J2ME Low-Level Network Programming with MIDP 2.0

by Qusay H. Mahmoud
April 2003

Version 1.0 of the Mobile Information Device Profile (MIDP) lacks low-level networking support for TCP/IP sockets and UDP/IP datagrams, but the MIDP 2.0 specification (JSR 118) has responded to the needs of the 2.5G and 3G networks now being deployed by adding support for sockets and datagrams, thus providing mobile applications more capable networking interfaces. This support is based on the Generic Connection Framework (GCF) of the Connected Limited Device Configuration (CLDC). The word "optional" here means that the specification does not mandate such support; it is entirely up to the handset manufacturers and network operators to deploy these capabilities if doing so makes sense to them. Consequently, platform developers must carefully weigh the cost of particular phones in terms of footprint, processing power, and network capability.

This article provides an overview of the CLDC Generic Connection Framework and examines the new support for sockets and datagrams. The article:

Discusses the new support for low-level networking
Describes how to run the networking demos in the J2ME Wireless Toolkit 2.0
Demonstrates how to use the new low-level networking support
Provides sample code listings for a time MIDlet and an email MIDlet

Overview of CLDC Generic Connection Framework

The `java.io.*` and `java.net.*` packages of J2SE require much more memory than many hand-held wireless devices can afford. In addition, the following requirements for various communication mechanisms call for the design of a new set of abstractions that can be used at the programming level:

Device manufacturers that work with circuit-switched networks require stream-based connections such as the Transport Control Protocol (TCP), a connection-oriented protocol.
Device manufacturers that work with packet-switched networks require datagram-based connections such as the User Datagram Protocol (UDP), a connectionless protocol.
Other hand-held devices have specific mechanisms for communications.
These variations make the design of networking facilities for CLDC-based devices quite a challenge, and shaped the design of the Generic Connection Framework. The GCF provides a single set of related abstractions you can use at the programming level to handle multiple forms of communications, instead of using different abstractions for different protocols.

In this framework, all connections are created using the `open` static method of the `Connector` class. If successful, this method returns an object that implements one of the generic connection interfaces. Figure 1 shows the *is-a* relationships of these interfaces in an inheritance hierarchy. The `Connection` interface is the base interface. `StreamConnectionNotifier` *is a* `Connection` and `InputConnection` *is a* `Connection` too. In this figure all yellow interfaces are part of CLDC 1.0, the `HttpConnection` interface was added by MIDP 1.0, and the blue interfaces are among the interfaces added by MIDP2.0.

Interface Hierarchy of the Generic Connection Framework

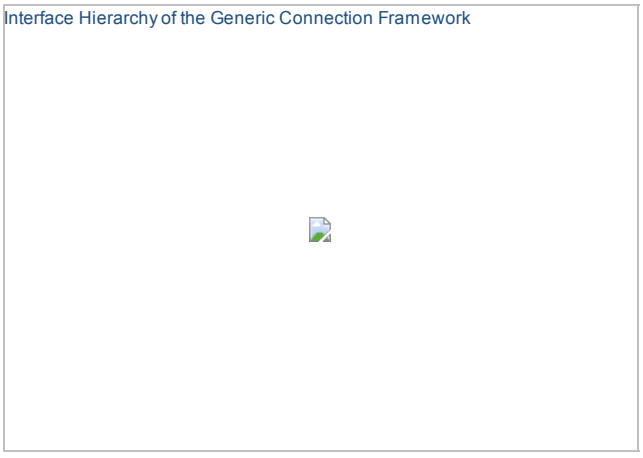

Figure 1: The Interface Hierarchy of the Generic Connection Framework
(Click to Enlarge)

You use `Connector.open` as in the next code fragment, supplying a single `String` parameter that specifies a protocol, an address, and parameters:

```
...
try {
   Connector.open("
                     protocol:
                     address;
                     parameters");
} catch (ConnectionNotFoundExeption e) {
   // no handler available for socket connections
}
...
```

Note: The original design of the Generic Connection Framework implied that, at deployment time, URL connection strings might be used as interchangeable access mechanisms; all `StreamConnection` applications would use convenience methods such as `Connector.openInputStream(url)`. In practice, however, the connections are rarely used in such a generic fashion. When using an `HttpConnection`, for example, your code should always call the `getResponseCode` method before assuming the input stream returned from the server has the data the application expects, and that no proxy- or server-reported error or redirection must be handled before accessing the stream data. Therefore, connections aren't really generic.

