19

Text and Multimedia Messaging

THE Wireless Messaging API (WMA) is a bridge between your MIDlets and the wonderful world of text and multimedia messaging. Most mobile phones are capable of sending and receiving messages. WMA extends this capability to MIDlets.

JSR 120 defines WMA 1.1, which encompasses Short Message Service (SMS), commonly known as *text messaging* or *texting*. JSR 205 defines WMA 2.0, which adds support for Multimedia Messaging Service (MMS).

WMA 2.0 is a superset of WMA 1.1. MSA requires WMA 2.0, but most MIDP devices out in the world today support WMA 1.1 or WMA 1.0. If you are aiming your application at the widest possible audience, stick to the basic SMS functionality provided by WMA 1.1. On the other hand, if you are targeting MSA devices, you can use the full range of MMS supported by WMA 2.0.

Like other network communication, WMA is based on the Generic Connection Framework (GCF) that you read about in Chapter 18. Use `Connector` to get a `MessageConnection`. The `MessageConnection`, in turn, can be used to create, send, and receive messages.

WMA is a compact API that lives in `javax.wireless.messaging`.

19.1 Why Messaging?

WMA is a great solution for some kinds of network communication, although the usual caveats about device testing apply. SMS and MMS travel through a

store-and-forward network, which means that messages are not lost if the destination is unavailable. For example, if you send a message to your friend when your friend's phone is turned off, the network hangs on to the message until it can be delivered to the phone. One good application for WMA is to transmit turns between players in a slow-moving, turn-based game like chess.

Another advantage to SMS and MMS is that they do not involve a server. You can easily communicate between applications running on different devices with no server-side programming.

Finally, WMA combined with the PIM API is a powerful combination. The PIM API gives your application access to people your user cares about. WMA gives you the ability to send those people messages.

19.2 Sending Messages

It takes longer to explain how to send a message than it takes to write it out in code. Here is the short story:

```
public void sendText(String address, String text)
    throws IOException, InterruptedException {
  String cs = "sms://" + address + ":50000";
  MessageConnection mc = (MessageConnection)
      Connector.open(cs);
  TextMessage tm = (TextMessage)
      mc.newMessage(MessageConnection.TEXT_MESSAGE);
  tm.setPayloadText(text);
  mc.send(tm);
}
```

It's basically four lines, but the casts makes it look bulky. Here are the four steps:

1. Obtain a `MessageConnection` from `Connector`. In the example, `cs` is the connection string, which I'll talk about soon.
2. Get a `Message` from the `MessageConnection` by calling `newMessage()`. In this example, I asked for a text message.
3. Fill up the message. This works differently for different message types. Here I called `setPayloadText()`, a method that is specific to `TextMessage`.
4. Send the message by passing it to the `MessageConnection`'s `send()` method.

The address in the connection string is the telephone number of the device to which you are sending the message. The port number is optional. The example above automatically appends a port number of 50,000.

If you leave off the port number, your message is an ordinary SMS message and will end up in an inbox at the destination. This is nice if you want an actual person to read the message.

If, instead, you are trying to send the message to an application running on the destination device, include a port number. Some port numbers are reserved. They are listed in the WMA specification. Any free port number should work, as long as the sender and recipient agree on that number.

In most cases, the push registry will field an incoming message and launch a MIDlet to respond. Go back to Chapter 6 if you can't remember the push registry. It's possible, but unlikely, that the receiving MIDlet will actually be running and listening for incoming messages. If no running MIDlet is listening for messages on the right port, and if no MIDlet is in the push registry for the right port, the message is likely to disappear entirely.

The people who named SMS were not kidding about the "short" part. Text messages can be, at most, 160 bytes, which allows for 160 characters in a single English message. Different message encodings will decrease the maximum message length. Furthermore, specifying a port number eats up 8 bytes.

Although individual message are small, WMA will actually split apart longer payloads into multiple messages, which will be reassembled at the receiving end. The specification requires implementations to be able to split long payloads into at least three messages. Keep in mind that many users must pay a small fee for each message, so even if it seems like you're sending one payload, your user might have to pay for more than one message.

Check out Table A-1 in the JSR 205 specification for a list of message lengths under different conditions.

