FORUM NOKIA

Optimizing the Client/Server Communication for Mobile Applications, Part 3

Version 1.0; October 2, 2003

Java™

NOKIA

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.

Contents

1 Introduction 5

2 The Profiler Application 6

3 Installation and Deployment of the Applications 8

4 Client-Side Deployment 10

5 Performance Test and Resource Usage 11

6 Conclusion 15

7 Message Data Transferred 16

7.1 Text Protocol 16

7.1.1 Text request 16

7.1.2 Text response 16

7.2 XML-RPC Protocol 16

7.2.1 XML-RPC request 16

7.2.2 XML-RPC response 17

7.3 SOAP Protocol 17

7.3.1 SOAP request 17

7.3.2 SOAP response 18

8 WSDL File of the SOAP Web Service 19

9 Installed Software of the Test Environment 21

10 Source Code for the Application 22

10.1 ProfilerMIDlet.java 22

10.2 CurrencyConverter.jad 26

Change History

October 2, 2003	Version 1.0	Initial document release

1 Introduction

In Part 2 of this series of Java™ technology client/server documents, we implemented different protocols for the client/server communication and the client and server parts of the application. Now, in Part 3, we will take a detailed look at the deployment aspect of the application, including the setup of the server side using Apache Tomcat and various other open-source projects. This “reality check” document also demonstrates how to use the application in a real environment and focuses on performance issues.

Additionally, Part 3 provides another sample application for automated performance testing using the three different transport protocols that were implemented in Part 2. The entire framework is an ideal platform for deploying and testing a client/server application. The results can then be folded back to optimize any client/server application.

2 The Profiler Application

This document presents a CurrencyConverter application, as well as a second, independent MIDlet that easily enables a comparison of the three different protocols. Although the CurrencyConverterMIDlet states the time required for the procedure call in a dialog box, a direct comparison of all three transport protocol variants necessitates consecutive runs. The times for single remote procedure calls can sometimes deviate substantially from one another, consequently several consecutive calls should be made to the same Web Service before calculating their mean value. The benchmarking program ProfilerMIDlet performs this task, making it easy to compare the performance of various protocols over different network technologies.

The MIDlet is comprised mainly of a TextBox that displays the results. Two *Command* objects that provide buttons for starting the profiling procedure and closing the application are appended to this TextBox.

The MIDlet implements the interface *CommandListener*, and therefore handling of the event functions, exactly the same as in the case of the *CurrencyConverterMIDlet* in the commandAction() method. If the Exit button is pressed, the application is terminated. If the Start button is pressed, the profiling procedure is started inside a separate thread. This is necessary to prevent blocking the system thread by allowing the commandAction() method to return immediately as required by the MIDP specification. The network connection itself is started with a single execution of the Text protocol. The benchmarking does not include this initial connection setup time because this initial step involves the external network infrastructure, which can influence the entire profiling process.

In the next step, the method *profile()* is called successively for each protocol. The respective constants from the *RequestCommandFactory* class specify which protocol is to be applied. The method *profile()* returns the execution time determined for each specific protocol, which includes the construction of the message, the transmission time, and the server-side processing of the call. After each profiling step, an output is displayed by the *TextBox*. This indicates the current status and, at the end, displays the times required for the three protocols: Text, XML-RPC, and SOAP.

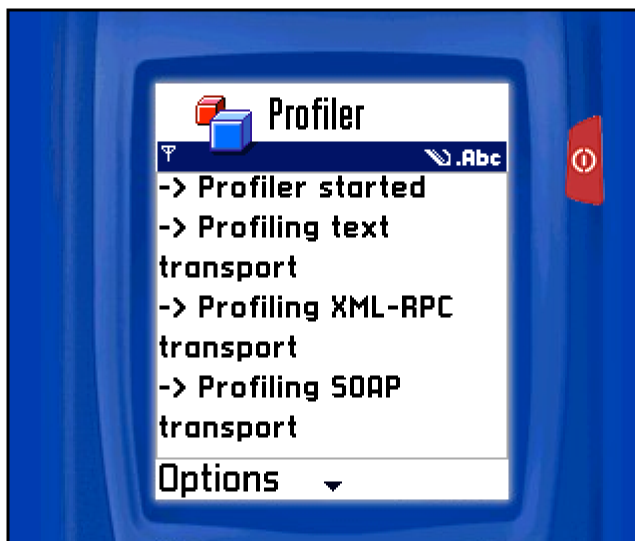

Figure 1: Output of the Profiler application's progress

The method *profile()* is built up in a manner similar to the remote procedure call in the program *CurrencyConverterMIDlet*, which was covered in Part 2. However, the call is run ten times consecutively in a for loop. It is also possible to modify the number of loops in the program code. (The number of executions is only an example used to get the results presented in this document.)

To make sure that the remote procedure call is not always identical, and thus to avoid the influence of any caching on the server or an intermediate HTTP proxy, the actual loop value is used as the parameter for the currency amount itself. The running time for each call is added to the variable *totaltimer*, which is eventually divided by the number of runs to obtain the mean value. This value is returned as the result of the *profile()* method. Finally, an alert notifies the user about the end of the profiling process and the calculated execution times are displayed inside the *TextBox*. If the profiling of one specific protocol fails, this is displayed accordingly.

