
# Strategies For J2ME MIDP/J2EE Integration Over HTTP

*prepared by Michael Taylor*

**3 July 2002**
**Version 1.1**

Developnet Consulting Limited
The Old School House
Grantley
Ripon
North Yorkshire
HG4 3PJ

+44 (0)7711 571 015

j2me@developnet.co.uk

http://wwww.developnet.co.uk/

# Copyright



# Warranty



# Trademarks



# Liability

# Change History

| Issue Date | Issue Number | Description of Changes |
|---|---|---|
| 01/07/2002 | 1.0 | Initial draft. |
| 03/07/2002 | 1.1 | Various typos fixed.<br>Minor re-work to several sections. |

 

# Table of Contents



 

# 1. Preface

This document examines the various methods available for passing data over HTTP for J2ME MIDP/J2EE applications.

The document begins with a brief refresher of J2ME MIDP and J2EE, and the benefits of their integration. Subsequently, the document discusses how the HTTP protocol is the pre-eminent transport for supporting integration of J2ME MIDP and J2EE applications.

With an understanding of the transport used for J2ME MIDP/J2EE integration, the bulk of the document moves on to compare and contrast the various options for messaging formats over HTTP available in MIDP.

 

# 2. Refreshers

## 2.1. J2ME MIDP

Java 2 Micro Edition (J2ME) is a platform for developing applications for mobile devices.

Within J2ME, configurations define the run-time environment for a set of devices by specifying the Java features that are available, as well as which virtual machine will be used. In J2ME, the Connected Limited Device Configuration (CLDC) defines a configuration targeted at devices that have limited processing power, display and memory. The majority of these devices will also be mobile.

On top of configurations are profiles. Profiles provide APIs for user interface design, network support, and persistent storage. The Mobile Information Device Profile (MIDP) is a set of Java APIs which, together with the CLDC, provides a complete J2ME application run-time environment targeted at mobile devices, such as mobile phones, pagers and entry level PDAs.

## 2.2. J2EE

Java 2 Enterprise Edition (J2EE) is an established standard for developing distributed, multi-tier applications. The current version of J2EE is 1.3.

J2EE provides developers with the development tools and run-time capabilities necessary to build applications meeting rigorous security, stability and maintainability requirements. J2EE applications consist of a number of server-side components utilising technologies such as:

- Servlets
- JavaServer Pages (JSPs)
- Enterprise JavaBeans (EJBs).
- Java Connectivity Architecture (JVA)
- Java Message Service (JMS)
- Java Management Extensions (JMX)
- Java Naming and Directory Interface ( JNDI)
- Java Database Connectivity (JDBC)

J2EE allows a wide variety of clients to interact with these server-side components, including web browsers, Java applets, standalone applications and wireless clients. Clients typically communicate to the server-side components using HTML or XML over HTTP.

 

# 3. J2ME MIDP/J2EE Integration

Using both the J2ME MIDP and J2EE platforms, it is possible to implement mobile enterprise solutions.

On the server-side, J2EE applications are developed and deployed to one of the many varied application servers available. The J2EE standards allow choice of an application server appropriate to the strategic purpose of the application.

On the client-side, MIDP applications are developed. These applications can be run on any MIDP-compliant device, whether it's a mobile phone, pager or PDA. This allows you to serve a wider range of clients, making your enterprise application even more accessible.

 

# 4. HTTP Network Transport

J2EE clients typically use HTTP as a network transport. HTTP has several advantages over other protocols:

- **Widely deployed.** Almost every computer has a web browser that can communicate over HTTP. Thus the deployment of the application is simplified.
- **Robust and simple.** The protocol is simple and well understood. Good implementations of HTTP servers and clients are widely available.
- **Passes through firewalls.** Due to the extensive use of HTTP, firewalls are typically set up to pass HTTP through. This makes it the protocol of choice on the Internet where the server and the client are separated by a firewall.

In MIDP, HTTP is the only network transport protocol that is guaranteed to be implemented. Consequently, HTTP provides the most suitable network transport between the MIDP application and the server-side J2EE application.

HTTP is a request/response protocol: a MIDP client sends a HTTP request to a J2EE server, which returns a HTTP response.

