FORUM NOKIA

NOKIA MMS JAVA LIBRARY

Version 1.1

March 4, 2002

Table of contents

Change history

March 4, 2002	Version 1.1	Document added into Forum Nokia

Disclaimer:
Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to the 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.

1. Purpose

This document describes the use of the Nokia MMS Java Library version 1.1. The library provides examples of the most common tasks applications perform through the Multimedia Messaging Service Center (MMSC). Examples include such tasks as message creation and encoding, decoding and sending messages to the MMSC.

The intended audience of this document is developers involved with the development of external applications that need to interface the Multimedia Messaging Service Center. The reader of this document is expected to be familiar with the different application types, HTTP protocol and the properties of the External Application Interface (EAIF). For more information about EAIF, please refer to [1].

2. Scope

This document contains the following information and examples:

- Application types

  Descriptions of application types that are provided
- Usage examples and sample code

  Originating application

  Terminating application

3. Overview

The MMS Center provides an external application interface (EAIF). It allows external applications (EA) to send and receive multimedia messages to / from the MMS Center.

This chapter reviews the EAIF of MMS Center in relation to the following concepts:

- Types of applications
- Synchronous and asynchronous applications
- Nokia MMS Java Library version 1.1

More information about EAIF and application types can be obtained from the External Applications Developer's Guide [1].

3.1 Application types

- Originating application

Originating applications send application originated (AO) multimedia messages. In the originating application case the external application acts as a web client and the MMSC acts as a web server.

An example of originating application is illustrated in section 8.1

- Terminating application

Terminating applications receive application terminated (AT) multimedia messages. In the terminating application case an external application acts as a web server receiving the messages from the MMSC, which acts as a web client.

An example of a terminating application is illustrated in section 8.2

- Filtering application

The functionality of a filtering application can be seen as a combination of the terminating and originating applications. Filtering applications receive a message from the MMS Center (terminating functionality), process the message, and then send the message (or status) back to the MMS Center (originating functionality) for further processing.

3.2 Synchronous and asynchronous

Synchronous applications are able to handle one message at a time. Essentially, such an application receives a message, process it, and returns the message status before accepting another message for processing.

Asynchronous external applications are able to receive a message, check whether the message can be processed, and send an interim status report to the EAIF. Such applications are able to handle several messages at the same time. Subsequent messages can also be received for processing without the first or previous message returning a status report prior to another message being processed. After the EA has processed the message, the EA sends a modified message or a final status report to EAIF.

An originating application can only be synchronous. Terminating and filtering applications can be either synchronous or asynchronous.

3.3 Nokia MMS Java library

The Nokia MMS Java Library facilitates 3rd party developers to create applications based on multimedia messages. The library provides examples of how to create, encode, send and decode multimedia messages according to the WAP-209 MMS encapsulation specification [2] and external application interface specification [1].

4. Functionality of the Nokia MMS Java library

The Nokia MMS Java Library provides examples from the following functionalities:

- Message creation and encoding

The Nokia MMS Java Library provides an example of how to create and encode a multimedia message from different content types. The Nokia Java Library illustrates how the message creation and encoding can be done according to the WAP-209-MMS encapsulation specification. The message creation functionality example can be used when designing originating or filtering applications.

- Message decoding

The Nokia MMS Java Library provides an example of how to decode the received multimedia message, encoded according to WAP-209-MMS encapsulation, and how to extract the

multimedia content obtained from the message body. The decoding functionality example can be used when designing terminating or filtering applications.

- Message sending to the MMS center

In order to send a multimedia message from the application to the MMSC, HTTP 1.1 protocol and EAIF should be used. The Nokia MMS Java Library provides an example of how this can be done according to the specifications.

5. Before you start

Before installing the Nokia MMS Java Library, the user should:

- Check software (see Software).

5.1 Software

The Nokia MMS Java Library requires the following software:

- A Java2 Runtime Environment (JRE) v1.3.1_02 (or higher) installed. The JRE can be downloaded from the Sun website (http://java.sun.com).

6. Installing the Nokia MMS Java Library

The Nokia MMS Java library can be downloaded from the web pages of Forum Nokia (http://www.forum.nokia.com) after registering to the system and accepting the common terms.

After download identify where the Nokia MMS Java Library will be installed. This path will be referenced as mmslibrarydir throughout the document.

To install the Nokia MMS Java Library, unzip the MMSLibrary.zip file into the directory specified by mmslibrarydir using WinZip or some other uncompressing software.

After the extracting process, the following directories will be created in the mmslibrarydir:/mmslibrary containing the MMSLibrary.jar file

- /samples — containing two sample applications (originating and terminating)
- /doc — containing javadoc pages
- /src — containing source files
- /mmslibrary — containing jar file

In order to complete the installation you have to add in your CLASSPATH (environment variable) the following path:

mmslibrarydir/mmslibrary/MMSLibrary.jar

In Unix (C shell) you can do it with following command:

setenv CLASSPATH mmslibrarydir/mmslibrary/MMSLibrary.jar

In Windows NT and Windows 2000, you should open System from Control Panel. From there open Environment and edit or add your CLASSPATH environment. The following figure illustrates how to add the CLASSPATH environment in Windows NT.

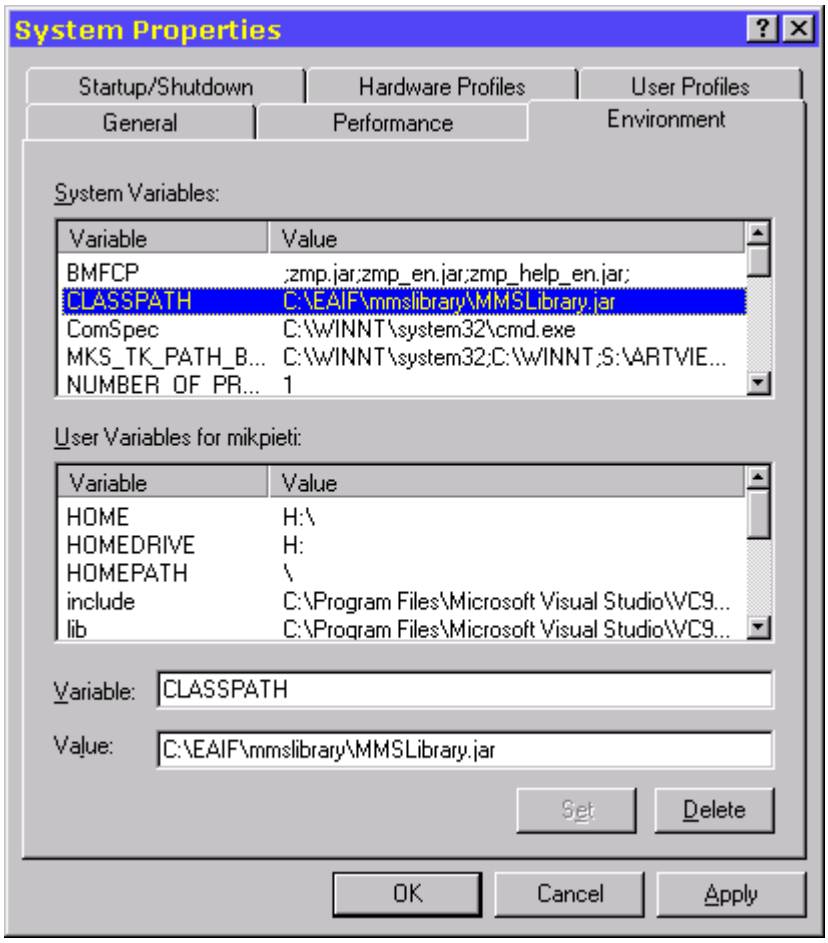

Figure 1 System properties window

For other operating systems, please refer to operating system manuals for how to set the environment variable.

7. Description of the classes

This chapter defines the classes included in the Nokia MMS Java Library version 1.1. In order to get a detailed description of the classes please refer to the Javadoc pages contained in the Library.

7.1 IMMConstants

IMMConstant is an interface that includes all the constants that are useful for the treatment of a multimedia message (MM); see [2] to get more information about the different fields in the message. The class contains the following constants:

- numbers assigned to the field names that compose the mm-header
- address types such as “TYPE=IPv4”, “TYPE=IPv6”, “TYPE=PLMN”

- most common content types used for the entries of the MM
- different kinds of MM classes, such as “personal”, “advertisement”, etc.
- different levels of priority that can be assigned to a MM

7.2 MMMessage

The MMMessage class represents a Multimedia Message. It contains all the methods to set and get the mm-header fields and to add the contents (represented by the MMContent class) included in the body of the MM.

7.3 MMEncoder

The MMEncoder class encodes Multimedia Message object (represented by MMMessage class) into an array of bytes according to the specification in [2].

7.4 MMDecoder

The MMDecoder class decodes an array of bytes representing a multimedia message (MMMessage) according with the specification in [2]. The class can be used to obtain an MMMessage object from which it can be possible access to each field and content of the MM.

7.5 MMAddress

The MMAddress class represents a generic address for a sender or a receiver of a MM.

7.6 MMContent

The MMContent class represents a generic entry for a MM. It contains methods to set and get the array of bytes representing the content, to set and get the content type and to save the content into a binary file.

7.7 MMSender

The MMSender class provides methods to send MMs to a predefined Multimedia Service Center (MMSC). The MM can be sent either as an MMMessage object (decoded format) or as an array of bytes (encoded format).

8. Using the Nokia MMS Java library

The following tutorial walks through the sample applications included with the Nokia MMS Java Library. You have the complete source listing in mmslibrarydir/samples directory. The sample applications utilize the classes presented in the previous chapter.

8.1 Example: Originating application

The sample application demonstrates the use of the Nokia MMS Java Library to create a simple originating application. See the OriginatingApp.java example in the samples folder to view the code of the application.

The Nokia MMS Java Library can be added to applications using the standard Java import mechanism:

```
import com.nokia.mms.*;
```

These other imports are necessary for utilities:

```
import java.io.*;
import java.util.*;
import java.net.*;
```

When the sample application is executed, a new `OriginatingApp` object is instantiated.

In the `OriginatingApp()` constructor, we first create a `MMMessage` object that represents a multimedia message.

```
MMMessage mm = new MMMessage();
```

Once we have a `MMMessage` object, we have to set the properties of the message. To do this, we will invoke the `SetMessage(mm)` method, passing as parameter the `mm` object (`MMMessage` object).

In the `MMMessage` object we are going to set both mandatory and optional properties. See [2] to get more information about the fields.

```
mm.setVersion(IMMConstants.MMS_VERSION_11);
mm.setMessageType(IMMConstants.MESSAGE_TYPE_M_SEND_REQ);
mm.setTransactionId("0000000066");
mm.setDate(new Date(System.currentTimeMillis()));
mm.setFrom("+358990000003/TYPE=PLMN");
mm.addToAddress("+358990000005/TYPE=PLMN");
mm.addToAddress("john.doe@nokia.com");
mm.setDeliveryReport(true);
mm.setReadReply(false);
mm.setSenderVisibility(IMMConstants.SENDER_VISIBILITY_SHOW);
mm.setSubject("This is a nice message!!");
mm.setMessageClass(IMMConstants.MESSAGE_CLASS_PERSONAL);
mm.setPriority(IMMConstants.PRIORITY_LOW);
mm.setContentType(IMMConstants.CT_APPLICATION_MULTIPART_MIXED);
```

After setting the message properties, we can add contents to the body part of the message.

```
. . .
AddContents(mm);
```

In order to add content to the body part of the message, we have to create an `MMContent` object.

```
MMContent part1 = new MMContent();
```

The MMContent object provides the methods to set the content and its properties.

```
part1.setContent(buf1, 0, buf1.length);
part1.setContentId("<0>");
part1.setType(IMMConstants.CT_TEXT_PLAIN);
```

In the example, `buf1` is an array of bytes containing the content (`sample_text.txt` file).

At this point we can add the content to the message.

```
mm.addContent(part1);
```

We can add more contents to the message. Here we only add a second content, but we could add more.

```
MMContent part2 = new MMContent();
byte[] buf2 = readFile(path + "sample_image.jpg");
part2.setContent(buf2, 0, buf2.length);
part1.setContentId("<1>");
part2.setType(IMMConstants.CT_IMAGE_JPEG);
mm.addContent(part2);
```

After adding the content(s) to the message, we have to encode the message according the WAP-209 MMS encapsulation specification [2].

In order to encode the message we first create the `MMEncoder` object and then set the message.

```
MMEncoder encoder=new MMEncoder();
encoder.setMessage(mm);
```

Now we can encode the message and get the encoded message.

```
encoder.encodeMessage();
byte[] out = encoder.getMessage();
```

Once we have obtained the encoded message, we can send it to the EAIF of the MMS Center. Here we use localhost's port number 8189, but we can use also other port numbers depending on the configuration of the receiving MMSC. Typically in real programs the address should be configurable or parameterized.

```
MMSender sender = new MMSender();
sender.setMMSCURL("http://127.0.0.1:8189");
sender.addHeader("X-NOKIA-MMSC-Charging", "100");
```

In the `MMSender` object we first set the address of the MMS Center and then we can add the Nokia HTTP extension headers. To get more information about the HTTP extension headers and the corresponding values, please see [1].

At this point we can send the message and get the response from the MMS Center.

```
. . .
MMResponse mmResponse = sender.send(out);

System.out.println("Message sent to " + sender.getMMSCURL());

System.out.println("Response code: " +
            mmResponse.getResponseCode() + " " +
            mmResponse.getResponseMessage());

Enumeration keys = mmResponse.getHeadersList();

while (keys.hasMoreElements()){
  String key = (String) keys.nextElement();
  String value = (String) mmResponse.getHeaderValue(key);
  System.out.println(key + ": " + value);
}
. . .
```

For an exhaustive documentation of the Nokia MMS Java Library, please refer to the javadoc pages.

8.2 Example: Terminating application

The sample application demonstrates the use of the Nokia MMS Java Library to create a simple terminating application. See the TerminatingApp.java example in the samples folder to view the code of the application.

The Nokia MMS Java Library can be added to the application using the standard Java import mechanism:

```
import com.nokia.mms.*;
```

These other imports are necessary for utilities:

```
import java.io.*;
import java.util.*;
import java.net.*;
```

When the sample application is executed, a new `TerminatingApp` object is instantiated.

```
public static void main (String[] args) {
    // Default port for creating the server socket.
    int port = 7000;
    int mode = SYNC_MODE;

. . .

     TerminatingApp ta = new TerminatingApp(port, mode);
}
```

`TerminatingApp` requires two parameters:

- port number for creating the server socket
- mode SYNC_MODE (terminating application synchronous) or ASYNC_MODE (terminating application asynchronous)

SYNC_MODE is the default, ASYNC_MODE is given with parameter A. If you don't give the second parameter, SYNC_MODE is used.

In the `TerminatingApp(int aPort, int aMode)` constructor, we first create a server socket on the specified port, waiting for a request to come in over the network.

```
. . .

ServerSocket s = new ServerSocket(aPort);
```

Then we have to listen for a connection to be made to this socket.

```
Socket incoming = s.accept( );
```

After a connection has been established, we have to retrieve both an input stream and an output stream in order to read the message and write the response to the MMS Center.

```
in = new DataInputStream(incoming.getInputStream());
out = new DataOutputStream(incoming.getOutputStream());

client = incoming.getInetAddress().getHostAddress() + ":" + HOST_PORT;

System.out.println("New message received from " + client);

. . .
```

From the input stream we extract the message and all the headers (including Nokia HTTP extension headers) invoking the `parseRequest()` method.

```
. . .
// Decodes the multimedia message.
MMDecoder mmDecoder = new MMDecoder();
mmDecoder.setMessage(content);
mmDecoder.decodeMessage();
MMMessage mm = mmDecoder.getMessage();
. . .
```

Having the `MMMessage` object we can access all the information of the message. See `printMessage(…)` method.

Looking at the `printMessage(…)` method we can see the usage of the `MMAddress` object. This object is necessary to extract the originator's and recipient's addresses.

```
. . .
if (mm.isToAvailable()) {
      Vector list = mm.getTo();
      for (int n=0; n < list.size(); n++) {
        MMAddress to = (MMAddress)list.elementAt(n);
        System.out.println(" TO (" + n + "): " + to.getAddress() + " (type=" +
to.getType() + ")");
      }
    }
. . .
```

Having an `MMMessage` object, it is also possible to save the different contents contained in the message body. In order to do this, the `MMMessage` object provides the following methods:

- `getNumContents()` returns the number of contents contained in the message body.
- `getContent(int i)`returns an array of bytes containing the content number i. The index (i) of the first content will be 0.

In the `storeContents(…)` method, we can see how to extract and store the contents of the message.

```
. . .

MMContent mmContent = null;
String path = getPath();
```

```
for (int j=0; j<mm.getNumContents(); j++) {  mmContent = mm.getContent(j);
  System.out.println(" Content " + j + " --> ID = " +
        mmContent.getContentId() + ", TYPE = " + mmContent.getType() +
        ", LENGTH = " + mmContent.getLength());

if ((mmContent.getType()).compareTo(IMMConstants.CT_IMAGE_WBMP)==0)

mmContent.saveToFile(path+cleanString(mmContent.getContentId())+".wbmp");

. . .
```

After processing the message, we must send a response to the MMS Center.

```
. . .
if (success)
        response = "HTTP/1.1 " + (mode == SYNC_MODE ? "204 No Content" : "202
Accepted") + "\r\n";
  else
        response = "HTTP/1.1 400 Message Validation Failed\r\n";

out.writeBytes(response);
out.flush();
out.close();
in.close();
incoming.close();

. . .
```

In the case of an asynchronous application we have to send a response containing an interim response (phase 1) and then if phase 1 is ok we can send the status of the message to the MMS Center (phase 2).

```
 . . .
if (success && mode == ASYNC_MODE) {  //if interim response (phase 1) was OK
   Thread.sleep(RESPONSE_DELAY);
   sendAsyncResponse();               //phase 2
}
. . .
```

In the `sendAsyncResponse()` method we can see both how to create a post request and how to set the Nokia HTTP extension headers required for an asynchronous terminating application.

```
. . .
URL url = new URL("http://" + client);
HttpURLConnection uc = (HttpURLConnection) url.openConnection();
uc.setDoInput(true);
uc.setDoOutput(true);
uc.setUseCaches(false);
uc.setRequestProperty("X-NOKIA-MMSC-Message-Id",(String)headers.get("x-nokia-
mmsc-message-id"));
uc.setRequestProperty("X-NOKIA-MMSC-Message-Type", "MultiMediaMessage");
uc.setRequestProperty("X-NOKIA-MMSC-Status", "200");

. . .
```

For an exhaustive documentation of the Nokia MMS Java Library, please refer to the javadoc pages.

REFERENCES

[1] External Applications Developer's Guide, DN00148759, Nokia Oyj

[2] WAP-209-MMSEncapsulation-20010601-a, WAP Forum specification, available from WAP Forum web pages http://www.wapforum.org

LIST OF TERMS AND ABBREVIATIONS

Term or abbreviation	Description
EA	External Application
EAIF	External Application Interface
JRE	Java2 Runtime Environment
MMS	Multimedia Messaging Service
MMSC	Multimedia Messaging Service Center