Advanced MIDP Networking, Accessing Using Sockets and RMI from MIDP-enabled Devices

By Qusay Mahmoud
January 2002

The Connected Limited Device Configuration (CLDC) provides a Generic Connection Framework that can be used to develop network-based applications. In addition, the Mobile Information Device Profile (MIDP) provides the `HttpConnection` interface, which is part of the `javax.microedition.io` package, that defines the necessary methods and constants for an HTTP connection. HTTP is the only protocol a MIDP implementation must support, all other protocols are optional. For example, in the reference implementation from Sun, there is no support for either Transport Control Protocol (TCP) sockets or User Datagram Protocol (UDP) datagrams.

In addition, the K Virtual Machine (KVM) does not support all the Java language and virtual machine features, either because they are too expensive to implement or their presence would impose security issues. For example, there is no support for object serialization, and consequently there is no support for Remote Method Invocation (RMI) either. The benefit of mobile applications, however, becomes real when you can access critical business data and Internet resources efficiently from anywhere you go.

This article presents a quick overview of the CLDC/MIDP networking mechanisms, then it discusses a middleman architecture that enables the use of Java sockets and Remote Method Invocation (RMI) from MIDP-enabled devices. This article:

Gives you a brief overview of CLDC and MIDP Networking
Discusses a middleman architecture for using sockets, RMI, and other Internet services
Shows you how to use sockets from MIDP-enabled devices
Shows you how to send emails from your MIDP-enabled device
Shows you how to use RMI from MIDP

Overview of CLDC and MIDP Networking

The CLDC inherited some of the classes in the `java.io` package, but it did not inherit classes related to file I/O mainly because not all devices support the concept of file I/O. The Java 2 Standard Edition (J2SE) provides several classes for network connectivity, however, none of these classes have been inherited simply because not all devices require TCP/IP or UDP/IP; some devices may not even have an IP stack.

The I/O and network connectivity challenge is solved by defining a new set of classes for I/O and network connectivity. These classes are known as the generic connection framework. This platform-independent framework provides its functionality without dependence on specific features of a device. It provides a hierarchy of connectivity interfaces as shown in Figure 1, but it does not implement any of them. Implementations are to be provided by profiles (such as MIDP).

Figure 1: CLDC Generic Connection Framework

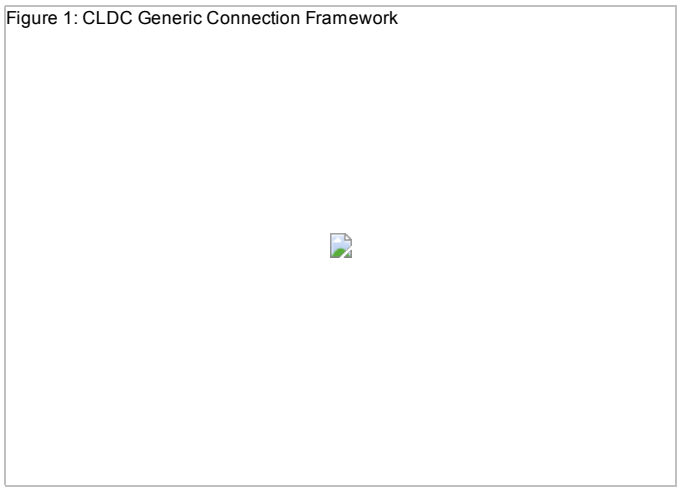

Figure 1: CLDC Generic Connection Framework

All connections are created using the `open` static method from the `Connector` class. If successful, this method returns an object that implements one of the generic connection interfaces.

The MIDP extends the CLDC connectivity to provide support for the HTTP protocol. MIDP provides the `HttpConnection` interface, which is a subclass of `ContentConnection`. The reason behind the HTTP support is the fact that HTTP can either be implemented using IP protocols (such as TCP/IP) or non-IP protocols (such as WAP). For example, a MIDP-enabled device may have no built-in support for the IP protocol. In such a case, it would utilize a gateway responsible for URL naming resolution to access the Internet. The idea of having MIDP supporting the HTTP protocol is very clever. For network programming, you can revert to the HTTP programming model, and your applications will run on any MIDP device, whether it is a GSM phone with a WAP stack, a Palm OS handheld, or a handheld device with Bluetooth. For more information on MIDP Networking, please see MIDP Network Programming.

