

Optimizing the Client/Server Communication for Mobile Applications Part 1

Version 1.0; February 28, 2003

Java™

NOKIA

Contents

1	Introduction	4
1.1	Mobile Client/Server Applications	4
1.2	Different Types of Protocols	5
1.2.1	Proprietary binary format	6
1.2.2	Object serialization	7
1.2.3	Message exchange by XML	7
1.2.4	Binary compression of XML	8
1.2.5	XML-RPC	8
1.2.6	SOAP	9
1.3	Optimization by Proxy	9
1.4	Additional Abstraction Layer	10
1.5	Conclusion	12
2	References	13

Change History

28 February 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 1

Version 1.0; February 28, 2003

1 Introduction

1.1 Mobile Client/Server Applications

In addition to stand-alone applications running on mobile handsets without any need of interaction with external resources, there is a need for a distributed environment where the client needs to communicate with the server using an IP connection. This white paper begins by pointing out some of the typical client/server communication problems that can arise during the active connection between a Java™ 2 Platform, Enterprise Edition (J2EE™), platform server and a MIDlet. The next step compares the various protocols, which can be considered for developing these types of distributed applications **[MIDSERV]**.

We then demonstrate how a programmer can use a supplementary abstraction layer between the transport protocol, based on HTTP, and the application itself to build a flexible architecture that can be optimized. With this approach in place, the selected transport protocol is relatively easily exchangeable, without the need for any modifications in the application logic.

Addressed herein is the introduction of a proxy server that can enhance the performance of mobile client/server applications. This document concludes by showing how, with the aid of known design patterns, one can relatively easily attain an additional abstraction layer in a Java application. Part 2 of this article then demonstrates this with the help of an example.

In practice, a multitude of Mobile Information Device Profile (MIDP) applications are not just implemented on mobile devices, but have access to a server as well, and thus represent a distributed application. Many mobile applications really make sense only with a connection to a server environment. The connection may be “always on” or just active when the application needs to communicate with the server. Using the distributed approach, access is enabled to external databases that are comprehensive and current, while tasks too complicated for the MIDP device’s limited capabilities can be transferred to a more powerful server environment. A mobile enterprise solution can, hence, occur only through interaction between the J2EE and the Java 2 Platform, Micro Edition (J2ME™), application layers. During data exchange between the server and the mobile client, attention should nevertheless be paid to certain aspects, especially involving the transmission performance and processing of data on the device **[PERFAN]**.

For a mobile enterprise solution based on J2ME technology, it is important to take into account that both the network connection and the device’s resources are limited and do not reflect the typical higher standards of a desktop computer that uses a fixed networking environment. This means that one should anticipate long latency periods along with a limited bandwidth. In addition, one should not expect the mobile device to have an always-on network connection under all circumstances. With respect to the device’s resources, one faces the problem of a limited computation capability under a relatively low-storage capacity level of the device. Consequently, before developing a distributed application for a mobile client, one should consider the factors described herein when choosing the protocol, since this decision can have a major impact on the application’s performance.

HTTP is an ideal client/server communication protocol for mobile Java applications [HTTPMIDP]. Per specification, every MIDP 1.0 compatible device must support HTTP. Other protocols such as TCP or UDP (User Datagram Protocol is a connectionless transport) are basically optional for a MIDP 1.0 device. Since not all MIDP devices support a socket or datagram-based communication, HTTP deployment on the mobile device permits optimal portability between devices from different manufacturers. Although some devices like the Nokia 6800 handset support socket connections, for maximum compatibility HTTP should be used as the transport protocol between client and server.

Another advantageous feature for development is that the HTTP protocol enjoys trouble-free access through firewalls. Because the server and mobile client are mostly separated through a firewall, HTTP eliminates the need for any extra configuration effort. Still, one should be aware of the security risks that come along with any open HTTP connection to the outside world. Another argument for HTTP is that the Java environment provides an extensive networking API, which supports the HTTP 1.1 protocol. It is an easy task to generate GET, POST, and HEAD requests in a Java application. The utilization of HTTP is not a problem on the J2EE application end either, since HTTP has become the typical network protocol for the J2EE platform environment. Moreover, it is widespread, and has been kept robust and easy – thereby offering a range of implementations on both the server and client side.

