3.3.4 Connection Framework

3.3.4.1 What’s Optional and What’s Not

The CLDC provides a Generic Connection Framework (GCF), which is an extensible framework that can be customized by a J2ME profile to support the necessary networking protocols required by that vertical device category. The MIDP 1.0 specification only required support (in other words, a concrete implementation) for the HTTP protocol. The MIDP 2.0 specification extends the support required for networking protocols to include mandatory support for HTTPS. The MIDP 2.0 specification also states that implementations should (where “should” implies a recommended practice that can be ignored only in exceptional circumstances) provide support for sockets, secure sockets, server sockets and datagrams. Support for serial port access via the CommConnection interface is optional under the MIDP 2.0 specification.

Symbian’s implementation of MIDP 2.0 complies with the specification, providing implementations for all of the above except the optional serial port access. So, Symbian’s MIDP 2.0 currently provides implementations of the following protocols:

- HTTP
- HTTPS
- sockets
- server sockets
- secure sockets
- datagrams.

In the following sections we will explore using these connections in a little more detail.

3.3.4.2 HTTP and HTTPS Support

HTTP connections have been supported since MIDP 1.0. To open an HTTP connection we use the Connector.open() method with a URL of the form www.myserver.com.

So code to open an HttpConnection and obtain an InputStream would look something like this.

```java
try{
    String url = "www.myserver.com";
    HttpConnection conn = (HttpConnection)Connector.open(url);
    InputStream is = conn.openInputStream();
    ...
    conn.close()
} catch(IOException ioe){...}
```
Under the MIDP 2.0 security model, untrusted MIDlets can open an HTTP connection only with explicit user confirmation. Signed MIDlets that require access to an HTTP connection must explicitly request the `javax.microedition.io.Connector.http` permission in the MIDlet-Permissions attribute:

```
MIDlet-Permissions: javax.microedition.io.Connector.http, ...
```

The MIDP 2.0 specification adds the requirement that implementations must support the HTTPS protocol, which implements HTTP over a secure network connection via the Secure Sockets Layer (SSL). Opening an HTTPS connection follows the same pattern as a normal HTTP connection, with the exception that we pass in a connection URL of the form `https://www.mysecureserver.com` and cast the returned instance to an `HttpsConnection` object, as in the following example of code for interrogating a secure server for security information associated with the connection.

```
try{
    String url = "https://www.mysecureserver.com";
    HttpsConnection hc = (HttpsConnection)Connector.open(url);
    SecurityInfo info = hc.getSecurityInfo();
    String protocolName = info.getProtocolName();
    String protocolVersion = info.getProtocolVersion();
    String cipherSuite = info.getCipherSuite();
    Certificate c = info.getServerCertificate();
    String name = c.getIssuer();
    ...
}catch(IOException ioe){...}
```

The MIDP 2.0 specification requires that MIDlets in untrusted MIDlet suites be able to open HTTPS connections with User permission. A signed MIDlet suite which contains MIDlets that open HTTPS connections must explicitly request the `javax.microedition.io.Connector.https` permission in its MIDlet-Permissions attribute:

```
MIDlet-Permissions: javax.microedition.io.Connector.https, ...
```

### 3.3.4.3 Socket and Server Socket Support

Although support for socket connections was an optional part of the MIDP 1.0 specification, MIDP 2.0 now makes support for socket connections a recommended practice. Socket connections come in two forms: client connections in which a socket connection is opened to another host; and server connections in which the system listens on a particular port for
incoming connections from other hosts. The connections are specified using Universal Resource Identifiers (URI).

You should be familiar with the syntax of a URI from Web browsing. They have the format `<string1>://<string2>` where `<string1>` identifies the communication protocol to be used (e.g. `http`) and `<string2>` provides specific details about the connection. The protocol may be one of those supported by the Generic Connection Framework (see Section 2.1.3.2).

