



The network connection

- 13.1 About the Generic Connection Framework 366
- 13.2 Using the Generic Connection Framework 372
- 13.3 HTTP-based connections 372
- 13.4 Socket-based connections 377
- 13.5 Datagram-based connections 394
- 13.6 Summary 406

One of the most critical aspects of J2ME is network connectivity. Although J2ME devices can be useful when they are not connected to a network, the ability to make a network connection provides a means to tap into the powerful resources available on a network. Even more significant are the emerging capabilities to establish a wireless network connection. Many J2ME devices support this capability, which opens the door to providing features on devices that go beyond sending and receiving email, such as extending the enterprise into the mobile space. J2ME applications, in this regard, become more than simple communication devices. They become another client capable of interfacing with the enterprise systems, databases, corporate intranets and the Internet. An insurance agent could file and adjust claims interactively while talking to customers. Medical staff could interact with the hospital and clinical systems at the point of care. Inspectors could file reports on site. Salespeople could submit orders, check inventory, and calculate deals in the field. Schedules could be dynamically updated and adjusted for mobile workers.

Table 13.1 GCF interfaces

GCF Interface	Purpose
Connection	The most basic type of connection in the GCF. All other connection types extend Connection.
ContentConnection	Manages a connection, such as HTTP, for passing content, such as HTML or XML. Provides basic methods for inspecting the content length, encoding and type of content.
Datagram	Acts as a container for the data passed on a Datagram Connection.
DatagramConnection	Manages a datagram connection.
InputConnection	Manages an input stream-based connection.
OutputConnection	Manages an output stream-based connection.
StreamConnection	Manages the capabilities of a stream. Combines the methods of both InputConnection and OutputConnection.
StreamConnectionNotifier	Listens to a specific port and creates a StreamConnection as soon as activity on the port is detected.

While mobile applications are nothing new to the technology market, J2ME provides the ability for organizations with a commitment to Java to easily move into the mobile space. J2ME also makes it possible to run the same application on multiple devices, providing flexibility among vendors. In cases where applications are publicly released, the number of devices on which the application can run becomes an important selling point.

The network capabilities of J2ME complements other emerging technologies such as Bluetooth, which provides wireless local area network capabilities through radio frequency communication, and Jini, which provides spontaneous networking capabilities. Using Jini and Bluetooth, a J2ME device could automatically register itself on a wireless local area network as the user enters a room and unregister the user when he leaves the room. While connected to the wireless network, the user would have access to a number of network services such as printers, fax machines, email, network file systems, databases, enterprise systems, and other devices currently registered with the network. Which services a user has available depends on who the user is, of course, and how the user or device is presented to the network.

Since network connectivity is so vital to J2ME it is important that the architecture be extendible to many different protocols while allowing applications to be portable across many devices. The piece of software within the J2ME architecture that addresses network connectivity is called the Generic Connection Framework (GCF).

13.1 ABOUT THE GENERIC CONNECTION FRAMEWORK

The Generic Connection Framework provides the foundation for all network communications within the J2ME architecture. Within the configuration layer the Generic Connection Framework interface is defined along with a number of basic interfaces. The Generic Connection Framework provides no protocol implementations.

The vendors supplying the profile must implement the necessary Generic Connection Framework interfaces.

The Generic Connection Framework resides in the `javax.microedition.io` package and consists of:

- one class (`Connector`)
- one exception (`ConnectionNotFoundException`)
- eight interfaces (table 13.1)

The relationships of these interfaces are depicted in figure 13.1.

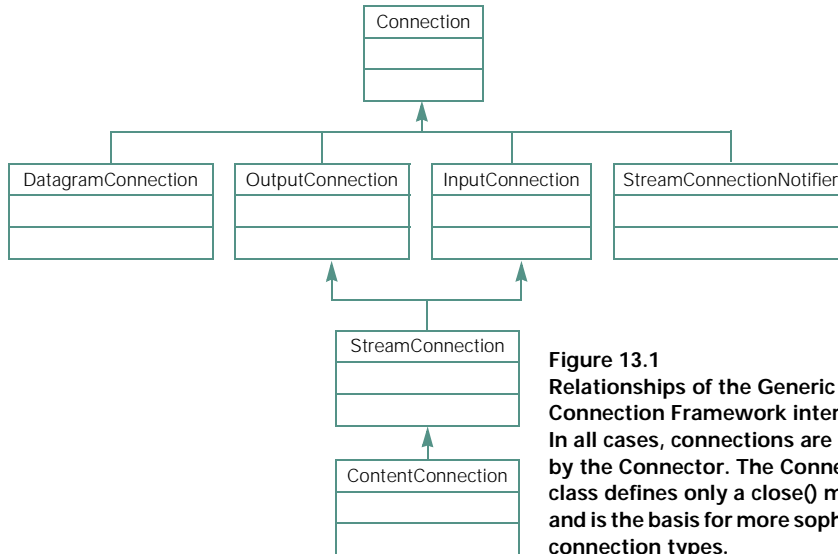


Figure 13.1
Relationships of the Generic Connection Framework interfaces. In all cases, connections are opened by the `Connector`. The `Connection` class defines only a `close()` method and is the basis for more sophisticated connection types.

13.1.1 Where the Generic Connection Framework lives

The Generic Connection Framework is defined at the configuration layer of the J2ME architecture. By implementing the framework at this level, the same `Connector` and interfaces are available across all the profiles.

Both the CDC and the CLDC support the Generic Connection Framework. Due to the nested arrangement of configurations, the connection interfaces are provided throughout the J2ME architecture. This increases the compatibility across configurations.

NOTE By definition, all J2ME configurations must adhere to a nested relationship. In other words, the CLDC fits completely inside the CDC. There are no classes, methods or other functionality in the CLDC that are not also in the CDC.

13.1.2 Working with the Connector class

The `Connector` class is used to create instances of a connection protocol using one of `Connector`'s static methods. The instance returned is an implementation supporting the `Connection` interface or one of its descendents.

The `Connector` class is not designed to be instantiated. It is simply used to create instances of a connection protocol. All of the methods `Connector` defines are static and serve this purpose.

The `Connector` defines three variations of an `open()` that return a `Connection` instance. The `Connector` also defines methods that return input and output streams. These methods will be discussed later in this chapter. For now, we will concentrate on the `open()` method.

The `open()` method returns an instance of type of `Connection`. However, the instance returned is most likely to be a subclass of `Connection` that is more sophisticated. It is the responsibility of the calling application to know what class to expect and to coerce the returned instance to the correct object type as necessary. The `open` method has the following signatures:

- `open(String name)`
- `open(String name, int mode)`
- `open(String name, int mode, boolean timeouts)`

The name is essentially a URI and is composed of three parts: a scheme, an address, and a parameter list. The general form of the name parameter is as follows:

```
<scheme>:<address>;<parameters>
```

The scheme identifies how the connection is made (socket, http, file, datagram, etc.). The address identifies what to connect to (www.ctimn.com, myfile.txt, etc.) and the parameters identify other information that is required by the protocol to establish a connection such as a connection speed. The parameters, when needed, are specified as name=value pairs. Some examples of the name URI are shown in the following list. Note that in some cases the parameter is not necessary and thus the “;” is not always present:

- `http://www.ctimn.com:8080`
- `socket://localhost:8080`
- `file:c:/myfile.txt` (Windows only)
- `file:/myfile.txt` (Unix)
- `datagram://127.0.0.1:8099`
- `comm:0;baudrate=9600`

The `mode` parameter allows the connection to be established in various access modes, such as read-only, read-write and write-only. These modes are defined by the `Connector` constants `READ`, `READ_WRITE`, and `WRITE`.

The `timeouts` parameter is a flag indicating whether or not the connection should throw an `InterruptedIOException` if a timeout occurs. The application is then responsible for handling this exception gracefully.

The `Connector` class is the only mechanism for creating the various types of connections using the Generic Connection Framework. Specific protocol implementations are designed to be created directly.

The other methods defined by the `Connector` interface are:

- `openInputStream()`
- `openOutputStream()`
- `openDataInputStream()`
- `openDataOutputStream()`

These are convenient methods for creating different types of input and output streams at the same time the `Connection` is created. In most cases, applications are not concerned with the `Connection` instance itself, but rather the stream that can be read from or written to. By using one of these four methods, the application can obtain the stream directly, without needing to be concerned about the connection instance. The following example illustrates the difference between the two ways of obtaining streams:

```
try {
    OutputConnection connection =
        (OutputConnection)Connector.open("socket://127.0.0.1:8888");
    OutputStream os = connection.openOutputStream();
    os.close();
    connection.close();
} catch (IOException x) {
    //Handle the exception
}
```

This first way to obtain an `OutputStream` is rather verbose and requires us to deal with the `Connection` simply for the purpose of calling the `openOutputStream()` method. We are also forced to cast the return type to `OutputConnection`. If you do not need to interact with the `Connection` instance itself, you can abbreviate how a stream is obtained.

```
try {
    OutputStream os =
        Connector.openOutputStream("socket://127.0.0.1:8888");
    os.close();
} catch (IOException x) {
    //Handle Exception
}
```

This second way of obtaining the `OutputStream` connection is much more concise and eliminates lines of code that deal directly with the `Connection` instance. Furthermore, there is no coercing of the return type on our part. However, there is one

troubling aspect that comes into the picture when using `openOutputStream()`: Who closes the connection? In this case, the connection has already been closed by the `openOutputStream()` method when the stream is returned. A connection is established just long enough to obtain an output stream. Once the stream has been obtained, the connection can be closed. It is important to understand that this works for stream-based connections only. The connection must remain open for some connection types, such as `Datagrams`, that rely more heavily on the underlying connection.