19.3 Sending Binary Messages

The payload of an SMS message can carry binary data. You will need a `BinaryMessage` instead of a `TextMessage`, but the basic procedure is the same.

```
public void sendBinary(String address, byte[] data)
    throws IOException, InterruptedException {
  String cs = "sms://" + address + ":50001";
  MessageConnection mc = (MessageConnection)
      Connector.open(cs);
  BinaryMessage bm = (BinaryMessage)
      mc.newMessage(MessageConnection.BINARY_MESSAGE);
```

continued

```
    bm.setPayloadData(data);
    mc.send(bm);
  }
```

Remember, you've only got 152 bytes in a message, although longer payloads will be split into multiple messages.

19.4 Sending Multipart Messages

MMS is the next generation of SMS. It enables larger messages, with multiple addresses, a subject, and different parts. In essence, you can use MMS to send one or more files from one device to another. One powerful application for MMS is sending pictures you have just taken with your phone's camera around the world to your friends. Be aware that getting MMS working between different carriers might be a challenge.

The exact size limit on MMS messages depends on the phone and the wireless carrier. Maximum sizes of 100 kB are typical, but some phones and carriers can go as high as 300 kB or 1 MB.

JSR 205, WMA 2.0, adds support for MMS with the `MultipartMessage` class, a sibling of `TextMessage` and `BinaryMessage`. A `MultipartMessage` keeps track of multiple addresses, a message subject, and some number of `MessageParts`, which are essentially files with associated content types.

Before you learn about `MessagePart`, take a look at the basic structure for sending a `MultipartMessage`.

```
  public void sendMultipart(String address, String subject,
      MessagePart[] parts)
      throws IOException, InterruptedException {
    String cs = "mms://+" + address +
        ":com.jonathanknudsen.Hermes";
    MessageConnection mc = (MessageConnection)
        Connector.open(cs);
    MultipartMessage mm = (MultipartMessage)
        mc.newMessage(MessageConnection.MULTIPART_MESSAGE);
    mm.setSubject(subject);
    for (int i = 0; i < parts.length; i++)
      mm.addMessagePart(parts[i]);
    mc.send(mm);
  }
```

This should look familiar by now: get a `MessageConnection`, get a message, populate it, and kick it out the door.

The connection string is a little different for multipart messages. It starts with mms instead of sms, for one thing, and it includes an *application identifier*, which is a lot like the port number for SMS messages, but more reliable. In SMS messages, port numbers are used to map to a specific application. For example, if you send a message on port 50,000, you're expecting the receiving application to be listening on port 50,000. There are lots of port numbers from which to choose, but let's face it: sooner or later some kids in a garage are going to write an application that uses the same port number as yours. No central registry exists that ensures port numbers do not overlap between applications.

Instead of port numbers, MMS messages use an application identifier. Messages are sent and received for a specific application identifier. The convention for application identifiers is an inverted domain name plus an application name. In the earlier example, I used com.jonathanknudsen.Hermes to identify my application. The chances of someone else using the same application identifier accidentally are very slim. This is a good system, but note that the application identifier can be a maximum of 32 characters, which is kind of short.

MMS messages are souped-up compared to SMS messages. You can set a message subject, for one thing. Furthermore, you can add more destination addresses. In the earlier example, the message is going to a single address as specified in the connection string. Use addAddress() in MultipartMessage to supply more addresses. You have to specify both an address type and the address itself. The address type is a string, one of to, cc, or bcc, which have the same meanings as for e-mail.

Each file in a multipart message has a *content type*, a *content ID*, an optional *location*, and an optional encoding scheme. The content type is specified as a MIME type, the same type scheme used by Web servers and in e-mail attachments. The content ID should be unique for each part of a multipart message. The location is usually just a filename, and the encoding scheme is useful if you are including encoded text in a part.

Here's a simple case, which encodes a Unicode string using UTF-8 and creates a MessagePart for a text string:

```
private MessagePart makePart(String text, String id)
    throws SizeExceededException,
           UnsupportedEncodingException {
  String encoding = "UTF-8";
  byte[] encoded = text.getBytes(encoding);
  return new MessagePart(encoded, "text/plain", id,
      null, encoding);
}
```

This method creates a MessagePart from a byte array. A more common case is to create a MessagePart from an InputStream. The other parameters in the constructor are the same. Here is a method that creates a MessagePart from a resource file in the MIDlet suite:

```
private MessagePart makePart(String filename,
    String type, String id)
    throws IOException, SizeExceededException {
  InputStream in =
      this.getClass().getResourceAsStream(filename);
  return new MessagePart(in, type, id, filename, null);
}
```

19.5 Receiving Messages

To receive messages, create a MessageConnection using a messaging connection string without a destination address. For example, to receive SMS messages on port 50,000, use this connection string:

```
sms://:50000
```

To receive MMS messages for the com.jonathanknudsen.Hermes application identifier, use this connection string:

```
mms://:com.jonathanknudsen.Hermes
```

Of course, you won't catch most messages with a live MessageConnection. Instead, you will put your MIDlet in the push registry, most likely with a static registration in the MIDlet suite descriptor. If this all sounds like gobbledygook, go back to Chapter 6 and read slower.

Picking up incoming messages is a lot like retrieving incoming datagrams or incoming socket connections. There are two ways to do this, both of which involve an additional thread that listens for incoming messages.

The first method is to create a thread that loops on calling MessageConnection's receive() method. This works pretty well, except that your thread is blocked at receive() until something comes in.

The second method is to use a MessageListener. You supply the listener and register it with the MessageConnection. Whenever a message arrives, your listener's notifyIncomingMessage() is called. Then you go ahead and call receive(), knowing full well that there really is a message waiting.

Unfortunately, it's not quite that simple, because you should not call `receive()` from the same system thread that calls `notifyIncomingMessage()`. Remember, when the system invokes one of your callback methods, you should respect its thread and do your own work in your own threads.

It's possible to set up a `MessageListener` in a rational way using `wait()` and `notify()`, but it's not trivial. The example later in this chapter includes such a listener.

`MessageConnection` gives you a `Message` when you call `receive()`, so you might have to do `instanceof` tests and casts if you're not sure what kind of message you are expecting.

19.6 A Simple Messaging Application

`HermesMIDlet` (see Figure 19.1) demonstrates how to send and receive SMS messages. It sends and receives on port 50,000. If you download the example from the book's Web site, the MIDlet suite descriptor has a static push registry entry, but `HermesMIDlet` also attempts to register itself dynamically if the static registration fails for any reason.

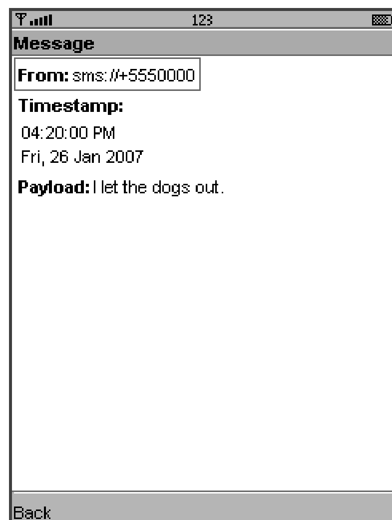

Figure 19.1 Running `HermesMIDlet`

The entire application consists of four classes:

1. `HermesMIDlet` runs the show. It handles commands and knows how to send messages.
2. `HermesMessageReader` receives incoming messages in its own thread. It is a `MessageListener`.
3. `HermesForm` is the user interface for creating a message.
4. `HermesMessageForm` is the user interface for displaying an incoming message.

The whole source code is lengthy, so only the interesting parts are here.

Sending a message is straightforward. Two methods in `HermesMIDlet` get the job done, and that includes the `sendText()` method you've already seen.

```
public void run() {
  String cs = "sms://" + mHermesForm.getAddress() + ":50000";
  String text = mHermesForm.getMessage();
  try {
    sendText(cs, text);
  }      mDisplay.setCurrent(mHermesForm);
  catch (IOException ioe) {
    showErrorAlert(ioe);
  }
  catch (InterruptedException ie) {
    showErrorAlert(ie);
  }
}

public void sendText(String cs, String text)
    throws IOException, InterruptedException {
  MessageConnection mc =
      (MessageConnection)Connector.open(cs);
  TextMessage tm = (TextMessage)
      mc.newMessage(MessageConnection.TEXT_MESSAGE);
  tm.setPayloadText(text);
  mc.send(tm);
}
```

`HermesMessageReader` handles incoming messages. Its `start()` method attempts to place an entry in the `PushRegistry` if that entry is not already present. In addition, it opens the incoming `MessageConnection` and registers itself as a listener.

