigned to the string result if the stream was opened successfully. The *DataInputStream* and the *HttpConnection* are then closed. It is important to catch any likely *IOException* and to verify if the *DataInputStream* is not equal to a null value. A -1 value will always be returned if an error occurs. (More precise handling of connection errors is definitely recommended if the program is used in a production environment.) The method *getResult()* then returns the result of the remote procedure call, converting the string result into a byte array, and is returned to the *CurrencyConverterMIDlet* class.

Meanwhile, at the other end of the network connection, the Servlet *CurrencyServletText* derived from the *HttpServlet* class is running on the Tomcat server. Here, the method *doGet()*, in which the HTTP request resulting from the call on the mobile client is processed, is important.

First, a new object from the *CurrencyDB* class is instantiated, as explained later in more detail. This sets up the link to the database. If the database has been initiated successfully with the method *initDatabase()*, the currency conversion can now be carried out by calling the method *calcCurrency()*:

```
str =  
Double.toString (db.calcCurrency (Double.parseDouble (request.getParameter ("va  
l")), request.getParameter ("src"), request.getParameter ("dst")));
```

A double value and two strings with the source and target currencies are passed to the method *calcCurrency()*. The method then automatically calculates the result based on the information in the database, and returns it as a double value. The method *getParameter()* of the *HttpServletRequest* class is used to retrieve the parameters transmitted using the URL; these are subsequently passed as parameters to the method *calcCurrency()*. However it is important to note that the parameter *val* must first be converted into a double value. This is done with the static method *parseDouble()* of the class *Double*. In precisely the same manner, the result returned by the method *calcCurrency()* must again be converted from a double type into a *String* object. This is achieved with the static method *toString()*, likewise of the *Double* class.

In the next step, a new *DataOutputStream* based on a similarly created *ByteArrayOutputStream* is instantiated. These two streams are used to serialize the *String* object into a byte array, which thereafter can be transmitted to the client. This is achieved by first executing the method *writeUTF()* of the class *DataOutputStream* and then calling the method *toByteArray()* of the underlying *ByteArrayOutputStream* class. After closing both streams, the content of the string is available in serialized form in the byte array *data*.

As with the client side, there are attributes assigned to the HTTP header of the response message:

```
response.setContentType("application/octet-stream");
response.setContentLength(data.length);
```

This is actually the content type requested by the client as well as the length of the transmitted message. The content type in this case has to be “application/octet-stream” because of the binary data that is transferred over the HTTP connection. The reason for serializing the *String* first into a byte array and not directly into the connection stream to the client is the necessity of determining its length in serialized form.

Now the data is ready to be transferred to the mobile client. An *OutputStream* for the connection is determined through the method *getOutputStream()*:

```
OutputStream out = response.getOutputStream();
```

This stream serves to return the result to the client again. Finally, the result is written into the *OutputStream* by its *write()* method, which is closed thereafter. The server side of the call is also complete.

2.2 XML-RPC Protocol

A comprehensive look at the implementation of the XML-RPC protocol is next. As stated earlier, this relates to a client-side application of the XML-RPC implementation KXML-RPC from Enhydra, introduced briefly in Part 1. It is based on the KXML parser and must be deployed together with the parser on the device. However, here the kxml-min variant, which does not integrate DOM and sacrifices the WBXML [COMPXML] support, suffices.

Again, development of the XML-RPC implementation in the framework is very similar to that of a proprietary protocol. Likewise, the class *XMLRPCRequestCommand* implements the interface *RequestCommand* and thereby provides the same methods that are recognizable from the class *TextRequestCommand*. The most important variables in the class are the vector *params* (subsequently instantiated in the constructor), the *XmlRpcClient xmlrpc*, and the string *result* for holding the outcome of the procedure call. In this class as well, the method *setOperation()* must first be called; this then verifies if the operation “convert” is involved and whether or not it should be called. If it should be called, a new

instance of the *XmlRpcClient* class will be created. Accordingly, the Servlet's complete URL will be passed as a parameter, from which the server side XML-RPC service will be provided:

```
if(operation.equals("convert")) xmlrpc = new XmlRpcClient (
"http://127.0.0.1:8080/currency/servlet/CurrencyServletXmlRpc" );
```

Developers will only be directly involved with the *XmlRpcClient* class of the KXML-RPC API. It is already obvious that the API is very compact, since it sacrifices the entire overhead. The object of the *XmlRpcClient* class instantiated as such is assigned to the class variable *xmlrpc*, so that it can be worked with at a later point in time.