Invoking HTTP-based Services

Using HTTP, it is possible to call Internet services that are based on the HTTP programming model, such as CGI scripts and Servlets. For more information on how to invoke CGI scripts and Servlets using HTTP GET and POST request methods, please see MIDP Inter-Communication with CGI and Servlets.

Middleman Architecture

CLDC and MIDP have no support for socket or datagram connections. Also, as I mentioned earlier, there is no support for RMI. Therefore, it is not possible to directly invoke socket- and RMI-based applications. As a solution we propose a middleman architecture that involves a servlet in the middle that accepts a request to open a socket connection or invoke a remote method. The servlet parses the client's request (if necessary), processes it, and returns the results back to the client. This architecture is shown in Figure 2.

Figure 2: Middleman Architecture

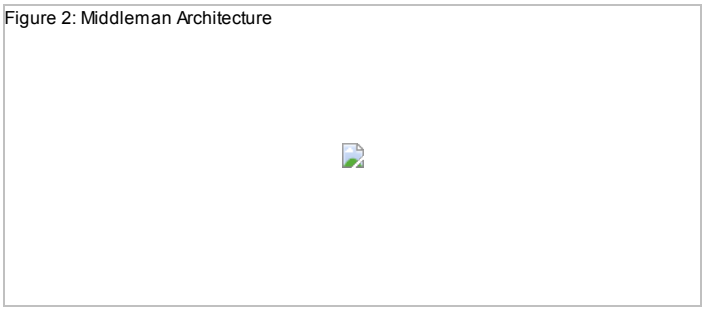

Figure 2: Middleman Architecture

Note: The client may use the eXtensible Markup Language (XML) to specify the request along with any parameters. The servlet would then parse the request, process it, collect the results, package them as an XML message, and serves it to the client. The examples here do not use XML, however.

Sockets

Now, let's see how to use the middleman architecture with sockets to send emails. Here we want to develop an application that allows us to send emails from a MIDP-enabled device (such as the Motorola i85s cellular phone or a Palm OS handheld). Here are the steps of how this application would work:

The device user starts a MIDlet that presents a mail form. The user would enter the email address of the recipient, the subject of the message, and the message body. Then presses a button to send the request to the servlet.
The servlet retrieves the values of the request and uses a utility class, which connects to a Simple Mail Transfer Protocol (SMTP) on port 25.
The utility class carries a conversation with the SMTP protocol to send an email message to the email address provided by the user of the device.
Once the message is sent, the servlet sends a confirmation message to the user.
For this application, we need a MIDlet, a servlet, and a utility class. Let's start with the MIDlet.

The MIDlet

We need a MIDlet which lets the user fill out a mail form providing information as to where the message is going (the email address of the recipient), the subject of the message, and the message body. The MIDlet will use a number of user interface classes for creating text fields and handling events. This MIDlet, `EmailMidlet`, is shown in CodeSample 1.

CodeSample 1: 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);
       toField = new TextField("To:", "", 25, TextField.EMAILADDR);
       subjectField = new TextField("Subject:", "", 15, TextField.ANY);
       msgField = new TextField("MsgBody:", "", 90, TextField.ANY);
       form = new Form("Fill Form");

    }
```

```
    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();
    }


    void invokeServlet(String url) throws IOException {
        HttpConnection c = null;
        InputStream is = null;
        OutputStream os = null;
        StringBuffer b = new StringBuffer();
        TextBox t = null;
        try {
          c = (HttpConnection)Connector.open(url);
          c.setRequestMethod(HttpConnection.POST);
          c.setRequestProperty("IF-Modified-Since", "20 Oct 2001 16:19:14 GMT");
          c.setRequestProperty("User-Agent","Profile/MIDP-1.0 Configuration/CLDC-1.0");
          c.setRequestProperty("Content-Language", "en-CA");
          c.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");

          os = c.openOutputStream();
          // encode data
          os.write(("to="+to).getBytes());
          os.write(("&subject="+subject).getBytes());
          os.write(("&msg="+msg).getBytes());
          os.flush();

          is = c.openDataInputStream();
          int ch;
          while ((ch = is.read()) != -1) {
            b.append((char) ch);
            System.out.print((char)ch);
          }
          t = new TextBox("Confirmation", b.toString(), 1024, 0);
          t.setCommandListener(this);
        } finally {
           if(is!= null) {
              is.close();
           }
           if(os != null) {
              os.close();
           }
           if(c != null) {
              c.close();
           }
        }
        display.setCurrent(t);
    }