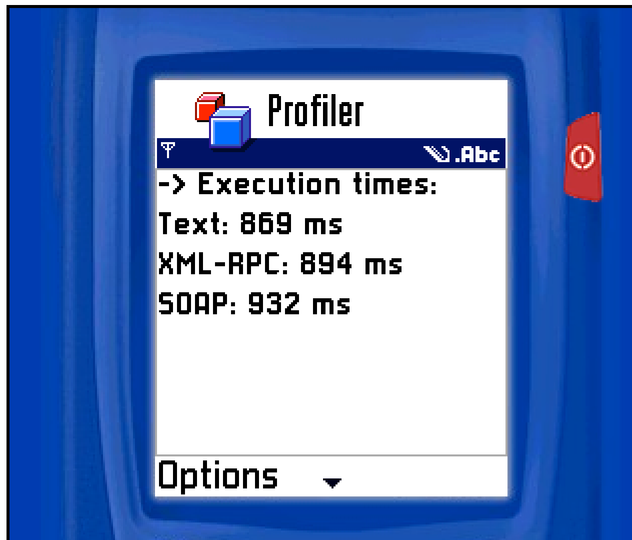

Figure 2: Execution times of the three transport protocols

3 Installation and Deployment of the Applications

Having described in detail how the program works, we will now focus on the details of setting up and running the application. As a prerequisite, a variety of software components must be installed on the server. During the development phase it is practical and helpful to have the appropriate tools available on the development workstation for running the emulator, as well as for monitoring communication between the client and server.

Chapter 9 gives an overview of all the software components used and the versions deployed for the test environment. Also included are links to each Web site for downloading the relevant software. It is best, but not mandatory, to use a development environment such as Borland JBuilder with the MobileSet extension, or Sun ONE Studio Mobile.

It makes sense to configure the server first and install the necessary software. The server should have a public IP address, so that later the mobile device can call Web Services via a GSM or GPRS connection. During test and development stages, developers can work with an internal server or install the server software on the development computer for emulating the client with the respective emulator.

A Linux server with SuSE Linux 8.2, a 700 MHz Celeron processor, and 384 MB RAM was used for our performance tests. Since the applied server software components are provided for a multitude of different systems, developers can use another operating system like Microsoft Windows or Sun Solaris. However, given that the installation steps covered here are based on a Linux system, minor deviations may occur in other operating systems, especially with regard to the needed scripts.

The MySQL database and Tomcat server are the first items to be installed on the server. The latest versions should be downloaded. The MySQL Control Center tool can also be installed on the development computer to ease the MySQL database's administration task. A typical Linux distribution such as SuSE already provides a MySQL installation. In addition to the basic database system, this setup needs the JDBC driver MySQL Connector/J to make the servlet subsequently capable of accessing the database. It is possible to use different JDBC drivers, but there may be differences with regard to the actual configuration and API.

When installing the MySQL database, it is best to follow the instructions provided with the software or deploy a Linux distribution that already comes with the MySQL installation package. Although the same applies to the Tomcat server, it should be noted that the latest JDK for running the Tomcat must be installed on the server first.

After completing a standard installation of MySQL and Tomcat, the next step is to create the *CurrencyDB* database on the server. To assure trouble-free access to the development computer, developers should create a new user who is privileged to access the database via the network. In the standard installation, the user *root* is already set up without a password but is not authorized for network access to the database. A new user is created on the command line with the following instructions, intended for access afterwards through JDBC:

```
mysql --user=root mysql
mysql> GRANT ALL PRIVILEGES ON *.* TO remote@localhost IDENTIFIED BY
'remote' WITH GRANT OPTION;
mysql> GRANT ALL PRIVILEGES ON *.* TO remote@"%" IDENTIFIED BY 'remote'
WITH GRANT OPTION;
```

Here, a new user remote is created with the password remote assigned local and remote access rights. In a production environment, this password should be changed to prevent full access to the database by unauthorized persons. For the execution of this instruction, it must first be verified that the MySQL command line tool is installed correctly on the system.

The newly set-up user can be used to connect to the server with the MySQL Control Center and then create the *CurrencyDB* database. The following SQL script is run to create the tables in the database applicable to the sample application:

```
CREATE TABLE `currencies` (
 `Currency` varchar(3) NOT NULL default '',
  `Value` double NOT NULL default '0'
) TYPE=MyISAM;
```

The table "*currencies"* with the columns *Currency* and *Value* has been created by this SQL statement. To test the example later, it is necessary to enter at least a few pairs of test values of any preferred currency, comprised of three-letter currency codes and the respective exchange rates. This can be achieved easily with the Control Center or an SQL Script in a text file.

Next, the JDBC driver MySQL Connector/J should be installed, to allow database access through JDBC from the servlet as well. For this, a developer simply unpacks the JAR archive, which in this case is named mysql-connector-java-3.0.6-stable-bin.jar, from the downloaded archive, and copies it into the folder $CATALINA_HOME/shared/lib. The environment variable $CATALINA_HOME specifies the installation folder for Tomcat. By providing the jar file inside the lib folder, the JDBC driver classes are made available to all applications run on the Tomcat server.