Note also a difference between message- and stream-oriented connections: For message-oriented connections, not all of the `Connector` convenience methods are supported.

A few examples:

HTTP Connection: `Connector.open("#")`
Socket Connection: `Connector.open("#: port")`
Datagram Connection: `Connector.open("#: port")`
The syntax of `open` isolates the differences between the setup of one protocol and the setup of another into a simple string that characterizes the type of connection. Most of the application's code remains the same regardless of the protocol you use.

Note: The `Connector.open` method is a *factory* for connections. You do not use the `new` operator to instantiate connections. This is a smart mechanism, which keeps the static footprint small when the platform does not include implementations for specific connections. The method throws `ConnectionNotFoundException` if the desired protocol handler is not available.

Remember that implementers of the MIDP specification are required to provide support for HTTP connections. Requiring MIDP support of HTTP was a very clever idea. When doing network programming, you can always revert to the HTTP programming model when you need to, confident that your application will run on any MIDP device, whether it is a GSM phone with a WAP stack, a phone with i-mode, a Palm VII wireless, or a hand-held device with Bluetooth. In addition, HTTP is firewall-friendly.

Note: MIDP 2.0 implementers *must* provide support for HTTP 1.1 servers as well as secure HTTP connections. They *should* also provide support for low-level IP networking, but this decision is left to the handset manufacturers and network operators. In certain circumstances they may have valid reasons to ignore this recommendation, but they must understand and carefully weigh the implications before choosing a different communication mechanism.

Low-Level IP Networking Support in MIDP 2.0

The new interfaces added to enable low-level IP networking are:

```
javax.microedition.io.SocketConnection
javax.microedition.io.ServerSocketConnection
javax.microedition.io.UDPDatagramConnection
```

Note that `Datagram` and `DatagramConnection` were included in the CLDC 1.0 specification.

A call like `Connector.open("socket:// host: port")` returns a `SocketConnection`, and a call like `Connector.open("socket://: port")` returns a `ServerSocketConnection`. A MIDlet should specify a host when requesting an outbound client connection and omit the host when requesting an inbound server connection. If you leave out the port parameter when obtaining a server socket – as in `Connector.open("socket://")` – an available port number is assigned dynamically. You can use the `getLocalPort` method to discover the assigned port number, and the `getLocalAddress` method to discover the local address to which the socket is bound. Note well that `host` and `port` are variables, which you must replace when you construct the URL string.

A call like `Connector.open("datagram:// host: port")` returns a `UDPDatagramConnection`. Note, however, that the UDP protocol is transaction-oriented, and delivery and duplicate protection are not guaranteed. Therefore, if your applications require ordered, reliable delivery and streams of data, you should use TCP/IP stream connections. This article will discuss only the TCP/IP stream connections. If you'd like to see an example of UDP datagrams, the networking demo in the J2ME Wireless Toolkit v2.0, discussed later, includes one.