To open a client socket connection to another host we pass a URI of the following form to the connector’s `open()` method:

```
socket://www.symbian.com:80
```

The host may be specified as a fully qualified hostname or IPv4 address and the port number refers to the connection endpoint on the remote peer. Some sample code is shown below:

```
SocketConnection sc = null;
OutputStream out = null;
try{
    sc = (SocketConnection)Connector.open("socket://localhost:79");
    ...
    out = c.openOutputStream();
    ...
}catch(IOException ioe){...}
```

A server socket connection is used for listening for inbound socket connections. To obtain a server socket connection we can pass a URI in either of the following forms to the connector’s `open()` method:

```
socket://:79
socket://
```

In the first case the system listens for incoming connections on port 79 (of the local host). In the latter case, the system allocates an available port for the incoming connections.

```
ServerSocketConnection ssc = null;
InputStream is = null;
try{
    ssc = (ServerSocketConnection)Connector.open("socket://:1234");
    SocketConnection sc = (SocketConnection)ssc.acceptAndOpen();
    ...
    is = sc.openInputStream();
    ...
}catch(IOException ioe){...}
```
The ServerSocketConnection interface extends the StreamConnectionNotifier interface. To obtain a connection object for an incoming connection the acceptAndOpen() method must be called on the ServerSocketConnection instance. An inbound socket connection results in the call to the acceptAndOpen() method, returning a StreamConnection object which can be cast to a SocketConnection as desired.

The SocketConnection interface defines several useful methods including:

```java
public void setSocketOption(byte option, int value)
```

This allows the developer to set several socket options using the following public static final byte constants defined in SocketConnection:

- **DELAY**
  A value of zero disables the use of Nagle’s algorithm – written data is not buffered pending acknowledgement of previously written data. This may be desirable when sending and receiving small packets of data, for instance, in a peer-to-peer messenger application.

- **LINGER**
  A non-zero value represents the interval in seconds that the system will continue to try to process queued data after the close() method has been called. After the interval has elapsed the connection will be forcefully closed with a TCP RST. A value of zero disables linger on close.

- **KEEPALIVE**
  If enabled (by a non-zero value), a keepalive probe will be sent to the remote peer after an implementation-specific time interval (the default is two hours) if no other data has been sent or received on the socket during that time interval. The purpose of the probe is to detect if the peer has become unreachable. The peer can respond in one of three ways: a TCP ACK response indicating all is well – no action is taken; a TCP RST response indicating the peer has crashed and been rebooted in which case the socket is closed; no response from the remote peer – the socket is closed. A value of zero disables this feature.

- **RCVBUF**
  This option is used by the platform’s networking code as a hint for the size at which to set the underlying network I/O receiving buffer.

- **SNDBUF**
  This option is used by the platform’s networking code as a hint for the size to set the underlying network I/O sending buffer.
A signed MIDlet suite which contains MIDlets which open socket connections must explicitly request the `javax.microedition.io.Connector.socket` permission (needed to open client connections) and if required the `javax.microedition.io.Connector.serversocket` permission (needed to open server connections), in its MIDlet-Permissions attribute, for example:

```
MIDlet-Permissions: javax.microedition.io.Connector.socket, ...
```

or:

```
MIDlet-Permissions: javax.microedition.io.Connector.socket,
javax.microedition.io.Connector.serversocket, ...
```

If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, these permissions, then the MIDlet suite will be installed and the MIDlets it contains will be able to open socket connections, either automatically or with user permission, depending upon the security policy in effect on the device for the protection domain to which the MIDlet suite has been bound.

Whether MIDlets in untrusted MIDlet suites can open socket connections depends on the security policy relating to the untrusted domain in force on the device.

### 3.3.4.4 Secure Socket Support

Secure socket connections are client socket connections over SSL. To open a secure socket connection we pass in a hostname (or IPv4 address) and port number to the connector’s `open()` method using the following URI syntax:

```
ssl://hostname:port
```

We can then use the secure socket connection in the same manner as a normal socket connection, for example:

```
try{
    SecureConnection sc = (SecureConnection)
        Connector.open("ssl://www.secureserver.com:443");
    ...
    OutputStream out = sc.openOutputStream();
    ...
    InputStream in = sc.openInputStream();
    ...
}catch(IOException ioe){...}
```
A signed MIDlet suite that contains MIDlets which open secure connections must explicitly request the `javax.microedition.io.Connector.ssl` permission in its MIDlet-Permissions attribute, for example:

```
MIDlet-Permissions: javax.microedition.io.Connector.ssl, ...
```

If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, this permission, the MIDlet suite can be installed and the MIDlets it contains will be able to open secure connections, either automatically or with user permission, depending on the security policy in effect.

Whether MIDlets in untrusted MIDlet suites can open secure connections depends on the permissions granted in the untrusted protection domain.