Now, the server application is deployed on the Tomcat server. The best approach is to use the IDE that the developer is already familiar with or an Ant task to build a WAR archive named *currency.war* containing the JAR archive with the XML-RPC implementation from Apache, in addition to the previously described classes. This file, with the name *xmlrpc-1.1.jar*, is located in the archive that can be downloaded from Apache, and should be copied into the folder *WEB-INF/lib* of the WAR archive to give the XML-RPC Servlet access to its classes. The archive *currency.war* can then be copied into the subfolder *webapps* inside the Tomcat installation folder. The folder axis, found in the subfolder *webapps* on unpacking the Axis binaries, should also be copied there.

Finally, the classes *CurrencyDB.class* and *CurrencyWsSoap.class* need to be copied into the folder *$CATALINA_HOME/webapps/axis/WEB-INF/classes*, to allow the application to be published by Axis as a SOAP Web Service. Once this step has been completed, the Tomcat server can be started. The best approach is to use the script *startup.sh* provided in the folder *$CATALINA_HOME/bin*. Starting the server leads to unpacking of the archive *currency.war* into the subfolder currency of the *webapps* folder.

Having started Tomcat, it is possible to access the Axis server using the local URL http://localhost:8080/axis. The link “Validate Configuration” should be clicked to verify if the installation was successful or to get detailed information if the server lacks packages that need to be provided by the system. To enable the Axis server to publish the *CurrencyWsSoap* class as a Web Service, it must be deployed first. This is done with the file *deploy.wsdd* and the Axis administration client, called on the command line:

```
java org.apache.axis.client.AdminClient deploy.wsdd
```

The *AdminClient* is started and the file containing the deployment descriptor for the Web Service is passed as a parameter. Since a couple of JAR archives containing the Axis packages are necessary to call the *AdminClient* and are supposed to be on the *CLASSPATH*, it is much easier to execute the *deploy.sh* shell script that appropriately sets up the *CLASSPATH* and then runs the *AdminClient*. The proper execution privileges must exist for running this file as a shell script. When the *AdminClient* is executed, the relevant service entry is made in the file *server-config.wsdd* located in the *$CATALINA_HOME/webapps/axis/WEB-INF* folder. To check whether the service has been deployed correctly, the link “View Deployed Web Services” is called on the Axis administration Web site. The service *CurrencyWsSoap* should also appear here.

4 Client-Side Deployment

A suitable MIDlet suite that can run on either an emulator or an actual device must be built, using either the IDE or a tool like the Nokia Developer Suite. The content of the MIDlet suite depends on the transport protocol to be supported. Developers can include either the *CurrencyConverterMIDlet* from Part 2, or the *ProfilerMIDlet* class, or both, in the MIDlet suite.

The classes *RequestCommand* and *RequestCommandFactory* must be included in the MIDlet suite. Depending on which protocol is to be supported and/or tested, the classes *TextRequestCommand*, *XMLRPCRequestCommand*, and *SOAPRequestCommand* should be integrated into the suite. To facilitate performance comparisons of all three protocols with *ProfilerMIDlet*, it is obviously necessary to incorporate all three classes into the suite.

As stated earlier, to support both the XML-RPC and SOAP transport protocols, the packages and classes of KXML, KXML-RPC, and KSOAP should also be integrated into the MIDlet suite.

The complete set of API classes is not necessary here, and some memory can be saved by leaving out the *kdom* and *wap* packages right from the start. Use of the *XMLRPCRequestCommand* class forces inclusion of the KXML-RPC classes, whereas the class *SOAPRequestCommand* needs the KSOAP classes. Memory can be saved by removing the package *org.ksoap.marshal*, which provides optional classes that are not needed to run this example. If the classes of all protocol implementations are not installed on the device, the missing classes must be deleted in the switch statement of the class *RequestCommandFactory*. Otherwise, execution of its method *getRequestCommand()* will fail due to lack of dynamic class loading support by the Java 2 Platform, Micro Edition (J2ME™).

To integrate the KXML, KXML-RPC, and KSOAP packages into the MIDlet suite, it is possible to either use the JAR archive of the binary distribution provided, or take the source code from the source distribution and compile and integrate it directly into the MIDlet suite. The latter approach offers more flexibility in optimizing the number of classes by integrating only those that are really necessary into the MIDlet suite. This approach also allows an obfuscator to be applied to further reduce the size of internal as well as external classes.

Normally, the IDE and/or Nokia Developer's Suite for J2ME™ are applied to compile the classes and build the MIDlet suite. Thus, it becomes easy to create the Java application descriptor *CurrencyConverter.jad*, which is required in addition to the archive *CurrencyConverter.jar*. Here, due consideration must be paid to ensure that all classes, including those from external packages, pass the preverifier prior to being deployed on the emulator or the device.

Both examples were tested on the Nokia 3650 device and on the emulator for the Series 60 MIDP SDK 1.0 for Symbian OS, Nokia edition, to obtain the most realistic comparison of the two run-time environments.

5 Performance Test and Resource Usage

Now let's take another, closer look at the differences between the various transport protocols. Given that the three protocols can be interchanged within the application, it becomes easier to determine their performance disparities, since the general running conditions remain constant. In addition to establishing the execution times for the protocols used, attention should be paid to the size of the applications, which can vary greatly. Unfortunately, for this example, it is not possible to subdivide the run time into latency time and the time required for parsing the transmitted data. To do so, it would be necessary to modify the source of the SOAP and KXML-RPC implementations, which is not our goal. Given the fact that the source code for both packages is available, this leaves enough room for further optimization.