The `SocketConnection` Interface

The `SocketConnection` interface defines the socket stream connection. You use it when writing MIDlets that access TCP/IP servers, as in the following code snippet:

```
...
SocketConnection client = (SocketConnection) Connector.open("socket://" + hostname + ":" + port);
// set application-specific options on the socket. Call setSocketOption to set other options
client.setSocketOption(DELAY, 0);
client.setSocketOption(KEEPALIVE, 0);
InputStream is = client.openInputStream();
OutputStream os = client.openOutputStream();
// send something to server
os.write("some string".getBytes());
// read server response
int c = 0;
while((c = is.read()) != -1) {
   // do something with the response
}
```

```
}
// close streams and connection
is.close();
os.close();
client.close();
...
```

`SocketConnection` provides some useful constants, `DELAY` and `KEEPALIVE`, among others. Please refer to the MIDP 2.0 specification for a full description of these options, which are outside the scope of this article.

The `ServerSocketConnection` Interface

The `ServerSocketConnection` interface defines the server socket stream connection. You use it when requesting an inbound server connection, as in the following snippet:

```
... // create a server to listen on port 2500
ServerSocketConnection server = (ServerSocketConnection) Connector.open("socket://:2500");
// wait for a connection
SocketConnection client = (SocketConnection) server.acceptAndOpen();
// set application-specific options on the socket;
// call setSocketOption to set other options
client.setSocketOption(DELAY, 0);
client.setSocketOption(KEEPALIVE, 0);
// open streams
DataInputStream dis = client.openDataInputStream();
DataOutputStream dos = client.openDataOutputStream();
// read client request
String result = is.readUTF();
// process request and send response
os.writeUTF(...);
// close streams and connections
is.close();
os.close();
client.close();
server.close();
...
```

Running the Sample Programs

If you want to run the example programs, and you haven't already installed the J2ME Wireless Toolkit 2.0, you'll need to download and install it now.

To run the sample low-level IP network application, open the `NetworkDemo` project and run it as shown in Figure 2.

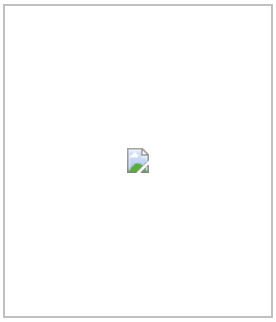

Figure 2: J2ME Wireless Toolkit Network Demo

Select Datagram Demo. You're going to run two phone instances, as shown in Figure 3. In the first instance select the server peer. Once it is running, start another phone instance as in Figure 2, run the Datagram Demo again, and this time select the client peer. The client connects to the server and they can exchange messages as shown:

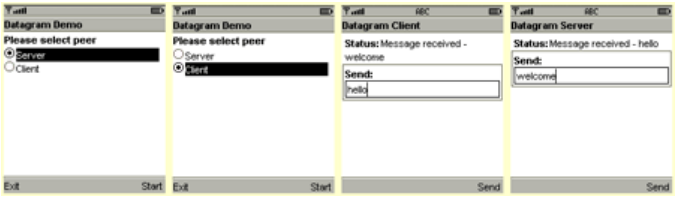

Figure 3: J2ME Wireless Toolkit Datagram Demo
(Click to Enlarge)

Developing Sample Applications

Now, let's look at a couple of sample applications that make use of the new support for low-level IP networking. I'll show the source code for a time MIDlet and an email MIDlet.

Time MIDlet

Imagine a simple but useful location-dependent mobile service: When you get off a plane in a different time zone, your phone adjusts to the local time automatically. The MIDlet can retrieve the current time in several ways. In this example, when the phone is co-located or turned on, it connects to a *daytime server*.

The daytime protocol is widely used by computers running UNIX and other operating systems. In this protocol, the server listens for client requests on port 13. When a client opens a connection to port 13, the server immediately supplies the current date and time. You can try it out simply by telneting to port 13 of a machine that is running the daytime protocol. Try this:

```
c:\> telnet prep.ai.mit.edu 13
```

If all goes well, you should see a message, *Connection to host lost*, along with the current day and time of the remote machine.

Code Sample 1 shows the front end of the Time MIDlet. When the user runs `TimeMIDlet`, it establishes a connection to the time server by creating an instance of `TimeClient`, which you'll see in Code Sample 2.