   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();
         try {
           invokeServlet(url);
         }catch(IOException e) {}
      }
   }
}
```

A few points to note about the `EmailMidlet`:

The `EmailMidlet` invokes the `EmailServlet`
The HTTP request method used is POST
The `EmailMidlet` accepts three pieces of information for `to`, `subject`, and `msg`. The information is sent to the `EmailServlet` in a message of the form: `to=value&subject=hi+there&msg=msg`
In order to allow the servlet to use the `getParameter` method to retrieve corresponding values, we set the content type of `application/x-www-form-urlencoded`.

The Servlet

The servlet, `EmailServlet`, handles POST requests. It retrieves the values of the variables `to`, `subject`, and `msg`; then Constructs an instance of a utility class responsible for communicating with a SMTP server; and finally sends a confirmation message to the device that the message was delivered successfully. The `EmailServlet` is shown in CodeSample 2. From this listing note that we assume that the `from` email address field is always `qmahmoud@yahoo.com`. Please change this value to your email address.

Note: In the future if user profiles can be customized on devices, then the email address for the `from` field can be retrieved from the user profile!

CodeSample 2: EmailServlet

```
import java.io.*;
import java.text.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class EmailServlet extends HttpServlet {

   public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/plain");
        PrintWriter out = response.getWriter();

        String to = request.getParameter("to");
        String subject = request.getParameter("subject");
        String msg = request.getParameter("msg");
        // construct an instance of EmailSender
        EmailSender es = new EmailSender();
        es.send("qmahmoud@yahoo.com", to, subject, msg);
        out.println("mail sent....");

    }
}
```

Now, let's discuss the `EmailSender` utility class that is used by the `EmailServlet`.

The Utility Class

The `EmailSender` utility class is responsible for establishing a socket connection with the SMTP protocol on port 25 and carrying a conversation to send email. This class is shown in CodeSample 3. The `EmailSender` class provides one method `send`, which establishes a socket connection with an SMTP server and carries a conversation to send email. The servlet will have to provide the email address of the sender, receiver, the subject of the message, and the body of the message.

Note: To use this class you need to have access to an SMTP server. If you do, then replace the string **IP address for SMTP server** in CodeSample 3 with the symbolic name or IP address of the SMTP server.

CodeSample 3: EmailSender.java

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

public class EmailSender {

  public EmailSender() {
  }

  public void send(String from, String to, String subject, String msg) {
     Socket smtpSocket = null;
     DataOutputStream os = null;
     DataInputStream is = null;

     try {
       smtpSocket = new Socket("Address for SMTP server", 25);
       os = new DataOutputStream(smtpSocket.getOutputStream());
       is = new DataInputStream(smtpSocket.getInputStream());
     } catch (UnknownHostException e) {
       System.err.println("Don't know about host: hostname");
     } catch (IOException e) {
       System.err.println("Couldn't get I/O for the connection to: hostname");
     }

     if (smtpSocket != null && os != null && is != null) {
      try {
        os.writeBytes("HELO there"+"\r\n");
        os.writeBytes("MAIL FROM: "+ from +"\r\n");
        os.writeBytes("RCPT TO: "+ to + "\r\n");
        os.writeBytes("DATA\r\n");
        os.writeBytes("Date: "+ new Date() + "\r\n"); // stamp the msg with date
        os.writeBytes("From: "+from+"\r\n");
        os.writeBytes("To: "+to+"\r\n");
        os.writeBytes("Subject: "+subject+"\r\n");
        os.writeBytes(msg+"\r\n"); // message body
```

```
        os.writeBytes(".\r\n");
        os.writeBytes("QUIT\r\n");
        // debugging
        String responseLine;
        while ((responseLine = is.readLine()) != null) {
          System.out.println("Server: " + responseLine);
          if (responseLine.indexOf("delivery") != -1) {
             break;
          }
        }

        os.close();
        is.close();
        smtpSocket.close();
      } catch (UnknownHostException e) {
        System.err.println("Trying to connect to unknown host: " + e);
      } catch (IOException e) {
        System.err.println("IOException: " + e);
      }
    }
  }
}
```