In conclusion, it can be said that the entire client/server communication must be handled over an HTTP gateway, which is usually a Web server or a J2EE server with a servlet engine. To be able to work with the same technology on the two sides, it is a straightforward approach to implement server communication with the help of servlets or JavaServer Pages™ (JSPs). As such, one can develop in the Java environment in both the client and the server.

1.2 Different Types of Protocols

Now that HTTP has crystallized as the preferred transport protocol, it is the developer's role to decide on the message format for data exchange between the server and the client, which is not predefined [WCLIENT]. Given that the J2ME platform does not offer standardized (but very resource-hungry) mechanisms like the Java Remote Method Invocation (RMI) and the Java API for the XML-based Remote Procedure Calls (JAX-RPC), the developer alone must define the format and the communication layer of the transfer protocol on top of the HTTP transport layer. There are numerous options, as examined in more detail in the following section.

Principally, there are two extremes for defining the messaging format: One is a pure and optimized binary format that guarantees the highest efficiency, while the other is a complex XML-based format such as SOAP, which offers maximum portability and readability, but performs extremely poorly, particularly with mobile devices due to the limited bandwidth and processing power that they are able to provide. A multitude of intermediate solutions do exist between these two extremes. The challenge facing the developer is to find the best solution for the special application at hand. Fundamentally, one can say that the size of the protocol grows directly proportional to the degree of self description, thereby lowering the transmission efficiency over narrowband mobile phone networks. With increasing human-readability, one sees simultaneous growth in XML-based formats, as well as in the computation performance of mobile devices necessary for generating a message and parsing incoming messages.