Code Sample 1: TimeMIDlet.java

```
import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import java.io.*;

public class TimeMIDlet extends MIDlet implements CommandListener {

   private static Display display;
   private Form f;
   private boolean isPaused;
   private StringItem si;
   private TimeClient client;
   private Command exitCommand = new Command("Exit", Command.EXIT, 1);
   private Command startCommand = new Command("GetTime", Command.ITEM, 1);

   public TimeMIDlet() {
      display = Display.getDisplay(this);
      f = new Form("Time Demo");
      si = new StringItem("Select GetTime to get the current Time! ", " ");
      f.append(si);
      f.addCommand(exitCommand);
      f.addCommand(startCommand);
      f.setCommandListener(this);
      display.setCurrent(f);
   }
   public void startApp() {
      isPaused = false;
   }

   public void pauseApp() {
      isPaused = true;
   }

   public void destroyApp(boolean unconditional) {
   }

   public void commandAction(Command c, Displayable s) {
      if (c == exitCommand) {
         destroyApp(true);
         notifyDestroyed();
      } else if (c == startCommand) {
         client = new TimeClient(this);
         client.start();
      }
   }
}
```

In Code Sample 2, `TimeClient.java` establishes a socket connection with a time server running on port 13 on a remote machine. The client doesn't send any data; the server treats the new connection itself as a service request, and immediately responds with the current date and time.

Code Sample 2: TimeClient.java

```
import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import java.io.*;

public class TimeClient implements Runnable {
   private TimeMIDlet parent;
   private Display display;
   private Form f;
   private StringItem si;
   private SocketConnection sc;
   private InputStream is;
   private String remoteTimeServerAddress;

   public TimeClient(TimeMIDlet m) {
      parent = m;
```

```
      display = Display.getDisplay(parent);
      f = new Form("Time Client");
      si = new StringItem("Time:" , " ");
      f.append(si);
      display.setCurrent(f);
   }

    public void start() {
       Thread t = new Thread(this);
       t.start();
    }

   public void run() {

      try {
        sc = (SocketConnection)
           Connector.open("socket://"+remoteTimeServerAddress+":13");

        is = sc.openInputStream();
        StringBuffer sb = new StringBuffer();
        int c = 0;
        while (((c = is.read()) != '\n') && (c != -1)) {
           sb.append((char) c);
        }
        si.setText(sb.toString());

      } catch(IOException e) {

         Alert a = new Alert
            ("TimeClient", "Cannot connect to server. Ping the server
              to make sure it is running...", null, AlertType.ERROR);
         a.setTimeout(Alert.FOREVER);
         display.setCurrent(a);

      } finally {
         try {
           if(is != null) {
             is.close();
           }
           if(sc != null) {
             sc.close();
           }
         } catch(IOException e) {
           e.printStackTrace();
         }
      }
   }

   public void commandAction(Command c, Displayable s) {
      if (c == Alert.DISMISS_COMMAND) {
         parent.notifyDestroyed();
         parent.destroyApp(true);
      }
   }
}
```

Run the `TimeMIDlet` and you'll see a series of screens (as in Figure 4) that you need to go through to connect to a daytime server and retrieve the current date and time. Easier screen navigation would be nice – and is left to you as an exercise. The security permission screen is worth noting.

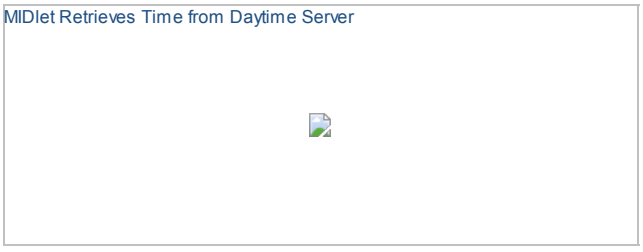

Figure 4: MIDlet Retrieves Time from Daytime Server
(Click to Enlarge)

When you choose the **GetTime** command to establish the connection, the MIDlet invokes `Connector.open`. The toolkit's emulator prompts you to grant or deny permission to make the connection. If you allow the connection, the MIDlet continues normally. If you deny permission, `Connector.open` throws a `SecurityException`. For more information on permissions and protection domains, please see Understanding MIDP 2.0's Security Architecture.