Ultimately, the degree of self-descriptiveness of the three different protocols should also be assessed by comparing their request and response messages. Chapter 7 presents a fairly good overview of the data transmitted by each protocol. This data is based on a sample call where a value of "2" is converted from euros into U.S. dollars. The following comparison of the transport protocols is also based on the data of this sample call:

	Text	XML-RPC	SOAP	
Request Size (HTML body without header)	0 Bytes	329 Bytes	600 Bytes	Request Size (HTML body without header)
Response Size (HTML body without header)	6 Bytes	127 Bytes	474 Bytes	Response Size (HTML body without header)
Total Transfer (HTML body without header)	6 Bytes	456 Bytes	1074 Bytes	Total Transfer (HTML body without header)
Client Application Size (without obfuscation)	4.71 Kbytes	35.0 Kbytes	48.5 Kbytes	Client Application Size (without obfuscation)
Execution Time Emulator (LAN, 100 Mbit/s)	869 ms	894 ms	932 ms	Execution Time Emulator (LAN, 100 Mbit/s)
Execution Time Nokia 3650 (GSM, 9.6 Kbit/s, analog)	3318 ms	3892 ms	4742 ms	Execution Time Nokia 3650 (GSM, 9.6 Kbit/s, analog)
Execution Time Nokia 3650 (GPRS 40.2 Kbit/s, class 6)	2978 ms	3489 ms	4350 ms	Execution Time Nokia 3650 (GPRS 40.2 Kbit/s, class 6)

Figure 3: Comparison of the different transport protocols

It is quite clear that the volume of data transferred depends on the protocol selected. Whereas a proprietary, text-based protocol sends just a short header with URL parameters, XML-RPC transmits 329 bytes, and SOAP as many as 596 bytes in a separate stream in addition to the URL and header data. These values are not directly comparable, since the longer URL length is not accounted for in the text-based protocol – but this amount of data overhead is minimal relative to the XML data.

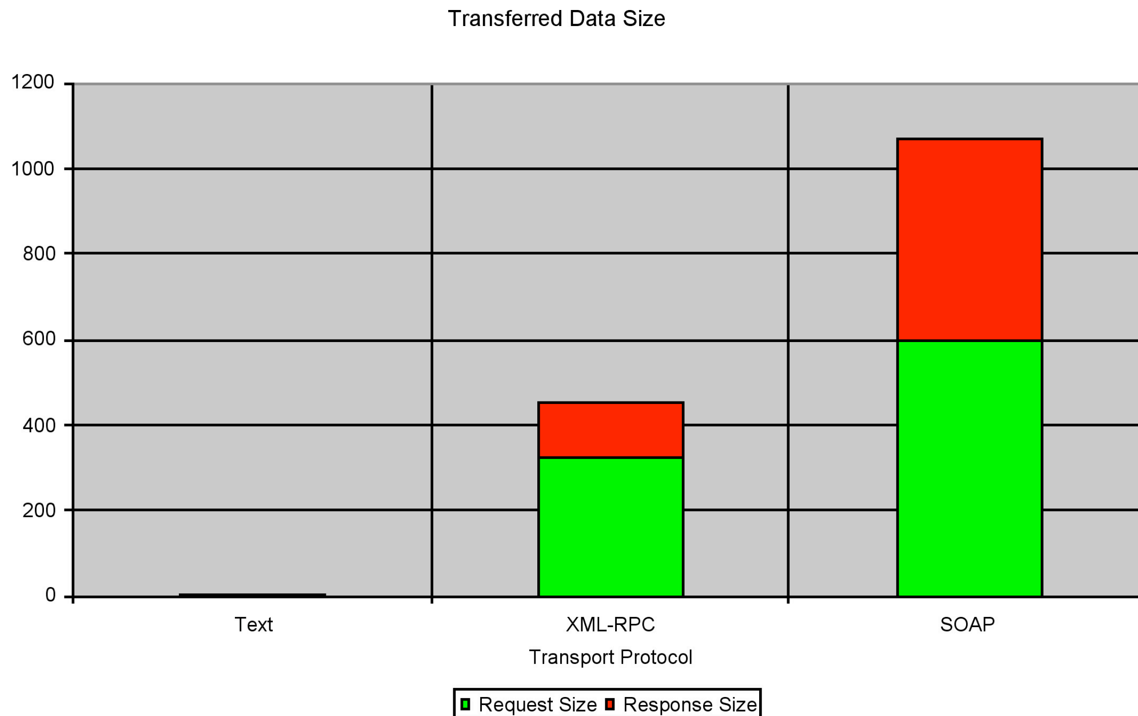

Figure 4: Size of the request and response data of the three protocols

The correlation is similar for responses. The Text protocol returns a string as a 6-byte-long octet stream, whereas the XML-RPC response message is 127 bytes, and the SOAP response encompasses as many as 472 bytes.

The SOAP protocol response message, generated by the Axis server, has a special characteristic. According to the HTTP 1.1 specification, the response can be transmitted as a chunked message. This means that the message length is not known when the transmission is initiated, hence it is sent in blocks, marked before and after the actual data with hexadecimal numbers. The Axis server takes advantage of such a transmission scheme, as is apparent from the protocol extract. Note: Some devices might have problems with chunked HTTP responses.