Testing the Email Application

The email application was tested using the J2ME Wireless Toolkit, version 1.0.3, and Tomcat server, version 3.2.1. When the `EmailMidlet` is started in an i85s device, you would see something similar to Figure 3.

Figure 3: Starting the midlet in i85s

I have also tested the MIDlet on a Palm OS handheld device as shown in Figure 4.

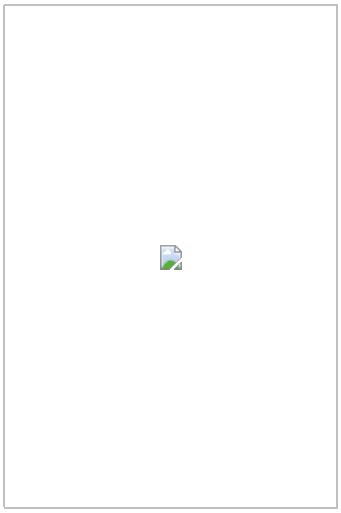

Figure 4: Starting the midlet on a Palm OS handheld device

The message in Figure 4 was sent from the Palm OS handheld device and received in my mailbox as shown in Figure 5.

Figure 5: Email message received from the Palm OS handheld

RMI

Now, let's see how the middleman architecture can be used to invoke a remote method. In this example, we assume you are familiar with RMI and that there exists an RMI-based application. In this application, a math server accepts two integers, adds them up and returns the sum to the client. In other words, a user starts a midlet and enters two integers to be sent to a servlet, which in turn retrieves the two numbers, gets a reference to a remote object, and invokes a method to add the two integers. The sum is then sent back to the device.

The Math Service

The interface for the math service is shown in CodeSample 4. In this remote interface, an `add` method, which takes in two integers and returns an integer, is declared.

CodeSample 4: Math.java

```
public interface Math extends java.rmi.Remote {
    int add(int a, int b) throws java.rmi.RemoteException;
}
```

The implementation of the `Math` interface, and a simple RMI server is shown in CodeSample 5.

CodeSample 5: MathImpl.java

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class MathImpl extends UnicastRemoteObject implements Math {
    private String name;

    public MathImpl(String s) throws RemoteException {
       super();
       name = s;
    }

    public int add(int a, int b) throws RemoteException {
       return a+b;
    }

    public static void main(String argv[]) {
        System.setSecurityManager(new RMISecurityManager());