If `TimeMIDlet` can't connect to the time server (perhaps because it's not running or can't be reached), you see the alert shown in Figure 5:

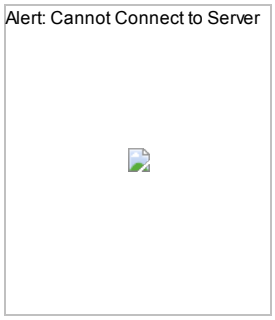

Figure 5: Alert: Cannot Connect to Server

SMTP Email MIDlet

As its name suggests, the Email MIDlet allows you to send email messages using the Simple Mail Transfer Protocol (SMTP). The SMTP service runs on port 25.

The `EmailMIDlet`'s GUI enables you to compose an email message, entering the recipient, subject, and body, as in Code Sample 3.

Code Sample 3: EmailMIDlet.java

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.io.*;
import java.io.*;

public class EmailMIDlet extends MIDlet implements CommandListener {
   Display display = null;

   // email form fields
   TextField toField = null;
   TextField subjectField = null;
   TextField msgField = null;
   Form form;

   static final Command sendCommand = new Command("send", Command.OK, 2);
   static final Command clearCommand = new Command("clear", Command.STOP, 3);
   String to;
   String subject;
   String msg;

   public EmailMIDlet() {
      display = Display.getDisplay(this);
      form = new Form("Compose Message");
      toField = new TextField("To:", "", 25, TextField.EMAILADDR);
      subjectField = new TextField("Subject:", "", 15, TextField.ANY);
      msgField = new TextField("MsgBody:", "", 90, TextField.ANY);
   }

   public void startApp() throws MIDletStateChangeException {
      form.append(toField);
      form.append(subjectField);
      form.append(msgField);
      form.addCommand(clearCommand);
      form.addCommand(sendCommand);
      form.setCommandListener(this);
      display.setCurrent(form);
   }

   public void pauseApp() {
   }

   public void destroyApp(boolean unconditional) {
      notifyDestroyed();
   }

   public void commandAction(Command c, Displayable d) {
      String label = c.getLabel();
      if(label.equals("clear")) {
         destroyApp(true);
      } else if (label.equals("send")) {
         to = toField.getString();
         subject = subjectField.getString();
         msg = msgField.getString();
         EmailClient client = new EmailClient
            (this,"qmahmoud@javacourses.com", to, subject, msg);
         client.start();
      }
   }
}
```

When you choose the **Send** button, the `EmailMIDlet` retrieves the information you entered and creates an instance of the `EmailClient` class,

which establishes a connection to the SMTP server and communicates with it, as in Code Sample 4.

Code Sample 4: EmailClient.java

```
import javax.microedition.midlet.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import java.io.*;
import java.util.*;

public class EmailClient implements Runnable {
   private EmailMIDlet parent;
   private Display display;
   private Form f;
   private StringItem si;
   private SocketConnection sc;
   private InputStream is;
   private OutputStream os;
   private String smtpServerAddress;

   private String from, to, subject, msg;

   public EmailClient
      (EmailMIDlet m, String from, String to, String subject, String msg) {
      parent = m;
      this.from = from;
      this.to = to;
      this.subject = subject;
      this.msg = msg;

      display = Display.getDisplay(parent);
      f = new Form("Email Client");
      si = new StringItem("Response:" , " ");
      f.append(si);
      display.setCurrent(f);
   }

   public void start() {
      Thread t = new Thread(this);
      t.start();
   }

   public void run() {

      try {
         sc = (SocketConnection)
            Connector.open("socket://"+smtpServerAddress+":25");
         is = sc.openInputStream();
         os = sc.openOutputStream();