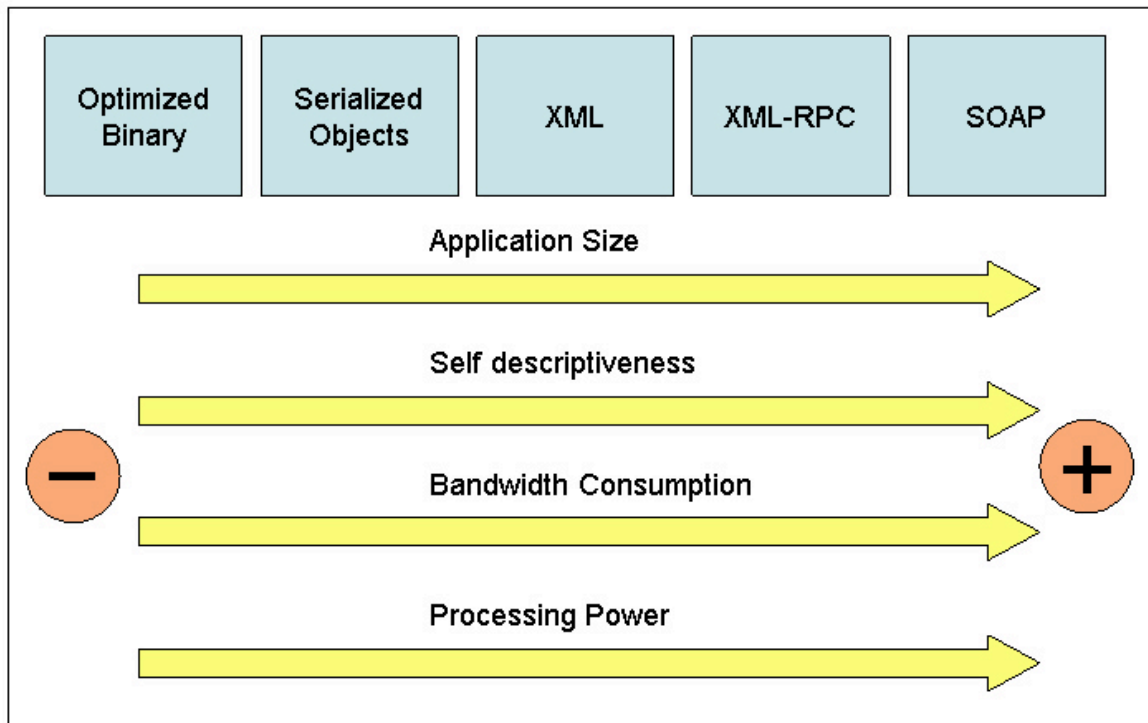


Figure 1: Comparison chart of different communication protocols

1.2.1 Proprietary Binary Format

The document will now deal with efficient proprietary protocols and then move forward step by step into more complex messaging formats. A proprietary request/response format, as stated earlier, is highly flexible and can be generated very easily. It has essentially to do with sending HTTP GET or HTTP POST requests to a J2EE platform server, such as a Tomcat from Apache Software Foundation [TOMCAT].

The main difference between GET and POST requests is that with HTTP GET, all parameters or data that have to be transmitted to a server are included in the URL itself. This means that one can direct a call for remote server procedures through the URL and its parameters. For transferring the parameters, however, one is limited to the simple text format with parameter lengths restricted by the actual size of the maximum request line length of the Web server. On the Apache Web server, for example, the default maximum size of the request-line is set to 8190 bytes and may be modified on special occasions by the LimitRequestLine directive in the server's configuration file. As for the needed bandwidth, sending a message in HTTP GET to the server is the most efficient way.

The HTTP POST technique is better suited for transporting larger amounts of data or even binary data from a mobile device to a server, since the proprietary data is sent to the server independently of the URL. This has the advantage that the data volume is not constrained, as in the case of the GET procedure. The implementation in the Java environment is achieved by opening a separate stream, on which the standard methods for streams in Java are employed, and thus one is able to transmit binary data to the server without any trouble; this is demonstrated by the sample code in Part 2 of this white paper. Hence, the data exchange between the client and server can take place entirely using streams. The result of both types of HTTP requests is always a separate stream that can naturally contain user-defined binary data. It is not relevant at this point if the request is sent through GET or PUT. The data exchange itself can be achieved with the help of the DataInputStream and DataOutputStream classes. This has the advantage,

among others, that it permits the usage of handy methods like `readInt()` and `writeInt()` for transferring each data type over the HTTP connection.

The major advantage of this form of communication using HTTP vs. the procedure presented in Part 2 is its high transmission efficiency and the compact payload size. Its disadvantage, on the other hand, is that it is not self-descriptive, and also prior knowledge of the format in both the client and the server is requisite, before the application development can even be initiated. This naturally results in the problem that any changes made to the message format must be consistent between the client and the server. Besides, with the increasing number of dissimilar messages that the server needs to handle, the program code becomes increasingly complex.

1.2.2 Object serialization

An interesting approach to overcome these difficulties is the use of serialized objects. In addition to standard data types supported by various read and write methods within the stream classes, this technique makes it feasible to serialize any objects for easy transmission between the client and server. A problem is that MIDP 1.0, unlike the more complete Java 2 Platform, Standard Edition (J2SE™), environment, does not support object serialization and reflection by default. Due to the lack of a general mechanism, each class entails an individual implementation of such a feature.

Serializing is best attained through a proper interface that requires any object that implements it to provide the respective methods necessary for serializing and deserializing it. It is important in this situation to realize that the classes employed — each with its proprietary serialization mechanism — must be present in identical versions on both the client and the server side. The easiest solution for ensuring consistency is usually the deployment of such a library in an equivalent Java Application Descriptor (JAR) archive. The specific environment we are dealing with here adds some complexity to such a straightforward approach since the J2ME platform version, as opposed to the J2EE platform version, requires preverified classes.

