FORUM NOKIA

MIDP: Bluetooth API Developer's Guide

Version 2.0; October 31st, 2006

Java™

Change history

October 31st, 2006	Version 2.0	This document is based on two documents earlier published on Forum Nokia at http://www.forum.nokia.com/ • MIDP: OBEX API Developer's Guide, version 1.0 • Introduction To Developing Networked MIDlets Using Bluetooth The documents have been combined and harmonized and other minor updates have been made. In addition, some code comments have been added.

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

1 About this document

This document covers the Java™ APIs for Bluetooth Wireless Technology (JSR-82 specification at http://www.jcp.org/en/jsr/detail?id=82).

The document describes the basics of Bluetooth connections and protocols. The RFCOMM and OBEX protocols are described in more detail. The document also provides step-by-step guides to help you to build the example MIDlets described in the document.

Intended audience

This document is intended for MIDP developers, as well as Java™ EE and Java SE developers wishing to develop mobile Java applications or services. The reader should be familiar with MIDP 2.0 (especially the LCDUI API) before attempting to understand this example.

Scope

This document assumes a good knowledge of application and service development and the Java programming language. Furthermore, it assumes familiarity with enterprise application development. However, previous knowledge of developing for the mobile environment is not necessary.

This document focuses on Bluetooth and therefore explaining the Java technology is out of the scope of this documentation. For information on Java technology, see Java Technology web site at http://java.sun.com/ and Java Mobility Development Center at http://developers.sun.com/techtopics/mobility/. For additional information on MIDP tools and documentation, see Forum Nokia, Mobile Java section at http://www.forum.nokia.com/java.

2 Networking in Java™ ME

Mobile Information Device 2.0 (JSR-118) specification at http://www.jcp.org/en/jsr/detail?id=118 includes support for the Generic Connection Framework (GCF) at http://developers.sun.com/techtopics/mobility/midp/articles/genericframework/index.html, defined in the `javax.microedition.io` package of the Connected Limited Device Configuration (CLDC) 1.0 specification. This package allows you to use networking and push registry functions.

These networking functions include support for HTTP and HTTPS connections, as well as support for User Datagram Protocol (UDP), straight TCP, and serial port communication. In addition to those, with the support for push registry MIDlets can receive incoming communication requests and register for network connection events.

Extended networking features

The `javax.microedition.io` package only provides very basic networking possibilities. However, APIs such as Web Services, SIP, and Bluetooth have been created to allow for more extended features.

The J2ME™ Web Services specification at http://jcp.org/aboutJava/communityprocess/final/jsr172/index.html was created in order to allow support for XML parsing and the ability to invoke Web Services. The API contains two optional packages for XML processing and Web Services. These two packages are independent and thus a device may have either one or both of them.

The Session Initiation Protocol (SIP) is an IETF signaling protocol for establishing real-time calls and conferences over Internet Protocol networks. The SIP API for Java 2 Micro Edition (JSR-180) specification at http://www.jcp.org/en/jsr/detail?id=180 defines a Java Platform Micro Edition (Java ME) Optional Package, that enables resource limited devices (referred to as 'terminals' in the following) to send and receive SIP messages. The API is designed to be a compact and generic SIP API, which provides SIP functionality on a transaction level. The SIP API is integrated into the Generic Connection Framework defined in Connected, Limited Device Configuration (CLDC).

SIP in itself is an application-layer control protocol that can establish, modify, and terminate communication sessions. SIP can also invite participants to already existing sessions. SIP transparently supports name mapping and redirection services, which supports personal mobility - users can maintain a single externally visible identifier regardless of their network location (see RFC 3261 at http://www.ietf.org/rfc/rfc3261.txt).

The Java APIs for Bluetooth (JSR-82) specification at http://www.jcp.org/en/jsr/detail?id=82 defines APIs that can be used to exercise certain Bluetooth protocols defined in the Bluetooth specification volume 1 at http://www.bluetooth.org/spec/, and certain profiles defined in the Bluetooth specification volume 2 at http://www.bluetooth.org/spec/. For more information about these profiles, see the Java APIs for Bluetooth (JSR-82) specification at http://www.jcp.org/en/jsr/detail?id=82. The Bluetooth API is defined in such a way as to make it possible for additional and future profiles to be built on top of this API. This assumes that future changes to the Bluetooth specification remain compatible with this API.