`HermesMessageReader` maintains two lists, implemented as `Vectors`. The first is a list of `MessageConnections` with incoming data. Every time the listener call-

back method `notifyIncomingMessage()` is called, the connection is added to the list. In addition, the waiting thread in `run()` is awakened.

The second list contains received messages. When the `run()` thread wakes up, it retrieves a message from a waiting `MessageConnection`. The received message is put in the message list. The MIDlet retrieves available messages using the `hasMore()` and `next()` methods.

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

import javax.microedition.io.*;

import javax.wireless.messaging.*;

public class HermesMessageReader
    implements MessageListener, Runnable {
  private volatile boolean mTrucking;
  private Vector mConnections;
  private Vector mMessages;
  private HermesMIDlet mMIDlet;

  private MessageConnection mConnection;

  public HermesMessageReader(HermesMIDlet midlet) {
    mConnections = new Vector();
    mMessages = new Vector();
    mMIDlet = midlet;
  }

  public void start() throws IOException {
    mTrucking = true;
    Thread t = new Thread(this);
    t.start();

    // Try push registry registration if static registration
    // did not work.
    String[] registered = PushRegistry.listConnections(false);
    if (registered == null || registered.length == 0) {
      try {
        PushRegistry.registerConnection(
            "sms://:50000", "HermesMIDlet", "*");
      }
      catch (IOException ioe) {
        System.out.println(ioe);
      }
```

continued

```
      catch (ClassNotFoundException cnfe) {
        System.out.println(cnfe);
      }
    }

    // Register as a listener for our connection.
    mConnection =
        (MessageConnection)Connector.open("sms://:50000");
    mConnection.setMessageListener(this);
  }

  public synchronized void stop() {
    mTrucking = false;
    notify();
    try { mConnection.close(); }
    catch (IOException ioe) {
      System.out.println(ioe);
    }
  }

  // MessageListener method.

  public synchronized void notifyIncomingMessage(
      MessageConnection mc) {
    mConnections.addElement(mc);
    notify();
  }

  // Runnable method.

  public synchronized void run() {
    while (mTrucking) {
      try {
        MessageConnection mc = null;
        if (mConnections.size() == 0)
          wait();
        if (mTrucking == false) break;
        mc = (MessageConnection)mConnections.elementAt(0);
        mConnections.removeElementAt(0);
        Message m = mc.receive();
        mMessages.addElement(m);
        mMIDlet.notifyAvailableMessage();
      }
      catch (InterruptedException ie) {
        mTrucking = false;
      }
      catch (IOException ioe) {
        System.out.println("MessageReader.run(): " +
            ioe.toString());
```

```
        }
      }
    }

    // Public API.

    public boolean hasMore() {
      return mMessages.size() > 0;
    }

    public synchronized Message next() {
      Message m = (Message)mMessages.elementAt(0);
      mMessages.removeElementAt(0);
      return m;
    }
  }
```

If you don't have real devices for testing, you can use the Sun Java Wireless Toolkit to simulate a message network. If you launch more than one emulator, you can send messages between emulators. You can also use the WMA Console, available from **File > Utilities**, to send and receive messages.

With my Motorola V3, I was able to send a message to myself, and the message was sent and received by `HermesMIDlet`. However, when I tried to exchange messages between the V3 and my Nokia 6030 (on different wireless networks), the messages were successfully sent but ended up in the recipient's default message box rather than being retrieved by `HermesMIDlet`. The destination port number was lost in transit.

19.7 Summary

WMA gives your application the power to send and receive SMS and MMS messages. Use GCF to obtain a `MessageConnection` that can be used to send and receive messages. SMS messages can include a port number for delivery to a specific application. MMS messages use an application identifier for the same purpose. Use the push registry to invoke specific applications for specific types of incoming messages.