1.2.3 Message exchange by XML

XML represents the next step in the self-description of messaging formats. XML does not automatically require the deployment of an RPC protocol like XML-based SOAP. The developer can just as well build up a proprietary messaging format on the basis of XML to realize any data exchange between the client and server. The decisive advantage of XML is that it is standardized and thus highly portable. Moreover, it is text-based and, as such, structured data can be depicted in a self-explanatory manner. In the enterprise business field, XML has attained a preferred status for messaging protocols mainly due to the good support provided by the J2EE platform.

Since standard APIs for HTTP connections are used when applying XML, a programmer again has the option of handling requests through either HTML GET or HTML PUT methods. Deploying HTML GET, nevertheless, implies that all the parameters to be transferred from the client to the server can be sent in the URL. In this case, instead of utilizing XML, it makes more sense to transfer the parameters in a proprietary format and to transmit only the return values from the server in XML. Working with the HTML PUT approach, on the contrary, allows messages to be in an XML format in both directions, that is, from the client to the server and back. This is viable due to the presence of a separate stream during message transmission, enabling the transfer of XML formatted data.

In addition to resulting in the transfer of larger data volumes in XML as a result of its text format plus the XML-based overhead, one encounters another problem with mobile devices. Whereas the J2EE platform normally supports XML (with usually at least one XML parser available to any `servlet` application by default), the MIDP 1.0 environment does not provide integrated support for XML parsing. Any XML-based solution therefore needs to include the XML parser on the MIDP client `[XMLCLDC]` `[XMLMIDP]`. Although there

are many open source parsers available, such as NanoXML, TinyXML, or KXML [KXML], that can be set up for this purpose and are typically designed for minimal resource usage, these nonetheless demand storage space in the devices, which often have little to spare.

The KXML from Enhydra [ENHYDRAME], for instance, comes into play here since it occupies just 21 KB of memory in its minimal version as a pure pull parser. If a programmer wishes to have the full version available that also supports DOM and WBXML, which will be addressed later, one requires an extra 37 KB of memory in addition to the program code. It is important to take this into account when developing applications for MIDP 1.0 devices, since some of them are limited to a total application size of 64 KB. One additional factor could be the maximum download size of MIDlets through OTA that sometimes still do not allow more than 30 KB for the whole transfer. A larger application must be installed via a PC connection (direct, IrDA, Bluetooth) to solve this potential problem.

Extra memory and an adequate computation capability are also essential for parsing XML documents, otherwise, the mobile unit's limited processor power can lead to significant delays in transmission. As mentioned, in most cases XML messages are one order of magnitude larger than proprietary binary messages – predominantly due to the verbosity of the XML format. In spite of that, there's a solution even for this problem that frequently makes it impractical to deploy XML for transmitting data between the J2ME and J2EE platform servers.

1.2.4 Binary compression of XML

The WBXML format helps reduce the XML document size considerably, in that the XML document's text format is converted into a binary form [COMPXML]. Using this format, also employed for transmitting WML pages, the size is cut down greatly through the replacement of common tags, attributes, and values with a configurable set of tokens. Just as the encoding and decoding of messages in a WAP-technology device is done through a WAP gateway, communication between a MIDP device and a J2EE platform server also may occur directly. The parser accordingly undertakes encoding and decoding of the messages, which must be supported here by WBXML. For example, the KXML parser from Enhydra supports this protocol and allows efficient data transfer between the client and server. The server naturally needs to understand the WBXML format. Alternatively, the communication must be handled through a proxy or a WBXML gateway, which are described later in detail.

1.2.5 XML-RPC

Having shown how to set up an XML-based proprietary messaging format, the following discussion proceeds to the next stage by looking at the two XML-based RPC protocols, XML-RPC and SOAP. These add another standardized layer to the XML format, thereby offering the developer a flexible messaging architecture, also generally designated as Web services [WEBMIDP]. XML-RPC [XMLRPC] is an extremely lightweight messaging protocol enabling the execution of remote procedures over the network through HTTP. The client thus places an XML message through HTTP POST that the server parses. This results in a local procedure that returns the response, also in the form of an XML message, back to the client.

XML-RPC thus takes advantage of XML's features. It builds minimal functionality through this format, enabling data types to be specified and transmitted as parameters for invoking remote procedures in a platform-neutral approach. The format's scope is consciously kept as small as possible by limiting the protocol to six primitive data types enhanced through two complex ones. This compactness makes the protocol particularly suited for J2ME applications and narrowband mobile phone network links.

The disadvantage here again, just as with XML, is that the J2ME platform does not offer any integrated support for XML-RPC, and it is necessary to rely on a supplementary package like Enhydra's KXML-RPC [KXMLRPC] for deployment on the device with the application. The KXML-RPC package is built on top of the previously stated KXML project and it manages quite well with just 24 KB of the device's resources

including KXML. Relevant open source implementations of the XML-RPC protocol are also available for setting up the corresponding communication point at the server. One example of that is the Java platform-based Apache XML-RPC from Apache Software Foundation [XMLRPCASF], which can be applied together with the Tomcat Servlet engine.

1.2.6 SOAP

The next step forward finally brings us to the other extreme of the proprietary binary format, the Simple Object Access Protocol (SOAP). Microsoft originally developed this protocol together with Userland (which is also the major force behind the XML-RPC protocol) to meet the needs of developers interested in coming up with distributed applications associated with Microsoft technologies. SOAP and earlier XML-RPC versions have the same roots. However, unlike SOAP, XML-RPC has been improved over time and has thus not become increasingly complex. The greater complexity definitely makes the open protocol very flexible, which is why it has established itself as the de facto standard for remote procedure calls in XML over HTTP.

Unfortunately, the inherent disadvantage for the mobile segment in particular, is that it comes with plenty of overhead that is not always needed, but takes up much of the resources. The protocol's current specification is 1.2 and, in addition to the feature set of XML-RPC, it offers extra functionality such as namespace awareness, data-typing mechanism, and the transferring of supplementary header data. These features, along with the marketing pressure behind major SOAP companies, have been responsible for the protocol's popularity over others like XML-RPC. SOAP also has the same drawback, in that the J2ME platform provides no native support for it. Any client that needs SOAP support needs to integrate the application's functionality. On the client end, one solution available is the result of the Enhydra ME project – kSOAP [KSOAP]. The kSOAP package, however, requires at least 41 KB of memory, which is an additional burden for the application on the device.

In this case, one must definitively assess if the extra functionality is justified by the overhead memory and bandwidth requirements. Support for SOAP is naturally needed at the server, too, when dealing with client/server applications. This can be tackled by various solutions like the older Open Source Project Apache SOAP or the newer Axis [AXIS] project, which should be used for new projects. Axis also stems from Apache [WSAPACHE].

1.3 Optimization by Proxy

Up to this point part 1, the series has demonstrated the existence of various options for client/server data exchange through HTTP. Depending on the application's purpose, one should decide between the use of a compact custom protocol, vs. a flexible and verbose one that sticks to standards widely used in the open enterprise field. If one wishes to combine the advantages of these contradictory approaches, there is an interesting solution – the use of a proxy server or gateway between the MIDP client and the J2EE server. As mentioned previously, XML messages can be compressed through a WBXML gateway.

Similarly, one can also implement a proxy for a proprietary protocol that transforms it into a standard protocol like XML-RPC or SOAP. Accordingly, the data is transferred from a MIDP client to the proxy through a lightweight protocol such as a customized binary one or WBXML, for minimized message size and better utilization of the limited bandwidth. The proxy, in turn, acts as a client for the J2EE platform server and translates the lightweight protocol's requests into the server's XML-based protocol and vice versa.

One has the possibility of linking an application to an existing server with the assistance of an optimized custom protocol, that could, for instance, handle SOAP or XML-RPC requests. This has the advantage that both interoperability and performance are concurrently raised. On the other hand, it is necessary to consider the fact that the entire system's increased complexity could very well encumber the application's development effort itself.

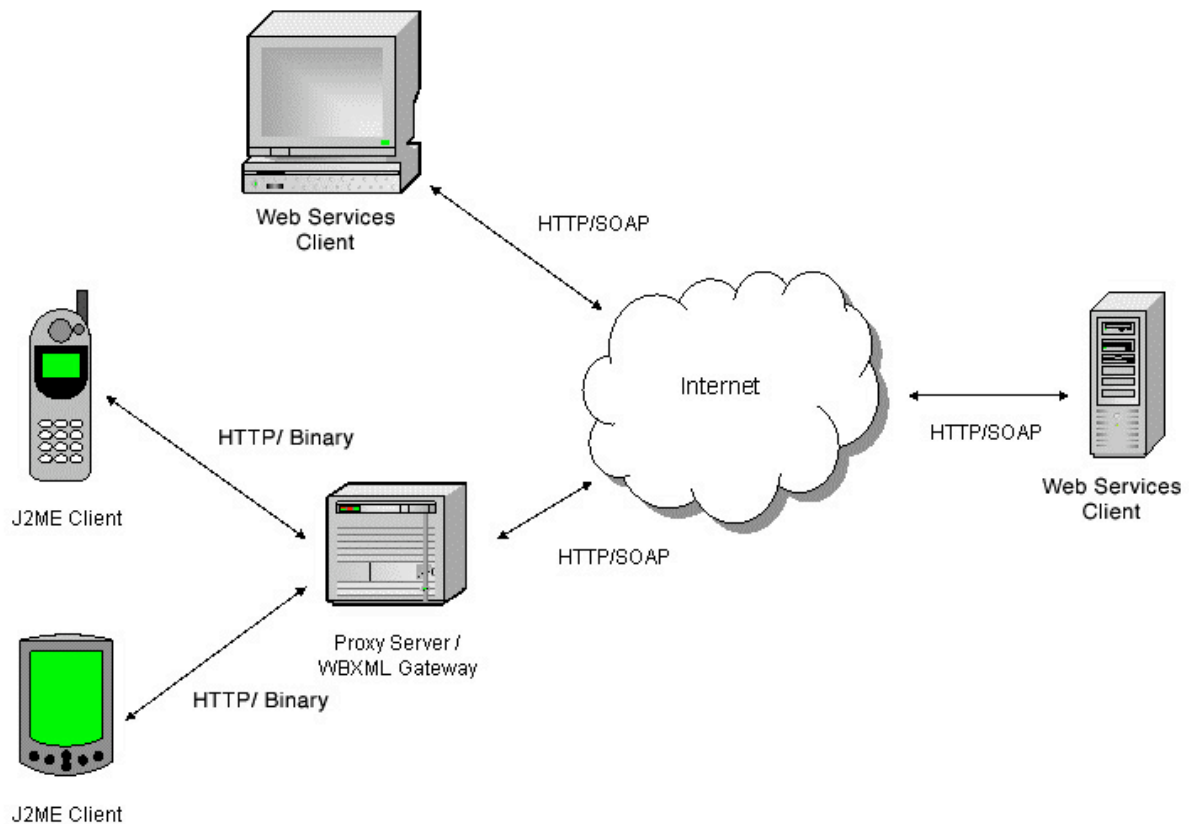


Figure 2: Sample configuration using a proxy

1.4 Additional Abstraction Layer

The discussion so far has highlighted the pros and cons of different protocols as well as the options available for optimizing a distributed application with the help of a proxy. The following discussion deals with the possibility of introducing a supplementary abstraction layer between the application and the transport protocol, to be utilized by the client and server for communication purposes. This makes it feasible to decouple the application logic from the transport protocol and to even make it exchangeable.

The major benefit of this approach is that, depending on the area of application, one can work with various transport protocols and can simply alter or optimize them without having to modify the program logic itself. This layering makes real sense, for example, in cases where the application needs to be optimized subsequently with the help of a proxy, thereby avoiding a complete rewrite. In such a situation, a standardized protocol like SOAP can be simply switched with an optimized proprietary one that communicates with the proxy. This additional abstraction layer offers a good basis for conducting performance tests of disparate protocols. The various protocols can then be tried out in each of their unique environments, to identify the best performer within the special application's environment.

The following section focuses briefly on how to attain such a supplementary abstraction layer in the Java environment, while expending the least possible effort and without unnecessarily using too many of the resources. The first step is to apply a general concept for the client/server communication. This step is also known as the Command design pattern. This covers primarily the definition of a class for executing an operation that can be set up with a number of parameters.

Appropriate setting methods are available for this purpose. Once the operation and the parameter have been defined, another class method is employed to initiate the command's execution. After that, the command is transferred to the server by its communication protocol, where the respective procedure is executed. The result is delivered back to the client and it can then be requested by the appropriate method. The development of such a class can be described very simply in terms of the following interface:

```
public interface RequestCommand
{
    public void setOperation(String operation);
    public void setParameter(String key, String value);
    public int execute();
    public byte[] getResult();
}
```

The Interface RequestCommand requires the implementation of the setOperation() and setParameter() methods to define the operation and the parameter. The setParameter() method can be invoked repeatedly to define several parameters, each being generated from a key/value pair. The execute() method executes the command on the server and the result is finally requested with the getResult() method and, in this simplified example, returned as a byte array.

The idea now is to handle every piece of communication with the server through classes that implement the Interface RequestCommand. For enabling the use of disparate protocols mentioned before, different classes can be defined for each protocol, while implementing the same interface for all of them. Hence, one could implement a class called TextRequestCommand and another named SOAPRequestCommand, for example, which would allow communication with the server over either a custom text-based protocol or over SOAP.

A Factory, representing a known design pattern, can be defined for guaranteed simplicity in handling the classes among the diverse types of protocols. This is a class with a static method designed to generate dissimilar objects that implement the predefined interface. This Factory class may look roughly as follows:

```
public class RequestCommandFactory
{
    public static RequestCommand getRequestCommand(int requestProtocol)
    {
        switch (requestProtocol)
        {
            case REQUEST_PROTOCOL_TEXT:
                return new TextRequestCommand();
            case REQUEST_PROTOCOL_XML:
                return new XMLRequestCommand();
            case REQUEST_PROTOCOL_SOAP:
                return new SOAPRequestCommand();
            default:
                return new TextRequestCommand();
        }
    }
}
```

```
    }  
  }  
}
```

The class RequestCommandFactory itself is never instantiated. The static getRequestCommand() method is merely invoked and the desired communications protocol is selected by a constant that is passed as a parameter. Subject to the passed constant, a new object of the related class is then instantiated for implementing the desired protocol. This approach allows easy setup of an additional abstraction layer on the client side.

It doesn't always make sense to deploy all the protocols on the device, since they will require extra memory. The wisdom of evaluating the performance of the various protocols during testing goes without saying. On the server side, one can of course set up a counterpart to the abstract level, but this isn't always logical. Here, one often finds another abstraction layer in the form of a JDBC driver or an EJB, such that on the server the different implementations of various protocols are already able to access a standardized interface.

1.5 Conclusion

A multitude of alternatives exist for HTTP-based data exchange between a MIDP device and a server. There are decisive distinctions with respect to the performance and resource usage on the mobile device. It has also been shown how a proxy server can help improve performance and enable a combination of the advantages of optimized proprietary protocols with ones that are more readable and standardized.

Having already dealt with the approach utilizing a supplementary abstraction layer between the application and the transport layer in this first part, the second part of this article will show how to realize this approach in practice. We will present an example application for accessing a database from a mobile client. In the next stage, this sample program will be utilized to demonstrate, on the basis of field measurements, how significant the disparities truly are with regard to performance and resource usage between a customized protocol and a standardized one like SOAP.

2 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** *A Performance Analysis of Web Services on Wireless PDAs*
<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

1

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