The method *setParameter()* that was found in the *TextRequestCommand* class will also be implemented in a very simple way by the *XMLRPCRequestCommand*. The previously instantiated vector *params* for holding the procedure call parameters, is assigned another parameter with the method *addElement()*:

```
params.addElement(value);
```

Here, too, the development of KXML-RPC is stripped to its essentials. Instead of defining a new class for holding the procedure call parameter, this solution uses the class vector that exists in MIDP. The parameter value itself is simply stored in the vector. The string *key* passed through the method *setParameter()* is not considered at all, since for this XML-RPC implementation it is merely the sequence of the elements within the vector that is important. However, the parameters of the procedure call on the server must also be set in the proper sequence with help of the *setParameter()* method. Although the sequence of setting the parameters through the framework is not relevant for both the Text and SOAP protocols, it is important to pay close attention to this detail when using the XML-RPC implementation.

The remote procedure call on the server is again called through the method *execute()* of the framework class. This basically consists of a call of the same named method *execute()* of the *XmlRpcClient* class, which is passed a string as a parameter. This string specifies the XML-RPC handler on the server and the remote method name that should be called. These are separated by a dot within the string:

```
result = (String) xmlrpc.execute( "CurrencyWsXmlRpc.calcCurrency", params
);
```

This XML-RPC call results in the method *calcCurrency()* being called on the server, which is provided by the handler *CurrencyWsXmlRpc* of the respective Servlet. The vector object, which contains the previously passed parameters for the remote procedure, is passed as the second parameter to the method *execute()*. Following a successful call, the result is returned as a string and passed to the variable *result*. This call again requires the handling of potential exceptions. If an error occurs, a -1 is returned, otherwise a 0 is returned. In this case, the method *getResult()* for requesting the result after the procedure call, looks exactly the same as when implementing the Text protocol.

On the server side of the XML-RPC implementation, again a Servlet is set up for accepting client requests. The Servlet is represented by the *CurrencyServletXmlRpc* class and is responsible for the procedure call on the server. The wrapper class *CurrencyWsXmlRpc* represents the remote object itself. As previously stated, the XML-RPC project from Apache is applied for accomplishing the server-side part of the application.

When the Servlet is loaded into the Tomcat server its method *init()* is called. This instantiates a new object of the class *XmlRpcServer*, which stems from the Apache project and is part of the *org.apache.xmlrpc* package. The object is then assigned to the class variable *xmlrpc* for further use. The next step is to assign an XML-RPC handler to the instance of *XmlRpcServer* through its method *addHandler()*:

```
xmlrpc.addHandler ("CurrencyWsXmlRpc", new CurrencyWsXmlRpc());
```

Here, the handler name along with a new instance of the prescribed wrapper class is passed, the latter representing the remote object. Through object introspection, the Apache XML-RPC implementation is capable of automatically establishing which methods are provided by the object, and can publish them externally as an XML-RPC Web service. In this sample program, however, only the single public method `calcCurrency()` is published as a Web service.

Calling the Web service leads to running of the Servlet's method `doPost()`. Inside this method the procedure call finally occurs through the method `execute()` of the `XmlRpcServer` class:

```
byte[] result = xmlrpc.execute (request.getInputStream ());
```

An `InputStream` is now passed as a parameter containing the XML-RPC request in the form of an XML message transmitted from the client to the Servlet. For returning the `InputStream`, the method `getInputStream()` of the `HttpServletRequest` class is called. The `execute()` method finally returns the result of the remote procedure call as a byte array. An `OutputStream` is then ascertained with the help of the method `getOutputStream()` of the class `HttpServletResponse`, and used to return the information to the client. The method `write()` of the `OutputStream` transmits the returned byte array, incorporating the XML message containing the result of the procedure call, back to the client for additional processing as described above.

It is also necessary to ensure that the result's content type and the length are defined before being returned to the client:

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

We'll now take a brief look at the wrapper class `CurrencyWsXmlRpc`, mentioned earlier. This incorporates the method `calcCurrency()`, which is called by the Web service. The currency conversion is done just as with the Text protocol. A new instance of the `CurrencyDB` class is first created and then the calculation is done if the connection to the database is initialized successfully:

```
return Double.toString(db.calcCurrency(Double.parseDouble(val), src, dst));
```

It is immediately apparent that the wrapper class neither receives nor returns the currency values as doubles. The call will return parameters in the more generic String format. The reason for this is that although the general XML-RPC protocol supports double data types, the actual KXML-RPC implementation's latest version 0.6 still does not support double values on the client side. These data types ought be supported in a future version and integrated into KXML-RPC. Until then, as for the sample program presented, the double data types must still be sent as strings over the HTTP connection. This is not really a disadvantage, but rather a minor compatibility related limitation when accessing third-party XML-RPC Web services.