         os.write(("HELO there" + "\r\n").getBytes());
         os.write(("MAIL FROM: "+ from +"\r\n").getBytes());
         os.write(("RCPT TO: "+ to + "\r\n").getBytes());
         os.write("DATA\r\n".getBytes());
         // stamp the msg with date
         os.write(("Date: " + new Date() + "\r\n").getBytes());
         os.write(("From: "+from+"\r\n").getBytes());
         os.write(("To: "+to+"\r\n").getBytes());
         os.write(("Subject: "+subject+"\r\n").getBytes());
         os.write((msg+"\r\n").getBytes()); // message body
         os.write(".\r\n".getBytes());
         os.write("QUIT\r\n".getBytes());

         // debug
         StringBuffer sb = new StringBuffer();
         int c = 0;
         while (((c = is.read()) != -1) ) {
            sb.append((char) c);
         }
         si.setText("SMTP server response - " + sb.toString());

      } catch(IOException e) {

         Alert a = new Alert
            ("TimeClient", "Cannot connect to SMTP server. Ping the server
              to make sure it is running...", null, AlertType.ERROR);
         a.setTimeout(Alert.FOREVER);
         display.setCurrent(a);
      } finally {
         try {
            if(is != null) {
               is.close();
            }
            if(os != null) {
               os.close();
            }
            if(sc != null) {
```

```
                sc.close();
             }
          } catch(IOException e) {
             e.printStackTrace();
          }
       }
    }

    public void commandAction(Command c, Displayable s) {
       if (c == Alert.DISMISS_COMMAND) {
          parent.notifyDestroyed();
          parent.destroyApp(true);
       }
    }
}
```

Figure 6 shows the Email MIDlet in action. As in `TimeMidlet`, when you select the **Send** command, the emulator prompts you for permission to connect. If you allow the connection to be made, the MIDlet continues processing. When the message is sent you'll see the SMTP server's response to each SMTP command. If `EmailClient` cannot establish a connection with the SMTP server, it displays an error message like the one in Figure 5.

Figure 6: Email MIDlet

Now you may wonder: How can such an Email MIDlet be implemented if the MIDP implementer has elected not to provide support for low-level IP networking? The answer: The MIDlet can communicate with a servlet using HTTP, and let the servlet handle all the low-level IP networking stuff. For more information on this, including sample code, please see Advanced MIDP Networking, Accessing Using Sockets and RMI from MIDP-enabled Devices

Conclusion:

MIDP 2.0 has added support for low-level IP networking based on the CLDC Generic Connection Framework. This article discussed the new classes that support low-level IP networking and described how to use them. The code samples in this article demonstrated how straightforward it is to use the new classes.

MIDP 2.0 implementers are required to provide support for HTTP 1.1 communication, but support for low-level TCP/IP sockets and UDP/IP datagrams is optional. Wireless software developers, therefore, should develop their applications with this uncertainty in mind -- write your applications in such a way that they can revert to the HTTP communication protocol if TCP/IP sockets or UDP/IP datagrams cannot be used. Sometimes, however, it may be that only a socket will do. Therefore, not all applications can run on all platforms.

For more information

Download the J2ME Wireless Toolkit 2.0
MIDP 2.0 (JSR 118)
CLDC 1.1 (JSR 139)
Understanding MIDP 2.0's Security Architecture
MIDP Network Programming using HTTP and the Connection Framework
Advanced MIDP Networking, Accessing Using Sockets and RMI from MIDP-enabled Devices

Acknowledgments

Special thanks to Gary Adams of Sun Microsystems, whose feedback helped improve the article.

About the Author: Qusay H. Mahmoud provides Java consulting and training services. He has published dozens of articles on Java, and is the author of *Distributed Programming with Java* (Manning Publications, 1999) and *Learning Wireless Java* (O'Reilly & Associates, 2002).