MIDP includes standard support for HTTP 1.1, and APIs for generating HTTP GET, POST and HEAD requests, basic header manipulation, and stream-based consumption and generation of messages.

In MIDP, there are no requirements about the format of messages that flow between a MIDP client and J2EE server. The next section of this document discusses the various messaging formats available.

 

# 5. Messaging Formats

## 5.1. Proprietary Request/Response Data

The simplest and most flexible messaging format involves sending HTTP GET or POST requests to a J2EE server in a proprietary format. The J2EE server then responds with a HTTP response in a proprietary format.

In a HTTP GET request, the proprietary request data is encoded into the URL. For example, if a servlet requires forename and surname parameters, the URL would be:

http://www.developnet.co.uk/TestServlet?forename=Mike&surname=Taylor

In a HTTP POST request, the proprietary request data is sent separately from the URL. This has two major benefits:

- POST has no limit to the amount of data that can be sent. This is not the case for GET. When a server processes a HTTP GET request, the data from the end of the URL is stored in an environment variable. When sending a large amount of data, there is a risk of over-running this environment variable.
- POST sends the data as a separate stream. Therefore, the data can be in any format (including binary) and is not visible as part of the URL.

The source code fragment below shows a MIDP application sending a HTTP POST request to a servlet. The HTTP POST request contains two parameters `forename` and `surname` encoded using the form-urlencoded MIME type. The servlet takes the two parameters and concatenates them together to form a full name. This full name is returned as a HTTP response to the MIDP client using the binary MIME type.

```
String url = "http://www.developnet.co.uk/TestServlet";

String forename = "Mike";
String surname = "Taylor";

HttpConnection conn = (HttpConnection) Connector.open(url);

conn.setRequestMethod(HttpConnection.POST);
conn.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
conn.setRequestProperty("User-Agent",
                        "Profile/MIDP-1.0 Configuration/CLDC-1.0");
conn.setRequestProperty("Content-Language", "en-US");
conn.setRequestProperty("Accept", "application/octet-stream");
conn.setRequestProperty("Connection", "close"); // optional

String formData = "forename=" + forename + "&surname=" + surname;
Byte[] data = formData.getBytes();
conn.setRequestProperty("Content-Length", Integer.toString(data.length));

OutputStream os = conn.openOutputStream();
os.write(data);
os.close();

int rc = conn.getResponseCode();
if (rc == HttpConnection.HTTP_OK)
{
  DataInputStream dis = new DataInputStream(conn.openInputStream());
```

 

```
  String fullName = dis.readUTF();
  dis.close();
}
```

The related servlet source code fragment is:

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
{
  String forename = request.getParameter("forename");
  String surname = request.getParameter("surname");

  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  DataOutputStream dos = new DataOutputStream(baos);
  dos.writeUTF(forename + " " + surname);

  byte[] data = baos.toByteArray();

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

  OutputStream os = response.getOutputStream();
  os.write(data);
  os.close();
}
```

## 5.2. Serialised Java Objects

A common messaging format involves passing Java objects from client to server and vice versa.

Unfortunately, MIDP does not support the standard object serialisation and reflection facilities offered in Java 2 Standard Edition (J2SE).

Without standard object serialisation, there is no built-in way to serialise objects into a byte stream. Lack of reflection means it is impossible to build general serialisation support that works with arbritary objects.

However, it is fairly easy to implement serialisation by providing serialisation methods in the class to be persisted. One way to achieve this is to provide a Java interface for these methods:

```
public interface KSerializable
{
  byte[] serialize() throws IOException;
  void deserialize(byte[] data) throws IOException;
}
```

Any serialisable classes should implement this interface and as a result provide implementations for the `seralize` and `deseralize` methods:

```
public class Person implements KSerializable
{
  private String _forename;
  private String _surname;

  public Person()
  {
  }

  public Person(String forename, String surname)
  {
```

 

```
    _forename = forename;
    _surname = surname;
  }

  public String getForename()
  {
    return _forename;
  }

  public void setForename(String value)
  {
    _forename = value;
  }

  public String getSurname()
  {
    return _surname;
  }

  public void setSurname(String value)
  {
    _surname = value;
  }

  public byte[] serialize() throws IOException
  {
    ByteArrayOutputStream bout = new ByteArrayOutputStream();
    DataOutputStream dout = new DataOutputStream(bout);

    dout.writeUTF(_forename);
    dout.writeUTF(_surname);
    dout.flush();

    return bout.toByteArray();
  }

  public void deserialize(byte[] data) throws IOException
  {
    ByteArrayInputStream bin = new ByteArrayInputStream(data);
    DataInputStream din = new DataInputStream(bin);