The Bluetooth technology itself is an industrial specification for wireless personal area networks (PANs). The Bluetooth technology provides a way to connect and exchange information between devices such as personal digital assistants (PDAs), mobile devices, laptops, PCs, printers, digital cameras and video game consoles via a secure, globally unlicensed short-range radio frequency.

3 Bluetooth technology and Java™ APIs for Bluetooth (JSR-82) overview

This section gives an overview of the Bluetooth technology.

3.1 Bluetooth connections

Bluetooth devices need to exchange information before actual data transmission. This information exchange involves using the following:

- Device inquiry on page 9
- Service discovery on page 9
- Universally Unique Identifier (UUID) on page 9
- Name discovery on page 10
- Master/slave roles on page 10
- Security on page 10

3.1.1 Device inquiry

Since Bluetooth devices are mobile and form networks dynamically, they need a way of finding other nearby devices to connect to. This process is called the inquiry procedure in the Bluetooth Baseband Specification at http://www.bluetooth.org/spec/ (Specification of the Bluetooth System, Version 1.1, Volume 1: Core, Part B: Baseband Specification). A discovered device is identified by its Bluetooth device address, as well as its class of device (CoD) value.

The CoD value indicates the capability of the device. The CoD field shows if a device has rendering capabilities, like a printer, or is an audio device, like a headset. Some numerical values are reserved by the Bluetooth specification.

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).

Normally, an inquiry is done with the GIAC. But the Java™ and Symbian Bluetooth APIs also offer the possibility of using a LIAC.

Note: The S60 3rd Edition FP 1 implementation of the Bluetooth API does not support using LIAC because the Symbian Bluetooth stack does not support the operation. Also changing the discovery mode is not supported at all.

3.1.2 Service discovery

Service discovery is based on the Service Discovery Protocol (SDP) on page 12. In wired networks, there is normally a central infrastructure for managing connections; Bluetooth doesn't provide this. Instead, connections are made dynamically, and devices must determine what services are provided on discovered devices. This is called service discovery, which is the process by which applications locate and gather information about other services in the vicinity. Once a desired service has been discovered on a nearby device, an application may attempt to connect to the service and use it.

3.1.3 Universally Unique Identifier (UUID)

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

You need to generate unique application-specific UUIDs for your application(s). A variety of different ways of generating UUIDs are possible:

- Using the "uuidgen" utility of a local Linux-based computer
- Using other similar tools that are also available in Microsoft 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 at http://www.forum.nokia.com/info/sw.nokia.com/id/2b17fb6f-b9a4-4cd8-80fd-94b8251a048e/Games_Over_Bluetooth_v1_0_en.zip.html.

3.1.4 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. To obtain user-friendly names for remote devices, the remote name request procedure is used.

3.1.5 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 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.

Bluetooth Clients and Servers

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.

For a good overview of some general steps that are useful in creating a connection between client/server applications in Bluetooth, see Games Over Bluetooth: Recommendations To Game Developers at http://www.forum.nokia.com/info/sw.nokia.com/id/2b17fb6f-b9a4-4cd8-80fd-94b8251a048e/Games_Over_Bluetooth_v1_0_en.zip.html.

3.1.6 Security

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.

3.2 Bluetooth protocols

Additional protocols have been defined for different purposes. This section describes those protocols.

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.