### 3.3.4.5 Datagram Support

Symbian’s MIDP 2.0 implementation includes support for sending and receiving UDP datagrams. A datagram connection can be opened in client or server mode. Client mode is for sending datagrams to a remote device. To open a client mode datagram connection we use the following URI format:

```
datagram://localhost:1234
```

Here the port number indicates the port on the target device to which the datagram will be sent. Sample code for sending a datagram is shown below:

```
String message = "Hello!";
byte[] payload = message.toString();
try{
    UDPDatagramConnection conn = null;
    conn = (UDPDatagramConnection)
    Connector.open("datagram://localhost:1234");
    Datagram datagram = conn.newDatagram(payload, payload.length);
    conn.send(datagram);
}catch(IOException ioe){...}
```

Server mode connections are for receiving (and replying to) incoming datagrams. To open a datagram connection in server mode we use a URI of the following form:

```
datagram://:1234
```
The port number in this case refers to the port on which the local device is listening for incoming datagrams. Sample code for receiving incoming datagrams is given below:

```java
try{
    UDPDatagramConnection dconn = null;
    dconn = (UDPDatagramConnection)Connector.open("datagram://:1234");
    Datagram dg = dconn.newDatagram(300);
    while(true){
        dconn.receive(dg);
        byte[] data = dg.getData();
        ...
    }
}catch(IOException ioe){...}
```

A signed MIDlet suite which contains MIDlets that open datagram connections must explicitly request the `javax.microedition.io.Connector.datagram` permission (needed to open client connections) and the `javax.microedition.io.Connector.datagramreceiver` permission (needed to open server connections) in its `MIDlet-Permissions` attribute, for example:

```
MIDlet-Permissions: javax.microedition.io.Connector.datagram, ...
```

or:

```
MIDlet-Permissions: javax.microedition.io.Connector.datagramreceiver,
...
```

or:

```
MIDlet-Permissions: javax.microedition.io.Connector.datagram,
javax.microedition.io.Connector.datagramreceiver, ...
```

If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, the requested permissions, the MIDlet suite can be installed and the MIDlets it contains will be able to open datagram connections, either automatically or with user permission, depending on the security policy in effect.

Whether MIDlets in untrusted MIDlet suites can open datagram connections depends on permissions granted to MIDlet suites bound to the untrusted protection domain.

### 3.3.4.6 Security Policy for Network Connections

The connections discussed above are part of the Net Access function group (see the `Recommended Security Policy for GSM/UMTS Compliant...`
Devices addendum to the MIDP 2.0 specification). On the Nokia 6600 and Sony Ericsson P900/P908, MIDlets in untrusted MIDlet suites can access the Net Access function group with User permission (explicit confirmation required from the user). On the Sony Ericsson P900/P908, the default User permission is set to session (and is not customizable by the user). On the Nokia 6600, the default User permission is set to oneshot, but can be changed by the user to session or disallowed.

The Sony Ericsson P900/P908 supports the trusted protection domain on Organizer firmware versions R2B02 or later. The security policy in effect for MIDlets in MIDlet suites bound to the trusted protection domain on the P900/P908 allows automatic access (Allowed permission) to the Net Access function group connections. At the time of writing, the available firmware release (3.42.1) on the Nokia 6600 only supported the untrusted domain, although future releases will add support for trusted protection domains.

3.3.4.7 Practical Networking using Wireless Networks

In the spirit of providing practical information, we shall now digress slightly into a discussion of networking on wireless data networks. The most common GSM networks at the time of writing are 2.5 G General Packet Radio Service (GPRS) networks. GPRS networks can be regarded as a private sub-network behind a gateway to the Internet. All current GPRS network providers operate their consumer networks behind Network Address Translation (NAT) gateways and dynamically allocate private IP addresses to mobile terminals on each PDP activation (data session).

This has important consequences for application developers wishing to use wireless networking. One consequence is that mobile terminals on a GPRS network typically are unable to receive inbound connections since their private IP addresses are not visible on the Internet. Another issue relates to connection-less communications protocols such as UDP. When a terminal on a GPRS network sends a UDP packet to a remote host on the Internet, the sender address is stripped out of the packet and replaced with the IP address of the gateway and a port number representing the terminal data session. How long this session information remains valid (enabling the remote host to reply to the sender) depends on the NAT gateway. After a limited period of time the gateway will re-allocate that port to another GPRS terminal. Some NAT policies allow for the session information (and thus the allocated port) to remain associated with the GPRS terminal as long as traffic flows through it. Such inactivity timeouts though, vary quite significantly between operators.