    _forename = din.readUTF();
    _surname = din.readUTF();
  }
}
```

The serialisable classes are used by both the MIDP client and J2EE server, so it is a good idea to package them into a JAR file. Unfortunately, we need a separate JAR file for the MIDP client and the J2EE server because of the extra pre-verification and obfuscation steps required for MIDP.

The source code fragment below shows a MIDP application sending a HTTP POST request to a servlet. The HTTP POST request contains the serialised `Person` object encoded using the binary MIME type. The servlet takes the `Person` object and deserialises it. It then sets the forename and surname to different values, re-serialises the `Person` object and returns it to the MIDP client in a HTTP response using the binary MIME type.

```
String url = "http://www.developnet.co.uk/SerializeServlet";

HttpConnection conn = (HttpConnection) Connector.open(url);

conn.setRequestMethod(HttpConnection.POST);
conn.setRequestProperty("Content-Type", "application/octet-stream ");
```

 

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

Person personIn = new Person("Mike", "Taylor");

byte[] data = personIn.serialize();
conn.setRequestProperty("Content-Length", Integer.toString(data.length));

OutputStream os = conn.openOutputStream();
os.write(data);
os.close();

int rc = conn.getResponseCode();
if (rc == HttpConnection.HTTP_OK)
{
  int len = (int)conn.getLength();
  InputStream in = conn.openInputStream();

  if (len != -1)
  {
    int total = 0;
    data = new byte[len];

    while (total < len)
    {
      total += in.read(data, total, len - total);
    }
  }
  else
  {
    ByteArrayOutputStream tmp = new ByteArrayOutputStream();
    int ch;

    while ((ch = in.read()) != -1)
    {
      tmp.write(ch);
    }

    data = tmp.toByteArray();
  }

  Person personOut = new Person();
  personOut.deserialize(data);
}
```

The related servlet source code fragment is:

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
{
  int len = (int)request.getContentLength() ;
  ServletInputStream in = request.getInputStream();

  byte[] data;

  if (len != -1)
  {
    int total = 0;
    data = new byte[len];

    while (total < len)
```

```
      {
        total += in.read(data, total, len - total);
      }
    }
    else
    {
      ByteArrayOutputStream tmp = new ByteArrayOutputStream();
      int ch;
      while ((ch = in.read()) != -1)
      {
        tmp.write(ch);
      }

      data = tmp.toByteArray();
    }

    Person person = new Person();
    person.deserialize(data);

    person.setForename("Nicola");
    person.setSurname("Oakley");

    data = person.serialize();

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

    OutputStream os = response.getOutputStream();
    os.write(data);
    os.close();
  }
```

## 5.3. XML

Extensible Mark-up Language (XML) is a standardised, portable, text-based method of representing structured data. XML is rapidly becoming the preferred messaging protocol for enterprise applications wishing to exchange data. Furthermore, XML is the messaging protocol of choice for Web services.

Due to the clear benefits of using XML, many J2EE applications publish an XML interface for utilisation by clients. The J2EE platform provides a rich set of libraries for building XML-based server applications including Java APIs for XML Messaging (JAXM), XML-based RPC (JAX-RPC), and XML Registries (JAXR). The lower-level Java API for XML Processing (JAXP) allows developers to manipulate XML documents using the Document Object Model (DOM) and Simple API for XML (SAX), and to transform XML documents using Extensible Stylesheet Language Transformations (XSLT).

MIDP does not provide native support for XML parsing. However, several open source XML parsers exist which are especially suitable for MIDP/ Due to the resource constraints of MIDP, these XML parsers are generally non-validating and have minimal features.

kXML[1] provides a non-validating XML pull parser and writer targeted specifically at the MIDP platform. The minimal kXML JAR file weighs in at 21 kilobytes, with the complete kXML JAR (including kDOM and WBXML) at 37 kilobytes.

[1] This document covers kXML 1.*x*. kXML 2 is a new version of kXML, based on a common XML pull API. kXML 2 is currently in beta state.

 

Providing a pull model, the code for processing state can be implemented in a natural and efficient manner using local variables and recursions. In contrast, the push model (such as that provided by SAX parsers) requires a more complicated implementation because internal state has to be maintained to be able to handle an event correctly. Due to this complication, many implementers choose to use DOM to process the XML document.

The following code demonstrates the usage of the kXML pull model to parse a document we have received over the network from a server. We assume the `conn` object exists and the server has sent us a HTTP response containing a valid XML document.