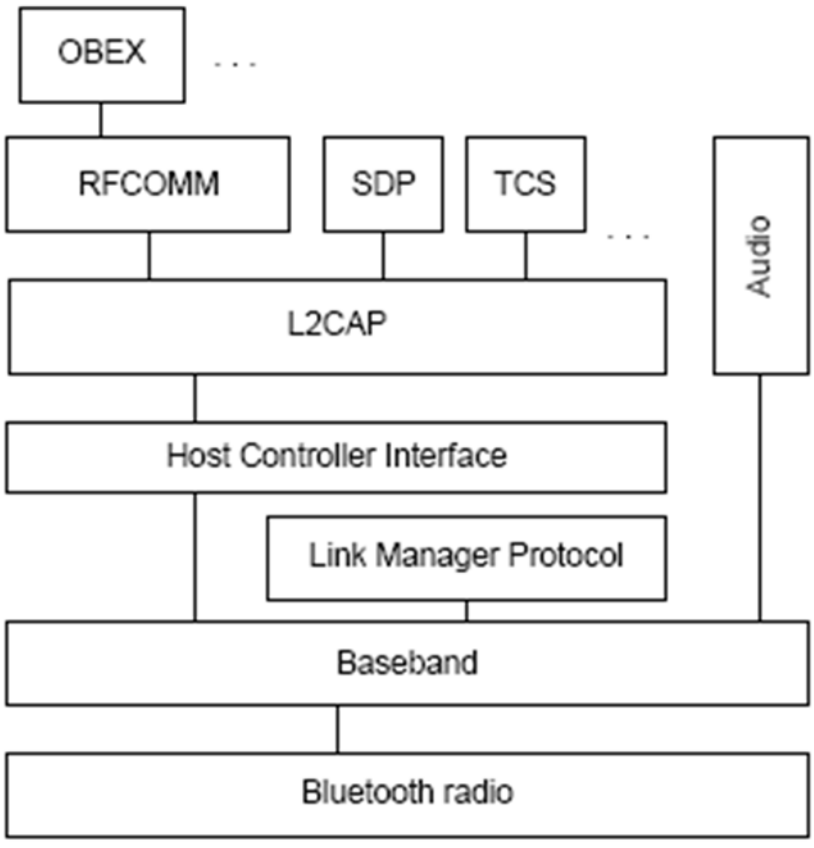

Figure 1: Bluetooth protocol stack

The following protocols are described here:

- L2CAP on page 12
- RFCOMM on page 12
- Service discovery protocol (SDP) on page 12

- OBEX on page 12

3.2.1 L2CAP

Logical Link Control and Adaptation Protocol (L2CAP) is the lowest-level Bluetooth protocol that can be accessed by an application. The protocol overhead for L2CAP is 4 bytes. L2CAP is recommended if you have a small amount of data and you need fast response times.

3.2.2 RFCOMM

RFCOMM is a Bluetooth protocol based on L2CAP. The protocol overhead for RFCOMM is between 4 and 5 bytes for small packets. For every 127 bytes of data, the header increases in size by 1 byte. So the overall protocol overhead is about 8 to 9 bytes for data less than 127 bytes (4 bytes from L2CAP and 4 to 5 bytes from RFCOMM).

3.2.3 Service discovery protocol (SDP)

Service discovery protocol (SDP) enables clients to discover services on other nearby Bluetooth devices or servers, as well as detecting when the services are no longer available. SDP considers any feature usable by other Bluetooth devices to be a service.

For more information on SDP, see the Specification of the Bluetooth System, Core, part E at http://www.bluetooth.org/spec/

3.2.4 OBEX

> **Note:** To see the OBEX protocol being used in an application example, see section Example: Creating an OBEX connection on page 78.

The OBEX protocol is a compact binary, session-layer protocol for exchanging complex data in a simple, efficient manner. Originally it was created for infrared connectivity and was used in the Infrared Data Association (IrDA) protocol stack. Bluetooth Special Interest Group (SIG) at https://www.bluetooth.org/ has defined the use of the OBEX protocol over Bluetooth connections in the IrDA Interoperability Specification at https://www.bluetooth.org/.

The purpose of the protocol is to support sending and receiving objects in a simple and spontaneous manner. For example, pushing business cards or synchronizing calendars on multiple devices is handled with this protocol.

A major use of OBEX is a "Push" or "Pull" application. However, OBEX is not limited to quick connect-transfer-disconnect scenarios - it also allows sessions in which transfers take place over a period of time, maintaining the connection even when it is idle.

OBEX performs a function similar to HTTP; however, OBEX can work on devices that cannot afford the substantial resources required for an HTTP server. It also targets devices with different usage models from the Web. The OBEX protocol, in its simplest form, is quite compact and requires a small amount of code to implement. It can reside on top of any reliable transport.