2.3 SOAP Protocol

As mentioned in the introduction, the third protocol example to be implemented for the framework is SOAP. The class `SOAPRequestCommand` is made available to implement the interface `RequestCommand` in precisely the same way as for the classes `TextRequestCommand` and `XMLRPCRequestCommand`. The API from the kSOAP project is now used on the client side to implement the SOAP client. The following are first defined as private class variables of the `SOAPRequestCommand` class – `ht` of the type `HttpTransport`, `rpc` of the type `SoapObject`, and `classMap` of the type `ClassMap`. For this implementation, the string `result` that again holds the result of the procedure call, is also defined.

To begin, in instantiating the *SOAPRequestCommand* class, a new object of the *ClassMap* class is created in its constructor. This is needed later to provide a *ClassMap*, when applying the data type *SoapPrimitive* to represent double data types. As for the other two implementations, the method *setOperation()* is again implemented here to verify if the operation “convert” was specified. After that, a new instance of the *SoapObject* class is created:

```
rpc = new SoapObject("CurrencyWS", "calcCurrency");
```

Two parameters are passed: the namespace of the SOAP object, in this case *CurrencyWS*, plus the name of the SOAP object designated *calcCurrency* in this example. Now an instance of the *HttpTransport* class actually responsible for setting up the connection with the server must be created:

```
ht = new HttpTransport("http://127.0.0.1:8080/axis/services/CurrencyWS",
"/"/");
```

The target URL of the SOAP Web service is specified; this is the address where the server implementation awaits the client’s request. It will become clear later how the server side is set up in this case. The SOAP action, which has to be included in the HTTP header, is the second parameter passed to the constructor of *HttpTransport*. However, it is not necessary in this example, and hence just a blank action should be passed. Due to a special behavior of various MIDP HTTP implementations on actual devices like the Nokia 3650 phone, two quotations marks must be included in the string, otherwise the SOAP action property is deleted in the HTTP header by these specific implementations. However, this is no problem for the client, because the SOAP specification allows both forms of defining an empty SOAP action – with and without quotation marks.

Now the previously created *ClassMap* is assigned to the *HttpTransport* object:

```
ht.setClassMap (classMap);
```

In the next step, each of the parameters for the remote procedure is again passed to the framework by the method *setParameter()*. Both parameters *src* and *dst* can then be passed directly to the *SoapObject* *rpc* through the method *addProperty()*.

```
rpc.addProperty ("src", value);
rpc.addProperty ("dst", value);
```

As mentioned, the parameter *val* calls for special treatment in this case—that is, it should not be passed as a string as with the XML-RPC implementation; it should instead appear as a double data type in the XML message of the SOAP protocol. Since kSOAP supports this data type directly, taking advantage of this option achieves a higher level of compatibility. Given that MIDP itself does not support any double data types, it is necessary to create an object of the *SoapPrimitive* class passing the double value as a parameter in the form of a string to the constructor containing the value read from the input field:

```
rpc.addProperty ("val", new SoapPrimitive (classMap.xsd, "double", value));
```

The previously created *ClassMap* must be specified, along with the data type that should be reflected by the string passed. This is passed as the second parameter in the form of a string.

If all parameters have been passed by *setParameter()*, the remote procedure call can take place with the help of the method *execute()*. The implementation of this method then uses the method *call()* of the *HttpTransport* class to call the SOAP Web service:

```
result = (ht.call(rpc)).toString();
```

The result returned is a double value as an object of the *SoapPrimitive* class, converted into a string with the method *toString()*, and passed to the variable *result*. Here, too, any exceptions that occur need to be caught. The result of the method *execute()* is -1 if there is an error, or 0 if the result is determined successfully. In this situation, the method *getResult()*, designed to request the results following a remote procedure call, looks the same as in the case of the Text and XML-RPC protocol implementations.

On the server side, the SOAP protocol implementation has proven to be much easier than the Text or XML-RPC protocols. Since the applied Axis SOAP server from Apache offers the option of publishing any desired Java class as a SOAP Web service, it is merely necessary to write an appropriate wrapper class incorporating the *calcCurrency()* method, which can be called automatically through SOAP. This saves developers the effort of implementing their own Servlet. In principle, the wrapper class named *CurrencyWsSoap* is built exactly like the wrapper class *CurrencyWsXmlRpc* of the XML-RPC Web service. The difference, however, is that the SOAP variant actually uses double data types for the parameter *val* and for the return value of the remote procedure. In the XML-RPC implementation, these are replaced with the type *String*, since the client implementation KXML-RPC does not support any double data types.

Once Axis has been installed on the Tomcat server, only the class must be deployed as a Web service, and the server side application is ready to receive requests. We will cover the installation and the deployment in detail in Part 3 of this series.