```
InputStreamReader doc = new InputStreamReader(conn.openInputStream());
XmlParser parser = new XmlParser(doc);

boolean keepParsing = true;

while (keepParsing)
{
  ParseEvent event = parser.read();

  switch (event.getType())
  {
    case Xml.START_TAG:
      // Handle start of an XML tag
      break;

    case Xml.END_TAG:
      // Handle end of an XML tag
      break;

    case Xml.TEXT:
      // Handle text within a tag
      break;

    case Xml.WHITESPACE:
      // Handle whitespace
      break;

    case Xml.COMMENT:
      // Handle comment
      break;

    case Xml.PROCESSING_INSTRUCTION:
      // Handle XML processing instruction
      break;

    case Xml.DOCTYPE:
      // Handle XML doctype
      break;

    case Xml.END_DOCUMENT:
      // Handle end of document;
      keepParsing = false;
      break;
  }
}
```

The pull model of kXML should handle most requirements for XML processing in MIDP applications. However, kXML provides an optional kDOM package which is a simplified version of DOM optimised for space efficiency.

 

kXML provides the `XMLWriter` class for writing XML. Using the writer ensures that any XML documents created adhere to the XML escaping rules. The following code demonstrates use of the `XMLWriter` class to send a HTTP POST request containing XML to a servlet:

```
String url = "http://www.developnet.co.uk/XMLServlet";

HttpConnection conn = (HttpConnection) Connector.open(url);

conn.setRequestMethod(HttpConnection.POST);
conn.setRequestProperty("Content-Type", "text/xml");
conn.setRequestProperty("User-Agent",
                        "Profile/MIDP-1.0 Configuration/CLDC-1.0");
conn.setRequestProperty("Content-Language", "en-US");
conn.setRequestProperty("Accept", "text/xml");
conn.setRequestProperty("Connection", "close"); // optional

ByteArrayOutputStream baos = new ByteArrayOutputStream();

XMLWriter writer = new XMLWriter(new OutputStreamWriter(baos));

writer.startTag("person");
writer.startTag("forename");
writer.write(forename);
write.endTag();
writer.startTag("surname");
writer.write(surname);
writer.endTag();

byte[] data = baos.toByteArray();

conn.setRequestProperty("Content-Length", Integer.toString(data.length));

OutputStream os = conn.openOutputStream();
os.write(data);
os.close();
```

XML messages can be orders of magnitude larger than the proprietary messages discussed in the previous sections. Due to the bandwidth limitations of wireless networks, passing XML between a MIDP client and J2EE server may become impractical because of the verbosity of XML documents.

However, there is a solution in the form of WBXML. WBXML is part of the WAP specification and is used in WAP to provide a compressed binary message encoding of WML documents which are passed between the WAP gateway and a mobile phone. WBXML can also be used to encode XML documents. WBXML works by replacing common tag and attribute names and/or values with a configurable set of tokens.

kXML provides support for both parsing and writing WBXML documents. Of course, if the MIDP client is using WBXML, the J2EE server must be able to generate and accept WBXML documents. This is not a problem in a closed environment where the client developer can dictate the message format. In an open environment, where the client developer has no control over the message format, it is possible to provide a WBXML proxy server. This accepts the clients WBXML documents and converts them to XML documents before passing them on to the J2EE server.

When implementing a J2EE server which generates and accepts proprietary XML, it is a good idea to provide a set of Document Type Definitions (DTDs) or schemas which client developers can use to ensure their XML requests are in the required format.

 

kXML is a non-validating parser, so it is not possible for MIDP clients to validate the XML against a DTD or schema. However, this is not normally a problem as validation can be handled exclusively by the J2EE server. As soon as a HTTP request containing XML arrives at the server, it is validated against the DTD or schema. If a problem is detected, a suitable HTTP response containing details of the validation error is returned to the client.

## 5.4. XML-RPC

XML-RPC is an extremely lightweight protocol that invokes remote procedures over a network by sending XML formatted messages over HTTP.

An XML-RPC message is contained within a HTTP POST request sent by a client. The server receives the HTTP POST request, parses the XML document, executes the appropriate procedure with the appropriate parameters and returns the results formatted as XML in a HTTP response back to the client.

XML-RPC leverages all the benefits of XML and provides the minimum functionality necessary to specify data types, pass parameters and invoke remote procedures in a platform-independent fashion. As such, XML-RPC is a popular protocol for use in Web services.

XML-RPC defines six primitive data types (int, boolean, string, double, datetime and base64) and two complex types (struct and array).

MIDP does not provide native support for XML-RPC. However, the open source kXML-RPC project provides an implementation of XML-RPC suitable for MIDP. This implementation is built on top of kXML which was covered in the previous section. The kXML-RPC JAR file (which includes the minimal version of kXML) weighs in at 24 kilobytes.

On the server-side, there are several Java implementations of XML-RPC available. This document will focus on using the Apache XML-RPC implementation to provide server-side XML-RPC support for a servlet.

The source code fragment below shows a MIDP application invoking a remote procedure using XML-RPC. An `XmlRpcClient` object is constructed passing in the URL to the servlet which is handling XML-RPC requests. Next, a `Vector` object is constructed and parameters for the method we wish to call are added. To execute a remote RPC, the `execute()` method is called on the `XmlRpcClient` object. The first parameter is a string containing the object name and method name separated by a full stop. The second parameter is the `Vector` object containing the method parameters.

```
XmlRpcClient xmlrpc = new XmlRpcClient( "http://localhost:8080/xmlrpcservlet" );

Vector params = new Vector();
params.addElement("Mike");
params.addElement("Taylor");
String fullname = (String) xmlrpc.execute("person.getFullname", params);
```

On the server-side, a class is provided which implements the remote procedure. This is known as a XML-RPC handler.

```
public class PersonHandler
{
  public PersonHandler()
  {
  }

  public String getFullname(String forename, String surname)
  {
```

 

```
    return forename + " " + surname;
  }
}
```

In the servlet, the `XmlRpcServer` object is constructed. Next, the `person` object name is registered against an instance of the `PersonHandler` class. The request is then passed in to the XML-RPC implementation which resolves incoming calls via object introspection.

```
XmlRpcServer xmlrpc = new XmlRpcServer ();

xmlrpc.addHandler ("person", new PersonHandler ());

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

response.setContentType ("text/xml");
response.setContentLength (result.length);
OutputStream out = response.getOutputStream();
out.write (result);
out.flush ();
```

The XML-RPC request passed to the server is:

```
<methodCall>
 <methodName>person.getFullname</methodName>
 <params>
  <param>
   <value>
    <string>Mike</string>
   </value>
  </param>
  <param>
   <value>
    <string>Taylor</string>
   </value>
  </param>
 </params>
</methodCall>
```

The XML-RPC response returned to the client is:

```
<?xml version="1.0" encoding="ISO88591"?>
<methodResponse>
 <params>
  <param>
   <value>Mike Taylor</value>
  </param>
 </params>
</methodResponse>
```

## 5.5. SOAP

Simple Object Access Protocol (SOAP) is, according to the W3C specification, “a lightweight protocol intended for exchanging structured information in a decentralised, distributed environment”.

SOAP was originally conceived and developed by Microsoft to meet the needs of developers who found distributed computing difficult to deploy with existing Microsoft technologies. The idea was to create an open protocol specification that defined a uniform mechanism for performing RPCs using HTTP as the underlying communication protocol and XML as the data serialisation format. SOAP began to gain widespread attention when DevelopMentor, IBM, Lotus and Microsoft submitted SOAP 1.1 to the W3C in April 2000. The current version of SOAP is 1.2.

 

SOAP is quite similar to XML-RPC, in that it provides an XML-based protocol for communication and data exchange. However, whereas XML-RPC provides an ultra-lightweight protocol for invoking remote procedures and passing complex data structures, SOAP provides a more versatile protocol in exchange for some additional overhead.

SOAP provides the following extra features over XML-RPC:

- **Namespace awareness.** SOAP provides namespace awareness via the W3C XML Namespaces specification.
- **Sophisticated data-typing mechanism.** SOAP supports over 40 standard data types and provides capabilities to define custom simple and complex data types using the W3C XML Schema specification.
- **Flexible messaging paradigm.** SOAP provides a flexible messaging architecture, supporting a wide variety of messaging paradigms, including uni-directional, bi-directional, multi-cast and sequential messaging. Furthermore, SOAP supports asynchronous messaging, whereas XML-RPC requires a synchronous RPC-style exchange between parties.
- **Supplementary header information.** SOAP messages contain a header which can provide security, transaction and routing information.

Elements of the SOAP specification were based on an earlier version of the XML-RPC specification. However, due to the extra features discussed above, SOAP has quickly overtaken XML-RPC as the most popular protocol for Web services.

MIDP does not provide native support for SOAP. However, the open source kSOAP project provides an implementation of SOAP suitable for MIDP. The SOAP JAR file (which includes the minimal version of kXML) weighs in at 41 kilobytes.

On the server-side, there are several Java implementations of SOAP available. This document will focus on using the Apache SOAP implementation to provide server-side SOAP support for a servlet.

The source code fragment below shows a MIDP application using kSOAP to invoke a remote procedure on a SOAP server.

```
HttpTransport transport =
  new HttpTransport("http://localhost:8080/soap/servlet/rpcrouter", "");

SoapObject request = new SoapObject("urn:developnet:testservlet", "getFullName");

request.addProperty("forename", "Michael");
request.addProperty("surname", "Taylor");

String result = transport.call(request).toString();
```

The related servlet source code is:

```
public class SOAPServlet
{
  public String getFullName(String forename, String surname)
  {
    return forename + " " + surname;
  }
}
```

The SOAP request passed to the server is:

```
<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/">
  <getFullName xmlns="urn:developnet:testservlet" id="o0" SOAP-ENC:root="1">
   <forename xmlns="" xsi:type="xsd:string">Michael</forename>
   <surname xmlns="" xsi:type="xsd:string">Taylor</surname>
  </getFullName>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAP response returned to the client is:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <SOAP-ENV:Body>
  <ns1:getFullNameResponse xmlns:ns1="urn:developnet:testservlet"
   <SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <return xsi:type="xsd:string">Michael Taylor</return>
  </ns1:getFullNameResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

 

# 6. Conclusion

One of the most important factors to consider when choosing a message format is the limited memory, processing power and network bandwidth available to MIDP devices.

Ensure that the device the client application is targeted at has enough memory to support the chosen message format. Many first-generation Java mobile phones have only 128 kilobytes of memory available for MIDP applications. To be able to handle SOAP messages, the mobile phone requires the kSOAP JAR which is 41 kilobytes in size. This JAR consumes over 30% of the devices memory, before taking into account memory used by the client application JAR.

Both XML-RPC and SOAP produce messages in XML format. These messages are significantly larger than equivalent messages in a proprietary binary format. Compressing XML messages using WBXML goes some way to solving the extra bandwidth consumed. However, the J2EE server must support WBXML and extra processing will be required on the MIDP client to compress the XML. However, XML benefits from being human readable and self-describing, making debugging and development easier.

One of the main benefits of an XML-based messaging solution, such as XML-RPC or SOAP, is the ability to integrate systems in a heterogeneous environment. However, if you are developing a 100% Java technology-based solution, the benefits of using XML are diminished.

In some cases, the J2EE server dictates the messaging format you choose. For example, a J2EE server developed by a third party may aim to maximise its interoperability with clients by providing an XML-based interface, such as XML-RPC or SOAP. However, because of the low network bandwidth and limited processing power of MIDP devices, there may be performance problems associated with a MIDP client connecting directly to the XML-based interface on the J2EE server.