The OBEX session protocol allows the higher layers of the stack to work with logical elements of a higher layer of abstraction than that of the packet formats used by the transport protocols, for example, RFCOMM. As the IrDA OBEX specification at http://www.irda.org/ states, this is ensured by two major elements of the OBEX protocol:

- A model for representing objects and information that describes the objects.

  The object model carries information about the objects being sent and contains the objects themselves. The object model is built entirely with parsable headers, similar to the headers in HTTP.
- A session protocol that provides a structure for the "conversation" between devices.

  The session protocol structures the dialogue between two devices. The session protocol uses a binary packet-based client-server request-response model.

The table below provides the exact definitions of OBEX terms.

Table 3.1: OBEX definitions. Source: IrDA OBEX specification

Term	Meaning
OBEX Client	An OBEX Client is the entity that initiates the underlying transport connection to an OBEX server and initiates OBEX operations.
OBEX Server	An OBEX Server is the entity that responds to OBEX operations. The OBEX server waits for the OBEX client to initiate the underlying transport connection.
OBEX Connection	An OBEX Connection is a virtual binding between two applications or services. An OBEX connection is initiated by sending an OBEX CONNECT packet. Once a connection is established, all operations sent over the connection are interpreted in a continuous context.
Application	An OBEX application communicates using a proprietary method known only by the manufacturer. Such applications can only expect to be understood by exact peers. Alternatively, an application may be a service with proprietary extensions. In this case the application must know if it is communicating with a service or application peer.

3.2.4.1 OBEX connections

The OBEX protocol is a client-server protocol. The protocol defines the notion of the client, which sends a request, and the server, which replies with a response. The client and server exchange the requests and responses during a connection session.

The OBEX protocol follows a client-server request-response paradigm for the conversation format. The request-response pair is referred to as an operation. The OBEX protocol defines two operations for controlling the connection, CONNECT and DISCONNECT. After the connection session is established, the client can perform various operations on the server. Two basic operations are the GET operation for downloading an object from the server and the PUT operation for uploading objects. During the connection session, these operations can be performed as many times as needed in any order.

The maximum size of the object that can be transmitted in the single PUT or GET operation is 65535 bytes. Maximum object size for the session is negotiated during the connection procedure. Both peers exchange their maximum allowed sizes and the smaller value is selected. When PUT and GET operations are requested to transmit large objects, the protocol handles transmission of the object in several packages. It helps to maintain good synchronization when requests and responses are broken into multiple OBEX packets that are limited to a size specified at connection time. Each request packet is acknowledged by the server with a response packet. The whole procedure is transparent to an application, so PUT or GET appears as one lengthy operation (for example, by using JSR-82 API an application developer can transmit large objects without bothering to split the objects into packages).

In order to cancel such lengthy operations, an ABORT command is defined. The ABORT operation may come in the middle of a request-response sequence. If the ABORT command is invoked while running an operation, the data transmission is canceled.

Request and response general formats are shown in the tables below.

Table 3.2: General form of a request packet. Source: IrDA OBEX specification]

Byte 0	Bytes 1, 2	Bytes 3 to n
opcode	packet length	headers or request data

Table 3.3: General form of a response packet. Source: IrDA OBEX specification

Byte 0	Bytes 1, 2	Bytes 3 to n
response code	response length	response data

Note: Each peer may implement both the client and the server, and thereby create a peer-to-peer relationship between applications by using a pair of OBEX sessions, one in each direction.

The aforementioned operations are the basic ones, but not the only ones defined in the OBEX protocol. The full list is given in the table below.

Table 3.4: OBEX operations. Source: IrDA OBEX specification

Opcode (w/high bit set)	Definition	Meaning
0x80 (high bit always set)	Connect	Choose your partner, negotiate capabilities
0x81 (high bit always set)	Disconnect	Signal the end of the session
0x02 (0x82)	Put	Send an object
0x03 (0x83)	Get	Get an object
0x04 (0x84)	Reserved	
0x85 (high bit always set)	SetPath	Modifies the current path on the receiving side
0xFF (high bit always set)	Abort	Abort the current operation
0x06 to 0x0F	Reserved	Not to be used without extension to IrDA OBEX specification
0x10 to 0x1F	User definable	Use as you wish with peer application
Bits 5 and 6 are reserved and should be set to zero. Bit 7 of the opcode means Final packet of request.		