3 The Database Layer

We will now take a look at accessing the database on the server with the *CurrencyDB* class. This class represents an interface to the database that can be accessed through the JDBC protocol. It helps to look up the latest exchange rates for the two specified currencies in the database, and to provide the conversion of the given sum into the target currency. With a more complicated business logic, it would make sense to encapsulate this functionality into an Enterprise Java Bean (EJB). However, since the logic involved here is easy and the program is supposed to be kept as simple as possible, this can be averted by simply providing an additional class. This class takes on the database's abstraction for the Web service.

Prior to applying the class *CurrencyDB* for accessing the database, it is necessary to call the method *initDatabase()*. In all examples, we call this function every time a request is made by the client to make the different protocols more comparable. In a production environment it would be better to make the initialization of the database once when the corresponding Servlet or wrapper class is first instantiated.

The method initializes the JDBC driver by calling the method *forName()* of the class *Class*:

```
Class.forName("com.mysql.jdbc.Driver");
```

Such a call normally returns the *Class* object of the specified class, but in this case it is applied to initialize the class of the given JDBC driver for the MySQL database. If the specified class of the JDBC driver cannot be found, a *ClassNotFoundException* is thrown and the initialization method returns with a false value, indicating an error during database initialization.

Once the JDBC driver has been initialized successfully, the method *calcCurrency()* can be called to perform the conversion. The *getCurrency()* method is called twice to read each of the exchange rates from the database, and to then return the value calculated:

```
double dsrc = getCurrency(src);
double ddst = getCurrency(dst);
```

```
return val * ddst / dsrc;
```

At this point a consistency check is made to test if one of the exchange rates contains a value of -1, which signifies that an error has occurred and the method *calcCurrency()* is terminated with a result value of -1.

The method *getCurrency()* arranges access to the database. A new connection is first established on a call, whereby the method *getConnection()* of the *DriverManager* class is called:

```
conn = DriverManager.getConnection(
    "jdbc:mysql://localhost/CurrencyDB?user=remote&password=remote");
```

In line with the syntax of the particular JDBC driver, both the database's host and name must be specified for the connection to be set up. The host name "localhost" is used, while the database is located on the same machine as the Tomcat server on which the Servlet will be run. If the database is located elsewhere, the applicable IP address or the host name of the database's server must be passed followed by the database's name and a valid user name and password. An object of the *Statement* class can then be created by calling the *createStatement()* method of the *Connection* class, once the connection is established successfully. The SQL instruction is then defined and run with the method *executeQuery()*:

```
String query = "SELECT * FROM currencies where Currency='" + curr + "'";
ResultSet rs = stmt.executeQuery(query);
```

The database named *CurrencyDB*, which must be installed on the MySQL database server, contains only one table with two columns. The fields of the column *Currency* contain the three-letter currency codes defined by the ISO 4217 standard, which must be input at the client later for the conversion. The second column named *Value* contains the exchange rates that obviously must all have the same reference currency to allow a conversion to be made. The value of the respective currency code is now determined through an SQL request, and provided as an object of the type *ResultSet*. With a call of the method *first()*, one can scroll within *ResultSet* to the first set of data, and look up the relevant value with the help of the method *getDouble()*:

```
rs.first();
double d = rs.getDouble("Value");
```

As long as the database does not have redundant entries, *ResultSet* should contain only this single value. It is important that for database operations, any exceptions that occur are handled and the connection to the database must be closed again using the *close()* method. In the event that an exception occurs and the requested exchange rate cannot be read from the database, a value of -1 will be returned to point out to the calling class that a result cannot be determined. Otherwise, the calculated result for the currency conversion is returned.

Having discussed in detail the implementation of the sample application and the different transport protocols, in addition to defining the framework in Part 1 of this series, we will focus on the performance issues of these protocols and implementation of a profiling application in Part 3.

4 Source Code of the Application

4.1 Client

The following section contains the source code of the *CurrencyConverter* application's classes, which run in the MIDP environment on the mobile client.

4.1.1 CurrencyConverterMIDlet.java

```

/*
 * CurrencyConverterMIDlet.java
 *
 * Client application for calculating currencies
 */

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class CurrencyConverterMIDlet extends MIDlet implements
CommandListener
{

    // Instantiate user interface components

    private Form mainForm = new Form ("Converter");

    private TextField numberField = new TextField ("Value 1: ", "2", 10,
    TextField.NUMERIC);

    private TextField resultField = new TextField ("Value 2: ", "", 10,
    TextField.ANY);

    private Command callCommand = new Command ("Calc", Command.SCREEN, 1);
    private Command exitCommand = new Command ("Exit", Command.SCREEN, 1);

    private TextField currField1 = new TextField ("Currency 1: ", "USD", 3,
    TextField.ANY);

    private TextField currField2 = new TextField ("Currency 2: ", "EUR", 3,
    TextField.ANY);

    private ChoiceGroup transportChoice = new ChoiceGroup ("Transport",
    Choice.EXCLUSIVE );

    private Display dsp;

    public CurrencyConverterMIDlet ()
    {
        // Define mainForm

        transportChoice.append ("TEXT", null);
        transportChoice.append ("XMLRPC", null);
        transportChoice.append ("SOAP", null);
    }
}

```

```
mainForm.append (transportChoice);

mainForm.append (currField1);
mainForm.append (numberField);
mainForm.append (currField2);
mainForm.append (resultField);

mainForm.addCommand (callCommand);
mainForm.addCommand (exitCommand);
mainForm.setCommandListener (this);

dsp = Display.getDisplay (this);
}

public void startApp ()
{
    dsp.setCurrent (mainForm);
}

public void pauseApp ()
{
}

public void destroyApp (boolean unconditional)
{
}

public void commandAction (Command c, Displayable d)
{
    if (c == exitCommand)
    {
        // Exit application
        destroyApp (false);
        notifyDestroyed ();
    }
    else if(c == callCommand)
```

```

    {
        // Get object which implements the requested protocol
        RequestCommand rc = RequestCommandFactory.getRequestCommand
            (transportChoice.getSelectedIndex ());

        // Define Operation
        rc.setOperation ("convert");
        rc.setParameter ("val", numberField.getString ());
        rc.setParameter ("src", currField1.getString ());
        rc.setParameter ("dst", currField2.getString ());

        long timer = System.currentTimeMillis ();

        // Execute remote procedure
        if (rc.execute ()==0)
        {
            timer = System.currentTimeMillis ()-timer;
            String number = new String (rc.getResult ());

            // Display the result of the call
            resultField.setString (number.substring (0, number.indexOf
                (".") + 3));

            // Display execution time
            dsp.setCurrent (new Alert ("Info", "Execution Time: " + new
                Long (timer), null, AlertType.INFO));
        }
        else
            dsp.setCurrent (new Alert ("Error", "Connection failed!", null,
                AlertType.ERROR));
    }
}
}
}

```

4.1.2 RequestCommandFactory.java

```

/*
 * RequestCommandFactory.java
 *
 * Factory class that returns an instance of a requested protocol

```

```

*/

public class RequestCommandFactory
{

    // Define constants for the supported protocol type
    static final int REQUEST_PROTOCOL_TEXT = 0;
    static final int REQUEST_PROTOCOL_XMLRPC = 1;
    static final int REQUEST_PROTOCOL_SOAP = 2;

    public static RequestCommand getRequestCommand (int requestProtocol)
    {

        // Check for requested protocol and return instance
        switch (requestProtocol)
        {
            case REQUEST_PROTOCOL_TEXT:
                return new TextRequestCommand ();
            case REQUEST_PROTOCOL_SOAP:
                return new SOAPRequestCommand ();
            case REQUEST_PROTOCOL_XMLRPC:
                return new XMLRPCRequestCommand ();
            default:
                // Default protocol
                return new TextRequestCommand ();
        }
    }
}

```

4.1.3 RequestCommand.java

```

/*
 * RequestCommand.java
 *
 * Interface that defines a request command
 */

public interface RequestCommand
{

    public void setOperation (String operation);
}

```

```

    public void setParameter (String key, String value);

    public int execute ();

    public byte[] getResult ();

}

```

4.1.4 TextRequestCommand.java

```

/*
 * TextRequestCommand.java
 *
 * Implementation of the Text protocol
 */

import java.io.*;
import javax.microedition.io.*;

public class TextRequestCommand implements RequestCommand
{

    private String URL = "";
    private String URLParam = "?";
    private int paramCount = 0;
    private String result = "";

    public TextRequestCommand ()
    {

    }

    public int execute ()
    {
        try
        {
            // Open new HTTP connection to the server
            HttpURLConnection conn = (HttpURLConnection) Connector.open (URL +
            URLParam);

            if(conn == null)
                return -1;
            else

```

```
{
    // Define request method
    conn.setRequestMethod (HttpConnection.GET);

    // Define attributes for HTTP header
    conn.setRequestProperty ( "User-Agent", "Profile/MIDP-1.0
    Configuration/CLDC-1.0");
    conn.setRequestProperty ( "Accept", "application/octet-
    stream");
    conn.setRequestProperty ( "Connection", "close");

    // Open DataInputStream to read result from server
    DataInputStream di = new DataInputStream (conn.openInputStream
    ());

    if(di == null)
        return -1;
    else
    {
        // Read result of the remote procedure call
        result = di.readUTF ();

        // Close all
        di.close ();
        conn.close ();
    }
}
}
catch(IOException e)
{
    return -1;
}

return 0;
}

public byte[] getResult ()
{
```

```

        // Return the result of the remote procedure call
        return result.getBytes ();
    }

    public void setOperation (String operation)
    {
        // Set the host and the servlet of the remote service
        if(operation.equals ("convert")) URL =
            "http://127.0.0.1:8080/currency/servlet/CurrencyServletText";
    }

    public void setParameter (String key, String value)
    {
        // Set parameters for remote procedure
        if (paramCount != 0) URLParam += "&";
        URLParam = URLParam + key + "=" +value;
        paramCount++;
    }
}