One solution to these problems is to introduce a proxy sever between the MIDP client and J2EE server. The MIDP client communicates with the proxy server using a lightweight protocol – perhaps WBXML or a proprietary format – which minimises message size, thus optimising use of the limited bandwidth available. The proxy server acts a client to the J2EE server, translating requests from the lightweight protocol to the XML-based interface exported by the J2EE server, and vice versa for responses. As the J2EE server provides an XML-based interface, interoperability with other clients is maximised. The clear benefits of a proxy server architecture are offset by the increased complexity of the solution and the considerable extra development effort for the proxy server and associated lightweight protocol.

If the requirements prescribe an XML-based solution, XML-RPC should be the automatic choice. SOAP should only be used if the extra features provided over and above XML-RPC are essential for the solution. However, due to the wide acceptance of SOAP, Web services more often than not export a SOAP interface in favour of an XML-RPC interface. If a high level of interoperability is required, then SOAP should be preferred.

Serialised Java objects provide a suitable message format within a 100% Java technology-based solution. However, they do introduce an extra layer of dependant objects between the client and server. This can often cause versioning problems, where a client has an old version of the serialisable objects. One major advantage of serialised objects is that features such as compression and encryption can be easily integrated via the serialise and

 

deserialise methods. Using serialised objects, it is possible to implement a lightweight generic command object architecture. This will be the subject of a future document.

In all but the simplest applications, we would recommend avoiding the proprietary request/response data message format described in section 5.1. The approach is not object-oriented and as the number of messages increases, the code becomes progressively more complicated and unmaintainable.

As a final point, it is important to performance test the chosen messaging architecture as early as possible during the implementation of the solution. The performance tests should be run in a live environment, with the MIDP application running on the target mobile device connected to the J2EE server over a live wireless network.

Better still, identify several candidate messaging architectures, implement prototypes, and run performance tests to discover which is the most appropriate.

 

# 7. Resources

## 7.1. Articles

*Designing Wireless Enterprse Application Using Java Technology*. Sun Microsystems.
http://developer.java.sun.com/developer/releases/smarticket/

*Programming Games in J2ME: The Battle for Market Share*. Sami Lababidi.
http://www.sys-con.com/java/articleprint.cfm?id=1488

*Master J2ME for Live Data Delivery*. John Chamberlain.
http://www.javaworld.com/javaworld/jw-05-2002/jw-0531-j2me_p.html

*J2ME and J2EE: Together at Last*. David Hemphill.
http://www.fawcette.com/javapro/2002_04/magazine/features/dhemphill/default_pf.asp

*Wireless Web Services With J2ME: Remote Possibilities*. Kyle Gabhart & Jason Gordon
http://www.sys-con.com/webservices/articleprint.cfm?id=136

*Wireless Web Services With J2ME Part II: SOAP or XML-RPC? The Answer Depends on Your Needs*. Kyle Gabhart & Jason Gordon
http://www.sys-con.com/webservices/articleprint.cfm?id=167

*A Performance Analysis of Web Services on Wireless PDAs*. Vinay Bansal & Angela Dalton.
http://www.cs.duke.edu/~vkb/advnw/project/

*Parsing XML in CLDC-Based Profiles*. Eric Giguere.
http://wireless.java.sun.com/configurations/ttips/xmlparse/

*Compressing XML for Faster Wireless Networking*. Eric Giguere.
http://wireless.java.sun.com/midp/ttips/compressxml/

*Object Serialization in CLDC-Based Profiles*. Eric Giguere.
http://wireless.java.sun.com/midp/ttips/serialization/

*Client-Server Communications Between MIDlets and Servlets*. Eric Giguere.
http://wireless.java.sun.com/midp/ttips/clientserv/

## 7.2. Software

Enhydra kXML
http://kxml.enhydra.org/index.html

Enhydra kXML-RPC
http://kxmlrpc.enhydra.org/index.html

Enhydra kSOAP
http://ksoap.enhydra.org/index.html

Apahce XML-RPC
http://xml.apache.org/xmlrpc/index.html

Enhydra kSOAP
http://xml.apache.org/soap/index.html

 