For more detailed definitions of each operation, see the IrDA OBEX specification at http://www.irda.org/.

IrDA OBEX Exchange Protocol (see IrDA OBEX specification at http://www.irda.org/) defines the application framework concept, which is built on top of the OBEX protocol. It ensures interoperability between devices, for example, sending messages to the Inbox of a device.

3.2.4.2 Headers

Headers allow transferring metadata together with the objects. By using headers, OBEX peers can transmit different kinds of information related to a transmitted object, session, etc. For example, the name of the object, its size, a description, and the type of an object can be defined in the headers. According to the IrDA OBEX specification at http://www.irda.org/ a header consists of two elements:

```
<HI, the header ID>
```

```
<HV, the header value>
```

`HI, the header ID`, is an unsigned one-byte integer that identifies what the header contains and how it is formatted. HV consists of one or more bytes in the format and meaning specified by HI. The table below provides information about the OBEX headers. Full information on each header can be found in the IrDA OBEX specification at http://www.irda.org/.

Table 3.5: List of OBEX headers. Source: IrDA OBEX specification

ID	Name	Description
0xC0	Count	Number of objects (used by Connect)
0x01	Name	Name of the object (often a file name)
0x42	Type	Type of object, e.g. text, html, binary, manufacturer-specific
0xC3	Length	The length of the object in bytes
0x440xC4	Time	Date/time stamp - ISO 8601 version - preferredDate/time stamp - 4 byte version (for compatibility only)
0x05	Description	Text description of the object
0x46	Target	Name of the service that an operation is targeted to
0x47	HTTP	An HTTP 1.x header
0x480x49	Body End of Body	A chunk of the object body. The final chunk of the object body
0x4A	Who	Identifies the OBEX application, used to tell if it is talking to a peer
0xCB	Connection Id	An identifier used for OBEX connection multiplexing
0x4C	App. Parameters	Extended application request & response information
0x4D	Auth. Challenge	Authentication digest-challenge
0x4E	Auth. Response	Authentication digest-response
0x4F	Object Class	OBEX Object class of object
0x10to 0x2F		Reserved
0x30 to 0x3F		User-defined

Note: Note that the OBEX specification denotes that all headers are optional.

3.2.4.3 OBEX profiles

The OBEX protocol is the basis for several profiles, which are meant to be commonly implemented on devices that employ the Bluetooth wireless technology. The profiles are:

- General Object Exchange Profile (GOEP) - the general profile that ensures generic interoperability for the application profiles using OBEX.
- File Transfer Profile - the profile that defines the ability to transfer a file considering the file system of both devices.
- Object Push Profile - the profile that allows a Bluetooth device to push an object to the Inbox of another Bluetooth device.
- Synchronization Profile - the profile that defines the requirements for the protocols and procedures to provide device-to-device synchronization of such information as phonebook, calendar, notes, etc.

Note: The Synchronization Profile is not supported in Nokia devices.

All these profiles are based on the principle that certain objects are pushed to and pulled from a device, and aim to allow rapid communications and interoperability among portable devices. For more information on the profiles, see the Bluetooth Specification for the following:

- General Object Exchange Profile (GOEP), v1.1 at https://www.bluetooth.org/
- File Transfer Profile, Bluetooth Profiles, v1.1 at https://www.bluetooth.org/
- File Transfer Profile, Bluetooth Profiles, v1.1 at https://www.bluetooth.org/
- Synchronization Profile, Bluetooth Profiles, v1.1 at https://www.bluetooth.org/

4 Using the Java™ APIs for Bluetooth (JSR-82)

This section provides information on how to create Bluetooth connections using the Bluetooth API (JSR-82) specification.

