

Working with the Wireless Messaging API

Author: Martin de Jode, Sunny Khaila
Status: Version 1.32
Date: 10/10/03

1. Introduction

The Wireless Messaging API (JSR 120) is an optional API targeted at devices supporting the Generic Connection Framework defined in the CLDC. The Wireless Messaging API (WMA) specification defines APIs for sending and receiving SMS messages and receiving CBS messages. At the time of writing the current release of the Wireless Messaging API is version 1.1. This contains minor modifications to the 1.0 specification to enable the API to be compatible with MIDP 2.0.

The Wireless Messaging API is one of the APIs mandated in the first release of the JTWI (JSR 185) roadmap for Java enabled mobile phones. Consequently the WMA should become an integral part of most MIDP enabled devices in the future. Naturally an implementation is available as part of Symbian's Java offering in the latest release of Symbian OS (Version 7.0s at the time of writing).

The WMA is a compact API containing just two packages:

```
javax.microedition.io  
javax.wireless.messaging
```

The first package contains the platform network interfaces modified for use on platforms supporting wireless messaging connection, in particular an implementation of the `Connector` class for creating new `MessageConnection` objects. The second package defines APIs which allow applications to send and receive wireless messages. It defines a base interface `Message` which `BinaryMessage` and `TextMessage` both derive from. It also defines a `MessageConnection` interface, which provides the basic functionality for sending and receiving messages, and a `MessageListener` interface for listening to incoming messages.

In this paper we consider sending and receiving SMS messages. We shall then go on to show how to use the Push Registry API of MIDP 2.0 to register an incoming SMS connection with a MIDlet.

2. Sending Messages

Sending an SMS message using the WMA could not be simpler as the code paragraph below shows.

```
String address = "sms://+447111222333";
MessageConnection smsconn = null;
try {
    //Open the message connection.
    smsconn = (MessageConnection)Connector.open(address);
    TextMessage txtMessage =
        (TextMessage) smsconn.newMessage(MessageConnection.TEXT_MESSAGE);
    txtmessage.setPayloadText("Hello World");
    smsconn.send(txtMessage);
    smsconn.close();
} catch (Exception e) {
    //handle
}
}
```

First we obtain a `MessageConnection` instance by invoking the `Connector.open(...)` method with an address of the appropriate syntax. A `MessageConnection` can operate in client or server mode depending on URL syntax of the address passed to the `open()` method. For a client mode connection (as used in the example code above) messages can only be sent. The URL address syntax for a client mode connection has the following possible formats:

```
sms://+447111222333
sms://+447111222333:1234
```

The first URL is used to open a connection for sending a normal SMS message to the end user (as the in the example above), which will be received in the end user's inbox. Note a Java application will not be able to receive a message sent using this URL format. If we wish a Java application to receive the message on the remote device we must use the second URL format to send the message. This includes a port number and is used to open a connection to send an SMS message to a particular application listening on the specified port.

The `MessageConnector` instance is then used to create a `Message` instance using the `newMessage(String type)` method. The `MessageConnection` interface defines two public static final `String` variables `BINARY_MESSAGE` and `TEXT_MESSAGE`. If type is equal to `BINARY_MESSAGE` an instance of `BinaryMessage` is returned, whereas if type equals `TEXT_MESSAGE` an instance of a `TextMessage` is returned. Both `BinaryMessage` and `TextMessage` implement the `Message` interface. In the above code we specify a type equal to `TEXT_MESSAGE` and cast the returned instance appropriately.

Now that we have a `TextMessage` object we use the `setPayloadText(String)` method to set the message text. We are now ready to send the message. This is achieved by invoking the `send(Message)` method of the `MessageConnection` class. Finally when we no longer need the connection we should close it using the `close()` method inherited from the `Connection` class.

3. Receiving Messages

Receiving a message is again straightforward. We illustrate this with a simple receiver class shown below that implements the `MessageListener` interface for notification of incoming messages.

```
import javax.wireless.messaging.*;
import java.io.*;
import javax.microedition.io.*;

public class Receiver implements MessageListener, Runnable {
    private String receivedMessage;
```

```

private String senderAddress;
private MessageConnection smsconn;

public void openReceiver(String smsPort){
    //smsPort has the form "1234"
    try {
        smsconn = (MessageConnection) Connector.open("sms://:" + smsPort);
        smsconn.setMessageListener(this);
    } catch (IOException ioe){
        //handle
    }
}

public void notifyIncomingMessage(MessageConnection conn) {
    new Thread(this).start();
}

public void run(){
    receiveMessage();
}

public void receiveMessage(){
    Message msg = null;
    try {
        msg = smsconn.receive();
        if (msg != null) {
            senderAddress = msg.getAddress();
            if (msg instanceof TextMessage) {
                //extract text message
                receivedMessage = ((TextMessage)msg).getPayloadText();
            }
            else {
                //do something with binary message
            }
        }
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}

public void closeReceiver() {
    if (smsconn != null) {
        try {
            smsconn.close();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
}

```

To receive an incoming message the first thing we must do is open a `MessageConnection`. This is performed in the `openReceiver()` method of our `Receiver` class. This time we want a server mode `MessageConnection` so we pass in an address to the `Connector.open(...)` method using the "sms://:1234" URL syntax. We cast the returned `Connection` instance to a `MessageConnection` and call the `setMessageListener(this)` method on it.

The `MessageListener` interface mandates the `notifyIncomingMessage(...)` method so this must be implemented. In the above example it simply calls the `receiveMessage(...)` method in a new thread.

The `receiveMethod(...)` method calls the `MessageConnection.receive()` method which returns an instance of the received `Message`. We can get the address of the sender of the message using the `getAddress()` method

of the `Message` class. This enables the receiver to send a reply – a server mode `MessageConnection` can be used to send as well as receive messages - however when creating a message to be sent the recipients address must be explicitly set using the `setAddress(String)` method of `Message`. We can test the received message to determine if it is an instance of `TextMessage` (as opposed to a `BinaryMessage`) and if so use the `getPayloadText()` method of `TextMessage` to extract the contents as a `String`. If the `Message` is an instance of `BinaryMessage` then the corresponding `getPayloadData()` method returns a byte array.

4. WMA in MIDP 2.0 - Using the Push Registry

The Wireless Messaging API can be implemented on a MIDP 1.0 or MIDP 2.0 platform.

When implemented in conjunction with MIDP 2.0 the Wireless Messaging API can take advantage of the new push registry technology that is part of MIDP 2.0. The push registry is an exciting addition to the MIDP specification allowing the Application Management System (AMS) to launch a MIDlet in response to an incoming connection. A full discussion of MIDP 2.0 is beyond the scope of this paper, however we shall briefly look at launching a MIDlet in response to an incoming SMS message.

The MIDlet registers its interest in incoming connections in the application descriptor (JAD) file, by specifying the protocol and port for the connection end point. The entry in the application descriptor has the following format:

```
MIDlet-Push-<n>: <ConnectionURL>, <MIDletClassName>, <AllowedSender>
```

In this example the entry in the JAD file would be as follows.

```
MIDlet-Push-1: sms://:1234, SMSMIDlet, *
```

The `<AllowedSender>` field acts as a filter indicating that the AMS should only respond to incoming connections from a specific sender. For the SMS protocol the `<AllowedSender>` entry is the phone number of the required sender (note the sender port number is not included in the filter). Here the wildcard character "*" indicates respond to any sender.

The AMS will respond to the appropriate incoming connection by launching the corresponding MIDlet (assuming it is not already running). The MIDlet should then respond to the incoming connection by launching a thread to handle the incoming data in the `startApp()` method. Using a separate thread is the recommended practice for avoiding conflicts between blocking I/O operations and the normal user interaction events. The `startApp()` method might look something like this.

```
public void startApp() {
    // List of active connections.
    String connections[];
    connections = PushRegistry.listConnections(true);

    if (connections.length != 0) {
        for (int i=0; i < connections.length; i++) {
            if (connections[i].equals("sms://:1234")) {
                new Thread() {
                    public void run() {
                        Receiver.openReceiver();
                    }
                }.start();
            }
        }
    }
    ...
}
```

5. WMA on Symbian OS

The first implementation of the Wireless Messaging API on Symbian OS shipped as part of Nokia's Series 60 v1.x platform. This WMA implementation supplemented the MIDP 1.0 environment available on this platform. The first phone to support this implementation of the WMA was the Nokia 3650. More recently a MIDP 2.0 compatible version of the WMA shipped as part of Symbian OS v7.0s which forms the basis for Nokia's Series 60 v2.0 platform. The first device announced using this and therefore supporting WMA in conjunction the push registry technology will be the Nokia 6600, available in Q4 2003. Symbian's implementation of the WMA currently does not support receiving CBS.

6. Example Code

A small example SMS chat application is available from Symbian's Developer Portal

http://www.symbian.com/developer/techlib/staffapps_java.html

illustrating the use of the WMA to send and receive SMS. The application was written for the Nokia 3650 and so does not include push functionality.

7. Resources

Wireless Messaging API

<http://java.sun.com/products/wma/>

J2ME Wireless Toolkit 2.0

<http://java.sun.com/products/j2mewtoolkit/>

MIDP 2.0 Specification

<http://java.sun.com/products/midp/>

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.