The most effective way of avoiding complications arising out of operating behind NAT gateways is for developers to use TCP-based protocols such as HTTP. As long as there is an active TCP session in place, the
gateway port will remain allocated to that GPRS terminal by the NAT gateway, enabling two-way traffic between the GPRS terminal and the remote device.

### 3.3.4.8 Socket Demo MIDlet

We will finish this section with a simple example using TCP sockets to interrogate a web browser. The Socket Demo MIDlet sends an HTTP GET request to a web server over a client socket connection and then reads and displays the response. The Socket Demo MIDlet consists of two classes, SocketMIDlet extending MIDlet and the ClientConnection class. The source code for the SocketMIDlet class is shown below.

```java
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class SocketMIDlet extends MIDlet implements CommandListener {
    private final static String defaultURL =
        "socket://www.symbian.com:80";
    private Command exitCommand, sendCommand;
    private Display display;
    public TextBox textBox;

    public SocketMIDlet() {
        display = Display.getDisplay(this);
        exitCommand = new Command("Exit", Command.EXIT, 2);
        sendCommand = new Command("Send request", Command.SCREEN, 1);
    }

    public void startApp() {
        textBox = new TextBox("Sockets Demo", defaultURL, 256,
            TextField.ANY);
        textBox.addCommand(exitCommand);
        textBox.addCommand(sendCommand);
        textBox.setCommandListener(this);
        display.setCurrent(textBox);
    }

    public void commandAction(Command c, Displayable s) {
        if (c == exitCommand) {
            notifyDestroyed();
        } else if (c == sendCommand) {
            ClientConnection socketConn = new ClientConnection(this);
            socketConn.sendMessage(textBox.getString());
            textBox.removeCommand(sendCommand);
        }
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }
}
```
SocketMIDlet sets up the UI and responds to the “Send request” Command by creating an instance of ClientConnection and invoking its sendMessage() method, passing in a String representing the URL of the required web server.

The main work is done in the ClientConnection class:

```java
import javax.microedition.io.*;
import java.io.*;
public class ClientConnection extends Thread {
    private final static String line1 = "GET /index.html\r\n";
    private final static String line2 = "Accept: */*\r\n";
    private final static String line3 = "Accept-Language: en-us\r\n";
    private final static String line4 = "Accept-Encoding: gzip, deflate\r\n";
    private final static String line5 = "User-Agent: Mozilla/4.0 (Compatible; MSIE 5.01; Windows NT)\r\n"
    private SocketMIDlet sM = null;
    private String url = null;
    private String request = null;

    public ClientConnection(SocketMIDlet sM) {
        this.sM = sM;
    } // Constructor

    public void sendMessage(String url) {
        this.url = url;
        String host = url.substring(url.lastIndexOf('/') + 1);
        System.out.println("host is " + host);
        String hostLine = "Host: " + host + "\r\n";
        request = line1 + line2 + line3 + line4 + line5 + hostLine;
        start();
    } // sendMessage

    public void run() {
        try{
            SocketConnection conn =
                (SocketConnection)Connector.open(url);
            DataOutputStream out = conn.openDataOutputStream();
            byte[] buf = request.getBytes();
            out.write(buf);
            out.flush();
            out.close();
            sM.textBox.insert("Finished request!\n" +
                "Receiving response...\n", sM.textBox.size());

            DataInputStream in = conn.openDataInputStream();
            int ch;
            while ( (ch = in.read()) != -1 &&
                sM.textBox.size() < sM.textBox.getMaxSize()) {
                String str = new Character((char) ch).toString();
                try {
                    sM.textBox.insert(str, sM.textBox.size());
                }catch(Exception e) {
                    e.printStackTrace();
                }
            }
        }catch(Exception e) {
            e.printStackTrace();
        }
    } // run
}
```
The `url` parameter of the `sendMessage()` method has the following form:

```
socket://www.symbian.com:80
```

The `sendMessage()` method creates a GET request and then starts a new Thread to create the connection, send the request and read the response. Let us look at the contents of the thread’s `run()` method in more detail.

```
SocketConnection conn = (SocketConnection)Connector.open(url);
DataOutputStream out = conn.openDataOutputStream();
byte[] buf = request.getBytes();
out.write(buf);
out.flush();
out.close();
```

A SocketConnection is opened using a URI of the form `socket://hostname:port` and the returned SocketConnection object is used to get a DataOutputStream. After converting the request to a byte array, this is written to the DataOutputStream using the `write()` method. The `flush()` method is then called on the DataOutputStream to ensure any buffered data is written to the connection endpoint. This last step is essential. Symbian’s implementation of OutputStream buffers data internally and only writes it to the connection endpoint when the buffer is full, or when the buffer is flushed. Failing to call `flush()` may result in data never being written to the connection endpoint. Once we have finished with the OutputStream we can close it.