The JSR-82 specification at http://www.jcp.org/en/jsr/detail?id=82 consists of two parts: the core Bluetooth API (`javax.bluetooth`) and the OBEX API (`javax.obex`). The specification was first introduced in the S60 2nd Edition and Series 40 2nd Edition.

4.1 Bluetooth connections

The following sections describe the process of creating Bluetooth connections with the Bluetooth API (JSR-82). Using the following concepts is explained:

4.1.1 Device inquiry

The first step in a networked Bluetooth application is often to discover other devices that are in range.

> **Note:** The S60 3rd Edition FP 1 implementation of the Bluetooth API does not support using LIAC because the Symbian Bluetooth stack does not support the operation. Also changing the discovery mode is not supported at all.

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 S60 and Series 40 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.

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. 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.)

4.1.2 Service discovery

Inquiry is followed by service discovery, which is based on the Service Discovery Protocol (SDP) on page 12

Your application will have a particular UUID on page 9 that you have associated with it. An UUID is needed in the service discovery process. The following snippet of code helps to illustrate this.

```
// 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.

An application normally performs service discovery on each found device of interest. The Java APIs For Bluetooth (JSR-82) 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 specification at http://www.jcp.org/en/jsr/detail?id=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). For related information on Nokia devices, see Games Over Bluetooth: Recommendations To Game Developers at http://www.forum.nokia.com/info/sw.nokia.com/id/2b17fb6f-b9a4-4cd8-80fd-94b8251a048e/Games_Over_Bluetooth_v1_0_en.zip.html.

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

4.1.3 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.

4.1.4 Security

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.

4.2 Bluetooth protocols in JSR-82

This section describes how to use the Bluetooth protocols with the Bluetooth API (JSR-82). Using the following protocols is explained:

4.2.1 L2CAP

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. 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.

4.2.2 RFCOMM

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).

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.

> **Note:** To see the RFCOMM protocol being used in an application example, see section Example: Creating a client server pair using RFCOMM on page 24.