The TCP monitor tool can be used to gain insight into the HTTP messages of the different protocols. This tool is part of the Axis server, and is ideal for observing not only the SOAP connections, but the Text and XML-RPC connections as well.

Figure 4 demonstrates that the total data volume transmitted by the streams for the Text protocol is just 6 bytes – miniscule compared to 456 bytes for an XML-RPC protocol. On the other hand, SOAP sends almost twice as much data as XML-RPC, which can have a significant impact on performance in slow cellular networks. This difference also plays an important role if the overall running cost of a client/server application needs to be assessed. Since most providers still charge for the amount of data using a GPRS connection, it can be extremely important to optimize the amount of data transferred per call.

Similar differences can be observed with respect to the MIDlet suite size. Whereas the MIDlet suite holding the *CurrencyConverterMIDlet* application and the text implementation demands just 4.71 Kbytes of memory from the mobile phone, the XML-RPC implementation needs at least 35 Kbytes, and the SOAP version as many as 48.5 Kbytes. These sizes can be somewhat reduced through obfuscation, but the limit is still reached very quickly given the fact that some mobile environments (WAP gateway) or devices will only install a maximum MIDlet size of 30 Kbytes over the air.

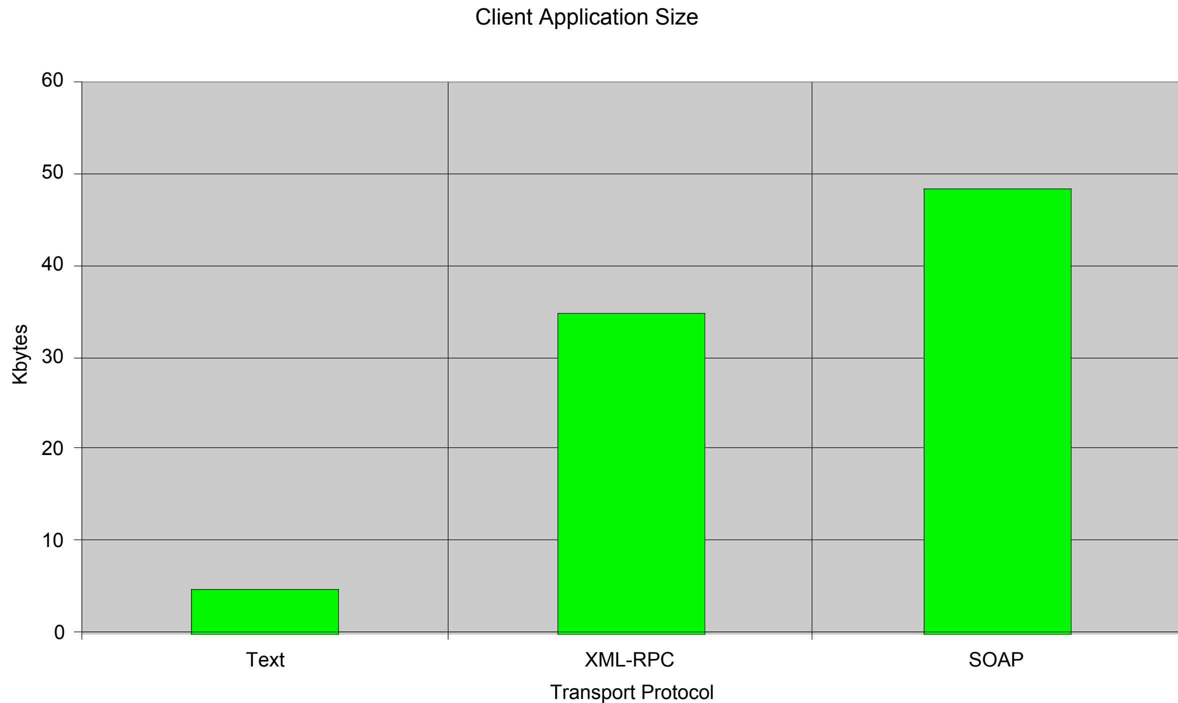

Figure 5: Size of the client application implementing only the specified protocol

The application at hand was tested on both an emulator and an actual device. The Nokia Series 60 emulator and a Nokia 3650 device were used to help achieve the best possible comparison. Even under the same apparent conditions, it is obvious that the times needed for a procedure call on the server deviate greatly from one another, because the transmission is dependent on a couple of factors.

One of these factors is the network infrastructure of the mobile operator. Some operators use additional gateways or proxies, which can have a tremendous influence on the performance and the successful transmission of data. In some cases, these gateways, which are usually used by WAP connections over GPRS, have an unwanted influence on the content of the HTTP header. The number of available GPRS time slots at the time of execution of the remote procedure is another factor.

Because the actual GPRS environment can vary enormously, connection times are rarely comparable; this means that the execution times presented in this document are only samples. Developers must make their own tests with the appropriate handset, operator, and connection type. The use of different firmware on seemingly similar devices can also influence the performance and functionality of an application.

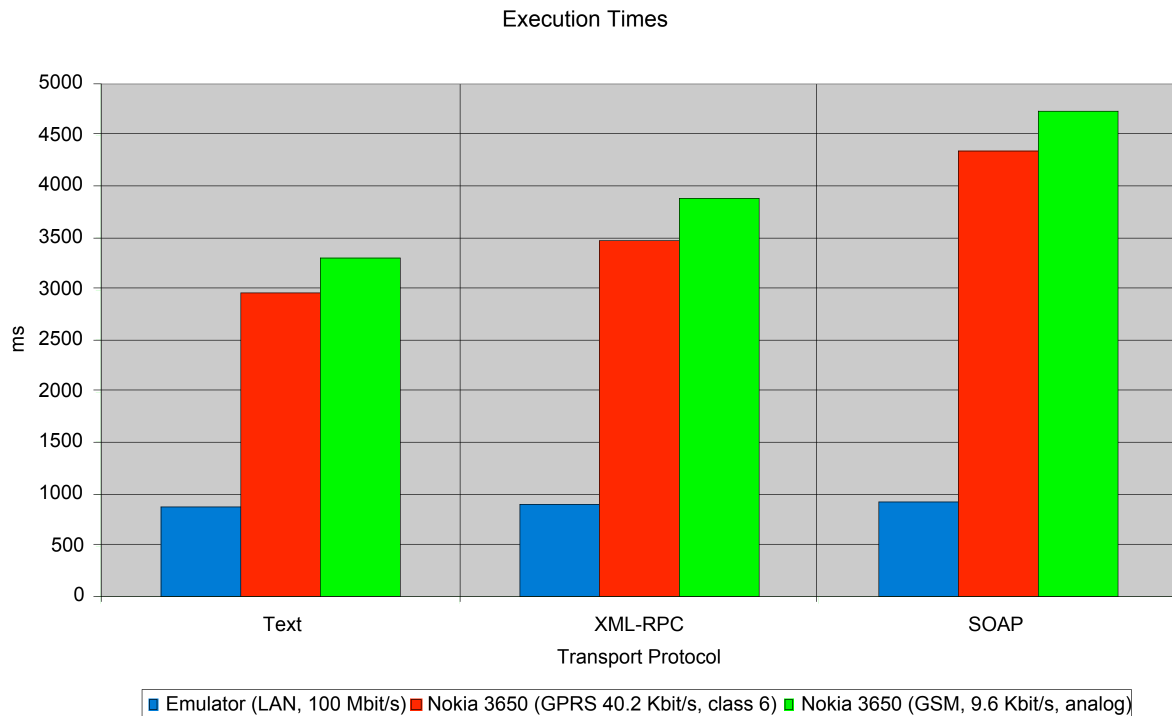

Figure 6: Execution times of a sample run of the profiler application

We used the analog mode for our GSM connection, as stated in the diagram. This means that the connection is established via an analog dial-in server. GSM allows different operation modes for data connections: One is analog and the other is ISDN, which allows connections via an ISDN dial-in server.

The overall trend comes as no surprise. The proprietary Text protocol in most cases is somewhat faster than XML-RPC and SOAP, with SOAP being the slowest due to the large amounts of data involved and the demands it places on the parser. If time is measured on a local system where both the server and the client's emulator run, the resulting differences are less meaningful, because the transmission time is negligible and the computer (with greater CPU power versus that of a mobile phone) is barely affected by the parsing of some XML messages.

Also, the protocol with the shortest messages is not always the fastest on a mobile phone, because the influence of the network's latency is more important than the available bandwidth and, respectively, the amount of transferred data. Because all three protocols transmit relatively compact messages, the result depends on the network's latency, which may vary by operator.

6 Conclusion

Using the example described in Part 2 of this series, it is possible to demonstrate the practical advantages and disadvantages of different transport protocols for client/server communication via HTTP. The supplementary abstraction layer makes it possible to switch between these three protocols within the same application, for a proper comparison. In practice, this abstraction layer enables a client to be connected to different types of Web Services, without having to alter the program logic. This permits convenient linking to third-party services, while allowing optimal client/server communication through adaptation to the respective requirements. Hence, if someone has specific reasons for using SOAP, the protocol can be deployed without precluding subsequent deployment of an optimized variant of the client based on a proprietary protocol.

Prior to developing a client/server application and before commencing final development, practical field tests with the data to be transmitted are recommended. The "reality" of using an emulator versus an actual device is sometimes worlds apart. Hence, it makes even more sense to develop an application based on a supplementary abstraction layer, and to conduct practical tests before selecting a specific protocol – one that not only delivers acceptable performance, but also adequately fulfills the requirements in terms of self-descriptiveness and conformity to standards.

7 Message Data Transferred

The following chapter provides an overview of data transmitted in the request and response messages of the various protocols. Differences in the size of the messages are obvious. With regard to differences of content length in the HTTP header and the actual content length of the message, the code reproduced here is presented aesthetically, and therefore may differ in size compared with the actual values.

7.1 Text Protocol

7.1.1 Text request

```
GET /currency/servlet/CurrencyServletText?val=2&src=EUR&dst=USD HTTP/1.1
User-Agent: Profile/MIDP-1.0 Configuration/CLDC-1.0
Accept: application/octet-stream
Host: 127.0.0.1
Connection: close
```

7.1.2 Text response

```
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: 6
Date: Sat, 12 Apr 2003 20:55:12 GMT
Server: Apache Coyote/1.0
Connection: close
```

7.2 XML-RPC Protocol

7.2.1 XML-RPC request

```
POST /currency/servlet/CurrencyServletXmlRpc HTTP/1.1
Host: 127.0.0.1
Connection: close
Content-Type: text/xml
Content-Length: 329

<methodCall>
   <methodName>CurrencyWsXmlRpc.calcCurrency</methodName>
   <params>
      <param>
         <value>
         <string>2</string>
         </value>
      </param>
      <param>
         <value>
```

```
        <string>EUR</string>
        </value>
      </param>
      <param>
        <value>
        <string>USD</string>
        </value>
      </param>
   </params>
```

</methodCall>

7.2.2 XML-RPC response

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 127
Date: Sat, 12 Apr 2003 20:56:18 GMT
Server: Apache Coyote/1.0
Connection: close

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodResponse>
   <params>
      <param>
        <value>2.12</value>
      </param>
   </params>
</methodResponse>
```

7.3 SOAP Protocol

7.3.1 SOAP request

```
POST /axis/services/CurrencyWsSoap HTTP/1.1
User-Agent: kSOAP/1.0
SOAPAction: ""
Host: 127.0.0.1
Content-Type: text/xml
Content-Length: 600

<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body SOAP-
   ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
```

```
      <calcCurrency xmlns="CurrencyWsSoap" id="o0" SOAP-ENC:root="1">
        <val xmlns="" xsi:type="xsd:double">2</val>
        <src xmlns="" xsi:type="xsd:string">EUR</src>
        <dst xmlns="" xsi:type="xsd:string">USD</dst>
      </calcCurrency>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

7.3.2 SOAP response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Transfer-Encoding: chunked
Date: Sat, 12 Apr 2003 20:56:59 GMT
Server: Apache Coyote/1.0
Connection: close


<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <ns1:calcCurrencyResponse
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="CurrencyWsSoap">
        <calcCurrencyReturn xsi:type="xsd:double">2.12</calcCurrencyReturn>
      </ns1:calcCurrencyResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

8 WSDL File of the SOAP Web Service

The following WSDL file describes the SOAP Web Service provided by the axis server:

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions
targetNamespace="http://127.0.0.1:8080/axis/services/CurrencyWsSoap"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://127.0.0.1:8080/axis/services/CurrencyWsSoap"
xmlns:intf="http://127.0.0.1:8080/axis/services/CurrencyWsSoap"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <wsdl:types />

   <wsdl:message name="calcCurrencyRequest">

      <wsdl:part name="val" type="xsd:double" />

      <wsdl:part name="src" type="xsd:string" />

      <wsdl:part name="dst" type="xsd:string" />

   </wsdl:message>

   <wsdl:message name="calcCurrencyResponse">

      <wsdl:part name="calcCurrencyReturn" type="xsd:double" />

   </wsdl:message>

   <wsdl:portType name="CurrencyWsSoap">

      <wsdl:operation name="calcCurrency" parameterOrder="val src dst">

         <wsdl:input message="impl:calcCurrencyRequest"
         name="calcCurrencyRequest" />

         <wsdl:output message="impl:calcCurrencyResponse"
         name="calcCurrencyResponse" />

      </wsdl:operation>

   </wsdl:portType>

   <wsdl:binding name="CurrencyWsSoapSoapBinding"
   type="impl:CurrencyWsSoap">

      <wsdlsoap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http" />

      <wsdl:operation name="calcCurrency">

      <wsdlsoap:operation soapAction="" />

         <wsdl:input name="calcCurrencyRequest">

            <wsdlsoap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://127.0.0.1:8080/axis/services/CurrencyWsSoap"
            use="encoded" />
```

```
        </wsdl:input>

        <wsdl:output name="calcCurrencyResponse">

           <wsdlsoap:body
           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
           namespace="http://127.0.0.1:8080/axis/services/CurrencyWsSoap"
           use="encoded" />

        </wsdl:output>

      </wsdl:operation>

   </wsdl:binding>

   <wsdl:service name="CurrencyWsSoapService">

      <wsdl:port binding="impl:CurrencyWsSoapSoapBinding"
      name="CurrencyWsSoap">

        <wsdlsoap:address
        location="http://127.0.0.1:8080/axis/services/CurrencyWsSoap" />

      </wsdl:port>

   </wsdl:service>

</wsdl:definitions>
```

9 Installed Software of the Test Environment

The following table offers an overview of the different software components that have been installed for the test environment. The list also provides the version of each component installed. For easier configuration of a similar execution environment, the appropriate links for download pages are provided.

Software	Version	Download
MySQL	3.23.55	http://www.mysql.com/
MySQL Connector/J	3.0.6	http://www.mysql.com/
MySQL Control Center	0.8.10 beta	http://www.mysql.com/
Sun JDK	1.4.1_02	http://java.sun.com/j2se/
Apache Tomcat	4.1.18	http://jakarta.apache.org/tomcat/
Apache Axis	1.0	http://ws.apache.org/axis/
Apache XML-RPC	1.1	http://ws.apache.org/xmlrpc/
KSOAP	1.2	http://ksoap.enhydra.org/
KXML-RPC	0.6	http://kxmlrpc.enhydra.org/
KXML	1.2.1	http://kxml.enhydra.org/
Nokia Devel-oper's Suite for J2ME™	1.1	http://www.forum.nokia.com/main/1%2c6566%2c030%2c00.html
Series 60 MIDP SDK for Symbian OS, Nokia edition	1.0	http://www.forum.nokia.com/main/1%2c6566%2c030%2c00.html

10 Source Code for the Application

The source code of the Profiler application's classes, which are running in the MIDP environment on the mobile client, is shown below.

10.1 ProfilerMIDlet.java