Having written the request we are now ready to read the response. We use our SocketConnection to get a DataInputStream and use the `read()` method to read from it in the standard manner.

```
DataInputStream in = conn.openDataInputStream();
int ch;
while ( (ch = in.read()) != -1 &&
     sM.textBox.size() < sM.textBox.getMaxSize()) {
    ...
}
```
The response from the web server should be a stream of raw HTML. We read the stream until our MIDlet’s TextBox is full and then close the connection (reading the response in its entirety is likely to be a lengthy process for most web sites!).

The screenshots in Figure 3.6 show the Socket Demo MIDlet running on a Nokia 6600.

Note that the purpose of this sample code is to demonstrate how to use client TCP socket connections. Normally, to make requests to a HTTP server we would use an HttpConnection or HttpsConnection. Also, under the JT WI security policy for GSM/UMTS compliant devices, the implementation of SocketConnection using TCP sockets must throw a SecurityException when an untrusted MIDlet suite attempts to connect on ports 80, 8080 (HTTP) and 443 (HTTPS). Hence the above code is not future-proof for untrusted MIDlet suites.

### 3.3.5 The Push Registry

#### 3.3.5.1 Introduction

One of the exciting new additions to MIDP 2.0 is the Push Registry API, which allows MIDlets to be launched in response to incoming network connections. Many applications, particularly messaging applications, need to be continuously listening for incoming messages. Previously, to achieve this a Java application would have had to be continually running in the background. Although the listening Java application may itself be small, it would still require an instance of the virtual machine to be running, thus appropriating some of the mobile phone’s scarce resources. The JSR 118 recognized the need for an alternative, more resource-effective solution for MIDP 2.0 and so introduced the push registry.
### 3.3.5.2 Using the Push Registry

The Push Registry API is encapsulated in the `javax.microedition.io.PushRegistry` class. The push registry maintains a list of inbound connections that have been previously registered by installed MIDlets. A MIDlet registers an incoming connection with the push registry either statically at installation via an entry in the JAD file or dynamically (programmatically) via the `registerConnection()` method.

When a MIDlet is running, it handles all the incoming connections (whether registered with the push registry or not). If, however, the MIDlet is not running, the AMS listens for registered incoming connections and launches the MIDlet in response to an incoming connection previously registered by that MIDlet, by invoking the `startApp()` method. The AMS then hands off the connection to the MIDlet which is then responsible for opening the appropriate connection and handling the I/O.

In the case of static registration, the MIDlet registers its interest in incoming connections in the JAD file, in the following format:

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

- The `<ConnectionURL>` field specifies the protocol and port for the connection end point in the same URI syntax used as the argument to the `Connector.open()` method that is used by the MIDlet to process the incoming connection. Examples of `<ConnectionURL>` entries might be:
  ```
sms://:1234
socket://:1234
```

- The `<MIDletClassName>` field contains the package-qualified name of the class that extends `javax.microedition.midlet.MIDlet`. This would be the name of the MIDlet class as listed in the application descriptor or manifest file under the `MIDlet-<n>` entry.

- 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. For a server socket connection endpoint the `<AllowedSender>` entry would be an IP address (note in both cases that the sender port number is not included in the filter). The `<AllowedSender>` syntax supports two wildcard characters: `*` matches any string including an empty string and `?` matches any character. Hence the following would be valid entries for the `<AllowedSender>` field:

  ```
  *
  129.70.40.*
  129.70.40.23?
  ```
The first entry indicates any IP address, the second entry allows the last three digits of the IP address to take any value, while the last entry allows only the last digit to have any value.

So the full entry for the MIDlet-Push-<n> attribute in a JAD file may look something like this:

```
MIDlet-Push-1: sms://:1234, com.symbian.devnet.ChatMIDlet, *
```

If the request for a static connection registration can not be fulfilled then the AMS must not install the MIDlet. Examples of when a registration request might fail include the requested protocol not being supported by the device, or the requested port number being already allocated to another application.