NOTE In the case where a stream is obtained directly, using `openInputStream()` or `openOutputStream()`, the connection `close()` is called immediately after the stream is obtained. However, the actual connection remains open until all the streams are closed as well. This is handled internally by the `Connection`, using a counter to track the number of opens and closes performed. When `open()` is called, the counter is incremented. When `close()` is called the counter is decremented. When the number of closes returns to zero, the connection is actually closed. This is why invoking `close()` on the `Connection` can take place without affecting the streams and still allow the connection resources to be cleaned up properly.

13.1.3 The Connector is a factory

The concept employed by `Connector` for creating connection protocol instances is often referred to as a factory. A factory is a class that has the sole purpose of creating and possibly configuring a set of classes supporting a common interface. Factories provide the ability to return different implementations of an interface while hiding these details from the application code. The actual implementation returned depends on what parameters are passed into the static method (an `open` method in this case) and possibly the state of the system. The factory then deciphers the parameters and system state and determines which class to create. The object created must implement the interface specified by the static method's return type (`Connection`, in this case). However, the interface does not need to be directly supported. For example `DatagramConnection` subclasses `Connection` and is therefore a `Connection` as well.

Factories provide a level of indirection or decoupling that allows the implementation of the interface to vary somewhat independently of the class using the interface. Put more simply, the class using `Connector` does not need to know about the actual (concrete) class that is created. This example of loose coupling is an extremely important aspect of the Generic Connection Framework because it provides flexibility and extendibility.

13.1.4 How the Connector finds the correct class

When the URI (name) is passed to the `Connector.open()` method, the `Connector` parses the URI into its various parts `<scheme>:<address>;<parameters>`. The key piece of information that the `Connector` is looking for at this point is the `scheme`. It is the `scheme`, in combination with other information such as the root package name and a platform identifier that allows the `Connector` to

determine the appropriate `Connection` implementation to create. Once this information is determined, a fully qualified class name is concatenated.

The root package and platform information are system properties identified by `microedition.protocolpath` and `microedition.platform`, respectively. The values of these properties are obtained using the `System` class.

```
String rootPackage = System.getProperty("microedition.protocolpath");
String platform = System.getProperty("microedition.platform");
```

By design, the Generic Connection Framework distinguishes different protocol implementations by package name rather than class name. This is necessary since every protocol implementation is written in a class named `Protocol.java`. Keeping the class name the same relieves the `Connector` class from having to know the names of each implemented class. The classes are differentiated by the location in which they reside. For example, a socket protocol could be defined by the class `com.sun.cldc.io.j2me.socket.Protocol.class` and an http protocol could be defined by the class `com.sun.cldc.io.j2me.http.Protocol.class`. Even though the names of the classes are identical, the full qualification (package name and class name) of the class allows the two implementations to be distinguished from one another. In this example, the root package is `com.sun.cldc.io` and the platform is “j2me”.

In the case of the CLDC reference implementation, the fully qualified class name for a protocol is constructed as follows using the root package name + platform + protocol name + `Protocol`, or more specifically `com.sun.cldc.io.[j2se, j2me, palm].[socket, datagram, http].Protocol`. The following example illustrates the process for creating a datagram protocol instance:

```
DatagramConnection connection =
    (DatagramConnection)Connector.open("datagram://127.0.0.1:9090");
```

The `Connector` extracts the scheme “datagram” and obtains the platform from the `System` properties. In this case, we will assume the platform is j2me. The fully-qualified class name is `com.sun.cldc.io.j2me.datagram.Protocol`. The `Connector` then loads this class into the virtual machine and creates an instance using statements similar to the following.

```
Class c = Class.forName("com.sun.cldc.io.j2me.datagram.Protocol");
Connection connection = (Connection)c.newInstance();
```

Once the `Protocol` instance is created the `open` method of the actual `Protocol` class is called to configure the instance. The `Protocol`’s `open` method returns the configured instance to the caller. The caller then coerces the instance to the expected connection type, which is `DatagramConnection` in this case.

It is important to note that the protocol implementations do not reside in the `java.*` or `javax.*` package. This is due to the fact that protocol implementations, such as HTTP, are the responsibility of the profile implementer, not the creators of J2ME. Furthermore, protocols must be implemented differently for specific platforms,

such as Palm OS or a Motorola phone. In many cases native calls into the underlying device APIs are required. Because of this device dependency of protocol implementations, the Generic Connection Framework does not provide any specific protocol implementations.

The primary goal of the Generic Connection Framework is to generalize how connectivity is handled so that it is extensible and coherent.

13.2 USING THE GENERIC CONNECTION FRAMEWORK

In the sections that follow, various protocols supported by the Generic Connection Framework are examined using examples for each protocol. The example will demonstrate the ability to send and receive messages over a network connection. For simplicity, and ease of learning, the same implementation will be provided for HTTP, socket and datagram connections.

The Generic Connection Framework is available in both the CDC and the CLDC configurations. For these examples, the MIDP will be used, which uses the CLDC at the configuration. Since the CLDC is completely nested inside the CDC, the GCF functionality in these examples will work for both configurations.

We begin by examining HTTP. This example application will be expanded to illustrate sockets and datagrams later on in the chapter.

13.3 HTTP-BASED CONNECTIONS

To establish HTTP `GET` connections, when you are not interested in HTTP-specific information, such as HTTP header information, the `GCF ContentConnection` interface can be used. This example shows how a `MIDlet` can be used to read a page of HTML from a website and display the title of the document in a `Form`.

13.3.1 Establishing a connection

The `ContentConnection` interface enables connections to deal with rich data exchanges, such as HTTP. This type of connection extends the `StreamConnection` interface and defines methods that allow an application to determine the type of connection, the length of content and the encoding used through the following methods:

- `String getEncoding()`
- `long getLength()`
- `String getType()`

In most cases, `ContentConnection` is not used directly but serves as a base interface for a protocol-specific connection types, such as an `HttpConnection` interface.

The following example shows how to use `ContentConnection` with HTTP as the underlying protocol:

```
ContentConnection connection = (ContentConnection) Connector.open(
    "http://www.catapult-technologies.com/ctimain.htm", Connector.READ);
```

In this example we want to read a page of HTML at a specified URL (www.catapult-technologies.com). The connection takes place over the default HTTP port, port 80. Since this is the default port it does not need to be specified. Since this example only reads the page, the connection is opened in read-only mode.

Once the connection is established, we obtain the input stream:

```
InputStream is = connection.openDataInputStream();
```

In this particular example we could have simply obtained an input stream without creating the connection directly, using the `openDataInputStream()` method of the `Connector`. However, the methods stated earlier that provide content length, content type and the encoding reside on the `ContentConnection` instance. Therefore it is necessary to hang onto the connection instance beyond the creation of the input stream.

The `ContentConnection` methods can be used as follows to obtain information about the connection:

```
System.out.println("encoding: " + connection.getEncoding());
System.out.println("length: " + connection.getLength());
System.out.println("type: " + connection.getType());
```

13.3.2 Using the connection

Once a connection is established it can be used to retrieve data. The following example illustrates how to use the content connection to read a page of HTML from a network connection. Since the amount of data retrieved is substantial in this case, our application will parse out only a meaningful portion of the data returned to display on the cell phone emulator. The entire contents of the read, however, will be displayed to the console so the progress can be monitored. The class we will create is named `MsgClient` and will be used throughout this chapter as a basis for demonstrating different types of connections. The full source listing for the `MsgClient` is shown in listing 13.1.

Listing 13.1 `MsgClient.java`

```
package com.ctimn;

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

public class MsgClient extends MIDlet implements CommandListener {

    private Form outputForm;
    private Display display;
    private List menu;
    private Command okCmd = new Command("OK", Command.OK, 1);
    private Command exitCmd = new Command("Exit", Command.EXIT, 1);
```

```

private static final String[] choices = {
    "1 HTTP Example"
};

protected void startApp() throws MIDletStateChangeException {
    display = Display.getDisplay(this);
    outputForm = new Form("Server Messages");
    menu = new List("Select:", List.IMPLICIT, choices, null);
    menu.addCommand(okCmd);
    outputForm.addCommand(okCmd);
    outputForm.addCommand(exitCmd);
    menu.addCommand(exitCmd);
    outputForm.setCommandListener(this);
    menu.setCommandListener(this);
    display.setCurrent(menu);
}

protected void pauseApp() {
}

protected void destroyApp(boolean unconditional)
    throws MIDletStateChangeException {
}

public void commandAction(Command cmd, Displayable displayable) {
    if (cmd == exitCmd){
        handleExit();
    } else if ((displayable == menu) && (cmd == okCmd)) {
        handleOK(((List)displayable).getSelectedIndex());
    } else {
        display.setCurrent(menu);
    }
}

private void handleExit(){
    try {
        notifyDestroyed();
        destroyApp(true);
    } catch (MIDletStateChangeException x) {
        x.printStackTrace();
    }
}

private void handleOK(int idx){
    display.setCurrent(outputForm);
    getHttpMessage();
}

private void getHttpMessage(){
    int c = 0;
    String dataIn = null;
    StringItem item = new StringItem("Reading from URL", "");
    outputForm.append(item);
}

```

1

```

try {
    ContentConnection connection = (ContentConnection)
        Connector.open(
            "http://www.catapult-technologies.com/ctimain.htm",
            Connector.READ);
    DataInputStream is = connection.openDataInputStream();
    try {
        System.out.println("encoding: " + connection.getEncoding());
        System.out.println("length: " + connection.getLength());
        System.out.println("type: " + connection.getType());
        StringBuffer sb = new StringBuffer("");
        for (int ccnt=0; ccnt < connection.getLength(); ccnt++){
            c = is.read();
            sb.append((char)c);
        }
        dataIn = sb.toString();
        item = new StringItem("Title: ", getTitle(dataIn));
        outputForm.append(item);
    } finally {
        is.close();
    }
} catch (IOException x) {
    System.out.println("Problems sending or receiving data.");
    x.printStackTrace();
}

private String getTitle(String data){
    String titleTag = "<TITLE>";
    int idx1 = data.indexOf(titleTag);
    int idx2 = data.indexOf("</TITLE>");
    return data.substring(idx1 + titleTag.length(), idx2);
}

```

- ❶ Set up the user interface
- ❷ Open a connection
- ❸ Open an input stream
- ❹ Display connection information
- ❺ Read the input
- ❻ Convert bytes to characters
- ❼ Extract the title

BEST PRACTICE Note the use of the `try..finally` construct in the example. As a general practice, it is a good idea to use a `try..finally` construct to handle the closing of resources. This ensures that the `close()` operation always takes place, whether an exception is thrown or not.

The `try` should be placed immediately after the resource has been opened. Placing the open statement within the `try..finally` block is likely to cause problems if the open process throws an exception. This is because the flow of control would be routed through the `close()` statement while the connection is in an unstable state, and was probably never opened.

The steps to compile and run this example follow. Since we are only dealing with a single class, the `MsgClient`, there is no need to JAR this application in order to run it.

13.3.3 Compiling and running the application

Use the following command line to compile the application:

```
>e:\jdk1.3\bin\javac -g:none -bootclasspath e:\midp-fcs\classes -classpath
.\build -d .\build MsgClient.java
```

Use the following command to preverify the application:

```
>e:\midp-fcs\bin\preverify.exe
-classpath e:\midp-fcs\classes;..\build .\build
```

Use the following command to run the application:

```
>e:\midp-fcs\bin\midp.exe -classpath e:\midp-fcs\classes;
.com.ctimn.MsgClient
```

The first screen to appear when running the application is the menu options of the `MsgClient` application. This is shown in figure 13.2.

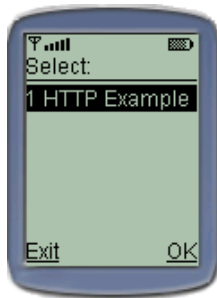


Figure 13.2
MsgClient main menu

Pressing the OK button reads the information from the URL provided. The encoding, content length, and content type are displayed to the console so that we can inspect the values. The “Title” of the HTML page is then parsed and displayed to the screen. The MIDlet output is depicted in figure 13.3.

The output to the console is as follows:

```
encoding: null
length: 176
type: text/html
```

In this case, the encoding is not known, the length, in bytes, is 176 and the content type is HTML.

13.4 SOCKET-BASED CONNECTIONS

Sockets can be used to implement communication between two systems and allow the network connection to be treated as a stream. Once a socket is established the stream can be read from, using an `InputStream`, or written to, using an `OutputStream`.

The Generic Connection Framework provides two interfaces for working with streams, `StreamConnectionNotifier` and `StreamConnection`. `StreamConnectionNotifier` is used by a server to monitor a port for clients wanting to establish a connection. `StreamConnection` is used to establish a socket connection.

The `StreamConnection` interface extends both `InputConnection` and `OutputConnection`, allowing both read and write capability on a single connection. In general, the interfaces `InputConnection`, `DataInputConnection`, `OutputConnection`, and `DataOutputConnection` are used in combination throughout the Generic Connection Framework to build more sophisticated connection types. Individually, they are not terribly useful since most connection protocols support both read and write capabilities.

Before a client can request a socket connection to a listener, the listener must be listening to a designated port. To bind a socket listener application to a port using `StreamConnectionNotifier` the following syntax for the open command is used:

```
StreamConnectionNotifier connection = (StreamConnectionNotifier)
    Connector.open("serversocket://:4444", Connector.READ_WRITE);
```

The `Connector` knows to open a `StreamConnectionNotifier` by looking for the “serversocket” scheme. The port number is specified after the “:” in the address portion of the name. The port chosen is arbitrary so long as both the client and the socket listener use the same port number. Also, if the specified port number is unavailable, or another service is already bound to the port, the connection will be refused.

Once a `StreamConnectionNotifier` is established, the socket listener waits for a client to attempt a connection using the `acceptAndOpen()` method.

```
StreamConnection sc = connection.acceptAndOpen();
```

When a client attempts to connect to the socket listener, the `acceptAndOpen()` method verifies that the connection can be established and opens a `StreamCon-`

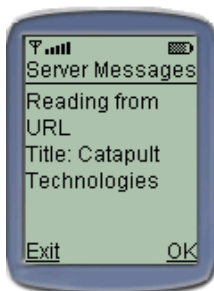


Figure 13.3
Reading the HTML page
with a `ContentConnection`.

nection socket. Once a `StreamConnection` is established, the socket listener is ready to read and write data from and to the stream. In order for the connection to be accepted by the listener, the client attempting the connection must be attempting to establish a socket connection. Other types of connections that the listener cannot handle or does not understand are refused.

Clients connect to socket listeners by directly opening a `StreamConnection`.

```
StreamConnection connection = (StreamConnection)
    Connector.open("socket://127.0.0.1:4444", Connector.READ_WRITE);
```

To open a socket from the client side, the "socket" scheme is used and both the host and port number must be specified in the address portion of the name. Note that the port numbers must be exactly the same when the connection is opened for both the client and the socket listener. If they are different you will not be able to establish a connection. Furthermore, if the socket listener at the host address is not listening to this port, or another type of service that cannot handle sockets is bound to this port, the connection will be refused with an exception stating that the connection was refused. If the connection is successful, the socket is ready to write and read data to and from the socket's input and output streams.

Generally, the client is the first to write data to the stream, even if only to issue a command to the listener; however, this is not a requirement of sockets. Once a connection is successful, either side may initiate the conversation.

Sockets provide a very useful means of communicating between systems; however, sockets only define the connection and the low-level data transport mechanisms, such as TCP/IP. How the client and socket listener deal with each other must be defined by the client and listener applications.

13.4.1 Writing to sockets

Once we have a successful connection, an output stream may be obtained from the `StreamConnection` using the `openOutputStream()` or `openDataOutputStream()` methods.

```
OutputStream os = connection.openOutputStream();
DataOutputStream os = connection.openDataOutputStream();
```

Once an output stream is obtained, the application can begin writing data using one of the various methods of the `OutputStream`. To make life easier, an `OutputStream` can be wrapped inside of other stream classes that provide richer data support when writing to the stream. For example, an `OutputStream` could be passed to a `OutputStreamWriter` to provide the means for dealing with character-based stream content rather than byte-based content. The `OutputStreamWriter` acts as a filter, converting the characters passed to the `OutputStreamWriter` methods into a byte representation of the data and passing this on to the appropriate `OutputStream` method.

OutputStream

`OutputStream` provides the basic methods necessary for writing byte data onto a stream, flushing the stream and closing the stream. All other output stream classes either extend or wrapper instances of `OutputStream`.

DataOutput

`DataOutput` is an interface that defines the methods for converting data from primitive types, such as `int`, `long`, `boolean`, etc. to an array of bytes that can be written to an output stream. This interface also provides the ability to convert Java `Strings` to UTF-8 format that in turn is written as a byte array to an output stream.

DataOutputStream

`DataOutputStream` extends `OutputStream` and implements the `DataOutput` interface to provide the ability to deal with byte, character and UTF encoded data in a machine-independent manner.

ByteArrayOutputStream

A `ByteArrayOutputStream` extends `OutputStream` and provides dynamic buffering capabilities for writing to byte streams. As data is written to the `ByteArrayOutputStream`, the buffer grows automatically. This class supplies two useful methods for retrieving the data as a byte array, using `toByteArray()`, or as a `String`, using the `toString()` method.

Writer

`Writer` is an abstract class that provides support for writing to character streams as opposed to bytes. Java uses a naming convention of “Writer” in the `java.io` package to denote classes that act as a bridge between byte streams and character-based streams. The fundamental benefit of using a `Writer` (or its counterpart, a `Reader`) is that character encoding is automatically translated between the byte representation of the data and the character representation of the data. All other writer classes extend `Writer`. Since this class is abstract it cannot be used directly by applications. Applications requiring `Writer` capabilities should use `OutputStreamWriter`.

OutputStreamWriter

`OutputStreamWriter` extends `Writer` and provides the necessary implementation for applications to write characters to an output stream. With the exception of the `write()` methods, all characters written to the stream through this class are buffered, requiring a call `flush()` in order to actually place the data on the stream.

PrintStream

`PrintStream` is a convenience class that extends `OutputStream` and provides means for easily printing stream data. Most notably, `PrintStream` introduces a `println()` with various signatures for printing different data types. The `println()` methods automatically append a ‘\n’ (new line) character to the data printed. Furthermore, `PrintStream` does not throw `IOExceptions` but rather sets an internal flag if errors occur. The error state is checked by a call to the `checkError()` method.

13.4.2 Reading from sockets

The API for reading from a socket is similar to the output APIs but performs reads instead. Once an input stream is obtained, there are a number of classes that help facilitate retrieving data from a stream.

InputStream

`InputStream` provides the basic methods necessary for reading byte data from a stream and closing the stream. All other input stream classes either extend or wrapper instances of `InputStream`.

DataInput

`DataInput` is an interface that defines methods for reading a stream of bytes and converting this series of bytes to Java primitive types such as `int`, `long`, `short`, `char`, `boolean`, etc. This interface also defines the ability for creating a Java `String` from a UTF-8 format.

DataInputStream

`DataInputStream` extends `InputStream` and implements the `DataInput` interface, providing an implementation for reading and converting java primitives from a series of bytes on an input stream. This class also provides the ability to deal with UTF encoded data in a machine independent manner.

ByteArrayInputStream

`ByteArrayInputStream` extends `InputStream` and provides buffering capabilities while reading from a byte input stream. The number of bytes read by this class is determined by the buffer size provided to the constructor.

Reader

`Reader` is an abstract base class for other readers in the API. Java uses a naming convention of “Reader” in the `java.io` package to denote classes that act as a bridge between byte streams and character-based streams. The fundamental benefit of using a `Reader` is that character encoding is automatically translated from the byte representation of the data to the character representation of the data during the read operations. All other reader classes extend `Reader`. Since this class is abstract it cannot be

used directly by applications. Applications requiring `Reader` capabilities should use `InputStreamReader`.