```
/*
 * ProfilerMIDlet.java
 *
 * Application for testing various protocols
 */

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

public class ProfilerMIDlet extends MIDlet implements CommandListener
{

   private Command exitCommand;
   private Command startCommand;
   private Display dsp;
   private String str="";
   private TextBox t;
   private boolean nwRun=false;

   public ProfilerMIDlet ()
   {
      dsp = Display.getDisplay (this);

      // Define Buttons
      exitCommand = new Command ("Exit", Command.SCREEN, 2);
      startCommand = new Command ("Start", Command.SCREEN, 2);

      // Instantiate TextBox
      t = new TextBox ("Profiler", "", 256, 0);

      //Define user interface
      t.addCommand (exitCommand);
      t.addCommand (startCommand);
      t.setCommandListener (this);
```

```
}

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

public void pauseApp ()
{
}

public void destroyApp (boolean unconditional)
{
}

public void commandAction (Command c, Displayable s)
{
   if (c == exitCommand)
   {
      // Exit application
      destroyApp (false);
      notifyDestroyed ();
   }
   else if ((c == startCommand) && (nwRun == false))
   {
      Thread nwThread = new Thread()
      {
         public void run()
         {
            nwRun = true;
            // Trigger network connection with a first call not
            profiled
            RequestCommand rc = RequestCommandFactory.getRequestCommand
            (RequestCommandFactory.REQUEST_PROTOCOL_TEXT);

            // Define Operation
            rc.setOperation ("convert");
```

```
rc.setParameter ("val", "0");
rc.setParameter ("src", "EUR");
rc.setParameter ("dst", "USD");

// Execute remote procedure
rc.execute ();

str = "-> Profiler started\n";

// Profile the Text protocol
str+="-> Profiling text transport\n";
t.setString (str);
long timerText = profile
(RequestCommandFactory.REQUEST_PROTOCOL_TEXT);

// Profile the XML-RPC protocol
str+="-> Profiling XML-RPC transport\n";
t.setString (str);
long timerXmlRpc = profile
(RequestCommandFactory.REQUEST_PROTOCOL_XMLRPC);

// Profile the SOAP protocol
str+="-> Profiling SOAP transport\n";
t.setString (str);
long timerSoap = profile
(RequestCommandFactory.REQUEST_PROTOCOL_SOAP);

// Notify user about end of profiling
dsp.setCurrent (new Alert ("Info", "Profiling finished!",
null, AlertType.INFO));

// Display result of profiling
str="-> Execution times:\nText: ";
if (timerText==-1) str+="failed\n"; else str+=(timerText +
" ms\n");

str+="XML-RPC: ";
if (timerText==-1) str+="failed\n"; else str+=(timerXmlRpc
+ " ms\n");
```

```
                str+="SOAP: ";
                if (timerText==-1) str+="failed"; else str+=(timerSoap + "
                ms");

                t.setString (str);
                nwRun = false;
            }
        };
        nwThread.start();
    }
}

public long profile (int type)
{
    long totaltimer = 0;

    // Loop for repeated remote procedure calls
    for (int i=1; i<=10; i++)
    {

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

        // Define Operation
        rc.setOperation ("convert");
        rc.setParameter ("val", Integer.toString (i));
        rc.setParameter ("src", "EUR");
        rc.setParameter ("dst", "USD");

        // Get system time
        long timer = System.currentTimeMillis ();

        // Execute remote procedure
        if (rc.execute ()==0)
        {
            // Calculate time for function call and new total time
            timer = System.currentTimeMillis () - timer;
```

```
                totaltimer += timer;
            }
            else
                return -1;
        }

        // Return average execution time
        return totaltimer/10;
    }
}
```

10.2 CurrencyConverter.jad

This code shows the Java application descriptor for the MIDlet suite *CurrencyConverter*, which in this case only contains the *ProfilerMIDlet*.

```
MIDlet-Version: 1.0
MIDlet-Vendor: Administrator
MIDlet-Jar-URL: CurrencyConverter.jar
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
MIDlet-1: Profiler, , ProfilerMIDlet
MIDlet-Name: CurrencyConverter
MIDlet-Jar-Size: 63102
```

Build > Test > Sell

Developing and marketing mobile applications and services with Nokia

1

Get started

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

http://www.forum.nokia.com

2

Download tools and SDKs

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

http://www.forum.nokia.com/tools

3

Get specifications and documentation

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

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

4

Get support and testing services

Access Nokia's wide range of technical support services, including case resolutions from our professional support staff and fee-based online support from our experts. Forum Nokia's Developer Hub services help with development and testing by providing both online and onsite access to servers, messaging centers, and the like.

http://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.

http://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 http://www.forum.nokia.com/business.

http://www.forum.nokia.com/business