```

4.1.5 XMLRPCRequestCommand.java

```

/*
 * XMLRPCRequestCommand.java
 *
 * Implementation of the XML-RPC protocol
 */
import java.util.Vector;
import org.kxmlrpc.*;

public class XMLRPCRequestCommand implements RequestCommand
{
    private Vector params;
    private XmlRpcClient xmlrpc;
    private String result="";

    public XMLRPCRequestCommand ()
    {
        params = new Vector ();
    }
}

```

```

public int execute ()
{
    try
    {
        // Execute remote procedure calcCurrency handled by the
        CurrencyWsXmlRpc handler
        result = (String) xmlrpc.execute ( "CurrencyWsXmlRpc.calcCurrency",
        params );
    }
    catch (Exception e)
    {
        return -1;
    }
    return 0;
}

public byte[] getResult ()
{
    // Return the result of the remote procedure call
    return result.getBytes ();
}

public void setOperation (String operation)
{
    // Set the host and the servlet of the remote service
    if(operation.equals ("convert")) xmlrpc = new XmlRpcClient (
    "http://127.0.0.1:8080/currency/servlet/CurrencyServletXmlRpc" );
}

public void setParameter (String key, String value)
{
    // Add the parameters for remote procedure
    params.addElement (value);
}
}

```

4.1.6 SOAPRequestCommand.java

```
/*
```

```
* SOAPRequestCommand.java
*
* Implementation of the SOAP protocol
*/

import java.io.*;
import org.ksoap.*;
import org.ksoap.transport.*;

public class SOAPRequestCommand implements RequestCommand
{

    private HttpTransport ht = null;
    private SoapObject rpc = null;
    private ClassMap classMap = null;
    private String result = "";

    public SOAPRequestCommand ()
    {
        classMap = new ClassMap ();
    }

    public int execute ()
    {
        try
        {
            // Execute remote procedure
            result = (ht.call (rpc)).toString ();
        }
        catch (IOException e)
        {
            return -1;
        }
        return 0;
    }

    public byte[] getResult ()
    {
        // Return the result of the remote procedure call
        return result.getBytes ();
    }
}
```

```

public void setOperation (String operation)
{
    if(operation.equals ("convert"))
    {
        // Create SOAP object
        rpc = new SoapObject ("CurrencyWsSoap", "calcCurrency");

        // Initilize connection to server
        ht = new HttpTransport
            ("http://127.0.0.1:8080/axis/services/CurrencyWsSoap", "\\\"");
        ht.setClassMap (classMap);
    }
}

public void setParameter (String key, String value)
{
    // Set parameters for remote procedure
    if (key.equals ("val")) rpc.addProperty ("val", new SoapPrimitive
        (classMap.xsd, "double", value));
    else if (key.equals ("src")) rpc.addProperty ("src", value);
    else if (key.equals ("dst")) rpc.addProperty ("dst", value);
}
}

```

4.1.7 CurrencyConverter.jad

```

MIDlet-Version: 1.0
MIDlet-Vendor: Administrator
MIDlet-Jar-URL: CurrencyConverter.jar
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
MIDlet-1: Converter, , CurrencyConverterMIDlet
MIDlet-Name: CurrencyConverter
MIDlet-Jar-Size: 63127

```

4.2 Server

The following section gives an overview of the source code of the Servlets running on the Tomcat server.

4.2.1 CurrencyServletText.java

```

/*

```

```
* CurrencyServletText.java
*
* Servlet providing the server side part of the Text protocol
*/

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CurrencyServletText extends HttpServlet
{

    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException
    {
        String str = "-1";
        CurrencyDB db = new CurrencyDB();

        if(db.initDatabase())
        {
            // Calculate currency if database has been initialiced succcessfully
            str = Double.toString( db.calcCurrency( Double.parseDouble(
            request.getParameter("val")), request.getParameter("src"),
            request.getParameter("dst")));
        }

        // Open output streams and serialize string into a byte array
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        DataOutputStream dout = new DataOutputStream( bout);
        dout.writeUTF(str);
        byte[] data = bout.toByteArray();
        bout.close();
        dout.close();

        // Set HTTP header attributes
        response.setContentType("application/octet-stream");
        response.setContentLength(data.length);
    }
}
```

```

        // Open output stream to client and write response
        OutputStream out = response.getOutputStream();
        out.write( data);
        out.close();
    }

    public String getServletInfo()
    {
        return "Servlet providing the server side part of the Text protocol";
    }
}

```

4.2.2 CurrencyServletXmlRpc.java

```

/*
 * CurrencyServletXmlRpc.java
 *
 * Servlet providing the server side part of the XML-RPC protocol
 */

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.xmlrpc.*;

public class CurrencyServletXmlRpc extends HttpServlet
{
    XmlRpcServer xmlrpc;

    public void init(ServletConfig config) throws ServletException
    {
        // instantiate XmlRpcServer and add handler
        xmlrpc = new XmlRpcServer();
        xmlrpc.addHandler("CurrencyWsXmlRpc", new CurrencyWsXmlRpc());

        super.init(config);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException
    {

```

```

// Get input stream from client and execute procedure
byte[] result = xmlrpc.execute(request.getInputStream());

// Set HTTP header attributes
response.setContentType("text/xml");
response.setContentLength(result.length);

// Open output stream to client and write response
OutputStream out = response.getOutputStream();
out.write(result);
out.close();
}

public String getServletInfo()
{
    return "Servlet providing the server side part of the XML-RPC
    protocol";
}
}

```

4.2.3 CurrencyWsXmlRpc.java

```

/*
 * CurrencyWsXmlRpc.java
 *
 * Wrapper class for XML-RPC webservice
 */

public class CurrencyWsXmlRpc
{
    public CurrencyWsXmlRpc()
    {
    }

    public String calcCurrency(String val, String src, String dst)
    {
        CurrencyDB db = new CurrencyDB();

        // Calculate currency if database has been initialized succcessfully
    }
}

```

```

        if (db.initDatabase()) return Double.toString( db.calcCurrency(
            Double.parseDouble(val), src, dst)); else return "-1";
    }
}

```

4.2.4 CurrencyWsSoap.java

```

/*
 * CurrencyWsSoap.java
 *
 * Wrapper class for SOAP webservice
 */

public class CurrencyWsSoap
{

    public CurrencyWsSoap()
    {
    }

    public double calcCurrency(double val, String src, String dst)
    {
        CurrencyDB db = new CurrencyDB();

        // Calculate currency if database has been initialized succcecsfully
        if (db.initDatabase())
            return db.calcCurrency(val, src, dst);
        else
            return -1;
    }
}

```

4.2.5 CurrencyDB.java

```

/*
 * CurrencyDB.java
 *
 * Class for accessing CurrencyDB on the MySQL Server
 */

import java.sql.*;

public class CurrencyDB
{

```

```
public CurrencyDB()
{
}

public boolean initDatabase()
{
    try
    {
        // Load JDBC Driver for MySQL
        Class.forName("com.mysql.jdbc.Driver");
    }
    catch(ClassNotFoundException e)
    {
        e.printStackTrace();
        return false;
    }
    return true;
}

public double calcCurrency(double val, String src, String dst)
{
    double dsrc = getCurrency(src);
    double ddst = getCurrency(dst);
    if ((dsrc != -1) & (ddst != -1))
        // Calculate value for destination currency
        return val * ddst / dsrc;
    else
        return -1;
}

public double getCurrency(String curr)
{
    Connection conn = null;
    try
    {
        // Connect to the database CurrencyDB
    }
}
```

```

conn = DriverManager.getConnection(
    "jdbc:mysql://localhost/CurrencyDB?user=remote&password=remote");

Statement stmt = conn.createStatement();

// Create query and execute it
String query = "SELECT * FROM currencies where Currency='" + curr +
    "'";
ResultSet rs = stmt.executeQuery(query);

// Goto first entry in resut set and return the value
rs.first();
double d = rs.getDouble("Value");
conn.close();
return d;
}
catch(SQLException e)
{
    e.printStackTrace();
    return -1;
}
}
}

```

4.2.6 deploy.sh

Here is the shell script for Linux, which executes the administration client of the Axis SOAP server to deploy the SOAP Web service defined in the file *deploy.wsdd*.

```

#!/bin/sh
CATALINA_HOME=/opt/jakarta/tomcat
AXIS_LIB=$CATALINA_HOME/webapps/axis/WEB-INF
CLASSPATH=$AXIS_LIB/lib/axis.jar
CLASSPATH=$CLASSPATH:$AXIS_LIB/lib/jaxrpc.jar
CLASSPATH=$CLASSPATH:$AXIS_LIB/lib/commons-discovery.jar
CLASSPATH=$CLASSPATH:$AXIS_LIB/lib/commons-logging.jar
CLASSPATH=$CLASSPATH:$AXIS_LIB/lib/saaj.jar
CLASSPATH=$CLASSPATH:$CATALINA_HOME/common/lib/servlet.jar
export CLASSPATH
java org.apache.axis.client.AdminClient deploy.wsdd

```

4.2.7 deploy.wsdd

This XML descriptor file defines the SOAP Web service named *CurrencyWsSoap*. All methods of the class *CurrencyWsSoap* are accessible via HTTP.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="CurrencyWsSoap" provider="java:RPC">
    <parameter name="className" value="CurrencyWsSoap"/>
    <parameter name="allowedMethods" value="*"/>
  </service>
</deployment>
```

5 Summary and Conclusion

In Part 2 of our three part series, we've provided the universal framework for a client/server application. Different protocols could be used as plug-ins without a need to change the client or the server application. This makes it easy to optimize the protocol layer according to the application itself as well as the transport protocol and the provider specific data transmission rates.

6 References

- AXIS** Web site of the Axis project of the Apache Software Foundation
<http://ws.apache.org/axis/index.html>
- COMPXML** *Compressing XML for Faster Wireless Networking*
<http://wireless.java.sun.com/midp/ttips/compressxml/>
- ENHYDME** Web site of Enhydra ME Software
<http://me.enhydra.org/software/index.html>
- HTTPMIDP** *Making HTTP Connections with MIDP*
<http://wireless.java.sun.com/midp/ttips/httpcon/>
- KSOAP** Web site of the Enhydra kSOAP project
<http://ksoap.enhydra.org/>
- KXML** Web site of the Enhydra KXML project
<http://kxml.enhydra.org/>
- KXMLRPC** Web site of the Enhydra KXML-RPC project
<http://kxmlrpc.enhydra.org/>
- MIDSERV** *Client-Server Communications between MIDlets and Servlets*

	http://wireless.java.sun.com/midp/ttips/clientserv/
MYSQL	Web site of the MySQL open source database project http://www.mysql.com/
PERFAN	<i>A Performance Analysis of Web Services on Wireless PDAs</i> http://www.cs.duke.edu/~vkb/advnw/project/index.html
TOMCAT	Web site of the Jakarta Tomcat project of the Apache Software Foundation http://jakarta.apache.org/tomcat/index.html
WCLIENT	Sun Blueprint “ <i>Designing Wireless Clients for Enterprise Applications with Java Technology</i> ” http://java.sun.com/blueprints/earlyaccess/wireless/designing/designing.html
WEBMIDP	<i>A Brief Introduction to MIDP Clients for Web Services v1.0, Forum Nokia, 2003</i> http://nds1.forum.nokia.com/nnds/ForumDownloadServlet?id=2907
WSAPACHE	Web Services Project @ Apache http://ws.apache.org/
XMLCLDC	<i>Parsing XML in CLDC-based Profiles</i> http://wireless.java.sun.com/midp/ttips/xmlprofiles/
XMLMIDP	<i>A Brief Introduction to XML Parsing in MIDlets v1.0, Forum Nokia, 2003</i> http://nds1.forum.nokia.com/nnds/ForumDownloadServlet?id=2908
XMLRPC	The central XML-RPC site with the specification and links to implementations http://www.xml-rpc.org
XMLRPCASF	Web site of the XML-RPC project of the Apache Software Foundation http://ws.apache.org/xmlrpc

Build > Test > Sell

Developing and marketing mobile applications with Nokia



Get started

Use free resources available through www.forum.nokia.com: articles covering the latest technical and business issues, tools for developing and deploying applications and services, bi-weekly newsletters, and active discussion boards.

www.forum.nokia.com

2

Download tools

Download free SDKs, emulators, simulators, and other tools that integrate with industry-leading integrated development environments.

www.forum.nokia.com/tools

3

Get specifications and documents

Get comprehensive device and platform specifications. Find detailed technical documentation – tutorials, technical notes, case studies, FAQs, and more.

www.forum.nokia.com/devices
www.forum.nokia.com/documents

4

Get support and testing services

Access our wide range of technical support services, including fee-based online support from Nokia's experts, case resolutions from our fee-based professional support services, and Nokia Developer Hub Services, including online and onsite access to mobile infrastructure elements, such as servers and messaging centers.

www.forum.nokia.com/support

5

Take your applications to market

Take your applications and services to market through Nokia Tradepoint and Nokia Software Market. Other opportunities are available through Nokia Mobile Phones and other Series 60 licensees.

www.forum.nokia.com/business

6

Build your business with Nokia

Nokia works with selected leading developers in the games, branded media and content, and enterprise software industries. Submit your application to Nokia at www.forum.nokia.com/business.

www.forum.nokia.com/business