        try {
           MathImpl obj = new MathImpl("MathServer");
           Naming.rebind("//machineName/MathServer", obj);
           System.out.println("AddServer bound in registry");
        } catch (Exception e) {
           System.out.println("ArithImpl err: " + e.getMessage());
           e.printStackTrace();
        }
    }
}
```

The MIDlet

The MIDlet, `RmiMidlet`, presents the user with two text fields through which the user can enter two integers. When the user presses the button associated with the `add`, a request is sent to a servlet that retrieves the two integers and then invokes a remote method. The source code of the `RmiMidlet` is shown in CodeSample 6.

CodeSample 6: RmiMidlet.java

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

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

    // email form fields
    TextField number1 = null;
    TextField number2 = null;
    Form form;

    static final Command sumCommand = new Command("add", Command.OK, 2);
    static final Command clearCommand = new Command("clear", Command.STOP, 3);

    String num1;
    String num2;


    public RmiMidlet() {
       display = Display.getDisplay(this);
       number1 = new TextField("Number1:", "", 10, TextField.NUMERIC);
       number2 = new TextField("Number2:", "", 10, TextField.NUMERIC);
       form = new Form("Enter Numbers");
```

```
    }

    public void startApp() throws MIDletStateChangeException {
      form.append(number1);
      form.append(number2);
      form.addCommand(clearCommand);
      form.addCommand(sumCommand);
      form.setCommandListener(this);
      display.setCurrent(form);
    }

    public void pauseApp() {
    }

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

    void invokeServlet(String url) throws IOException {
        HttpConnection c = null;
        InputStream is = null;
        OutputStream os = null;
        StringBuffer b = new StringBuffer();
        TextBox t = null;
        try {
          c = (HttpConnection)Connector.open(url);
          c.setRequestMethod(HttpConnection.POST);
          c.setRequestProperty("IF-Modified-Since", "20 Oct 2001 16:19:14 GMT");
          c.setRequestProperty("User-Agent","Profile/MIDP-1.0 Configuration/CLDC-1.0");
          c.setRequestProperty("Content-Language", "en-CA");
          c.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");

          os = c.openOutputStream();
          // encode data
          os.write(("num1="+num1).getBytes());
          os.write(("&num2="+num2).getBytes());
          os.flush();

          is = c.openDataInputStream();
          int ch;
          while ((ch = is.read()) != -1) {
            b.append((char) ch);
            System.out.print((char)ch);
          }
          t = new TextBox("Result", b.toString(), 1024, 0);
          t.setCommandListener(this);
        } finally {
           if(is!= null) {
              is.close();
           }
           if(os != null) {
              os.close();
           }
           if(c != null) {
              c.close();
           }
        }
        display.setCurrent(t);
    }

   public void commandAction(Command c, Displayable d) {
      String label = c.getLabel();
      if(label.equals("clear")) {
        destroyApp(true);
      } else if (label.equals("add")) {
         num1 = number1.getString();
         num2 = number2.getString();
         try {
           invokeServlet(url);
         }catch(IOException e) {}
      }
   }
}
```

The Servlet

As you can see in the `RmiMidlet`, a `MathServlet` is being invoked. This servlet, which is shown in CodeSample 7, retrieves two integers sent from the device then it gets a reference to the math server, and then invokes a method to add the two integers. The sum is then sent to the device.

CodeSample 7: MathServlet.java

```
import java.rmi.*;
import java.io.*;
import java.text.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
public class MathServlet extends HttpServlet {

   public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/plain");
        PrintWriter out = response.getWriter();

        String s1 = request.getParameter("num1");
        String s2 = request.getParameter("num2");
        int x = Integer.parseInt(s1);
        int y = Integer.parseInt(s2);
        int result = 0;
        try {
          Math obj = (Math)Naming.lookup("//machineName/MathServer");
          result = obj.add(x, y);
        } catch (Exception e) {
          System.out.println("MathServlet exception:"+e.getMessage());
          e.printStackTrace();
        }
        out.println("The sum of "+x+" and "+y+" is: "+result);
    }
}
```

Testing the Remote Math Application

Similar to the email application, this application was tested using the J2ME Wireless Toolkit and Tomcat. I started the `RmiMidlet` and entered two integers as shown in Figure 6.

Figure 6: The RmiMidlet in action

When the button associated with the `add` command is pressed, the two integers are sent to the `MathServlet`. A couple of seconds later, the sum of the two integers was displayed on the device's screen as shown in Figure 7.

Figure 7: Result of remote method invocation

Conclusion

This article presented an overview of the CLDC and MIDP networking mechanisms. The examples throughout this article demonstrate the effort involved in using sockets and RMI (two mechanisms that are not supported by the reference implementation from Sun) from MIDP-enabled devices using.

Some MIDP implementations, such as the one from Motorola, have support for TCP sockets and UDP datagrams and therefore a middleman is not needed. The middleman architecture, however, has its own advantages: (1) Not all devices are IP-enabled. (2) It promotes loose coupling and therefore simplifies interactions between objects. (3) Enables you to use RMI, CORBA, or any distributed technology from a MIDP-enabled device.

For more information

- CLDC
- MIDP
- Sockets Tutorial

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).

Reader Feedback
○ **Excellent** ○ **Good** ○ **Fair** ○ **Poor**
If you have other comments or ideas for future technical tips, please type them here:

Comments:

If you would like a reply to your comment, please submit your email address:
Note: We may not respond to all submitted comments.
Submit »

Back To Top

E-mail this page Printer View