InputStreamReader

`InputStreamReader` extends `Reader` and provides an implementation for reading character data from a byte stream, thus providing a layer of translation between byte and character data. This class also provides capabilities for returning UTF-encoded data. `InputStreamReader` automatically supports buffering. Each read operation will cause one or more bytes to actually be read from the stream, regardless of the data to be returned by the method.

13.4.3 When to use sockets

Sockets are a primitive but lightweight method of connecting two systems and exchanging data. Due to the low overhead of sockets, this can be one of the fastest methods of exchanging data and issuing commands between two systems. However, the lightweight nature of sockets comes at the price of needing to define a protocol of how the two systems communicate. In other words, sockets provide the connection, but the format of the information exchanged is something left to the implementer. As a result, there are few restrictions on what you can do with sockets; however, everything you do will need to be determined and built.

Sockets are useful in cases where speed is more important than adhering to open protocol standards. In other words, using sockets probably means you will be implementing a proprietary data transport mechanism. (Note, however, that the protocol HTTP can be, and often is, implemented using sockets.) If your system needs to subscribe to open communications standards or you are not in control of both the client and the server, sockets may not be a good way to implement communication capabilities in your application. In these cases, something like HTTP may be more appropriate. There are exceptions to this case, however. Since sockets purely provide the transportation mechanism, another data format or protocol could be used in conjunction with sockets. For example, sockets could be used in combination with XML. This would allow an application to take advantage of sockets while using a nonproprietary or publicly defined XML schema. You should be aware, however, that this still requires coordination between client and server applications since they both need to support the same connection types. If the systems implementing a standardized XML data exchange expect an HTTP connection, implementing a socket solution would not be acceptable.

13.4.4 Client-server socket example

The following example enhances the `MsgClient` application used earlier to include socket communication capabilities. This example illustrates how to establish a connection between a client and a socket listener (server) and send data between the two systems. Figures 13.4 and 13.5 illustrate communication links between two systems.

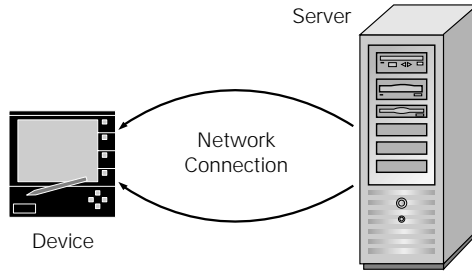


Figure 13.4
A client-server relationship that allows data to be exchanged between the two systems. Generally, the client (in this example, the device) participates by triggering the communication events and asking the server for information. The server simply listens for incoming messages and responds appropriately.

Since we are dealing with J2ME, the two systems in this case are cellular phones that send messages to each other. However, it is definitely possible to use sockets to connect to a J2SE or J2EE server application.

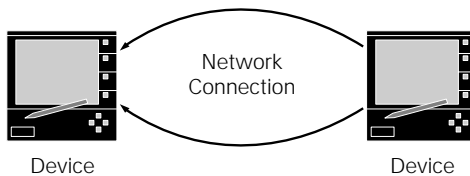


Figure 13.5 A client-server relationship where both systems are mobile devices. This is the scenario used for the socket and datagram examples so that both sending and receiving data in a J2ME environment can be demonstrated. In this situation, one of the devices would be designated the client and the other the server. The device acting as the server will listen for incoming messages and respond appropriately. For example, two devices could communicate with each other using an infrared link between the two devices.

Modifying the client

The first step in incorporating socket capabilities in the `MsgClient` application is to modify the user interface so that the socket behavior that we will be writing can be invoked. This requires two changes. First, we add a menu item to our list of menu choices called “Socket Message”. This becomes the second option in the list.

```
private static final String[] choices = {
    "1 HTTP Example",
    "2 Socket Message"
};
```

Next we enhance the `handleOK()` method to be able to respond appropriately when the “Socket Message” menu option is chosen.

```
private void handleOK(int idx){
    display.setCurrent(outputForm);

    switch (idx) {
        case 0:
            getHttpMessage();
            break;
```

```

        case 1:
            socketMessage();
            break;
    }
}

```

The only thing left on the client side is to implement the `socketMessage()` method to send and receive data. To open a socket connection, the following scheme is used.

```

StreamConnection connection = (StreamConnection)
    Connector.open("socket://localhost:4444", Connector.READ_WRITE);

```

The input and output streams are obtained immediately, since both will be required. Once this is done, the connection's `close()` method can be called.

```

DataOutputStream os = connection.openDataOutputStream();
DataInputStream is = connection.openDataInputStream();
connection.close();

```

The complete `socketMessage()` method of implementation is shown in listing 13.2:

Listing 13.2 `socketMessage()` method

```

private void socketMessage(){
    StringBuffer sb = new StringBuffer("");
    String dataIn = null;
    String dataOut = null;
    int c = 0;
    try {
        StreamConnection connection = (StreamConnection)
            Connector.open("socket://localhost:4444", Connector.READ_WRITE);
        DataOutputStream os = connection.openDataOutputStream();
        DataInputStream is = connection.openDataInputStream();
        connection.close(); -
        try {
            dataOut = "Message from the client.";
            os.writeUTF(dataOut);
            os.flush();
            dataIn = is.readUTF();
            System.out.println(dataIn);
            StringItem si = new StringItem("Msg: ", ""+dataIn+"");
            outputForm.append(si);
        } finally {
            is.close();
            os.close();
        }
    } catch (IOException x) {
        System.out.println("Problems sending or receiving data.");
        x.printStackTrace();
    }
}

```

1 Open a socket connection

2 Write data

3 Read data

4 Close the socket

Creating the socket listener

Now that we have a client that can connect to a service that is listening to a particular port, we need to create that service. To do so, we will implement another MIDlet that will listen to a designated port. When a client sends a message, the message is displayed by the MIDlet and a response is returned.

The listener MIDlet will be used in this example as well as the datagram example, so we will create the user interface with the ability to handle both cases. To modularize the design, the protocol-specific behavior will be encapsulated in separate classes. This allows the listener application to easily employ many different services without becoming monolithic. The listener application has the responsibilities of providing the means to start a particular service, socket listener, or datagram listener, and displaying the messages as they are handled by the service. The full source listing for the listener application is provided in listing 13.3.

Listing 13.3 MsgListener.java

```
package com.ctimn;

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

public class MsgListener extends MIDlet implements CommandListener {

    private Form outputForm;
    private Display display;
    private List menu;
    private Command okCmd = new Command("OK", Command.OK, 1);
    private Command exitCmd = new Command("Exit", Command.EXIT, 1);
    private SocketListener socketListener;

    private static final String[] choices = {
        "1 Socket Listener",
        "2 Datagram Listener"
    };

    protected void startApp() throws MIDletStateChangeException {
        display = Display.getDisplay(this);
        outputForm = new Form("Messages");
        menu = new List("Select:", List.IMPLICIT, choices, null);
        outputForm.addCommand(okCmd);
        menu.addCommand(okCmd);
        outputForm.addCommand(exitCmd);
        menu.addCommand(exitCmd);
        outputForm.setCommandListener(this);
        menu.setCommandListener(this);
        display.setCurrent(menu);
    }

    protected void pauseApp() {
    }
}
```

1 Provide menu options

```

protected void destroyApp(boolean unconditional)
    throws MIDletStateChangeException {
    System.out.println("Destroy App.");
    if (socketListener != null){
        socketListener.shutdown();
    }
}

public void commandAction(Command cmd, Displayable activeDisplay) {
    if (cmd == exitCmd) {
        handleExit();
    } else if ((activeDisplay == menu) && (cmd == okCmd)) {
        handleOK(((List)activeDisplay).getSelectedIndex());
        return;
    }
    display.setCurrent(menu);
}

private void handleExit(){
    try {
        System.out.println("exit.");
        destroyApp(true);
        notifyDestroyed();
    } catch (MIDletStateChangeException x){
        x.printStackTrace();
    }
}

private void handleOK(int idx){
    display.setCurrent(outputForm);
    switch (idx) {
        case 0:
            socketListener();
            break;
        case 1:
            datagramListener();
            break;
    }
}

private void socketListener(){
    if (socketListener == null){
        socketListener = new SocketListener(outputForm);
        socketListener.start();
    }
}

private void datagramListener(){
}

```

2 Handle the menu event

3 Start the socket listener service

4 Create a placeholder for the datagram example

Creating the service class

The next step is to implement the `SocketListener` service class. This is where all the socket listener behavior will be encapsulated. Listening for messages is slightly different than sending messages because the listener can never be sure of when or if the client will attempt to connect. As a result, if we simply listened for the connection and the data to be sent on the same thread that the application is running on, our application would appear to hang until the connection was terminated. For this reason, it is best to implement the connection listening part of the application on a separate thread. This allows the application to continue functioning while waiting and receiving messages. By handling messages on a separate thread, the data received by the message handler can be displayed immediately as well, rather than having to wait until the connection finished transmitting data. This is especially important if two devices require that the users provide input during the data exchange. If a separate thread is not used, a connection would need to be established each time the user entered a piece of data, since the connection and the data entry must share the same thread.

The `SocketListener` class extends `Thread` to provide the ability to create and run the listener on its own thread.

```
public class SocketListener extends Thread
```

The `Thread` class requires that the main thread loop be implemented in a method named `run()`. For our purposes, this is where the connection will be established and the listener loop will be implemented.

To start a thread in Java using the `Thread` class, the `start()` method is called. In our case, the `SocketListener` thread is started by the `MsgListener` application. This is already in place as we can see by revisiting the code listing for `MsgListener.java`.

Since the `SocketListener` will be running on its own thread, we need a way to shut down the thread when we are finished. A public `shutdown()` method is provided to perform this step from the `MsgListener` application. However, the best we will be able to do at this point is set a `boolean` flag indicating that the `SocketListener` should shut itself down at the earliest point possible. This is because the `SocketListener` class will be occupied by its task of listening for incoming messages. The best opportunity for checking the shutdown state is when the listener times out. We then have the opportunity to check the shutdown state and either exit the thread or restart the listener connection.

In some implementations of the J2ME virtual machines, there is only one underlying native thread that is actually being used on the device. This is the thread the virtual machine is running on. In these situations multithreading behavior is handled internally by the virtual machine using what are called green threads. Green threads are explained in more detail in chapter 14. In these situations all threads will be terminated automatically when the application exits without the need for invoking any thread shutdown operations. However, it is always a good idea to have thread shut-

down functionality in place, since each J2ME virtual machine has different characteristics and may implement threading differently. Failing to properly terminate threads could result in memory leaks or worse. In our example, failing to shut down a listener would lock the port that the listener is bound to, requiring the device to be restarted in order to release the port resource.

NOTE The KVM provided with the MIDP reference implementation implements multithreading entirely within the virtual machine (green threads). As a result, all listener threads are terminated when the application exits. Therefore, there is no way to actually illustrate the threads terminating on their own since they are shut down automatically.

Unlike a socket client, the socket listener must listen to a designated port for any clients wishing to establish communications. This is implemented using a `StreamConnectionNotifier`.

```
StreamConnectionNotifier connection = (StreamConnectionNotifier)
    Connector.open("serversocket://:4444", Connector.READ_WRITE);
StreamConnection socketConnection = connection.acceptAndOpen();
```

WARNING Sockets can be created on any port supported by the platform. However, it is important to note that on Unix you must be signed in as the root user to create a port below 1024. This means that your application would be required to run as root to create a `StreamConnection` or `StreamConnectionNotifier` using port 999 on Unix. Furthermore, you will be deploying applications into network environments where other services are also using ports. If you attempt to establish a connection to a port that is already in use an exception will be thrown and your application will be unable to bind to the port. Although changing a port may be an easy modification to your code, it is nice to find problems like this before your application ships. Therefore it is advisable to do a bit of research on the platforms and environments that an application is targeting before choosing the port number.

The `StreamConnectionNotifier` is first created using the `serversocket` scheme and designating a port. The mode in which to open socket connections is also specified. Once we have a `StreamConnectionNotifier`, the `acceptAndOpen()` method is called. The `acceptAndOpen()` method causes the server to sit idle until a client attempts to connect to the server. There is no socket connection at this point.

When a client contacts the server, the server determines if a socket connection can be established. This takes place within the `acceptAndOpen()` method. If a socket connection can be established, `acceptAndOpen()` returns a `SocketConnection`. Since we specified that `READ_WRITE` sockets should be created when we created the `StreamConnectionNotifier`, the `SocketConnection` returned can be used for both receiving and sending data to the client.

If the client connection cannot be established, the connection is refused. An exception is thrown on the client side. The server, however, continues listening for other clients.

With a client-server socket connection established, the server is reading to receive and send data. To do this, an `InputStream` and an `OutputStream` are obtained.

```
InputStream is = socketConnection.openInputStream();
OutputStream os = socketConnection.openOutputStream();
```

Reading from a stream

Data can be read from a `DataInputStream` as a UTF-8 encoded `String` using the method `readUTF()`. The data read using the method, however, must be sent using UTF-8 encoding in order to read the data successfully. This requires the client and server to coordinate on how the data is sent. Alternately, the data can be read as an array of bytes. Examples of both techniques are provided below. This first example reads UTF-encoded data:

```
String dataIn = is.readUTF();
```

The following example demonstrates reading bytes from the stream. This example assumes that the data is character data in byte form and coerces the data before appending it to a `StringBuffer`:

```
int ch = 0;
StringBuffer sb = new StringBuffer();
while (ch != -1) {
    ch = is.read();
    if (ch == -1){
        break;
    }
    sb.append((char)ch);
}
String dataIn = sb.toString();
```

Writing to a stream

To write data to the output stream we will use the `writeUTF()` method along with a `DataOutputStream`. This method encodes the data as UTF-8 before sending it to the destination. Since the listener is expecting a UTF-8 encoded `String` it is necessary that the client provide the data in this format. If the application on the other end is not expecting UTF-8 encoded data, it may not be able to handle the data properly. In this case, unexpected behavior in the stream interactions can result. If the client cannot handle UTF-encoded data, the data can be sent as a byte array.

The following code demonstrates how to write to the `DataOutputStream`. When you are finished, the output stream must be flushed. This forces bytes within the buffer to be written to the stream, if the stream supports buffering. How data is buffered in an output stream differs between different types of streams. In some cases, no buffering may be applied at all. However, it is always a good practice to call `flush()` when you have finished writing to a stream.

```
String data = "Test Message";
os.writeUTF(data);
os.flush();
```

The next example demonstrates writing a byte array to the stream. This is useful when sending raw bytes to another system.

```
String data = "Test Message";
byte[] b = data.getBytes();
os.write(b, 0, b.length);
os.flush();
```

Opening a client connection

Once an application is listening to a port, a client can attempt to open a connection to the listening application using the following syntax:

```
StreamConnection connection = (StreamConnection)
    Connector.open("socket://127.0.0.1:4444", Connector.READ_WRITE);
```

Unlike the socket listener example, the client must specify a host address as well as a port. If the `open()` is successful, a `StreamConnection` is returned. If the socket connection cannot be established, a `ConnectException` is thrown indicating that the connection was refused. A connection can be refused for a number of reasons. Typically, this occurs when there is no application listening to the port or another type service that does not or cannot deal with sockets is listening to the port and refuses the connection.

With a `StreamConnection` open, the client is ready to send and receive data. At this point, the code is the same as the server examples provided earlier.

Shutting down the listener thread

Since the `SocketListener` is implemented on its own thread, it supports the ability for an external object to shut it down. In our example, `MsgListener` makes the call to `shutdown()` when the application exits. Since the `shutdown()` method is invoked by a thread different from the one that our socket listening code is running, all we can really do is set a flag so that the thread within the listen-respond loop can check the status of this flag periodically during execution and take appropriate action.

There is a problem with this, however. Most likely, when `shutdown()` is called, the thread that we need to shut down is busy listening for connections within the `acceptAndOpen()` method. The thread will remain at this spot in the code until one of two things happens: a socket connection is accepted or a timeout occurs. Since these are the only two cases where the listening thread will have the opportunity to check the status of the shutdown flag, we use these situations to our advantage. The easiest way to trigger the shutdown process is for the listener to respond to timeout conditions. This is done by specifying the appropriate value in the `Connector`'s `open()` method.

```
StreamConnectionNotifier notifier = (StreamConnectionNotifier)
    Connector.open("serversocket://:4444", Connector.READ_WRITE, true);
```

This signature of the open method allows for setting a boolean indicating we want to be notified when a timeout occurs. The timeout can then be detected by catching an `InterruptedException`. When this exception occurs, after the timeout expires the `StreamConnectionNotifier` connection, the `SocketListener` can take appropriate action based on the value of the shutdown flag. If the flag is true, indicating we should shut down, the method exits (using a return statement). If the value is false, the `SocketListener` continues listening for connections. If the `SocketListener` is to continue listening, however, the `StreamConnectionNotifier` must be reestablished. The code to perform shutdown flag monitoring is as follows:

```
try {
    StreamConnection connection = notifier.acceptAndOpen();
} catch (InterruptedException x){
    if (shutdownFlag){
        return;
    } else {
        notifier = createNotifier();
    }
}
```

In this example the `createNotifier()` method returns an instance of `StreamConnectionNotifier`.

```
private StreamConnectionNotifier createNotifier()throws IOException {

    return (StreamConnectionNotifier)
        Connector.open("serversocket://:4444", Connector.READ_WRITE, true);
}
```

The full-source listing of `SocketListener` is shown in listing 13.4. To simulate connecting over a network connection, the localhost address will be used, 127.0.0.1.

WARNING In some implementations of the Generic Connection Framework it is necessary to specify the IP address rather than the domain name.

Listing 13.4 SocketListener.java

```
package com.ctimn;

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

public class SocketListener extends Thread{
    private Form outputForm;
    private boolean shutdownFlag = false;
    private StreamConnectionNotifier notifier;

    public SocketListener(Form outputForm){
        this.outputForm = outputForm;
    }
}
```

1 Reference to the `MsgListener` output form

2 Flag indicating if the service has been shut down

3 Socket connection

```

private StreamConnectionNotifier createNotifier() throws IOException {
    return (StreamConnectionNotifier)
        Connector.open("serversocket://:4444",
            Connector.READ_WRITE, true);
}

public void run(){
    String dataIn = null;
    String dataOut = null;
    int counter = 1;
    StringItem item = new StringItem("Listening to Socket", "");
    outputForm.append(item);
    StreamConnection connection = null;
    try {
        notifier = createNotifier();
        while (true) {
            try {
                connection = notifier.acceptAndOpen();
            } catch (InterruptedException x){
                if (shutdownFlag){
                    return;
                } else {
                    notifier = createNotifier();
                }
            }
            DataInputStream is = connection.openDataInputStream();
            DataOutputStream os = connection.openDataOutputStream();
            connection.close();
            try {
                dataIn = is.readUTF();
                System.out.println(dataIn);
                item = new StringItem("Msg: ", ""+dataIn+"");
                outputForm.append(item);
                dataOut = "Message " + counter + " from the server.";
                counter++;
                os.writeUTF(dataOut);
                os.flush();
            } finally {
                os.close();
                is.close();
            }
        } catch (IOException x) {
            System.out.println("Problems sending or receiving data.");
            x.printStackTrace();
        }
    }

    public void shutdown(){
        shutdownFlag = true;
    }
}

```

4 Create the socket connection

5 Establish the listener connection with timeout exceptions

6 Do these steps "forever"

7 Wait for a client message

8 Check shutdown status on timeout. Return if shutdown=true

9 Return if shutdown=false, restart the listener connection

10 Get the I/O streams

11 Call close on the connection obtained by acceptAndOpen()

12 Read and display the message from the client

13 Form and write a response message

14 Flush the output stream buffer

15 Close the I/O streams

16 Set the shutdown flag

Compiling and preverifying and running

The entire example, including the classes `MsgClient`, `MsgListener`, and `SocketListener` is now ready to be compiled and preverified. In order to run the application, since there is more than a single class involved, all the classes will need to be packaged into a JAR file. The following batch file shown in listing 13.5 can be used to build the application and bundle it as a single JAR file.

Listing 13.5 Build.bat

```
e:\jdk1.3\bin\javac -g:none
  -bootclasspath e:\midp-fcs\classes
  -classpath .\build
  -d .\build *.java

e:\midp-fcs\bin\preverify.exe
  -classpath e:\midp-fcs\classes;..\build .\build

jar cvf io.jar -C output .
```

The `MsgClient` and `MsgListener` applications must be run from separate command windows. The following commands are necessary for starting the `MsgListener`. The first command allows all protocols to be made available to our MIDlet. Without this flag set to true, only the HTTP connections are available since this is the only connection type that must be supported by MIDP.

```
>set ENABLE_CLDC_PROTOCOLS=true

>e:\midp-fcs\bin\midp -classpath e:\midp-fcs\classes;.\io.jar
com.ctimn.MsgListener
```

The `MsgListener` presents a menu listing the available options. Choosing the “Socket Listener” option invokes the `SocketListener` and binds the listener to port 4444. Figure 13.6 shows the running `MsgListener` application.

Use the following command to run the `MsgClient`:

```
>set ENABLE_CLDC_PROTOCOLS=true

>e:\midp-fcs\bin\midp -classpath e:\midp-fcs\classes;.\io.jar
com.ctimn.MsgClient
```



Figure 13.6
Running `MsgListener` for the first time. The `MsgListener` creates a `StreamConnectionNotifier` and waits for activity on the specified port.

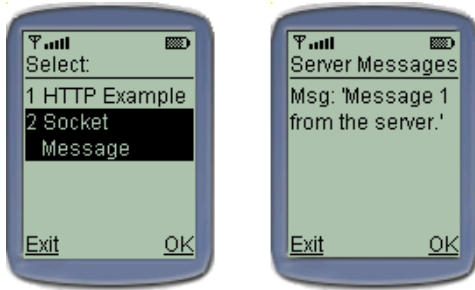


Figure 13.7
MsgClient sending and receiving messages using sockets. The MsgClient initiates communication by opening a socket connection. The receiving system acknowledges the connection, allowing the MsgClient to pass data onto the stream represented by the connection.

Choosing the second option sends a message from the `MsgClient` to the `MsgListener`. The `MsgListener` then returns a response to the `MsgClient`. This is shown in figure 13.7.

The client initially displays a menu as well, which is shown in figure 13.7. Selecting the second option “Socket Message” triggers the client code to send a message to the server. The server responds with a message. The messages are displayed in the output forms of each emulator. The output from the `MsgListener` is shown in figure 13.8.

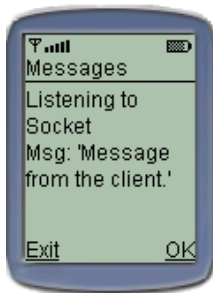


Figure 13.8
MsgListener receiving a message over a socket connection. The `MsgListener` monitors a specified port for socket activity. When the `MsgClient` initiates a connection, the `MsgListener` establishes the other half of the connection and gets ready to receive data. When the `MsgClient` transmits data, the `MsgListener` responds appropriately.

Additionally, we placed some `System.out.println()` statements into our code to monitor progress from the command line. These results are shown in figure 13.9.

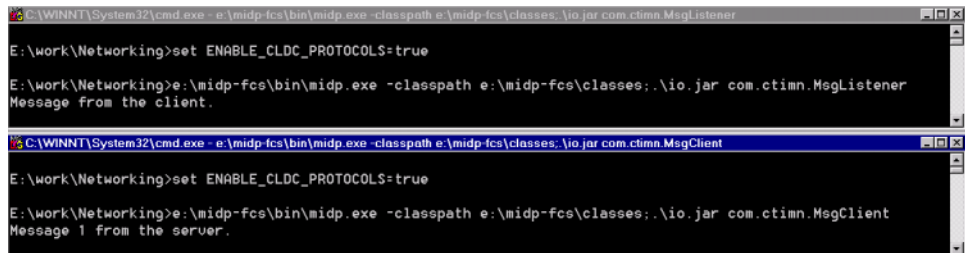


Figure 13.9 Command line results from running `MsgClient` and `MsgListener`.

13.5 DATAGRAM-BASED CONNECTIONS

Datagrams are designed for sending packets of data over a network. Datagrams work much differently than sockets in that a hard connection is not established between the two systems. In the case of sockets, if a client tries to connect to a system that does not support sockets or is not listening for socket connections, an exception is thrown. Datagrams, on the other hand, allow data to be sent over a connection regardless of whether the listener on the other end is capable of handling datagrams or even exists. In all cases, when sending data using Datagrams, the transmission is assumed to be successful. Furthermore, unlike sockets, the data sent using Datagrams is considered to be unreliable in that if a packet is lost it is not resent automatically by the protocol implementation, and when multiple packets are sent there is no guarantee that the packets will arrive in the same order they were sent. Datagrams do not provide support for reassembling data packets into the order in which they were sent. For these reasons, Datagrams are termed to be an unreliable data transport mechanism. The term unreliable in this case is not necessarily a negative term. It simply means that the protocol does not inherently support mechanisms to guarantee that data arrives in the order it was sent or that the data arrives at all. There is nothing stopping an application from implementing these features itself, however.

So why use Datagrams? Speed is one primary reason. Datagrams do not incur the overhead of ensuring that packets arrive in the correct order or that they arrive at all. In some applications, such as audio streaming, a missing data packet may appear as static. Raw speed is more important in this case than data integrity.

There are several datagram protocols available. The most common is User Datagram Protocol (UDP). This is the protocol implementation provided by the reference implementation of the Generic Connection Framework. However, the `Datagram` and `DatagramConnection` interfaces of the Generic Connection Framework are designed to allow implementations of different types of datagram protocols. Other such protocols include IP and WDP along with proprietary beaming protocols that take advantage of the packet nature of datagrams for transmitting data.

When to use datagrams

At first glance, datagrams seem to have a lot of marks against their use, especially since there is no reliability of data delivery, flow-control and error handling. However, the raw speed benefits of datagrams may outweigh the data integrity issues for some applications. Applications that stream real-time audio or video may be more concerned with speed than getting every byte of data transported and in a certain order. If data is missing there may be some static over the speaker or on the screen momentarily. Although static is not a desirable feature in such applications, the alternative would require the application to wait for all the data to arrive and to place it into the correct receiving order based on how the packets were sent before the data could be

officially received. This speed degradation is likely to be unacceptable in applications that are streaming audio or video content.

Since UDP does not provide guarantees of packet delivery or packet receiving order, the headers and metadata required are simpler than a reliable protocol such as TCP. Therefore, datagrams are most useful when speed of delivery is crucial. In the J2ME environment, datagrams can be useful due to their simplicity as a lighter weight data transport alternative to TCP. For example, datagrams might be useful when beaming data over an Ir port between two devices.

Another feature of datagrams is that the programmer controls the packet size of the transmission. If you want to send a large amount of data in a single packet, you can (up to 64kB). If you want to send a single byte in a packet, you can.

Handling datagram unreliability

Although UDP datagrams do not inherently provide guaranteed delivery and packet reordering, you can implement this at the application level. For example, a client that sends a datagram and does not receive a response for a specified period of time could assume the packet was not received and try to resend the information or indicate an error. Furthermore, the data encapsulated by the datagram could include tags indicating how to reassemble the data on the receiving side. For example, if the first packet received contains the information “packet 4 of 7” the receiver would understand it needs 7 packets in all before attempting to order the data. If less than 7 packets are received and a certain amount of time passed without receiving another packet, the receiver could ask the sender to resend the missing packets.

Alternatively, a client could send packets one at a time and wait for the receiver to respond with a success code indicating that the packet was correctly received before sending the next packet.

Of course in doing this the sender and receiver need to understand how to communicate. In other words, you need to define your own protocol. This does not mean, however, that you are duplicating the functionality of TCP and eliminating the benefits of datagrams. Obviously, there will be some additional overhead in providing flow-control and data delivery error handling in datagrams. However, a custom protocol has the advantage of accommodating a specific case, rather than the more generalized case that TCP is required to address, and this specificity can improve efficiency. If you are working in a closed system, where you have control of both the sender and the receiver, you also have the ability to define how the sender and receiver communicate.

How datagrams work in J2ME

Datagrams have been generalized in the Generic Connection Framework so that different types of datagram connections can be used. As a result, the datagram API is much different in J2ME than in J2SE.

The two classes involved with datagrams in the Generic Connection Framework are `DatagramConnection` and `Datagram`. The `DatagramConnection` class is

used to bind the application to a port and the `Datagram` class is used to transport data over this port connection. It is important to understand that datagrams do not behave like streams. Although the datagram is ultimately sent across the network connection in some fashion, datagrams themselves are packets of data placed onto the protocol as a whole. The difference is that with a stream, each byte written to a stream immediately becomes part of the stream and is sent to whatever the stream is hooked up to, assuming there is no buffering taking place. With datagrams, all of the data resides in the datagram buffer until the datagram is placed on the `DatagramConnection`. Once the datagram is placed on the connection, the connection transmits the data to the specified target.

NOTE The J2SE `DatagramSocket` class is analogous to the J2ME `DatagramConnection` class and the J2SE `DatagramPacket` class is analogous to the J2ME `Datagram` class. In J2SE, you bind the application to a socket using the `DatagramSocket` class and transport data over this connection using a `DatagramPacket` class.

To send a datagram using the J2ME API, your application needs to supply three things: the address to send the datagram, the port on which the receiving system is listening, and the data. The port used by the application sending data (the client) is always dynamically allocated.

The `DatagramConnection` instance is created using a slightly different name parameter depending on whether you are a datagram client or server. To open a connection as the client the target host must be specified. The following example opens a `DatagramConnection` in “client mode.”

```
datagram://127.0.0.1:5555
```

To open the connection on the receiving side, only the port is specified.

```
datagram://:5555.
```

When establishing a client connection, the application is specifying the host as well as the port on which the host is expected to be listening. The port that the client is using on the system is hidden from the developer and is dynamically assigned.

When opening the connection on the receiving side, the receiving application binds itself to a port. If these ports are not identical, data sent from the client will be lost since it will not be transmitting to a port on which the service is listening. Since datagrams do not guarantee packet delivery, the client sends the data once and is never informed that anything is wrong.

Once a `DatagramConnection` is established, multiple `Datagrams` can be sent over this single connection. However, a single datagram is good for one read and one write, allowing an application to receive a message and send a response. A datagram response is sent using the same `Datagram` instance used to receive data. This instance contains the necessary host and port information required to send the response message.

In this scenario, a new datagram is created for each incoming message. There is no attempt to prevent a single `Datagram` instance from incorrectly being used multiple times; however, unexpected results can occur since the buffer is not designed for reuse beyond the receive-response sequence. Therefore, it is the responsibility of the application to manage when a `Datagram` needs to be created and when a datagram can be reused.

In order for a receiving application to respond to a datagram, the same datagram instance must be used for the outgoing message. Although, in theory, it would be possible to construct a new `Datagram` for the send operation, the Generic Connection Framework `Datagram` interface does not support the ability to obtain the sender's port number. The port number for a `DatagramConnection` opened in client mode is dynamically assigned. Thus, there is no way to properly construct a new `Datagram` to reply to a client application even if we wanted to.

13.5.1 Datagram example

The following example is a simple application that uses datagrams as the means of transmitting data between two systems.

In this example we will create the client (the datagram sender) first. To begin, a `DatagramConnection` between the client to the listener must be established. This is done using the `Connector.open()` method and the datagram scheme.

```
DatagramConnection connection = (DatagramConnection)
    Connector.open("datagram://127.0.0.1:5555", Connector.READ_WRITE);
```

A client connection must specify both the host address as well as the port. The connection is opened in read-write mode, allowing the client and server to pass data back and forth.

The client, in this example, is the first to send a message. To send a message over a `DatagramConnection`, a datagram object is needed. Datagrams are created using the `DatagramConnection` method `newDatagram()`. There are several signatures of `newDatagram()` available. These are provided for convenience. At a minimum, `newDatagram()` requires a buffer size to be specified. We will create a `Datagram` with a buffer size of 100 bytes.

```
Datagram datagram = connection.newDatagram(100);
```

Once the datagram is created, the buffer must be populated with data. Datagrams only deal with data in a byte form. Below we create a `String` and convert it to a byte array. Once we have a byte array of data, this data can be placed into the datagram using the `setData()` method.

```
byte[] data = "Message 1 from the Client".getBytes();
datagram.setData(data, 0, data.length);
```

In the `setData(String)` method, the first parameter is the byte array of data, the second parameter is an offset, indicating where to begin sending data from when the datagram is actually sent. The third parameter is the actual length of the data.

With a `DatagramConnection` and a `Datagram` containing our data, the only thing left to do is send the data. The `DatagramConnection` class provides a `send(Datagram)` method to trigger the data transmission. This method is called by passing our `Datagram` instance as a parameter.

```
connection.send(datagram);
```

The `DatagramConnection send(Datagram)` method automatically flushes the buffer and transmits the data. There are no additional steps we need to take. As long as no exceptions were thrown, the data has been transmitted through the network. What we cannot assume, however, is that the message was actually received. As mentioned, datagrams do not guarantee data transmissions. If there is no system listening on the designated port or the system is unable to handle datagrams, the data is sent into empty space and our client receives no indication of this situation.

However, there are ways to detect if a message was actually received. To do this, our client must be capable of receiving a response from the system to which the message is sent. To set up the client to receive a message, the `DatagramConnection` method `receive(Datagram)` is used. We will also need a new `Datagram` instance to hold the incoming data.

Create a new `Datagram` using the `newDatagram()` method.

```
Datagram datagram = connection.newDatagram(100);
```

Then pass this new datagram to the receive method. The receive method waits idly for data to be sent.

```
connection.receive(datagram);
```

If there is no system listening on the host port, as currently is the case with our example, the client will appear to hang as it waits for the response. To handle this situation you could create the connection and specify that you wish to be notified if there is a timeout on the connection.

```
DatagramConnection connection = (DatagramConnection)
    Connector.open("datagram://127.0.0.1:5555", Connector.READ_WRITE, true);
```

By opening the connection with the ability to be notified of timeouts, the client will not wait indefinitely for the listener's response. However, our application must be ready to deal with the timeout situation as well by handling the exception thrown when the connection times out.

Once the datagram is received, the data can be extracted from the `Datagram` instance, which is accessed using the `Datagram` method `getData()` which returns the data as a byte array. This byte array can then be converted to an appropriate data type. The data type involved depends on what kind of data the transmitting system actually sent. In this example, the data is assumed to be a `String`.

```
String data = new String(datagram.getData());
```

Enhancing the MsgClient

To incorporate the ability to send and receive datagrams from our MsgClient, the following changes will be needed, starting with the user interface, then adding the datagram option to the menu as the third option:

```
private static final String[] choices = {
    "1 HTTP Example",
    "2 Socket Message",
    "3 Datagram Message"
};
```

Next, we add the necessary lines of code to the handleOK method to trigger a datagram message to be sent.

```
private void handleOK(int idx){
    display.setCurrent(outputForm);
    switch (idx) {
        case 0:
            getHttpMessage();
            break;
        case 1:
            socketMessage();
            break;
        case 2:
            datagramMessage();
            break;
    }
}
```

To handle the functionality of sending and receiving datagrams, three more methods are introduced to the MsgClient: `datagramMessage()`, `receiveDatagram()` and `sendDatagram()`. The `datagramMessage()` method is called to handle requests from the menu to send and receive a message. The latter two methods handle the receiving and sending of specific functionality. The code for sending and receiving datagrams is shown in listing 13.6.

Listing 13.6 Sending and receiving datagrams from the MsgClient

```
private void datagramMessage() {
    String msg = null;
    try {
        DatagramConnection connection =
            (DatagramConnection)Connector.open(
                "datagram://localhost:5555", Connector.READ_WRITE);
        Datagram datagram = null;
        try {
            datagram = connection.newDatagram(100);
            sendDatagram(connection, datagram, "Message from the Client");
            datagram = connection.newDatagram(100);
            msg = receiveDatagram(connection, datagram);
        } finally {
            connection.close();
        }
    }
```

```

    } catch (IOException x) {
        x.printStackTrace();
    }
    StringItem item = new StringItem("Msg: ", msg);
    outputForm.append(item);
}

private void sendDatagram(DatagramConnection connection,
    Datagram datagram, String msg) throws IOException{
    byte[] data = msg.getBytes();
    datagram.setData(data, 0, data.length);
    connection.send(datagram);
}

private String receiveDatagram(DatagramConnection connection,
    Datagram datagram) throws IOException{
    connection.receive(datagram);
    System.out.println("Address="+datagram.getAddress());
    System.out.println("Length="+datagram.getLength());
    System.out.println("Offset="+datagram.getOffset());
    byte[] byteData = datagram.getData();
    byte b = 0;
    StringBuffer sb = new StringBuffer();
    for (int ccnt=0; ccnt < byteData.length; ccnt++){
        if (byteData[ccnt] > 0){
            sb.append((char)byteData[ccnt]);
        } else {
            break;
        }
    }
    String data = sb.toString();
    System.out.println("Data="+data);
    return data;
}

```

- 1 Create the Datagram connection
- 2 Create a new datagram and send a message
- 3 Create a new datagram and receive a message
- 4 Close the connection
- 5 Display the message received
- 6 Prepare a datagram and send a message
- 7 Get the message in a byte format
- 8 Load the message into the datagram
- 9 Send the datagram
- 10 Receive and display a datagram message
- 11 Receive the incoming datagram
- 12 Retrieve the contents of the datagram

Datagram listener

Now that we have a client application sending datagrams to another system, we now need to implement a `DatagramListener` class that can handle the message and return a response from the `MsgListener` application we wrote earlier. This listener behaves a lot like the socket listener from the previous example only it handles datagrams and uses a different port. Listening for socket messages and datagram messages on different ports allows the listener application to monitor for both socket and datagram client connections simultaneously.

The only difference between a datagram client and a datagram receiver (server) in J2ME is how the connection is established. In the client example, we needed to specify both the host address as well as the host port. When establishing a listening connection, only the port needs to be specified.

```
DatagramConnection connection = (DatagramConnection)
    Connector.open("datagram://:5555", Connector.READ_WRITE, true);
```

Once the connection is established, the server is ready to read information sent from the client. Reading and writing data to a `DatagramConnection` on the server is exactly the same as using datagrams on the client. First a datagram is created in which to place the data, then the datagram's `receive()` method is invoked.

```
Datagram datagram = connection.newDatagram(100);
connection.receive(datagram);
```

To send data back to the client a `Datagram` is created, populated with the data and placed onto the connection using the `send(Datagram)` method.

```
Datagram datagram = connection.newDatagram(100);
byte[] data = "Message 1 from the Server".getBytes();
datagram.setData(data, 0, data.length);
connection.send(datagram);
```

Listing 13.7 shows the implementation of the `DatagramListener` class that handles receiving and responding to datagram messages on behalf of the `MsgListener` class we created earlier.

Listing 13.7 DatagramListener.java

```
package com.ctimn;

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

public class DatagramListener extends Thread {

    private Form outputForm;
    private boolean shutdownFlag = false;
```

```

public DatagramListener(Form outputForm){
    this.outputForm = outputForm;
}

public void run(){
    Datagram datagram = null;
    String msg = null;
    StringItem item = new StringItem("Listening for Datagrams", "");
    outputForm.append(item);
    try {
        DatagramConnection connection = (DatagramConnection)
            Connector.open("datagram://:5555", Connector.READ_WRITE);
        try {
            while (true) {
                datagram = connection.newDatagram(100);
                try {
                    msg = receiveDatagram(connection, datagram);
                } catch (InterruptedException x){
                    if (shutdownFlag){
                        return;
                    }
                }
                item = new StringItem("Msg: ", msg);
                outputForm.append(item);
                sendDatagram(connection, datagram, "Message from the server");
            }
        } finally {
            connection.close();
        }
    } catch (IOException x) {
        System.out.println("Problems sending or receiving data.");
        x.printStackTrace();
    }
}

private String receiveDatagram(DatagramConnection connection,
    Datagram datagram) throws IOException{
    connection.receive(datagram);
    System.out.println("Address="+datagram.getAddress());
    System.out.println("Length="+datagram.getLength());
    System.out.println("Offset="+datagram.getOffset());
    byte[] byteData = datagram.getData();
    byte b = 0;
    StringBuffer sb = new StringBuffer();
    for (int ccnt=0; ccnt < byteData.length; ccnt++){
        if (byteData[ccnt] > 0){
            sb.append((char)byteData[ccnt]);
        } else {
            break;
        }
    }
    String data = sb.toString();
    System.out.println("Data="+data);
    return data;
}

```

1 Create a Datagram connection

2 Run "forever"

3 Wait for a datagram

4 Receive a datagram

```

private void sendDatagram(DatagramConnection connection,
    Datagram datagram, String msg) throws IOException{
    byte[] data = msg.getBytes();
    datagram.setData(data, 0, data.length);
    connection.send(datagram);
}

public void shutdown(){
    shutdownFlag = true;
}
}

```

5 Send a response

6 Set the shutdown condition

In this example, the client sends a message to the `DatagramListener` and the `DatagramListener` responds with a message of its own. When the client finishes running, it closes the connections and the application exits. The `DatagramListener`, however, continues running and waits for another message.

Shutting down the listener thread

As with the `SocketListener`, `DatagramListener` is implemented on its own thread and therefore must support the ability for an external object to shut it down. In our example, `MsgListener` makes the call to `shutdown()` when the `MsgListener` application exits. Since the `shutdown()` method is invoked by a thread different from the one that our socket listening code is running, all we can really do is set a flag so that the thread within the listen-respond loop can check the status of this flag periodically during execution and take appropriate action.

The same problem that existed with `SocketListener` applies to the `DatagramListener`. At the point `shutdown()` is called the `DatagramListener` is most likely to be waiting for an incoming message. As a result, the listening thread is unavailable to check the status of the shutdown flag. The only events that allow the listener to stop listening for incoming messages and inspect the shutdown status are if a datagram connection is made or a timeout occurs. To take advantage of this situation, the listener requests to be notified of timeouts when the `Datagram` connection is opened. If a timeout occurs, an `InterruptedIOException` exception is thrown. By catching this exception and checking the status of the shutdown flag the `DatagramListener` can respond appropriately.

Enhancing the MessageListener

The next step is to enhance the `MessageListener` created during the socket example to be able to listen for datagrams as well. The following lines of code are required to create a `DatagramListener` instance when the datagram option is selected from the menu. The following method was intentionally left blank on the previous exercise. Now that we have a `DatagramListener`, we can fill in the details.

```

private datagramListener() {
    if (datagramListener == null) {
        datagramListener = new DatagramListener(outputForm);
        datagramListener.start();
    }
}

```

We will also need to add a member variable to hold onto the `DatagramListener` reference. The following line of code needs to be added to the top of the `MessageListener` class.

```
private DatagramListener datagramListener;
```

Building the applications

At this point we can compile, preverify, and run both the `MsgClient` and `MsgListener` (which invokes `DatagramListener`) and send messages between the two applications. The commands for building the examples are essentially the same as for building the `MsgClient`.

Compile the `DatagramListener` and `MsgListener` classes.

```
>e:\jdk1.3\bin\javac -g:none -bootclasspath e:\midp-fcs\classes
-classpath .\build -d .\build DatagramListener.java MsgListener.java
```

If the classes compiled successfully, preverify them using the following command. This command preverifies all classes in the `.\build` directory (the directory where we just compiled the code) and places the preverified version of the classes into an `\output` directory off of the current directory.

```
>e:\midp-fcs\bin\preverify.exe -classpath
e:\midp-fcs\classes;.\build .\build
```

Use the following to JAR the application into a file named `io.jar`.

```
jar cvf io.jar -C output .
```

Now we are ready to run both the client and the listener applications and send datagrams back and forth. Remember to set the `ENABLE_CLDC_PROTOCOLS` environment variable before running the applications. Without this variable set, the emulator will report that the datagram protocol is unavailable.

Since we are running two applications that need to communicate with each other, you will need to run each application from a separate command window. Use the following commands to run the applications. The example to run the listener is shown first.

```
>set ENABLE_CLDC_PROTOCOLS=true

>e:\midp-fcs\bin\midp.exe -classpath e:\midp-fcs\classes;.io.jar
com.ctimn.MsgListener
```



Figure 13.10
MsgListener menu.

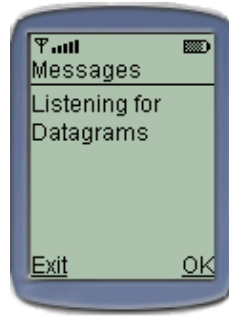


Figure 13.11
Listening for a Datagram.

The menu displays the options for listening. Choosing the second option, “Datagram Listener” invokes our code that binds to port 5555 and listens for an incoming datagram. Examples of running the datagram listening service from `MsgListener` are shown in figures 13.10 and 13.11.

With the `DatagramListener` running from within the `MsgListener` application we are ready to run the `MsgClient` application and send a datagram message. Run the `MsgClient` from the other window using the following commands. When the main menu appears, choose the second option, “Datagram Message”.

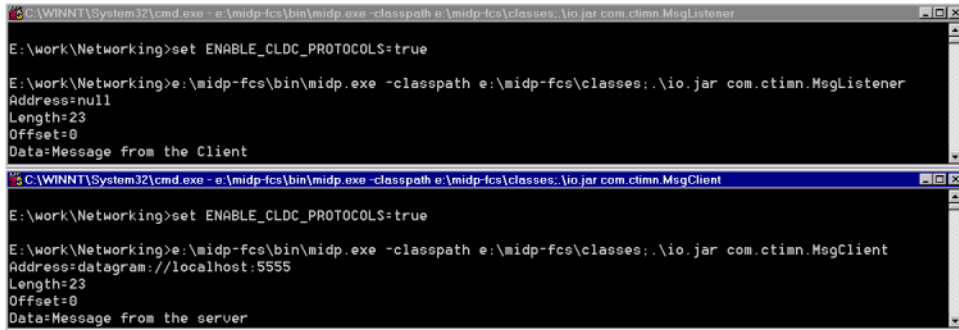
```
>set ENABLE_CLDC_PROTOCOLS=true
>e:\midp-fcs\bin\midp.exe -classpath e:\midp-fcs\classes;.io.jar
com.ctimn.MsgClient
```

Selecting the “Datagram Message” option sends a message to the listener. The listener then displays the message and responds with a message for the client. The results of sending and receiving messages using the `MsgClient` and `MsgListener` are shown in figure 13.12.

Additional information is shown on the command line as messages are sent back and forth. The output from the command windows is shown in figure 13.13.



Figure 13.12 Running the `MsgClient` and `MsgListener` passing datagrams. The `MsgListener` waits for activity on a specified port. When a datagram is sent from the `MsgClient`, the `MsgListener` receives the datagram and responds appropriately.



```
C:\WINNT\System32\cmd.exe - e:\midp-fcs\bin\midp.exe -classpath e:\midp-fcs\classes;.io.jar com.ctimn.MsgListener
E:\work\Networking>set ENABLE_CLDC_PROTOCOLS=true
E:\work\Networking>e:\midp-fcs\bin\midp.exe -classpath e:\midp-fcs\classes;.io.jar com.ctimn.MsgListener
Address=null
Length=23
Offset=0
Data:Message from the Client

C:\WINNT\System32\cmd.exe - e:\midp-fcs\bin\midp.exe -classpath e:\midp-fcs\classes;.io.jar com.ctimn.MsgClient
E:\work\Networking>set ENABLE_CLDC_PROTOCOLS=true
E:\work\Networking>e:\midp-fcs\bin\midp.exe -classpath e:\midp-fcs\classes;.io.jar com.ctimn.MsgClient
Address=datagram://localhost:5555
Length=23
Offset=0
Data:Message from the server
```

Figure 13.13 Command line results from running the MsgListener and MsgClient.

13.6 SUMMARY

This chapter explored the Generic Connection Framework in detail. The Connector class is the factory that allows vendor-specific implementations of each GCF connection to be instantiated at runtime. By using a factory to instantiate the actual classes, the GCF can effectively abstract the device-dependent aspects of establishing network connections. This architecture greatly enhances portability across applications that make use of the network since the connection implementations are allowed to vary but the interface remains consistent.

Examples were provided using common network connections such as HTTP, sockets and datagrams. In the case of sockets and datagrams, multiple threads were used to monitor a port for incoming messages in the background so that the use of the device would not be affected.

Since J2ME provides support for HTTP connections, J2ME clients can connect to any server environment that understands HTTP and is not restricted to interacting with Java services. As a result, a J2ME client can be used with non-Java technologies such as ASP and CGI.