To register a dynamic connection with the AMS we use the static registerConnection() method of PushRegistry:

```
PushRegistry.registerConnection("sms://:1234", 
   "com.symbian.devnet.ChatMIDlet", 
   "*");
```

The arguments take precisely the same format as those used to make up the MIDlet-Push-<n> entry in a JAD or manifest. Upon registration, the dynamic connection behaves in an identical manner to a static connection registered via the application descriptor.

To un-register a dynamic connection the static boolean unregisterConnection() method of PushRegistry is used:

```
boolean result = PushRegistry.unregisterConnection("sms://:1234");
```

If the dynamic connection was successfully unregistered a value of true is returned.

The AMS will respond to input activity on a registered connection by launching the corresponding MIDlet (assuming that the MIDlet 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. For a MIDlet registered for incoming SMS messages, the startApp() method might look something like this:

```
public void startApp() {
   // List of active connections.
   String[] connections = PushRegistry.listConnections(true);
```
for (int i=0; i < connections.length; i++) {
    if(connections[i].equals("sms://:1234")) {
        new Thread()
        {
            public void run()
            {
                Receiver.openReceiver();
            }
        }.start();
    }
}

One other use of the push registry should be mentioned before we leave this topic. The PushRegistry class provides the registerAlarm() method:

public static long registerAlarm(String midlet, long time)

This allows a running MIDlet to register itself or another MIDlet in the same suite for activation at a given time. The midlet argument is the class name of the MIDlet to be launched at the time specified by the time argument. The launch time is specified in milliseconds since January 1, 1970, 00:00:00 GMT. The push registry may contain only one outstanding activation time entry per MIDlet in each installed MIDlet suite. If a previous activation entry is registered, it will be replaced by the current invocation and the previous value returned. If no previous wakeup time has been set, a zero is returned.

3.3.5.3 The Push Registry and the Security Model

The PushRegistry is a protected API and, as such, a signed MIDlet suite which registers connections statically or contains MIDlets which register connections and/or alarms, must explicitly request the javax.microedition.io.PushRegistry permission in its MIDlet-Permissions attribute, for example:

MIDlet-Permissions: javax.microedition.io.PushRegistry, ...

Note that a signed MIDlet suite must also explicitly request the permissions necessary to open the connection types of any connections it wishes to register either statically or dynamically. If the protection domain to which the signed MIDlet suite would be bound grants, or potentially grants, the requested permission, the MIDlet suite can be installed and the MIDlets it contains will be able to register and deregister connections, and register alarms, either automatically, or with user permission, depending on the security policy in effect.
Untrusted MIDlets do not require a MIDlet-Permissions entry. Whether access is granted to the Push Registry API will depend on the security policy for the untrusted protection domain in effect on the device.

On the Sony Ericsson P900/P908 and Nokia 6600, MIDlets in untrusted MIDlet suites can use the Push Registry APIs (Application Auto-Invocation function group) with user permission. On both the 6600 and the P900/P908, the default user permission is set to session. On the Nokia 6600, the default value can be changed by the user to oneshot or disallowed. For the Sony Ericsson P900/P908, the security policy in effect for MIDlets in MIDlet suites bound to the trusted protection domain allows automatic access to the Push Registry API.

3.3.5.4 Symbian’s Implementation

At the time of writing, the current version of Symbian OS, Version 7.0s, supports the following connection types in its implementation of the MIDP 2.0 push architecture:

- Server socket
- Datagram
- SMS.

In Symbian’s implementation, all connections that can be registered as push connections are managed by the system AMS, even if they are not requested to be push-enabled by an application. In the case of server connections that spawn off a connected stream due to an incoming connection, the stream connections are also maintained through the system AMS.

Future releases of Symbian OS are likely to increase the types of connections supported by the push architecture to include Bluetooth, L2CAP and RFCOMM connections.

3.3.6 Additions to the LCDUI

3.3.6.1 A Quick Tour

The MIDP 2.0 specification introduces a number of new features to the LCDUI toolkit which are designed to give developers more control over their application’s user interface. In this section we briefly look at some of them.

Display

MIDP 2.0 introduces two useful new methods to the Display class which allow MIDlets to control the screen backlight and the vibration of the phone: