FORUM NOKIA

Introduction To Developing Networked MIDlets Using Bluetooth

Version 1.0; May 11, 2004

Java™

NOKIA

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

Change History

May 11, 2004	Version 1.0	Initial document release

1 Introduction

1.1 Document Scope and Purpose

Bluetooth enables “ad hoc” networks to be formed dynamically between Bluetooth-capable devices. This document presents an overview of the *Java APIs For Bluetooth* [JSR-82] and their use in an example MIDlet application.

The writers of this document assume that you are already familiar with Java™ programming and the basics of Mobile Information Device Profile (MIDP) programming, for example by having read the Forum Nokia document *MIDP 1.0: Brief Introduction To MIDlet Programming* [MIDPPROG]. It will also help if you are somewhat familiar with the basics of Bluetooth [BTSPEC] and the *Java APIs For Bluetooth*. The Forum Nokia document *Games Over Bluetooth: Recommendations To Game Developers* [BTGAMES] provides a general introduction to the subject and is recommended background reading.

This document will help a Java 2 Platform, Micro Edition (J2ME™) developer become familiar with the following concepts from the *Java APIs For Bluetooth*:

- Device inquiry
- Name discovery
- Service discovery
- Bluetooth master/slave roles
- Universally Unique Identifiers
- L2CAP channels and packets
- RFCOMM connections
- Authentication, encryption, and authorization

This document does not cover other features described in the *Java APIs For Bluetooth*, such as the Object Exchange (OBEX) protocol.

Within this document, the *Java APIs For Bluetooth* are sometimes referred to simply as “JSR-82.”

1.2 Background

This document does not give a general introduction to Bluetooth networking. Chapter 2 of the Forum Nokia document *Games Over Bluetooth: Recommendations To Game Developers* provides a good introduction to many Bluetooth concepts [BTGAMES]. It also presents an overview of other important topics such as latency and is recommended background reading.

1.2.1 Bluetooth protocol stack

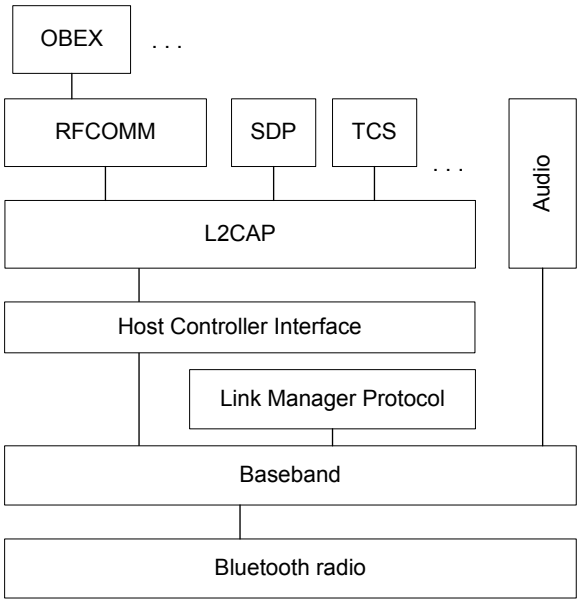

Figure 1.1 Bluetooth protocol stack

The Bluetooth specifications define the functionality of the Bluetooth protocol stack. The initial features of the protocol stack were based on known needs and usage models such as support for audio headsets, object transfer (that is, business cards, calendar entries, etc.), dial-up networking, file transfer, and serial port connections. More features are likely to be added in the future.

This document mainly discusses Bluetooth RFCOMM, Logical Link Control and Adaptation Protocol (L2CAP), and Service Discovery Protocol (SDP) in relation to the *Java APIs For Bluetooth*. It does not discuss these protocols more generally.

1.2.2 Device inquiry

Bluetooth devices can create dynamic ad hoc networks. To accomplish this, devices must be able to discover other nearby devices, a process referred to as *device inquiry*. A discovered device is identified by its Bluetooth device address.

For reasons of privacy, a Bluetooth device can have various settings for its "discoverable mode." For example, a device may be configured to not be discoverable. In this case, other Bluetooth devices that are within range won't detect it.

Alternatively, a Bluetooth device may be configured to be generally discoverable by other Bluetooth devices. In this case, the discoverable mode will be set using the General Unlimited Inquiry Access Code (GIAC).

A Bluetooth device may also be configured to be discoverable in a "limited" manner by other Bluetooth devices by using a limited inquiry. In this case, the discoverable mode is set using the Limited Dedicated Inquiry Access Code (LIAC). An analogy can help to explain LIAC use. Using LIAC as the discoverable mode is a bit like briefly sticking your head above the crowd in a crowded (with GIAC devices) area, in order to be temporarily more visible to someone who is trying to find you.

For a more detailed technical explanation about GIAC and LIAC, please refer to the Bluetooth specifications.

1.2.3 Name discovery

In addition to a Bluetooth address, a device may also have a "friendly name," which is easier to remember. *Name discovery* refers to the process of learning a Bluetooth device's friendly name.

1.2.4 Service discovery

Device inquiry is not enough to use a nearby service. Applications must also be able to find services hosted on nearby Bluetooth devices. This process is referred to as *service discovery*, and the Bluetooth Service Discovery Protocol (SDP) is used for this purpose. Once a desired service has been discovered on a nearby device, an application may attempt to connect to the service and use it.

1.2.5 Bluetooth master/slave roles

For every Bluetooth link between two devices, one device is defined as the *master* of the connection and the other as the *slave*. These terms are defined by the Bluetooth specifications.

Frequency hopping is used on Bluetooth links for reasons such as security, avoiding interference, etc. The Bluetooth specification defines that the master's clock and Bluetooth device (BD) address are used to calculate the frequency hopping sequence, and that slaves for Asynchronous Connectionless (ACL) links are allowed to start sending in odd slots of the clock when they have been addressed by the master in a previous slot. Thus, the master BD address and clock define the hopping sequence, and the master's transmission defines which slave is allowed to send in which time slot.

Some devices may support a feature referred to as "master/slave switching." For example, a device that is currently a slave may request to become the master. However not all devices support this feature. (See the local device property "`bluetooth.master.switch`" defined in JSR-82.)

If you are using a Bluetooth connection between just two devices, it probably doesn't matter which is the Bluetooth master or slave. However if you wish to connect more than two devices together in the same session (for example, a multiplayer game with more than two players), then it is likely that you will have to consider how the Bluetooth master/slave roles impact connection setup between networked peers. See Section 1.3, A Note About Bluetooth Client/Server, for more details.

1.2.6 UUID

A Universally Unique Identifier (UUID) is a 128-bit value that is guaranteed to be unique across all space and time [UUID]. UUIDs are used in the Bluetooth SDP to identify services. Some UUIDs are defined by the Bluetooth specification for specific protocols or uses.

You will need to generate unique application-specific UUIDs for your application(s).

A variety of different ways of generating UUIDs are possible. The application-specific UUIDs used in the example MIDlets presented in this paper were generated using the "`uuidgen`" utility of a local Linux-based computer. Similar tools are also available for use in Windows and other environments. You can also generate UUIDs manually, for example as described in the Forum Nokia document *Games Over Bluetooth: Recommendations To Game Developers* [BTGAMES].

1.2.7 L2CAP

The Logical Link Control and Adaptation Protocol (L2CAP) is a packet-oriented multiplexing layer that was designed to allow several different higher-level protocols (for example, SDP and RFCOMM) to use Bluetooth communication. All data-oriented protocols in Bluetooth are built on top of L2CAP.

L2CAP is the lowest-level Bluetooth communication protocol that can be used by a MIDlet application. You might use the L2CAP protocol in your application if it needs to control which bytes are sent

together in a single L2CAP packet. (If you use L2CAP, you will need to understand about L2CAP channel configuration, packet payload sizes, and MTU rules. The *Java APIs For Bluetooth* specification explains the MTU rules in more detail.)

L2CAP does not provide flow control by itself. It is up to protocols used above L2CAP to provide flow control if needed, for example so that data is not lost. This becomes important when large amounts of data are transmitted.

1.2.8 RFCOMM

The Bluetooth Serial Port Profile (SPP) is used for creating an RFCOMM connection between two devices. The RFCOMM protocol emulates a serial port connection and provides support for flow control. It is used for stream-oriented connections.

1.2.9 Authentication, encryption, and authorization

Bluetooth provides support for security in the form of device pairing, device authentication, communication encryption, and authorization.

When two Bluetooth devices come into contact with each other for the first time, they may establish a shared secret. The creation of a shared secret is called “pairing,” and it usually requires the users of both devices to enter a shared secret code in both devices using an appropriate user interface. After the pairing has been done successfully, the shared secret may be stored by each device and may be used for any future device authentication between the two devices.

Applications running on authenticated devices may optionally use encryption to prevent eavesdroppers from listening in on the shared communication between applications.

Another security option is called authorization. It is used on a per-connection basis to grant permission for a connection from another Bluetooth device. A trusted device is a device that always passes authorization and may connect to any service on the local device. A device that is not trusted usually requires the user to accept the connection.

Each Bluetooth security level requires the previous underlying security level. Device authentication requires device pairing. Encryption requires device authentication.

The *Java APIs For Bluetooth* specification defines how authentication, encryption, and authorization are specified as connection string parameters used by an application to open a client-role or server-role connection. The authorization parameter is only used when opening server-role connections and not for client-role connections. Appropriate exceptions are thrown if the parameters are used in an invalid manner (for example, “`encrypt=true`” and “`authenticate=false`”) in a request to open a specific connection.

1.3 A Note About Bluetooth Client/Server

Section 1.2.5, “Bluetooth master/slave roles,” briefly presented the master and slave device roles in Bluetooth. It is important to remember that the master defines the access and timing of a Bluetooth piconet. This is especially important when more than two Bluetooth devices are going to be interconnected.

In many traditional client/server architectures, the server starts running first and waits indefinitely to accept new client connection requests from remote clients.

In a Bluetooth piconet, the master adds each slave into the piconet. A common result of this is that client applications will be slaves and the server application will be the master. From the user’s point of view, the server (master) "initiates" connections to the clients (slaves). This is important to keep in

mind. (Figure 3.1 will also help explain some of the related design consequences of this for your MIDlet.)

Chapter 3 of the Forum Nokia document *Games Over Bluetooth: Recommendations To Game Developers* [BTGAMES] presents a good overview of some general steps that are useful in creating a connection between client/server applications in Bluetooth. The discussion of those steps is not repeated here.

2 Java APIs for Bluetooth Networking: A Brief Tour

2.1 Introduction

This section presents a brief overview of some of the main interfaces and classes from JSR-82 and their uses for networking.

2.2 Device Inquiry

The first step in a networked Bluetooth application is often to discover other devices that are in range. (Also see the discussion of using preknown or cached devices at the end of this section.)

A MIDlet application can use either the GIAC or the LIAC form of inquiry for device inquiry. A general inquiry is used to find all nearby GIAC-discoverable devices for an unlimited amount of time.

An alternative form is to use the limited form of inquiry. The *Java APIs For Bluetooth* specification may give the impression that when LIAC is used, the device is only discoverable for a limited amount of time (for example, one minute) and the device will automatically revert back to its previous discoverable mode after that time has expired. However when LIAC is used in Nokia's devices, it should remain in use until the MIDlet changes this (for example, to GIAC) or until the MIDlet terminates. This approach is easier for developers and end users to understand and utilize. Use of LIAC for the device inquiry phase can be especially useful when there are large numbers of GIAC-discoverable devices nearby. See Section 1.2.2 for more details.

The `LocalDevice` class methods `getDiscoverable` and `setDiscoverable` are used to get and set the discoverable mode for a device.

The `DiscoveryAgent` class provides the methods needed for performing device inquiry. Device inquiry is performed using the method `startInquiry`. Callbacks are made to an appropriate handler that implements the `DiscoveryListener` interface, via the methods `deviceDiscovered` (for each device found) and `inquiryComplete` (when the device inquiry process has ended).

Your application should also perform filtering of found devices based on their Class of Device (COD), before performing a service search on them. You will want to perform service searches only on found devices whose COD makes sense for the service. For example, if you are writing a game MIDlet, you can skip doing a service search on all the headsets, car kits, digital pens, and probably most laptops that are found[1]. The `DiscoveryListener` interface's callback method `deviceDiscovered` has a parameter called `DeviceClass` that can be used for such filtering.

As an alternative to performing device inquiry, an application might use a list of preknown or cached devices. The method `retrieveDevices` of the `DiscoveryAgent` class can be used to request a list of known `RemoteDevice` objects. However the method `retrieveDevices` does not provide the COD, nor does class `RemoteDevice` provide that information. (See the previous paragraph about the importance of filtering devices based on their COD.)

For the sake of simplicity, the MIDlet example presented in this document only performs device inquiry.

[1] In addition, many Bluetooth devices have link level security turned on, which requires pairing with a device even to perform a service search. (See the discussion about pairing in Section 1.2.9.) If COD filtering isn't used beforehand to limit service searches to an appropriate set of possible devices, the subsequent pairing request(s) that might arise could result in an unpleasant UI experience for an end user.

2.3 Name Discovery

A MIDlet application can use the method `getFriendlyName(boolean alwaysAsk)` of class `RemoteDevice` to perform name discovery. The `getFriendlyName` method may contact the device if the name is not known or if the parameter `alwaysAsk` is set to a value of "`true`". The associated delay may be nonzero, and could be cumulatively noticeable when performed for a large number of devices that are in range.

It may also be worth noting that it is possible for different phones to have identical friendly names (for example, "Nokia 6600," "My phone," "Jim's phone," etc.). Since many people enjoy uniquely personalizing their devices, one can assume that many phones will have unique friendly names.

The MIDlet example presented in this document doesn't use name discovery. Instead, only the unique Bluetooth device address is used to identify devices.

2.4 Service Discovery

The next step in a networked Bluetooth application is to discover which nearby devices support the service(s) that the user is interested in using.

Your application will have a particular UUID that you have associated with it. The following snippet of code helps to illustrate.

```
// UUID for this service:
String uuidString = "50FDB90ADBFB49b3AA71D6BA308E45F3";
String params = …; // set optional parameters as needed
// an RFCOMM (BTSPP) based service:
String url = "btspp://localhost:" + uuidString + params;
try
{
    StreamConnectionNotifier connectionNotifier =
       (StreamConnectionNotifier) Connector.open(url);
    StreamConnection connection =
       (StreamConnection) connectionNotifier.acceptAndOpen();
}
// send or receive messages to remote peer, etc.
```

A remote peer discovers the service by doing a service discovery on one or more found devices using the method `searchServices` in class `DiscoveryAgent`:

```
    public int searchServices(int[] attrSet,
                              UUID[] uuidSet,
                              RemoteDevice btDev,
                              DiscoveryListener discListener)
        throws BluetoothStateException
```

The input parameters of the method are the set of service attributes to be retrieved for matching services, the set of UUIDs that define a matching service, the device of interest for service discovery, and a listener for handling the appropriate discovery callbacks.

The application will normally perform service discovery on each found device of interest. The *Java APIs For Bluetooth* allow multiple simultaneous service discovery transactions. The maximum number of concurrent service discoveries allowed is defined by the property "`bluetooth.sd.trans.max`" (see JSR-82). An appropriate `BluetoothStateException` will be thrown if the application attempts to start more service discovery transactions than allowed. JSR-82 does not specify how the underlying implementation performs such transactions (for example, concurrently or sequentially). See [BTGAMES] for related information for Nokia devices.

Service discovery callbacks are made to an appropriate listener. The listener implements the `DiscoveryListener` interface, which defines callback methods `servicesDiscovered` and `serviceSearchCompleted`.

2.5 RFCOMM Connections

RFCOMM connections provide reliable, bidirectional, stream-oriented communication. In JSR-82, the API is based on the `StreamConnectionNotifier` and `StreamConnection` interfaces of the Generic Connection Framework defined for Connected Limited Device Configuration (CLDC) [CLDC].

An RFCOMM server is created by calling `Connector.open()` with an appropriate server connection string. The format of the string is defined by JSR-82. The connector string begins with the protocol prefix "`btspp://`" and may contain parameters related to a master/slave preference, or security parameters (authentication, encryption, and authorization). A `StreamConnectionNotifier` object is returned when opening a server connection, and its `acceptAndOpen()` method can be used to accept connections from remote clients. Input and output streams are used to read or write data on a connection.

RFCOMM clients are created in a similar way with method `Connector.open()` using a client connection string.

2.6 L2CAP Connections

JSR-82 defines the `L2CAPConnectionNotifier` and `L2CAPConnection` interfaces for sending and receiving packets over L2CAP channels. These are derived from the CLDC Generic Connection Framework interface `Connection`.

An L2CAP server is created by calling `Connector.open()` with an appropriate server connection string. The format of the string is defined by JSR-82. The connector string begins with the protocol prefix "`btl2cap://`" and may contain parameters related to a master/slave preference, or security parameters (authentication, encryption, and authorization). It may also contain parameters related to the application's preference for a desired Maximum Transmit Unit (MTU) size in the send and/or receive directions.[2] An `L2CAPConnectionNotifier` object is returned when opening a server connection. The method `acceptAndOpen` can be used to accept connections from remote clients.

L2CAP packets are sent or received using the `send` and `receive` methods of class L2CAPConnection. An application using L2CAP connections must provide its own flow control if needed.

L2CAP clients are created in a similar way with method `Connector.open()` using a client connection string.

[2] Section 10.2.1.1 of JSR-82 defines specific rules related to MTU size, for example when a connection is created. It is important to be familiar with these if you intend to use an `L2CAPConnection` in your application.

3 An Example MIDlet Using RFCOMM

3.1 MIDlet Design

3.1.1 Brief overview

The example MIDlet presented in this document implements a simple client/server pair:

- The server MIDlet receives messages from any connected client MIDlet. The server "echoes" received message back to all clients (including the original sender). This allows the user of a client MIDlet to send a message to all clients connected to a server.
- The user of the server MIDlet can send a message to all clients connected to the server.

The following figure illustrates various roles of the client and server MIDlet applications at different levels.

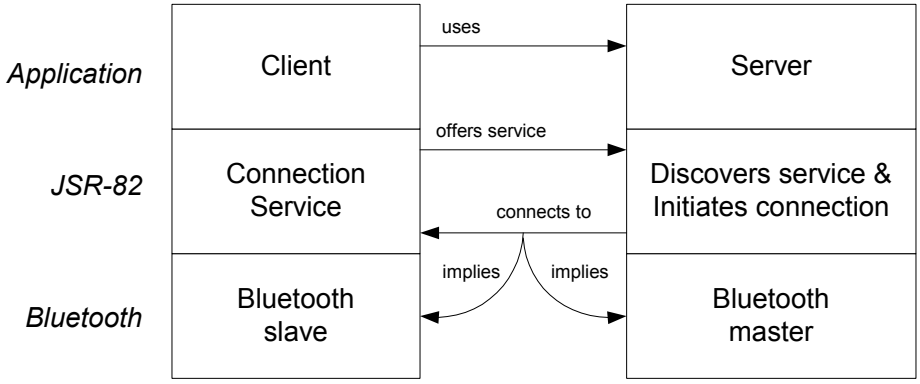

Figure 3.1 Different roles of the client/server application pair (at various levels)

Note: In the example MIDlet presented in this document, we have chosen to name the “client” and “server” MIDlets based on their application level (client and server) roles and relationship. This naming convention has the advantage of being independent of the network connectivity used (for example, if one someday changes a MIDlet to use an alternative network bearer, the client/server relationship and related naming convention do not necessarily have to change). This convention may feel a bit more “natural” for some types of applications (for example, multiplayer games over Bluetooth). However, it is worth noting that one can also easily find examples of other Bluetooth based applications, where the naming convention for the client/server pair has instead been made at the JSR-82 / Bluetooth connectivity level. Using the latter approach would reverse the naming convention (of which MIDlet is called the “client” or “server”) when compared the convention used in this example. In the design phase for your own MIDlets, this may be one topic to consider. Since you might choose either approach depending on your point-of-view, it may be useful (to avoid confusion) to clarify in your design documents why a particular approach and terminology was chosen.

3.1.2 Creating connections

Several "client" instances of the MIDlet are started in different Bluetooth MIDP devices. (These will be Bluetooth slaves.) Each client MIDlet waits to accept a connection from an appropriate remote "server" MIDlet.

The server MIDlet first performs a device inquiry. When the device inquiry has completed, the server has a list of possible nearby Bluetooth devices. It then performs service discovery on found devices having a suitable COD (that is, mobile phones). This allows the server to find reachable nearby clients.

After the service discovery phase has completed, the user of the server MIDlet is presented with a list of devices (having the right COD) where the client is available. The server MIDlet next initiates a connection to each client selected by the end user. The user of the server MIDlet selects which client to initiate a connection to and the order in which connections are created (that is, the connections are created from the server to clients, one by one). The server is the Bluetooth master for each connection.

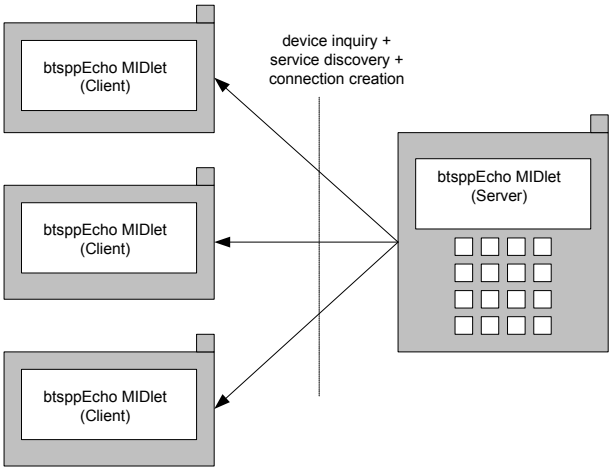

Figure 3.2 Device inquiry, service discovery, and connection creation

3.1.3 Client/server communication

A connected client MIDlet may send a message to its server MIDlet. The server simply echoes the message back to all connected clients. The message is also echoed back to the original sending client (just to keep the server MIDlet code trivially simple). This allows a client to send a message to all connected clients via the server.

Another alternative is that the server sends a "non-echo" message to all connected clients.

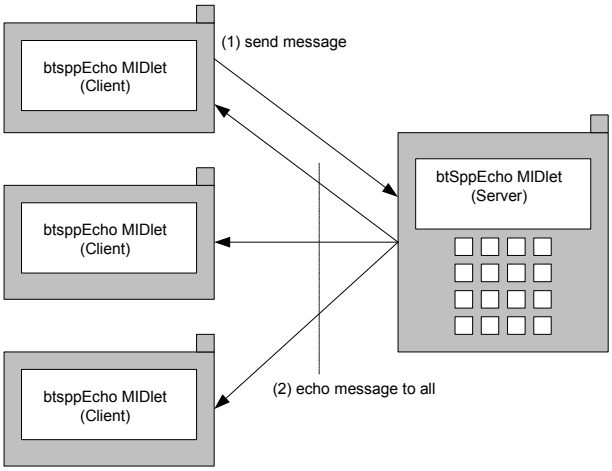

Figure 3.3 A client sends a message and the server sends that message to all clients

3.1.4 Dropping connections

If a client MIDlet exits or otherwise disconnects, the server remains connected to and in communication with the remaining clients. However, if the server MIDlet exits or otherwise disconnects, then all communication is lost between the clients (that is, all connections to the server are closed).

3.2 User Interface Design

The MIDlet can be started in either the client or server mode in the initial `SettingsList` screen. The `SettingsList` screen lets user choose the discoverable mode (server) or discovery mode (client) used for device inquiry, and if the general or limited inquiry access codes are used (that is, GIAC or LIAC). It also lets the user choose appropriate security settings (that is, authentication, encryption, and authorization).

The `SettingsList` screen also has a "Bluetooth properties" option to allow the user to examine the Bluetooth properties of the local device.

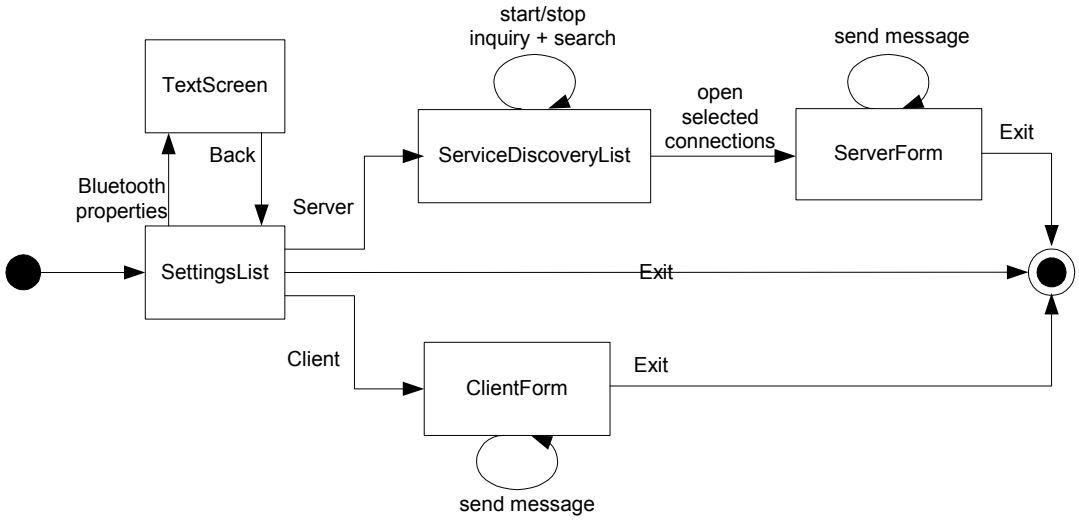

Figure 3.4 Screen map

When the MIDlet is started in client mode, the screen map transitions from the initial `SettingsList` screen to a `ClientForm`. The `ClientForm` uses a class called `ConnectionServer` to accept and open incoming connections from the remote server. Once a connection has been set up, the `ClientForm` allows the user to send messages to the server. It also displays any messages received from the server.

When the MIDlet is started in server mode, the screen map changes from the initial `SettingsList` screen to a `ServiceDiscoveryList`. It is used to initiate a device inquiry, and subsequent service discovery on any found devices that have a suitable COD. As matches (that is, found devices with a suitable COD where the client is running) are discovered, these are displayed to the user as elements of the `List`. The user can next choose to open a connection from the server to a client. When the connection has been created, the screen map changes to a `ServerForm` screen.

The `ServerForm` is used to display a text message received from any client and to allow the user to send a simple text message to all connected clients. It also displays how many clients are currently connected, and allows a new connection to be added by a screen transition back to the `ServiceDiscoveryList`. This is used to select and open the next connection to some other as-yet-unconnected client.

3.3 Class Diagram

The following two diagrams illustrate the class diagram for the MIDlet design. The JAR file for the MIDlet contains all the classes needed for it to be run in either client or server mode. (If needed, the MIDlet could easily be split into two separate MIDlets for each role.) The MIDlet's design is more clearly illustrated by showing the classes used in each of these modes separately.

3.3.1 Client

When the MIDlet first starts, it displays a `SettingsListScreen` that is used to set important settings for the MIDlet, and to start it in either server or client mode.

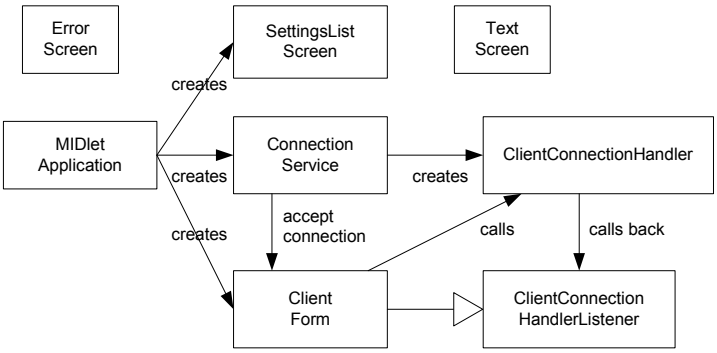

Figure 3.5 Class diagram of classes used by the client

When the MIDlet is started in client mode, it next creates an instance of a `ConnectionService` object and displays the `ClientForm` screen.

The `ConnectionService` runs continuously, waiting to accept new connections on the appropriate server connection string. When it accepts a new connection from a remote server, it notifies the `ClientForm` that it has accepted a new connection and created a new `ClientConnectionHandler` to handle that connection. A `ClientConnectionHandler` is runnable, so it may read and write from the appropriate input and output streams. The `ClientForm` starts the thread for each accepted `ClientConnectionHandler`. This causes the latter to instantiate the actual `InputStream` and `OutputStream` objects needed, and to start both a reader and a writer thread for them.

The `ClientForm` is a `ClientConnectionHandlerListener`, and so accepts callbacks from its `ClientConnectionHandler` objects related to opening the input and output streams, received messages, sent messages, and normal or abnormal connection closes. The `ClientForm` directly uses a `ClientConnectionHandler` to send messages.

3.3.2 Server

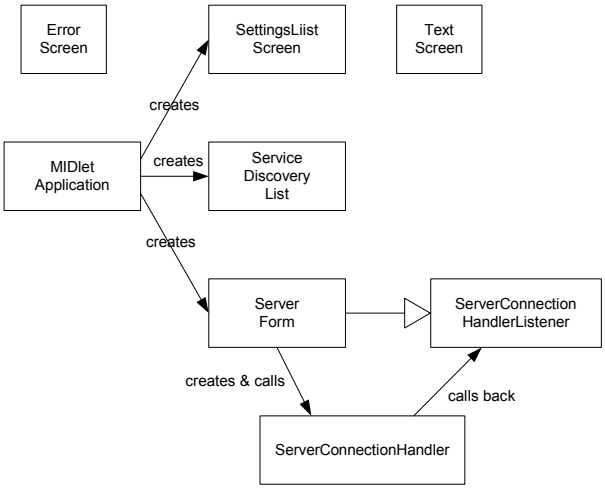

Figure 3.6 Class diagram of classes use by the server

When the MIDlet is started in server mode, it next changes the screen map to the `ServiceDiscoveryList`. From this screen, the user may begin a “search” (device inquiry plus

service searches). That phase returns matching devices and services. The end user may then select to open the first connection to a remote client, and the screen map is changed to the `ServerForm` screen.

The ServerForm screen creates the actual connection to a remote client, and maintains the list of all `ServerConnectionHandler` objects. (One is created per created connection.) The `ServerConnectionHandler` objects are runnable, and separate threads are used in each such object for reading and writing to input and output streams. The `ServerForm` is responsible for starting these threads when it creates a new `ServerConnectionHandler` object.

The `ServerForm` implements the interface `ServerConnectionHandlerListener`, and so accepts callbacks from its `ServerConnectionHandler` objects. The callbacks are related to opening the input and output streams, received messages, sent messages, and normal or abnormal connection closes. The `ServerForm` directly uses a `ServerConnectionHandler` to send messages.

The `ServerForm` also has an "Add connection" command, which temporarily causes a screen transition back to the `ServiceDiscoveryList` screen to select the next client to open a connection to.

3.4 MIDletApplication

The `MIDletApplication` class handles the MIDlet state model callbacks. The user interface is delegated to appropriate screens. The MIDlet provides a central "state model" for the screens, so that each screen calls back to the MIDlet and the MIDlet displays the appropriate next screen.

The UUID value was generated for this particular application and uniquely identifies it. For this MIDlet, it is worth noting that the UUID identifies the `ConnectionService` of the client. This is because the server initiates connections to the clients.

```
package example.btsppecho;


import java.io.IOException;
import java.util.Vector;
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.bluetooth.BluetoothStateException;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.ServiceRecord;

import example.btsppecho.client.ConnectionService;


public class MIDletApplication
    extends MIDlet
{
    private final static String UUID =
                                "A55665EE9F9146109085C2055C888B39";
    private final static String SERVICE_URL =
                                "btspp://localhost:" + UUID;
    private final SettingsList settingsList;
    private boolean restoreDiscoverableModeOnExit;
    private int initialDiscoverableMode;

    // Client server
    private ConnectionService ConnectionService = null;
    private ClientForm clientForm = null;

    // Server client
    private ServiceDiscoveryList serviceDiscoveryList = null;
    private ServerForm serverForm = null;
```

```
boolean serverUseAuthentication = false;
boolean serverUseEncryption = false;


public MIDletApplication()
{
    // Try to read the initial discoverable mode of
    // device, so it can be restored on exit.
    try
    {
        restoreDiscoverableModeOnExit = true;
        initialDiscoverableMode =
            LocalDevice.getLocalDevice()
                       .getDiscoverable();
    }
    catch (BluetoothStateException e)
    {
        restoreDiscoverableModeOnExit = false;
    }

    settingsList = new SettingsList(this);
}


private void exit()
{
    destroyApp(false);
    notifyDestroyed();
}


public void startApp()
{
    Display display = Display.getDisplay(this);
    display.setCurrent(settingsList);
    ErrorScreen.init(null, display);
}


public void pauseApp()
{
    // we can ignore this
}


public void destroyApp(boolean unconditional)
{
    // stop any networking, service discovery, etc.
    // threads that the MIDlet may have started

    if (serviceDiscoveryList != null)
    {
        serviceDiscoveryList.cancelPendingSearches();
        serviceDiscoveryList.abort();
    }

    if (ConnectionService != null)
    {
        ConnectionService.close();
    }

    if (clientForm != null)
    {
        clientForm.closeAll();
    }

    if (serverForm != null)
    {
        serverForm.closeAll();
```

```
        }

        // I restore the discoverable mode to the initial
        // value on exit, so the behaviour of this MIDlet
        // might be more similar in this respect on all
        // vendors' devices. (You might want to rethink
        // this in your MIDlet, for example if a device
        // allows you to run multiple simultaneous Bluetooth
        // applications, each having varying start &
        // exit times.)
        if (restoreDiscoverableModeOnExit)
        {
            try
            {
                LocalDevice ld = LocalDevice.getLocalDevice();
                ld.setDiscoverable(initialDiscoverableMode);
            }
            catch (BluetoothStateException e)
            {
                // there is nothing we can do
                // to handle this case: ignore it
            }
        }
    }


    // screen callbacks

    // ClientForm

    public void clientFormExitRequest()
    {
        exit();
    }


    public void clientFormViewLog(Displayable next)
    {
       LogScreen logScreen =
           new LogScreen(this,
                         next,
                         "Log",
                         "Back");
       Display.getDisplay(this).setCurrent(logScreen);
    }


    // SettingsList callbacks

    public void settingsListStart(boolean isServer,
                                  int inquiryAccessCode,
                                  boolean useAuthentication,
                                  boolean useAuthorization,
                                  boolean useEncryption)
    {
        // set inquiry access mode for ConnectionService
        if (!isServer)
        {
            try
            {
                LocalDevice.getLocalDevice()
                           .setDiscoverable(inquiryAccessCode);
            }
            catch(BluetoothStateException e)
            {
                String msg = "Error changing inquiry access type: " +
                             e.getMessage();
                ErrorScreen.showError(msg, settingsList);
            }
```

```
        }

        if (isServer)
        {
            // start application in server role

            // we only run one server at a time,
            // so the following is safe
            serverUseAuthentication = useAuthentication;
            serverUseEncryption = useEncryption;
            serviceDiscoveryList =
                new ServiceDiscoveryList(
                        this,
                        UUID,
                        inquiryAccessCode);
            Display.getDisplay(this)
                   .setCurrent(serviceDiscoveryList);
        }
        else
        {
            // start application in client role

            clientForm = new ClientForm(this);
            String url = SERVICE_URL;
            if (useAuthentication)
            {
                url += ";authenticate=true";
            }
            else
            {
                url += ";authenticate=false";
            }
            if (useAuthorization)
            {
                url += ";authorize=true";
            }
            else
            {
                url += ";authorize=false";
            }
            if (useEncryption)
            {
                url += ";encrypt=true";
            }
            else
            {
                url += ";encrypt=false";
            }

            url += ";name=btsppEcho";

            ConnectionService = new ConnectionService(url, clientForm);

            Display.getDisplay(this).setCurrent(clientForm);
        }
    }


    public void settingsListPropertiesRequest()
    {
        String[] keys =
        {
            "bluetooth.api.version",
            "bluetooth.connected.devices.max",
            "bluetooth.connected.inquiry",
            "bluetooth.connected.inquiry.scan",
            "bluetooth.connected.page",
            "bluetooth.connected.page.scan",
            "bluetooth.l2cap.receiveMTU.max",
```

```
            "bluetooth.master.switch",
            "bluetooth.sd.attr.retrievable.max",
            "bluetooth.sd.trans.max",
        };

        String str = "";
        try
        {
          str = "my bluetooth address: " +
                LocalDevice.getLocalDevice()
                           .getBluetoothAddress() + "\n";
        }
        catch (BluetoothStateException e)
        {
            // there is nothing we can do: ignore it
        }
        for (int i=0; i < keys.length; i++)
        {
           str += keys[i] + ": " +
                  LocalDevice.getProperty(keys[i]) + "\n";
        }

        TextScreen textScreen =
            new TextScreen(this,
                           settingsList,
                           "Device properties",
                           str,
                           "Back");
        Display.getDisplay(this).setCurrent(textScreen);
    }


    public void settingsListExitRequest()
    {
        exit();
    }


    // ServiceDiscoveryList callbacks

    public void serviceDiscoveryListFatalError(String errorMessage)
    {
        ErrorScreen.showError(errorMessage, serviceDiscoveryList);
        Display.getDisplay(this).setCurrent(settingsList);
    }


    public void serviceDiscoveryListError(String errorMessage)
    {
        ErrorScreen.showError(errorMessage, serviceDiscoveryList);
    }


    public void serviceDiscoveryListOpen(Vector selectedServiceRecords)
    {
        int security;
        if (serverUseAuthentication)
        {
           if (serverUseEncryption)
           {
               security = ServiceRecord.AUTHENTICATE_ENCRYPT;
           }
           else
           {
               security = ServiceRecord.AUTHENTICATE_NOENCRYPT;
           }
        }
        else
        {
```

```
            security = ServiceRecord.NOAUTHENTICATE_NOENCRYPT;
        }

        if (serverForm == null)
        {
            serverForm = new ServerForm(this);
        }
        serverForm.makeConnections(selectedServiceRecords, security);
        Display.getDisplay(this).setCurrent(serverForm);
    }


    public void serviceDiscoveryListExitRequest()
    {
        exit();
    }


    public void serviceDiscoveryListBackRequest(Displayable next)
    {
        Display.getDisplay(this).setCurrent(next);
    }


    public void serviceDiscoveryListViewLog(Displayable next)
    {
        LogScreen logScreen =
            new LogScreen(this,
                          next,
                          "Log",
                          "Back");
        Display.getDisplay(this).setCurrent(logScreen);
    }


    // TextScreen

    public void textScreenClosed(Displayable next)
    {
        Display.getDisplay(this).setCurrent(next);
    }


    // LogScreen

    public void logScreenClosed(Displayable next)
    {
        Display.getDisplay(this).setCurrent(next);
    }


    // ServerForm

    public void serverFormSearchRequest(int numConnectionsOpen)
    {
         // cleanup for new search
        serviceDiscoveryList.init(numConnectionsOpen);

        if (numConnectionsOpen > 0)
        {
            serviceDiscoveryList.addBackCommand(serverForm);
        }
        Display.getDisplay(this).setCurrent(serviceDiscoveryList);
    }


    public void serverFormExitRequest()
    {
        exit();
```

```
        }


        public void serverFormAddConnection(Vector alreadyOpen)
        {
            // I took a simple approach of simply changing the
            // screen to the ServiceDiscovery screen when the
            // user wants to try and add a new connection, or
            // perform both a new device inquiry + service search
            // and then add more connections.
            //
            // However, reality can be a bit more complicated:
            //   - How many previously discovered items (e.g. device
            //     running the desired service) have we already
            //     connected to, or not connected to?
            //   - How many additional new connections can this device
            //     open below its maximum limit? The maximum number
            //     of simultaneous connections can vary in different
            //     devices (i.e. see "bluetooth.connected.devices.max").
            //   - Can new inquiries/searches be started while
            //     already connected?
            // Depending on your MIDlet's needs + use cases, and
            // the devices it is likely to be deployed in: it
            // might employ a bit more user friendly approach than
            // the simplistic/generic one used here.

            serviceDiscoveryList.remove(alreadyOpen);
            serviceDiscoveryList.addBackCommand(serverForm);
            Display.getDisplay(this).setCurrent(serviceDiscoveryList);
        }


        public void serverFormViewLog()
        {
            LogScreen logScreen =
                new LogScreen(this,
                              serverForm,
                              "Log",
                              "Back");
            Display.getDisplay(this).setCurrent(logScreen);
        }
    }
```

3.5 SettingsList

This is the first screen displayed by the MIDlet. It is used to make a few settings and then start the MIDlet. The settings are as follows:

- An option to select either client or server mode
- The inquiry type used to be discoverable by a client, or when performing discovery by a server
- Use of authentication: true/false
- If authentication is used:
    - Use of encryption: true/false
    - Use of authorization: true/false

The Start command is used to start the MIDlet in the appropriate role and using the appropriate settings. This is done via an appropriate callback to the `MIDletApplication`, since it manages the screen transitions.

There is also a BT Properties command to examine certain system properties related to use of Bluetooth. For example, a Bluetooth device might only allow a limited number of other devices to be connected, etc.

```
package example.btsppecho;


import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.List;

import javax.bluetooth.DiscoveryAgent;


class SettingsList
     extends List
     implements CommandListener
{
    // UI strings
    private static final String SERVER = "Server";
    private static final String CLIENT = "Client";
    // (I abbreviated the strings "Authentication",
    // "Authorization" and "Encryption" because they are a
    // bit long on some MIDP device's List items.  Another
    // MIDlet might hard code its preference anyways.)
    private static final String AUTHENTICATION_TRUE =
                                "Authen.: true";
    private static final String AUTHENTICATION_FALSE =
                                "Authen.: false";
    private static final String AUTHORIZATION_TRUE =
                                "Authoriz.: true";
    private static final String AUTHORIZATION_FALSE =
                                "Authoriz.: false";
    private static final String ENCRYPTION_TRUE =
                                "Encrypt: true";
    private static final String ENCRYPTION_FALSE =
                                "Encrypt: false";
    private static final String RETRIEVE_DEVICES_TRUE =
                                "Use known devices";
    private static final String RETRIEVE_DEVICES_FALSE =
                                "Use inquiry";
    private static final String INQUIRY_TYPE_LIAC = "LIAC";
    private static final String INQUIRY_TYPE_GIAC = "GIAC";
    private static final String INQUIRY_TYPE_NOT_DISCOVERABLE =
                                "not discoverable";
    private static final String INQUIRY_TYPE_CACHED = "cached";
    private static final String INQUIRY_TYPE_PREKNOWN = "preknown";

    // settings
    private int inquiryType;
    private int protocol;
    private boolean isServer;
    private boolean useAuthorization;  // client-only
    private boolean useAuthentication;
    // when useAuthenication is false, useEncryption is also false
    private boolean useEncryption;

    // MIDlet stuff
    private final MIDletApplication midlet;
    private final Command startCommand;
    private final Command propCommand;
    private final Command exitCommand;


    SettingsList(MIDletApplication midlet)
    {
        super("Settings", List.IMPLICIT);
        this.midlet = midlet;

        // default setting values
```

```
        // You should think about what is the preferred
        // default inquiry type used to make your service
        // discoverable. This MIDlet uses LIAC as the default,
        // but you might want to use GIAC.
        inquiryType = DiscoveryAgent.LIAC;

        isServer = false; // client by default
        useAuthentication = false;
        useEncryption = false; // false when auth. not used
        useAuthorization = false; // false when auth. not used
        updateListElements();

        // add screen commands
        startCommand = new Command("Start application",
                                   Command.SCREEN,
                                   0);
        propCommand = new Command("BT properties",
                                  Command.SCREEN,
                                  1);
        exitCommand = new Command("Exit", Command.EXIT, 0);
        addCommand(startCommand);
        addCommand(propCommand);
        addCommand(exitCommand);
        setCommandListener(this);
    }


    private void updateListElements()
    {
        // remove all old list items
        while(size() > 0)
        {
            delete(0);
        }

        // Index 0: Server / Client
        String string;
        if (isServer)
        {
            string = SERVER;
        }
        else
        {
            string = CLIENT;
        }
        append(string, null);

        // Index 3: LIAC / GIAC
        if (inquiryType == DiscoveryAgent.LIAC)
        {
            append(makeInquiryLabel(isServer, INQUIRY_TYPE_LIAC),
                   null);
        }
        else if (inquiryType == DiscoveryAgent.GIAC)
        {
            append(makeInquiryLabel(isServer, INQUIRY_TYPE_GIAC),
                   null);
        }
        else if (inquiryType == DiscoveryAgent.PREKNOWN)
        {
            append(makeInquiryLabel(isServer,
                                    INQUIRY_TYPE_PREKNOWN),
                   null);
        }
        else if (inquiryType == DiscoveryAgent.CACHED)
        {
            append(makeInquiryLabel(isServer, INQUIRY_TYPE_CACHED),
                   null);
```

```
        }
        else if (inquiryType == DiscoveryAgent.NOT_DISCOVERABLE)
        {
            append(makeInquiryLabel(isServer, INQUIRY_TYPE_CACHED),
                   null);
        }


        // Index 2: use authentication true / false
        //          (encryption and authorization can only be
        //           used if authentication is used)
        if (useAuthentication)
        {
            append(AUTHENTICATION_TRUE, null);

            // Index 3: use encryption true / false
            if (useEncryption)
            {
                append(ENCRYPTION_TRUE, null);
            }
            else
            {
               append(ENCRYPTION_FALSE, null);
            }

            // Index 4: ConnectionService only : use auth. true / false
            if (!isServer)
            {
                if (useAuthorization)
                {
                    append(AUTHORIZATION_TRUE, null);
                }
                else
                {
                    append(AUTHORIZATION_FALSE, null);
                }
            }
        }
        else
        {
            useAuthentication = false;
            useEncryption = false;

            append(AUTHENTICATION_FALSE, null);
        }
    }


    private String makeInquiryLabel(boolean searching, String string)
    {
        if (searching)
        {
            // we are searching
            return "Discover: " + string;
        }
        else
        {
            // we will be searched for
            return "Discoverable: " + string;
        }
    }


    public void commandAction(Command command, Displayable d)
    {
        if (command == startCommand)
        {
            midlet.settingsListStart(isServer,
                                     inquiryType,
```

```
                                useAuthentication,
                                useAuthorization,
                                useEncryption);
        }
        else if (command == propCommand)
        {
            midlet.settingsListPropertiesRequest();
        }
        else if (command == exitCommand)
        {
            midlet.settingsListExitRequest();
        }
        else if (command == List.SELECT_COMMAND)
        {
            int index = getSelectedIndex();
            switch(index)
            {
                // Index 0: "Server client" (isServer=true) or
                //          "Client server" (isServer=false)
                case 0:
                  isServer = !isServer;
                  break;


                // Index 1: "Discovery mode: LIAC" or
                //          "Discovery mode: GIAC"
                case 1:
                    // toggle between LIAC and GIAC
                    if (inquiryType == DiscoveryAgent.LIAC)
                    {
                       inquiryType = DiscoveryAgent.GIAC;
                    }
                    else
                    {
                        inquiryType = DiscoveryAgent.LIAC;
                    }
                    break;


                // Index 2: "Authentication: true" or
                //          "Authentication: false"
                case 2:
                    // toggle
                    useAuthentication = !useAuthentication;
                    if (!useAuthentication)
                    {
                        // Authorization and encryption are only
                        // settable if authentication is true, otherwise
                        // they are false and we should remove them.
                        // (The order of removal is important.)

                        // Only a client has this setting option
                        // and not a server, thus the size check.
                        if (size() == 5)
                        {
                            delete(4); // remove authorization from List
                            useAuthorization = false;
                        }

                        this.delete(3); // remove encryption from List
                        useEncryption = false;
                    }
                    break;


                // Index 3: "Encryption: true" or
                //          "Encryption: false"
                case 3:
                    useEncryption = !useEncryption;  // toggle
```

```
                    break;


                // Index 4: "Authorization: true" or
                //          "Authorization: false"
                case 4:
                    // toggle
                    useAuthorization = !useAuthorization;
                    break;

            }
            updateListElements();
            setSelectedIndex(index, true);
        }
    }
}
```

3.6 TextScreen

This is a simple read-only text screen. It has just one command (for example, Back), which is used to transition to the next appropriate UI screen state via an appropriate `MIDletApplication` callback.

```
package example.btsppecho;


import javax.microedition.lcdui.*;


// A closeable screen for displaying text.

class TextScreen
    extends Form
    implements CommandListener
{
    private final MIDletApplication midlet;
    private final Displayable next;


    TextScreen(MIDletApplication midlet,
               Displayable next,
               String title,
               String text,
               String closeLabel)
    {
        super(title);
        this.midlet = midlet;
        this.next = next;
        append(text);
        addCommand(new Command(closeLabel, Command.BACK, 1));
        setCommandListener(this);
    }


    public void commandAction(Command c, Displayable d)
    {
        // The application code only adds a 'close' command.
        midlet.textScreenClosed(next);
    }
}
```

3.7 LogScreen

It can be useful to give end users a way to follow the progress of the device inquiry and service discovery phases.

The target end users for this example MIDlet are developers learning to use the *Java APIs For Bluetooth*. A log screen is used to follow the progress of these phases if desired. It is also a helpful debugging aid for these somewhat more sophisticated end users.

```
package example.btsppecho;


import java.util.Vector;
import javax.microedition.lcdui.*;

import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.DiscoveryListener;


// This class represents a simple log screen. In the ideal case,
// the actual log would be encapsulated by a different class
// than the presentation (LogScreen). It is a bit less elegant,
// but eliminates one class, to combine the log and log
// screen in the same class. This  MIDlet uses the LogScreen
// mainly as a simple aid during device inquiry + service discovery
// to help a user follow the progress if they wish to. So the
// log isn't persistently saved in the record store. (That would
// be easy to add if it were needed.) For unsophisticated users,
// a MIDlet would use some other visual aid rather than a log to show
// progress through the device inquiry + service discovery phases.
// The target users of this MIDlet are mainly developers learning
// to use Bluetooth, so a LogScreen is probably more helpful for them,
// as it helps show how which Bluetooth devices were found during
// device inquiry, which devices of those are running the desired
// service, and so on.

public class LogScreen
    extends Form
    implements CommandListener
{
    private static final Vector entries = new Vector();
    private static final String FIRST_ENTRY = "-- Log started: --\n\n";

    // We place a limit the maximum number of entries logged.
    // Only the 1 .. MAX_ENTRIES last entries will be kept
    // in the log. If the log exceeds MAX_ENTRIES, the
    // earliest entries will be deleted.
    private static final int MAX_ENTRIES = 300;

    static
    {
        log(FIRST_ENTRY);
    }


    private final MIDletApplication midlet;
    private final Displayable next;
    private final Command refreshCommand;
    private final Command deleteCommand;
    private final Command closeCommand;


    public LogScreen(MIDletApplication midlet,
                     Displayable next,
                     String title,
                     String closeLabel)
    {
        super(title);
        this.midlet = midlet;
        this.next = next;

        refresh(); // add any text already present
```

```
        refreshCommand = new Command("Refresh", Command.SCREEN, 1);
        deleteCommand = new Command("Delete", Command.SCREEN, 2);
        closeCommand = new Command(closeLabel, Command.SCREEN, 3);
        addCommand(refreshCommand);
        addCommand(deleteCommand);
        addCommand(closeCommand);
        setCommandListener(this);
    }


    public static void log (String string)
    {
        if (entries.size() > MAX_ENTRIES)
        {
            entries.removeElementAt(0);
        }
        entries.addElement(string);
    }


    private void refresh()
    {
        // clear the display's text
        while(size() > 0)
        {
            delete(0);
        }

        // get the lastest status and display that as text
        String text = "";
        for (int i=0; i < entries.size(); i++)
        {
            String str = (String) entries.elementAt(i);
            if (str != null)
            {
                text += str;
            }
        }
        append(text);
    }


    public void commandAction(Command command, Displayable d)
    {
        if (command == closeCommand)
        {
            midlet.logScreenClosed(next);
        }
        else if (command == refreshCommand)
        {
            refresh();
        }
        else if (command == deleteCommand)
        {
            // The deletion of all log strings affects
            // all LogScreen instances.

            synchronized(this)
            {
                entries.removeAllElements();
                log(FIRST_ENTRY);
            }

            refresh();
        }
    }


    // It was somewhat convenient to place these helper
```

```
    // methods inside the LogScreen class.

    public static String inquiryAccessCodeString(int iac)
    {
        String str = null;
        switch(iac)
        {
            case DiscoveryAgent.CACHED:
                str = "CACHED";
                break;

            case DiscoveryAgent.GIAC:
                str = "GIAC";
                break;

            case DiscoveryAgent.LIAC:
                str = "LIAC";
                break;

            case DiscoveryAgent.PREKNOWN:
                str = "PREKNOWN";
                break;
        }
        return str;
    }


    public static String responseCodeString(int responseCode)
    {
        String str = null;
        switch (responseCode)
        {
          case DiscoveryListener.SERVICE_SEARCH_COMPLETED:
              str = "SERVICE_SEARCH_COMPLETED";
              break;
          case DiscoveryListener.SERVICE_SEARCH_DEVICE_NOT_REACHABLE:
              str = "SERVICE_SEARCH_DEVICE_NOT_REACHABLE";
              break;
          case DiscoveryListener.SERVICE_SEARCH_ERROR:
              str = "SERVICE_SEARCH_ERROR";
              break;
          case DiscoveryListener.SERVICE_SEARCH_NO_RECORDS:
              str = "SERVICE_SEARCH_NO_RECORDS";
              break;
          case DiscoveryListener.SERVICE_SEARCH_TERMINATED:
              str = "SERVICE_SEARCH_TERMINATED";
              break;
        }
        return str;
    }
}
```

3.8 ClientForm

This screen is the main screen of the MIDlet when it is run in client mode. It displays the number of open connections, status strings as connections are created/deleted or as messages are sent, and simple text messages as they are received from a remote server.

```
package example.btsppecho;


import java.io.IOException;
import java.util.Vector; package example.btsppecho;


import java.io.IOException;
import java.util.Vector;
```

```
import javax.bluetooth.BluetoothStateException;
import javax.bluetooth.LocalDevice;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Item;
import javax.microedition.lcdui.StringItem;
import javax.microedition.lcdui.TextField;

import example.btsppecho.client.ClientConnectionHandler;
import example.btsppecho.client.ClientConnectionHandlerListener;
import example.btsppecho.client.ConnectionService;


public class ClientForm
    extends Form
    implements CommandListener, ClientConnectionHandlerListener
{
    private final MIDletApplication midlet;
    private final StringItem numConnectionsField;
    private final TextField sendDataField;
    private final StringItem receivedDataField;
    private final StringItem statusField;
    private final Command sendCommand;
    private final Command quitCommand;
    private final Command logCommand;
    private final Vector handlers = new Vector();

    private StringItem btAddressField = null;
    private volatile int numReceivedMessages = 0;
    private volatile int numSentMessages = 0;
    private int sendMessageId = 0;


    public ClientForm(MIDletApplication midlet)
    {
        super("Client");
        this.midlet = midlet;

        try
        {
            String address = LocalDevice.getLocalDevice()
                                        .getBluetoothAddress();
            btAddressField = new StringItem("My address", address);
            append(btAddressField);
        }
        catch (BluetoothStateException e)
        {
            // nothing we can do, don't add field
        }

        numConnectionsField = new StringItem("Connections", "0");
        append(numConnectionsField);
        statusField = new StringItem("Status", "listening");
        append(statusField);
        sendDataField = new TextField("Send data",
                                      "Client says: 'Hello, world.'",
                                      64,
                                      TextField.ANY);
        append(sendDataField);
        receivedDataField = new StringItem("Last received data", null);
        append(receivedDataField);

        sendCommand = new Command("Send", Command.SCREEN, 1);
        quitCommand = new Command("Exit", Command.EXIT, 1);
        logCommand = new Command("View log", Command.SCREEN, 2);
        addCommand(quitCommand);
        addCommand(logCommand);
```

```
            setCommandListener(this);
        }


        void closeAll()
        {
            for (int i=0; i < handlers.size(); i++)
            {
                ClientConnectionHandler handler =
                    (ClientConnectionHandler) handlers.elementAt(i);
                handler.close();
            }
        }


        public void commandAction(Command cmd, Displayable disp)
        {
            if (cmd == logCommand)
            {
                midlet.clientFormViewLog(this);
            }
            if (cmd == sendCommand)
            {
                String sendData = sendDataField.getString();
                try
                {
                    sendMessageToAllClients(++sendMessageId, sendData);
                }
                catch (IllegalArgumentException e)
                {
                    // Message length longer than
                    // ServerConnectionHandler.MAX_MESSAGE_LENGTH

                    String errorMessage =
                                "IllegalArgumentException while trying " +
                                "to send a message: " + e.getMessage();

                    handleError(null, errorMessage);
                }
            }
            else if (cmd == quitCommand)
            {
                // the MIDlet aborts the ConnectionService, etc.
                midlet.clientFormExitRequest();
            }
        }


        public void removeHandler(ClientConnectionHandler handler)
        {
            // Note: we assume the caller has aborted/closed/etc.
            // the handler if that needed to be done. This method
            // simply removes it from the list of handlers maintained
            // by the ConnectionService.
            handlers.removeElement(handler);
        }


        public void sendMessageToAllClients(int sendMessageId, String
    sendData)
            throws IllegalArgumentException
        {
            Integer id = new Integer(sendMessageId);

            for (int i=0; i < handlers.size(); i++)
            {
                ClientConnectionHandler handler =
                    (ClientConnectionHandler) handlers.elementAt(i);
```

```
            // throws IllegalArgumentException if message length
            // > ServerConnectionHandler.MAX_MESSAGE_LENGTH
            handler.queueMessageForSending(
                        id,
                        sendData.getBytes());
        }
    }


    // interface L2CAPConnectionListener

    public void handleAcceptAndOpen(ClientConnectionHandler handler)
    {
        handlers.addElement(handler);
        // start the reader and writer, it also causes underlying
        // InputStream and OutputStream to be opened.
        handler.start();

        statusField.setText("'Accept and open' for new connection");
    }


    public void handleStreamsOpen(ClientConnectionHandler handler)
    {

         // first connection
         if (handlers.size() == 1)
         {
             addCommand(sendCommand);
         }

         String str = Integer.toString(handlers.size());
         numConnectionsField.setText(str);
         statusField.setText("I/O streams opened on connection");
    }


    public void handleStreamsOpenError(ClientConnectionHandler handler,
                                       String errorMessage)
    {
        handlers.removeElement(handler);

        String str = Integer.toString(handlers.size());
        numConnectionsField.setText(str);
        statusField.setText("Error opening I/O streams: " +
                            errorMessage);
    }



    public void handleReceivedMessage(ClientConnectionHandler handler,
                                      byte[] messageBytes)
    {
        numReceivedMessages++;

        String msg = new String(messageBytes);
        receivedDataField.setText(msg);

        statusField.setText(
            "# messages read: " + numReceivedMessages + " " +
            "sent: " + numSentMessages);
    }


    public void handleQueuedMessageWasSent(
                    ClientConnectionHandler handler,
                    Integer id)
    {
        numSentMessages++;
```

```
            statusField.setText("# messages read: " +
                        numReceivedMessages + " " +
                        "sent: " + numSentMessages);
        }


        public void handleClose(ClientConnectionHandler handler)
        {
            removeHandler(handler);
            if (handlers.size() == 0)
            {
                removeCommand(sendCommand);
            }

            String str = Integer.toString(handlers.size());
            numConnectionsField.setText(str);
            statusField.setText("Connection closed");
        }


        public void handleErrorClose(ClientConnectionHandler handler,
                                     String errorMessage)
        {
            removeHandler(handler);
            if (handlers.size() == 0)
            {
                removeCommand(sendCommand);
            }

            statusField.setText("Error: (close)\"" + errorMessage + "\"");
        }


        public void handleError(ClientConnectionHandler hander,
                                String errorMessage)
        {
            statusField.setText("Error: \"" + errorMessage + "\"");
        }
    }
```

3.9 ServiceDiscoveryList

When the MIDlet is run in server mode, it must first search for suitable clients to create connections to. This screen is used to perform device inquiry, service discovery, and open connections to selected clients. After the connections are opened, the screen map is changed (via a callback to the MIDlet) to the `ServerForm` screen.

The `ServiceDiscoveryList` class uses the `setTitle` method to dynamically modify the screen's title string during device inquiry and service to indicate to the end user that some longer term activity is taking place. This approach for providing an 'activity indicator' was used to keep the MIDlet code relatively simple. Your MIDlet may want to use a separate transient canvas screen to graphically indicate that an activity (which may take some time to complete) is taking place. The amount of time that device inquiry and service search need to complete depends on how many Bluetooth devices are within range, and on the number of those which might be, and/or actually are, running the service. You are likely to want to test your Bluetooth MIDlets in locations where there are just a few nearby Bluetooth devices, and also in locations where there are many Bluetooth devices within range (for example, dozens of devices).

```
package example.btsppecho;


import java.util.Hashtable;
import java.util.Vector;
import java.util.Enumeration;
```

```
import javax.bluetooth.BluetoothStateException;
import javax.bluetooth.DataElement;
import javax.bluetooth.DeviceClass;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.DiscoveryListener;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.RemoteDevice;
import javax.bluetooth.ServiceRecord;
import javax.bluetooth.UUID;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.ImageItem;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.Form;

import example.btsppecho.MIDletApplication;


class ServiceDiscoveryList
    extends List
    implements CommandListener,
               DiscoveryListener,
               Runnable
{
    private final static String TITLE = "New search";
    private final static int WAIT_MILLIS = 500; // milliseconds
    private final static String[] ACTIVITY = { "",
                                               ".",
                                               "..",
                                               "...",
                                               "...." };

    private final UUID uuid;
    private final MIDletApplication midlet;
    private final Command backCommand;
    private final Command searchCommand;
    private final Command openCommand;
    private final Command stopCommand;
    private final Command logCommand;
    private final Command exitCommand;


    private final int sdTransMax;
    private final int inquiryAccessCode;

    private DiscoveryAgent discoveryAgent;
    private volatile boolean inquiryInProgress = false;
    private volatile int numServiceSearchesInProgress = 0;
    private volatile boolean aborting = false;
    private volatile Thread thread;
    private Displayable backDisplayable;

    private volatile Thread activityIndicatorThread;
    private boolean activityIndicatorRunning = false;

    private Vector unsearchedRemoteDevices = new Vector();
    private Vector transIds = new Vector();
    private Hashtable matchingServiceRecords = new Hashtable();

    private int numConnectionsAlreadyOpen = 0;


    ServiceDiscoveryList(MIDletApplication midlet,
                         String uuidString,
                         int inquiryAccessCode)
```

```
{
    super(TITLE, List.IMPLICIT);
    this.midlet = midlet;
    uuid = new UUID(uuidString, false);
    this.inquiryAccessCode = inquiryAccessCode;

    openCommand = new Command("Open connection",
                              Command.SCREEN,
                              1);
    searchCommand = new Command("Search", Command.SCREEN, 2);
    logCommand = new Command("View log", Command.SCREEN, 3);
    stopCommand = new Command("Stop", Command.SCREEN, 4);
    backCommand = new Command("Back", Command.SCREEN, 5);
    exitCommand = new Command("Exit", Command.EXIT, 0);


    String property =
           LocalDevice.getProperty("bluetooth.sd.trans.max");
    sdTransMax = Integer.parseInt(property);

    // create discovery agent
    try
    {
        discoveryAgent =
            LocalDevice.getLocalDevice().getDiscoveryAgent();

        addCommand(logCommand);
        addCommand(exitCommand);
        addCommand(searchCommand);
        setCommandListener(this);

        start();
    }
    catch(BluetoothStateException e)
    {
        midlet.serviceDiscoveryListFatalError(
                  "Couldn't get a discovery agent: '" +
                  e.getMessage() + "'");
    }
}

static Image makeImage(String filename)
{
    Image image = null;

    try
    {
        image = Image.createImage(filename);
    }
    catch (Exception e)
    {
        // there's nothing we can do, so ignore
    }
    return image;
}



public void addBackCommand(Displayable backDisplayable)
{
    this.backDisplayable = backDisplayable;
    removeCommand(backCommand);
    addCommand(backCommand);
}


public synchronized void start()
{
    thread = new Thread(this);
```

```
        thread.start();
    }


    public synchronized void abort()
    {
        thread = null;
    }


    public void init(int numConnectionsAlreadyOpen)
    {
        this.numConnectionsAlreadyOpen =
             numConnectionsAlreadyOpen;

        // stop any pending searches
        if (inquiryInProgress ||
            numServiceSearchesInProgress > 0)
        {
            cancelPendingSearches();
        }

        // remove any old list elements
        while (size() > 0)
        {
            delete(0);
        }
    }


    private synchronized void setInquiryInProgress(boolean bool)
    {
        inquiryInProgress = bool;
    }


    public void commandAction(Command command, Displayable d)
    {
        if (command == logCommand)
        {
            midlet.serviceDiscoveryListViewLog(this);
        }
        else if (command == searchCommand &&
                 !inquiryInProgress &&
                 (numServiceSearchesInProgress == 0))
        {
            // new inquiry started

            removeCommand(openCommand);

            // remove old device lists
            unsearchedRemoteDevices.removeAllElements();
            for (Enumeration keys = matchingServiceRecords.keys();
                 keys.hasMoreElements();)
            {
                matchingServiceRecords.remove(keys.nextElement());
            }

            // delete all old List items
            while (size() > 0)
            {
                delete(0);
            }

            try
            {
                // disable page scan and inquiry scan
                LocalDevice dev = LocalDevice.getLocalDevice();
                dev.setDiscoverable(DiscoveryAgent.NOT_DISCOVERABLE);
```

```
                String iacString =

LogScreen.inquiryAccessCodeString(inquiryAccessCode);
                LogScreen.log("startInquiry (" +
                              iacString + ")\n");

                //startActivityIndicator();
                // this is non-blocking
                discoveryAgent.startInquiry(inquiryAccessCode, this);

                setInquiryInProgress(true);
                addCommand(stopCommand);
                removeCommand(searchCommand);
            }
            catch (BluetoothStateException e)
            {
                addCommand(searchCommand);
                removeCommand(stopCommand);
                midlet.serviceDiscoveryListError(
                           "Error during startInquiry: '" +
                           e.getMessage() + "'");
            }
        }
        else if (command == stopCommand)
        {
            // stop searching
            if (cancelPendingSearches())
            {
                setInquiryInProgress(false);
                removeCommand(stopCommand);
                addCommand(searchCommand);
            }
        }
        else if (command == exitCommand)
        {
            midlet.serviceDiscoveryListExitRequest();
        }
        else if (command == openCommand)
        {
            int size = this.size();
            boolean[] flags = new boolean[size];
            getSelectedFlags(flags);
            Vector selectedServiceRecords = new Vector();
            for (int i=0; i < size; i++)
            {
                if (flags[i])
                {
                    String key = getString(i);
                    ServiceRecord rec =
                        (ServiceRecord) matchingServiceRecords.get(key);
                    selectedServiceRecords.addElement(rec);
                }
            }

            // try to perform an open on selected items
            if (selectedServiceRecords.size() > 0)
            {
                String value = LocalDevice.getProperty(
                                    "bluetooth.connected.devices.max");
                int maxNum = Integer.parseInt(value);

                int total = numConnectionsAlreadyOpen +
                            selectedServiceRecords.size();
                if (total > maxNum)
                {
                    midlet.serviceDiscoveryListError(
                               "Too many selected. " +
                               "This device can connect to at most " +
```

```
                                maxNum + " other devices");
                    }
                    else
                    {
                        midlet.serviceDiscoveryListOpen(
                                    selectedServiceRecords);
                    }
                }
                else
                {
                    midlet.serviceDiscoveryListError(
                              "Select at least one to open");
                }
            }
            else if (command == backCommand)
            {
                midlet.serviceDiscoveryListBackRequest(backDisplayable);
            }
        }


        boolean cancelPendingSearches()
        {
            LogScreen.log("Cancel pending inquiries and searches\n");

            boolean everythingCancelled = true;

            if (inquiryInProgress)
            {
                // Note: The BT API (v1.0) isn't completely clear
                // whether cancelInquiry is blocking or non-blocking.

                if (discoveryAgent.cancelInquiry(this))
                {
                    setInquiryInProgress(false);
                }
                else
                {
                    everythingCancelled = false;
                }
            }

            for (int i=0; i < transIds.size(); i++)
            {
                // Note: The BT API (v1.0) isn't completely clear
                // whether cancelServiceSearch is blocking or
                // non-blocking?

                Integer pendingId =
                    (Integer) transIds.elementAt(i);
                if (discoveryAgent.cancelServiceSearch(
                                        pendingId.intValue()))
                {
                    transIds.removeElement(pendingId);
                }
                else
                {
                    everythingCancelled = false;
                }
            }
            return everythingCancelled;
        }


        // DiscoveryListener callbacks

        public void deviceDiscovered(RemoteDevice remoteDevice,
                                     DeviceClass deviceClass)
        {
```

```
        LogScreen.log("deviceDiscovered: " +
                      remoteDevice.getBluetoothAddress() +
                      " major device class=" +
                      deviceClass.getMajorDeviceClass() +
                      " minor device class=" +
                      deviceClass.getMinorDeviceClass() + "\n");

        // Note: The following check has the effect of only
        // performing later service searches on phones.
        // If you intend to run the MIDlet on some other device (e.g.
        // handheld computer, PDA, etc. you have to add a check
        // for them as well.) You might also refine the check
        // using getMinorDeviceClass() to check only cellular phones.
        boolean isPhone =
            (deviceClass.getMajorDeviceClass() == 0x200);

        // Setting the following line to 'true' is a workaround
        // for some early beta SDK device emulators. Set it
        // to false when compiling the MIDlet for download to
        // real MIDP phones!
        boolean isEmulator = true; //false;

        if (isPhone || isEmulator)
        {
            unsearchedRemoteDevices.addElement(remoteDevice);
        }
    }


    public void inquiryCompleted(int discoveryType)
    {
        LogScreen.log("inquiryCompleted: " +
                      discoveryType + "\n");

        setInquiryInProgress(false);

        if (unsearchedRemoteDevices.size() == 0)
        {
            setTitle(TITLE);

            addCommand(searchCommand);
            removeCommand(stopCommand);

            midlet.serviceDiscoveryListError(
                       "No Bluetooth devices found");
        }
    }


    public void servicesDiscovered(int transId,
                                   ServiceRecord[] serviceRecords)
    {
        LogScreen.log("servicesDiscovered: transId=" +
                      transId +
                      " # serviceRecords=" +
                      serviceRecords.length + "\n");

        // Remove+Add: ensure there is at most one open command
        removeCommand(openCommand);
        addCommand(openCommand);

        // Use the bluetooth address
        // to identify the matching Device + ServiceRecord

        // there should only be one record
        if (serviceRecords.length == 1)
        {
            RemoteDevice device = serviceRecords[0].getHostDevice();
            String name = device.getBluetoothAddress();
```

```
            // This MIDlet only uses the first matching service
            // record found for a particular device.
            if (!matchingServiceRecords.containsKey(name))
            {
                matchingServiceRecords.put(name, serviceRecords[0]);
                append(name, null);

                // The List should have at least one entry,
                // before we add an open command.
                if (size() == 1)
                {
                    addCommand(openCommand);
                }
            }
        }
        else
        {
            midlet.serviceDiscoveryListError(
                        "Error: Unexpected number (" +
                        serviceRecords.length + ") of service records " +
                        "in servicesDiscovered callback, transId=" +
                        transId);
        }
    }


    public void serviceSearchCompleted(int transId, int responseCode)
    {
        setTitle("New search");

        String responseCodeString =
            LogScreen.responseCodeString(responseCode);
        LogScreen.log("serviceSearchCompleted: transId=" +
                    transId +
                    " (" +
                    responseCodeString + ")\n");

        // For any responseCode, decrement the counter
        numServiceSearchesInProgress--;

        // remove the transaction id from the pending list
        for (int i=0; i < transIds.size(); i++)
        {
            Integer pendingId = (Integer) transIds.elementAt(i);
            if (pendingId.intValue() == transId)
            {
                transIds.removeElement(pendingId);
                break;
            }
        }

        // all the searches have completed
        if (!inquiryInProgress && (transIds.size() == 0))
        {
           removeCommand(stopCommand);
           addCommand(searchCommand);

           if (matchingServiceRecords.size() == 0)
           {
                midlet.serviceDiscoveryListError(
                            "No matching services found");
           }
        }
    }


    // Interface Runnable
    public void run()
```

```
{
    Thread currentThread = Thread.currentThread();
    int i = 0;

running:
    while (thread == currentThread)
    {
        synchronized (this)
        {
            if (thread != currentThread)
            {
                break running;
            }
            else if (!inquiryInProgress)
            {
                doServiceSearch();
            }

            if (inquiryInProgress ||
                numServiceSearchesInProgress > 0)
            {
                setTitle("Searching " + ACTIVITY[i]);
                if (++i >= ACTIVITY.length)
                {
                    i = 0;
                }
            }

            try
            {
                wait(WAIT_MILLIS);
            }
            catch (InterruptedException e)
            {
                // we can safely ignore this
            }
        }
    }
}


public void doServiceSearch()
{
    if ((unsearchedRemoteDevices.size() > 0) &&
        (numServiceSearchesInProgress < sdTransMax))
    {
        synchronized(this)
        {
            RemoteDevice device =
                (RemoteDevice) unsearchedRemoteDevices
                                    .elementAt(0);

            UUID[] uuids = new UUID[1];
            uuids[0] = uuid;
            try
            {
                int[] attrSet = null; // default attrSet

                numServiceSearchesInProgress++;

                int transId = discoveryAgent.searchServices(
                                                attrSet,
                                                uuids,
                                                device,
                                                this);

                LogScreen.log("starting service search on device=" +
                              device.getBluetoothAddress() +
                              " transId=" + transId + "\n");
```

```
                    transIds.addElement(new Integer(transId));
                    unsearchedRemoteDevices.removeElementAt(0);
                }
                catch (BluetoothStateException e)
                {
                    numServiceSearchesInProgress--;

                    midlet.serviceDiscoveryListError(
                              "Error, could not perform " +
                              "service search: '" +
                              e.getMessage());
                }
            }
        }
    }


    public void remove(Vector alreadyOpen)
    {
        if (size() > 0)
        {
            for (int i=0; i < alreadyOpen.size(); i++)
            {
                // Bluetooth address of slave device that
                // we already have a connection open to
                String btAddress = (String) alreadyOpen.elementAt(i);

                boolean found = false;
                for (int j = 0; j < size(); j++)
                {
                    if (getString(j).equals(btAddress))
                    {
                        found = true;

                        // if the element we are about to
                        // remove is selected, select something else
                        if (getSelectedIndex() == j)
                        {
                            setSelectedIndex(j, false);
                        }
                        delete(j);
                        break;
                    }
                }
                if (found)
                {
                    break;
                }
            }
        }
    }
}
```

3.10 ServerForm

This is the main UI screen of a connected MIDlet when run in server mode. It is used to receive messages from a remote client and echo those back to all connected clients. This allows the application to be used a very simple chat-like server. This screen shows the number of connected clients. If one client disconnects, it doesn't affect the connectivity between the server and other clients.

```
package example.btsppecho;


import java.io.IOException;
import java.util.Vector;
```

```
import javax.bluetooth.BluetoothStateException;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.ServiceRecord;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Item;
import javax.microedition.lcdui.StringItem;
import javax.microedition.lcdui.TextField;

import example.btsppecho.server.ServerConnectionHandler;
import example.btsppecho.server.ServerConnectionHandlerListener;


class ServerForm
    extends Form
    implements ServerConnectionHandlerListener, CommandListener
{
    private final MIDletApplication midlet;
    private final StringItem numConnectionsField;
    private final TextField sendDataField;
    private final StringItem receivedDataField;
    private final StringItem statusField;
    private final Command sendCommand;
    private final Command addConnectionCommand;
    private final Command searchCommand;
    private final Command logCommand;
    private final Command quitCommand;
    private final Command clearStatusCommand;
    private final Vector handlers;

    private int maxConnections;
    private StringItem btAddressField = null;
    private volatile int numReceivedMessages = 0;
    private volatile int numSentMessages = 0;
    private int sendMessageId = 0;


    ServerForm(MIDletApplication midlet)
    {
        super("Server");
        this.midlet = midlet;
        handlers = new Vector();

        String value =
            LocalDevice.getProperty(
                        "bluetooth.connected.devices.max");
        try
        {
            maxConnections = Integer.parseInt(value);
        }
        catch (NumberFormatException e)
        {
            maxConnections = 0;
        }

        // 1. add Form items
        try
        {
            String address = LocalDevice.getLocalDevice()
                                        .getBluetoothAddress();
            btAddressField = new StringItem("My address", address);
            append(btAddressField);
        }
        catch (BluetoothStateException e)
        {
```

```
            // nothing we can do, don't add field
        }
        numConnectionsField = new StringItem("Connections", "0");
        append(numConnectionsField);
        statusField = new StringItem("Status", "");
        append(statusField);
        sendDataField = new TextField("Send data",
                                      "Server says: 'Hello, world.'",
                                      64,
                                      TextField.ANY);
        append(sendDataField);
        receivedDataField = new StringItem("Last received data",
                                           null);
        append(receivedDataField);


        // 2. create commands
        sendCommand = new Command("Send", Command.SCREEN, 1);
        searchCommand = new Command("Search for more",
                                    Command.SCREEN,
                                    1);
        addConnectionCommand = new Command("Add connection",
                                           Command.SCREEN,
                                           2);
        logCommand = new Command("View log", Command.SCREEN, 3);
        clearStatusCommand = new Command("Clear status", Command.SCREEN,
4);
        quitCommand = new Command("Quit", Command.EXIT, 1);


        // 3. add commands and set command listener
        addCommand(searchCommand);
        addCommand(addConnectionCommand);
        addCommand(logCommand);
        addCommand(clearStatusCommand);
        addCommand(quitCommand);
        // The 'sendCommand' is added later to screen,
        // if at least one connection is open.
        setCommandListener(this);
    }


    public void makeConnections(Vector serviceRecords, int security)
    {
        for (int i=0; i < serviceRecords.size(); i++)
        {
            ServiceRecord serviceRecord =
                (ServiceRecord) serviceRecords.elementAt(i);
            boolean found = false;
            for (int j=0; j < handlers.size(); j++)
            {
                ServerConnectionHandler old =
                    (ServerConnectionHandler) handlers.elementAt(j);
                String oldAddr = old.getServiceRecord().
                                     getHostDevice().
                                     getBluetoothAddress();
                String newAddr =
                       serviceRecord.getHostDevice()
                                    .getBluetoothAddress();
                if (oldAddr.equals(newAddr))
                {
                     found = true;
                     break;
                }
            }
            if (!found)
            {
                ServerConnectionHandler newHandler =
                    new ServerConnectionHandler(
```

```
                            this,
                            serviceRecord,
                            security);
                newHandler.start(); // start reader & writer
            }
        }
    }


    private void removeHandler(ServerConnectionHandler handler)
    {
        if (handlers.contains(handler))
        {
            handlers.removeElement(handler);
            String str = Integer.toString(handlers.size());
            numConnectionsField.setText(str);
            if (handlers.size() == 0)
            {
                removeCommand(sendCommand);
                addCommand(searchCommand);
            }
        }
    }


    void closeAll()
    {
        for (int i=0; i < handlers.size(); i++)
        {
            ServerConnectionHandler handler =
                (ServerConnectionHandler) handlers.elementAt(i);
            handler.close();
            removeHandler(handler);
        }
    }


    public void commandAction(Command cmd, Displayable disp)
    {
        if (cmd == addConnectionCommand)
        {
            Vector v = new Vector();
            for (int i=0; i < handlers.size(); i++)
            {
                ServerConnectionHandler handler =
                    (ServerConnectionHandler) handlers.elementAt(i);
                String btAddress = handler.getServiceRecord()
                                          .getHostDevice()
                                          .getBluetoothAddress();
                v.addElement(btAddress);
            }
            midlet.serverFormAddConnection(v);
        }
        else if (cmd == clearStatusCommand)
        {
            statusField.setText("");
        }
        else if (cmd == logCommand)
        {
            midlet.serverFormViewLog();
        }
        else if (cmd == sendCommand)
        {
            String sendData = sendDataField.getString();
            Integer id = new Integer(sendMessageId++);

            for (int i=0; i < handlers.size(); i++)
            {
                ServerConnectionHandler handler =
```

```
                    (ServerConnectionHandler) handlers.elementAt(i);
                try
                {
                    handler.queueMessageForSending(
                                id,
                                sendData.getBytes());
                    statusField.setText(
                                    "Queued a send message request");
                }
                catch(IllegalArgumentException e)
                {
                    // Message length longer than
                    // ServerConnectionHandler.MAX_MESSAGE_LENGTH

                    String errorMessage =
                        "IllegalArgumentException while trying " +
                        "to send a message: " + e.getMessage();
                    handleError(handler, errorMessage);
                }
            }
        }
        else if (cmd == searchCommand)
        {
            midlet.serverFormSearchRequest(handlers.size());
        }
        else if (cmd == quitCommand)
        {
            closeAll();
            midlet.serverFormExitRequest();
        }

        // To keep this MIDlet simple, I didn't add any way
        // to drop individual connections. But you might
        // want to do so.
    }


    // ServerConnectionHandlerListener interface methods

    public void handleOpen(ServerConnectionHandler handler)
    {
        handlers.addElement(handler);
        // for the first open connection
        if (handlers.size() == 1)
        {
            removeCommand(searchCommand);

            removeCommand(sendCommand);
            addCommand(sendCommand);
        }

        // Remove the 'Add connection' command
        // when the device already has open the
        // maximum number of connections it can
        // support.
        if (handlers.size() >= maxConnections)
        {
            removeCommand(addConnectionCommand);
        }

        statusField.setText("Connection opened");
        String str = Integer.toString(handlers.size());
        numConnectionsField.setText(str);
    }


    public void handleOpenError(
                    ServerConnectionHandler handler,
                    String errorMessage)
```

```
{
    statusField.setText("Error opening outbound connection: " +
                        errorMessage);
}


public void handleReceivedMessage(
                ServerConnectionHandler handler,
                byte[] messageBytes)
{
    numReceivedMessages++;

    String message = new String(messageBytes);
    receivedDataField.setText(message);

    statusField.setText(
        "# messages read: " + numReceivedMessages + " " +
        "sent: " + numSentMessages);

    // Broadcast message to all clients
    for (int i=0; i < handlers.size(); i++)
    {
        ServerConnectionHandler h =
            (ServerConnectionHandler) handlers.elementAt(i);

        Integer id = new Integer(sendMessageId++);
        try
        {
            h.queueMessageForSending(id, messageBytes);
        }
        catch (IllegalArgumentException e)
        {
            String errorMessage =
                "IllegalArgumentException while trying to " +
                "send message: " + e.getMessage();
            handleError(handler, errorMessage);
        }
    }
}


public void handleQueuedMessageWasSent(
                ServerConnectionHandler handler,
                Integer id)
{
    numSentMessages++;
    statusField.setText("# messages read: " +
                        numReceivedMessages + " " +
                        "sent: " + numSentMessages);
}


public void handleClose(ServerConnectionHandler handler)
{
    removeHandler(handler);
    if (handlers.size() == 0)
    {
        removeCommand(sendCommand);
        addCommand(searchCommand);
    }

    // If the number of currently open connections
    // drops below the maximum number that this
    // device could have open, restore
    // 'Add connection' to the screen commands.
    if (handlers.size() < maxConnections)
    {
        removeCommand(addConnectionCommand);
        addCommand(addConnectionCommand);
```

```
        }

        statusField.setText("Connection closed");
    }


    public void handleErrorClose(ServerConnectionHandler handler,
                                 String errorMessage)
    {
        removeHandler(handler);
        if (handlers.size() == 0)
        {
            removeCommand(sendCommand);
            addCommand(searchCommand);
        }

        statusField.setText("Error (close): '" + errorMessage + "'");
    }


    public void handleError(ServerConnectionHandler handler,
                            String errorMessage)
    {
        statusField.setText("Error: '" + errorMessage + "'");
    }
}
```

3.11 ConnectionService

A ConnectionService is a runnable object that runs continuously. It listens for new inbound connection requests from a remote server. It uses its listener to handle that request.

```
package example.btsppecho.client;


import java.io.IOException;
import javax.microedition.io.ConnectionNotFoundException;
import javax.microedition.io.Connector;
import javax.microedition.io.StreamConnection;
import javax.microedition.io.StreamConnectionNotifier;

import example.btsppecho.ClientForm;
import example.btsppecho.LogScreen;

public class ConnectionService
    implements Runnable
{
    private final ClientForm listener;
    private final String url;

    private StreamConnectionNotifier connectionNotifier = null;
    private volatile boolean aborting;


    public ConnectionService(String url,
                             ClientForm listener)
    {
        this.url = url;
        this.listener = listener;

        LogScreen.log("ConnectionService: waiting to " +
                      "accept connections on '" +
                      url + "'\n");

        // start waiting for a connection
        Thread thread = new Thread(this);
        thread.start();
    }
```

```
public String getClientURL()
{
    return url;
}


public void close()
{
    if (!aborting)
    {
        synchronized(this)
        {
            aborting = true;
        }

        // Ideally, one might want to give the run method's
        // loop a chance to abort before calling the
        // subsequent close, but the run loop is anyways
        // likely to be sitting on the acceptAndOpen
        // (i.e. blocked).

        try
        {
            connectionNotifier.close();
        }
        catch (IOException e)
        {
            // There is nothing very useful that
            // we can do for this case.
        }
    }
}


public void run()
{
    aborting = false;

    try
    {
        connectionNotifier =
            (StreamConnectionNotifier) Connector.open(url);

        // It might useful in some cases to add a service to the
        // 'Public Browse Group'. For example by doing something
        // approximately as follows:
        // ------------------------------------------------------
        // Retrieve the service record template
        // LocalDevice ld = LocalDevice.getLocalDevice();
        // ServiceRecord rec = ld.getRecord(connectionNotifier);
        // DataElement element =
        //             new DataElement(DataElement.DATSEQ);
        //
        // The service class for PublicBrowseGroup (0x1002)
        // is defined in the Bluetooth Assigned Numbers document.
        // element.addElement(new DataElement(DataElement.UUID,
        //                                    new UUID(0x1002)));
        //
        // Add to the public browse group:
        // rec.setAttributeValue(0x0005, element);
        // ------------------------------------------------------
    }
    catch (IOException e)
    {
        // ConnectionNotFoundException is an IOException
        String errorMessage =
               "Error while starting ConnectionService: " +
```

```
                e.getMessage();

        listener.handleError(null, errorMessage);

        aborting = true;
    }
    catch (SecurityException e)
    {
        String errorMessage =
            "SecurityException while starting ConnectionService: " +
        e.getMessage();

        listener.handleError(null, errorMessage);

        aborting = true;
    }

    while (!aborting)
    {
        try
        {
            // 1. wait to accept & open a new connection
            StreamConnection connection =
                (StreamConnection)
                connectionNotifier.acceptAndOpen();

            LogScreen.log("ConnectionService: new connection\n");

            // 2. create a handler to take care of
            // the new connection and inform
            // the listener
            if (!aborting)
            {
                ClientConnectionHandler handler =
                    new ClientConnectionHandler(this,
                                                connection,
                                                listener);
                listener.handleAcceptAndOpen(handler);
            }

            // One could consider exiting the
            // ConnectionService when the Client
            // reaches the maximum number of allowed
            // open connections. In that case (i.e.
            // when the maximum number of possible
            // connections is already open), the
            // ConnectionService will not be able
            // to accept any new connections and one
            // might possibly want to consider whether
            // or not the ConnectionService thread
            // could then be terminated.
            //
            // However, existing connections can also
            // be disconnected (e.g. the Server is
            // terminated or closes some/all of its
            // existing connections). In that case,
            // one may want to keep the
            // ConnectionService alive and running:
            // in order to accept later new connections
            // without the need to restart the
            // ConnectionService or MIDlet.
            //
            // (This MIDlet uses the latter approach.)
        }
        catch (IOException e)
        {
            if (!aborting)
            {
                String errorMessage =
```

```
                                    "IOException occurred during " +
                                    "accept and open: " +
                                    e.getMessage();

                            listener.handleError(null, errorMessage);
                        }
                    }
                    catch (SecurityException e)
                    {
                        if (!aborting)
                        {
                            String errorMessage =
                                    "IOException occurred during " +
                                    "accept and open: " +
                                    e.getMessage();

                            listener.handleError(null, errorMessage);
                        }
                    }
                }
            }
        }
```

3.12 ClientConnectionHandlerListener

This interface describes the callbacks of a ClientConnectionHandler.

```
package example.btsppecho.client;


public interface ClientConnectionHandlerListener
{
    // The handler's accept and open of a new connection
    // has occurred, but the I/O streams are not yet open.
    // The I/O streams must be open to send or receive
    // messages.
    public void handleAcceptAndOpen(ClientConnectionHandler handler);


    // The handler's I/O streams are now open and the
    // handler can now be used to send and receive messages.
    public void handleStreamsOpen(ClientConnectionHandler handler);


    // Opening of the handler's I/O streams failed. The handler has
    // closed any connections or streams that were open.
    // The handler should not be used anymore,
    // and should be discarded.
    public void handleStreamsOpenError(ClientConnectionHandler handler,
                                       String errorMessage);


    // The handler got an inbound message.
    public void handleReceivedMessage(
                    ClientConnectionHandler handler,
                    byte[] messageBytes);


    // A message that had been previously queued for sending
    // (identified by id) by the handler, has been sent successfully.
    public void handleQueuedMessageWasSent(
                    ClientConnectionHandler handler,
                    Integer id);


    // The handler has closed its connections and streams.
    // The handler should not be used anymore, and should be discarded.
    // Only handlers which have previously called 'handleOpen' may
```

```
    // call 'handleClose', and only just once.
    public void handleClose(ClientConnectionHandler handler);


    // An error related to the handler occurred. The handler
    // has closed the connection, and the handler should no
    // longer be used.
    public void handleErrorClose(ClientConnectionHandler handler,
                                 String errorMessage);

}
```

3.13 ClientConnectionHandler

When the MIDlet is run in client mode, each connection is represented by a ClientConnectionHandler.

```
package example.btsppecho.client;


import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Vector;

import javax.microedition.io.StreamConnection;
import javax.microedition.io.StreamConnectionNotifier;


public class ClientConnectionHandler
    implements Runnable
{
    private final static byte ZERO = (byte) '0';
    private final static int LENGTH_MAX_DIGITS = 5;

    // this is an arbitrarily chosen value:
    private final static int MAX_MESSAGE_LENGTH =
                             65536 - LENGTH_MAX_DIGITS;

    private final ConnectionService ConnectionService;
    private final ClientConnectionHandlerListener listener;
    private final Hashtable sendMessages = new Hashtable();

    private StreamConnection connection;
    private InputStream in;
    private OutputStream out;

    private volatile boolean aborting;


    public ClientConnectionHandler(
        ConnectionService ConnectionService,
        StreamConnection connection,
        ClientConnectionHandlerListener listener)
    {
        this.ConnectionService = ConnectionService;
        this.connection = connection;
        this.listener = listener;
        aborting = false;
        in = null;
        out = null;

        // Our caller uses method 'start' to start the reader
        // and writer. (This allows the ConnectionService a
        // chance to add us to its list of handlers before
        // the reader and writer start running.)
    }
```

```
ClientConnectionHandlerListener getListener()
{
    return listener;
}


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


public void close()
{
    if (!aborting)
    {
        synchronized(this)
        {
            aborting = true;
        }

        synchronized(sendMessages)
        {
            sendMessages.notify();
        }

        if (out != null)
        {
            try
            {
                out.close();
                synchronized (this)
                {
                    out = null;
                }
            }
            catch(IOException e)
            {
                // there is nothing we can do: ignore it
            }
        }

        if (in != null)
        {
            try
            {
                in.close();
                synchronized (this)
                {
                    in = null;
                }
            }
            catch(IOException e)
            {
                // there is nothing we can do: ignore it
            }
        }

        if (connection != null)
        {
            try
            {
                connection.close();
                synchronized (this)
                {
                    connection = null;
```

```
                }
            }
            catch (IOException e)
            {
                // there is nothing we can do: ignore it
            }
        }
    }
}


public void queueMessageForSending(Integer id, byte[] data)
{
    if (data.length > MAX_MESSAGE_LENGTH)
    {
        throw new IllegalArgumentException(
                    "Message too long: limit is " +
                    MAX_MESSAGE_LENGTH + " bytes");
    }

    synchronized(sendMessages)
    {
        sendMessages.put(id, data);
        sendMessages.notify();
    }
}


private void sendMessage(byte[] data)
    throws IOException
{
    byte[] buf = new byte[LENGTH_MAX_DIGITS + data.length];
    writeLength(data.length, buf);
    System.arraycopy(data,
                     0,
                     buf,
                     LENGTH_MAX_DIGITS,
                     data.length);
    out.write(buf);
    out.flush();
}


public void run()
{
    // the reader

    // 1. open the streams, start the writer
    try
    {
        in = connection.openInputStream();
        out = connection.openOutputStream();

        // start the writer
        Writer writer = new Writer(this);
        Thread writeThread = new Thread(writer);
        writeThread.start();

        listener.handleStreamsOpen(this);
    }
    catch(IOException e)
    {
        // open failed: close any connections/streams and
        // inform listener that the open failed

        close(); // also tells listener to delete handler

        listener.handleStreamsOpenError(this, e.getMessage());
        return;
```

```
        }


        // 2. wait to receive and read messages
        while (!aborting)
        {
            int length = 0;
            try
            {
                byte[] lengthBuf = new byte[LENGTH_MAX_DIGITS];
                readFully(in, lengthBuf);
                length = readLength(lengthBuf);
                byte[] temp = new byte[length];
                readFully(in, temp);

                listener.handleReceivedMessage(this, temp);
            }
            catch (IOException e)
            {
                close();
                if (length == 0)
                {
                   listener.handleClose(this);
                }
                else
                {
                   // we were in the middle of reading...
                   listener.handleErrorClose(this, e.getMessage());
                }
            }
        }
    }


    private static void readFully(InputStream in, byte[] buffer)
        throws IOException
    {
        int bytesRead = 0;

        while (bytesRead < buffer.length)
        {
            int count = in.read(buffer,
                                bytesRead,
                                buffer.length - bytesRead);

            if (count == -1)
            {
                throw new IOException("Input stream closed");
            }
            bytesRead += count;
        }
    }


    private static int readLength(byte[] buffer)
    {
        int value = 0;

        for (int i = 0; i < LENGTH_MAX_DIGITS; ++i)
        {
            value *= 10;
            value += buffer[i] - ZERO;
        }
        return value;
    }


    private void sendMessage(OutputStream out, byte[] data)
        throws IOException
```

```
{
    if (data.length > MAX_MESSAGE_LENGTH)
    {
        throw new IllegalArgumentException(
                    "Message too long: limit is: " +
                    MAX_MESSAGE_LENGTH + " bytes");
    }
    byte[] buf = new byte[LENGTH_MAX_DIGITS + data.length];
    writeLength(data.length, buf);
    System.arraycopy(data, 0, buf, LENGTH_MAX_DIGITS, data.length);
    out.write(buf);
    out.flush();
}


private static void writeLength(int value, byte[] buffer)
{
    for (int i = LENGTH_MAX_DIGITS -1; i >= 0; --i)
    {
        buffer[i] = (byte) (ZERO + value % 10);
        value = value / 10;
    }
}


private class Writer
    implements Runnable
{
    private final ClientConnectionHandler handler;


    Writer(ClientConnectionHandler handler)
    {
        this.handler = handler;
    }


    public void run()
    {
        while (!aborting)
        {
            synchronized(sendMessages)
            {
                Enumeration e = sendMessages.keys();
                if (e.hasMoreElements())
                {
                    // send any pending messages
                    Integer id = (Integer) e.nextElement();
                    byte[] sendData = (byte[]) sendMessages.get(id);
                    try
                    {
                        sendMessage(out, sendData);

                        // remove sent message from queue
                        sendMessages.remove(id);

                        // inform listener that it was sent
                        listener.handleQueuedMessageWasSent(
                                    handler,
                                    id);
                    }
                    catch (IOException ex)
                    {
                        close(); // stop the networking thread

                        // inform that we got an error close
                        listener.handleErrorClose(
                                    handler,
                                    ex.getMessage());
```

```
                        }
                    }

                    if (sendMessages.isEmpty())
                    {
                        try
                        {
                            sendMessages.wait();
                        }
                        catch (InterruptedException ex)
                        {
                            // this can't happen in MIDP: ignore it
                        }
                    }
                }
            }
        }
    }
}
```

3.14 ServerConnectionHandlerListener

This interface describes the callbacks of a ServerConnectionHandler.

```
package example.btsppecho.server;


public interface ServerConnectionHandlerListener
{
    // The handler's open succeeded. It can now be used for sending
    // and receiving messages.
    public void handleOpen(ServerConnectionHandler handler);


    // The hadler's open failed. It has closed any connections or
    // streams that were open. The handler should not be used anymore,
    // and should be discarded.
    public void handleOpenError(ServerConnectionHandler handler,
                                String errorMessage);


    // The handler got an inbound message.
    public void handleReceivedMessage(
                    ServerConnectionHandler handler,
                    byte[] messageBytes);


    // A message that had been previously queued for sending
    // (identified by id) by the handler, has been sent successfully.
    public void handleQueuedMessageWasSent(
                    ServerConnectionHandler handler,
                    Integer id);


    // The handler has closed its connections and streams.
    // The handler should not be used anymore, and should be discarded.
    // Only handlers which have previously called 'handleOpen' may
    // call 'handleClose', and only just once.
    public void handleClose(ServerConnectionHandler handler);


    // An error related to the handler occurred. The handler
    // has closed the connection, and the handler should no
    // longer be used.
    public void handleErrorClose(ServerConnectionHandler handler,
                                 String errorMessage);

}
```

3.15 ServerConnectionHandler

When the MIDlet is run in the server mode, each connection is represented by a ServerConnectionHandler.

```
package example.btsppecho.server;


import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Hashtable;
import java.util.Enumeration;
import javax.bluetooth.ServiceRecord;
import javax.microedition.io.Connector;
import javax.microedition.io.StreamConnection;
import javax.microedition.io.StreamConnectionNotifier;

import example.btsppecho.MIDletApplication;
import example.btsppecho.LogScreen;


public class ServerConnectionHandler
    implements Runnable
{
    private final static byte ZERO = (byte) '0';
    private final static int LENGTH_MAX_DIGITS = 5;

    // this is an arbitrarily chosen value:
    private final static int MAX_MESSAGE_LENGTH =
                              65536 - LENGTH_MAX_DIGITS;

    private final ServiceRecord serviceRecord;
    private final int requiredSecurity;
    private final ServerConnectionHandlerListener listener;
    private final Hashtable sendMessages = new Hashtable();

    private StreamConnection connection;
    private OutputStream out;
    private InputStream in;
    private volatile boolean aborting;
    private Writer writer;


    public ServerConnectionHandler(
               ServerConnectionHandlerListener listener,
               ServiceRecord serviceRecord,
               int requiredSecurity)
    {
        this.listener = listener;
        this.serviceRecord = serviceRecord;
        this.requiredSecurity = requiredSecurity;
        aborting = false;

        connection = null;
        out = null;
        in = null;
        listener = null;

        // the caller must call method 'start'
        // to start the reader and writer
    }


    public ServiceRecord getServiceRecord()
    {
        return serviceRecord;
    }
```

```
public synchronized void start()
{
    Thread thread = new Thread(this);
    thread.start();
}


public void close()
{
    if (!aborting)
    {
        synchronized(this)
        {
            aborting = true;
        }

        synchronized(sendMessages)
        {
            sendMessages.notify();
        }

        if (out != null)
        {
            try
            {
                out.close();
                synchronized (this)
                {
                    out = null;
                }
            }
            catch(IOException e)
            {
                // there is nothing we can do: ignore it
            }
        }

        if (in != null)
        {
            try
            {
                in.close();
                synchronized (this)
                {
                    in = null;
                }
            }
            catch(IOException e)
            {
                // there is nothing we can do: ignore it
            }
        }

        if (connection != null)
        {
            try
            {
                connection.close();
                synchronized (this)
                {
                    connection = null;
                }
            }
            catch (IOException e)
            {
                // there is nothing we can do: ignore it
            }
```

```
            }
        }
    }


    public void queueMessageForSending(Integer id, byte[] data)
    {
        if (data.length > MAX_MESSAGE_LENGTH)
        {
            throw new IllegalArgumentException(
                          "Message too long: limit is " +
                          MAX_MESSAGE_LENGTH + " bytes");
        }

        synchronized(sendMessages)
        {
            sendMessages.put(id, data);
            sendMessages.notify();
        }
    }


    private void sendMessage(byte[] data)
        throws IOException
    {
        byte[] buf = new byte[LENGTH_MAX_DIGITS + data.length];
        writeLength(data.length, buf);
        System.arraycopy(data,
                         0,
                         buf,
                         LENGTH_MAX_DIGITS,
                         data.length);
        out.write(buf);
        out.flush();
    }


    public void run()
    {
        // the reader

        // 1. open the connection and streams, start the writer
        String url = null;
        try
        {
            // 'must be master': false
            url = serviceRecord.getConnectionURL(
                                   requiredSecurity,
                                   false);

            connection = (StreamConnection) Connector.open(url);
            in = connection.openInputStream();
            out = connection.openOutputStream();

            LogScreen.log("Opened connection & streams to: '" +
                      url + "'\n");

            // start the writer
            Writer writer = new Writer(this);
            Thread writeThread = new Thread(writer);
            writeThread.start();

            LogScreen.log("Started a reader & writer for: '" +
                      url + "'\n");

            // open succeeded, inform listener
            listener.handleOpen(this);
        }
```

```
        catch(IOException e)
        {
            // open failed, close any connections/streams, and
            // inform listener that the open failed

            LogScreen.log("Failed to open " +
                          "connection or streams for '" +
                           url + "' , Error: " +
                           e.getMessage());

            close();

            listener.handleOpenError(
                        this,
                        "IOException: '" + e.getMessage() + "'");

            return;
        }
        catch (SecurityException e)
        {
            // open failed, close any connections/streams, and
            // inform listener that the open failed

            LogScreen.log("Failed to open " +
                          "connection or streams for '" +
                           url + "' , Error: " +
                           e.getMessage());

            close();

            listener.handleOpenError(
                        this,
                        "SecurityException: '" + e.getMessage() + "'");

            return;
        }

        // 2. wait to receive and read messages
        while (!aborting)
        {
            int length = 0;
            try
            {
                byte[] lengthBuf = new byte[LENGTH_MAX_DIGITS];
                readFully(in, lengthBuf);
                length = readLength(lengthBuf);
                byte[] temp = new byte[length];
                readFully(in, temp);

                listener.handleReceivedMessage(this, temp);
            }
            catch (IOException e)
            {
                close();
                if (length == 0)
                {
                   listener.handleClose(this);
                }
                else
                {
                   // we were in the middle of reading...
                   listener.handleErrorClose(this, e.getMessage());
                }
            }
        }
    }


    private static void readFully(InputStream in, byte[] buffer)
```

```
    throws IOException
{
    int bytesRead = 0;

    while (bytesRead < buffer.length)
    {
        int count = in.read(buffer,
                            bytesRead,
                            buffer.length - bytesRead);

        if (count == -1)
        {
            throw new IOException("Input stream closed");
        }
        bytesRead += count;
    }
}


private static int readLength(byte[] buffer)
{
    int value = 0;

    for (int i = 0; i < LENGTH_MAX_DIGITS; ++i)
    {
        value *= 10;
        value += buffer[i] - ZERO;
    }
    return value;
}


private void sendMessage(OutputStream out, byte[] data)
    throws IOException
{
    if (data.length > MAX_MESSAGE_LENGTH)
    {
        throw new IllegalArgumentException(
                       "Message too long: limit is: " +
                       MAX_MESSAGE_LENGTH + " bytes");
    }
    byte[] buf = new byte[LENGTH_MAX_DIGITS + data.length];
    writeLength(data.length, buf);
    System.arraycopy(data,
                     0,
                     buf,
                     LENGTH_MAX_DIGITS,
                     data.length);
    out.write(buf);
    out.flush();
}


private static void writeLength(int value, byte[] buffer)
{
    for (int i = LENGTH_MAX_DIGITS -1; i >= 0; --i)
    {
        buffer[i] = (byte) (ZERO + value % 10);
        value = value / 10;
    }
}


private class Writer
    implements Runnable
{
    private final ServerConnectionHandler handler;
```

```
        Writer(ServerConnectionHandler handler)
        {
            this.handler = handler;
        }


        public void run()
        {
            while (!aborting)
            {
                synchronized(sendMessages)
                {
                    Enumeration e = sendMessages.keys();
                    if (e.hasMoreElements())
                    {
                        // send any pending messages
                        Integer id = (Integer) e.nextElement();
                        byte[] sendData =
                                (byte[]) sendMessages.get(id);
                        try
                        {
                            sendMessage(out, sendData);

                            // remove sent message from queue
                            sendMessages.remove(id);

                            // inform listener that it was sent
                            listener.handleQueuedMessageWasSent(
                                        handler,
                                        id);
                        }
                        catch (IOException ex)
                        {
                            close(); // stop the networking thread

                            // inform that we got an error close
                            listener.handleErrorClose(handler,
                                                      ex.getMessage());
                        }
                    }

                    if (sendMessages.isEmpty())
                    {
                        try
                        {
                            sendMessages.wait();
                        }
                        catch (InterruptedException ex)
                        {
                            // this can't happen in MIDP: ignore it
                        }
                    }
                }
            }
        }
    }
}
```

4 Terms and Abbreviations

Term or abbreviation	Meaning
ACL	Asynchronous Connectionless Link
BD address	Bluetooth device address
COD	Class of Device
GIAC	General Unlimited Inquiry Access Code
HCI	Host Controller Interface
L2CAP	Logical Link and Adaptation Protocol
LIAC	Limited Dedicated Inquiry Access Code
OBEX	Object Exchange Protocol
RFCOMM	Bluetooth Serial Cable Emulation Protocol
SDP	Bluetooth Service Discovery Protocol
TCS	Telephony Control Protocol
UUID	Universally Unique Identifier

5 References

BTGAMES *Games Over Bluetooth: Recommendations To Game Developers*
Version 1.0; November 13, 2003
http://www.forum.nokia.com

BTSPEC *Specification Of The Bluetooth System, Version 1.1, Volume 1: Core*
http://www.bluetooth.org/spec

CLDC Connected Limited Device Configuration
v1.0 (JSR-30)
v1.1 (JSR-139)
http://www.jcp.org

JSR-82 *Java™ APIs For Bluetooth (JSR-82)*
http://www.jcp.org/en/jsr/detail?id=82

MIDPPROG *MIDP 1.0: Introduction To MIDlet Programming*
Forum Nokia, 2004
http://www.forum.nokia.com

UUID ISO/IEC 11578:1996 Information Technology — Open Systems Interconnection — Remote Procedure Call, http://www.iso.ch/cate/d2229.html