4.2.3 OBEX

The OBEX API (JSR-82) uses the Generic Connection Framework (GCF) for networking (for more details on GCF, see CLDC specification (JSR-139) at http://jcp.org/en/jsr/detail?id=139). In practice it means that some classes and interfaces from the `javax.microedition.io` package are used as illustrated in the figure below.

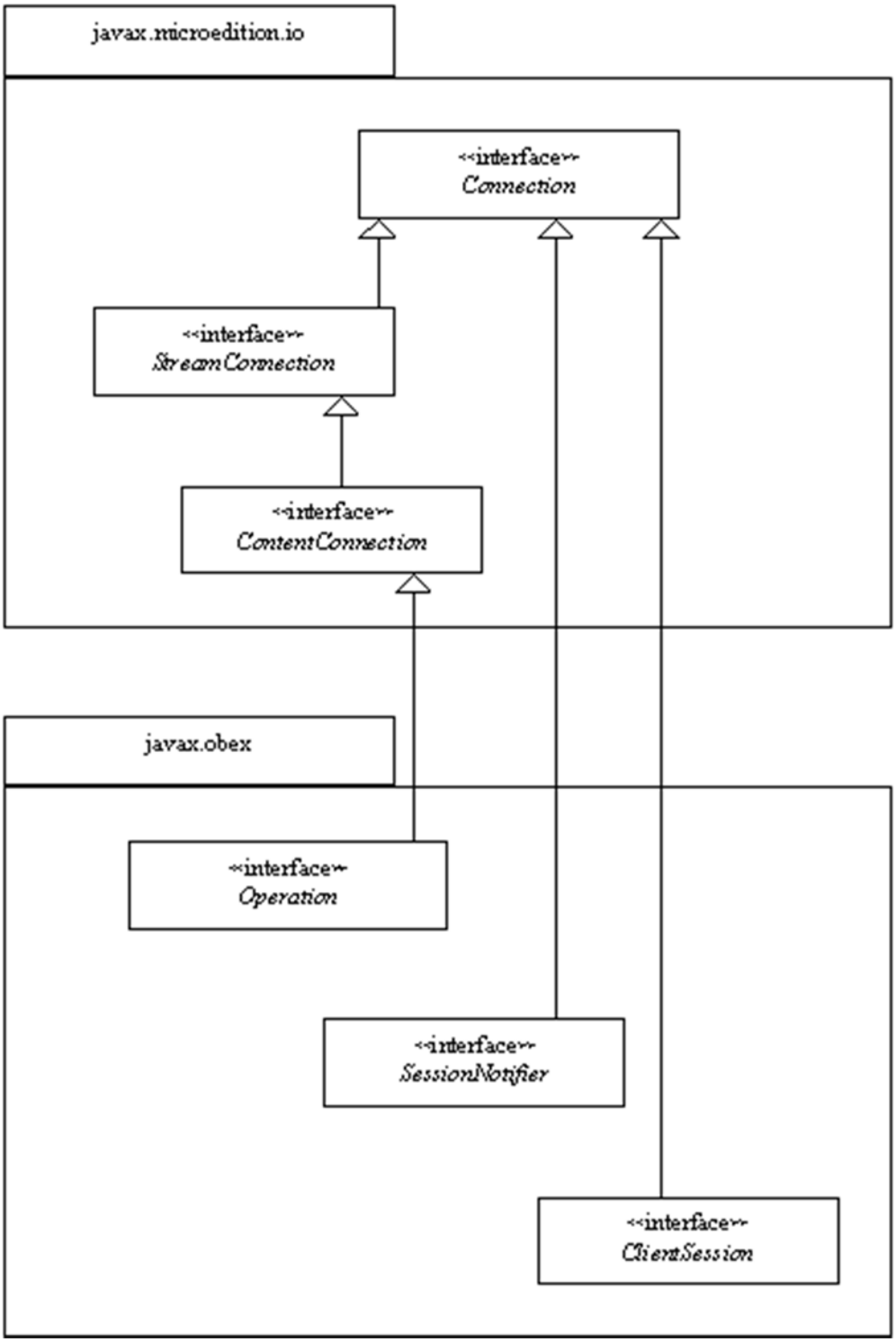

Figure 2: OBEX in the Generic Connection Framework. Source: [JSR-82]

The GCF is used to establish the transport layer communication. Once the connection is established, peers can start exchanging OBEX commands. (Section OBEX connectionsl on page 13 introduces the OBEX protocol operations.)

JSR-82 supports the following operations:

- CONNECT
- PUT
- GET
- SETPATH
- ABORT
- CREATE-EMPTY
- PUT-DELETE
- DISCONNECT

In addition, JSR-82 gives access to the following OBEX headers:

- Count
- Name
- Type
- Length
- Time
- Description
- Target
- HTTP
- Body
- End of Body
- Who
- Connection ID
- Application Parameters
- Authentication Challenge
- Authentication Response
- Object Class
- User defined

The class `HeaderSet` is used to manipulate most of the headers. Exceptions are:

- Body and End of Body headers, which are accessible using the `Operation` class.
- Connection ID header, which is accessible using the `ClientSession` class.
- Authentication Challenge and Authentication Response headers, which are accessible using both the `HeaderSet` and the `Authenticator` classes.

4.2.3.1 OBEX Client connections

Firstly, the client establishes the Bluetooth transport layer connection by using the GCF `Connector.open()` with OBEX client URL. The JSR-82 specification defines three types of URL for using OBEX over RFCOMM (Bluetooth), TCP/IP, and IrDA. For example, an OBEX client connection URL over RFCOMM may look like:

```
btgoep://00A3920B2C22:12
```

where:

- `btgoep` - is the protocol (Bluetooth Generic Object Exchange Profile)

- `00A3920B2C22` - is the Bluetooth Address of the target device
- `12` - is the RFCOMM server channel identifier (each OBEX service has its own identifier)

This information is obtained during device discovery and service discovery. The OBEX client URL is similar to other JSR-82 Bluetooth URL formats and allows using various optional parameters, as defined in the JSR-82 specification.

In case an application wants to connect to a standard OBEX service (for example, send the message to the Inbox or browse the file system of the remote device), it must use the UUID defined in Bluetooth Assigned Numbers at https://www.bluetooth.org/foundry/assignnumb/document/assigned_numbers. For example, the short Universally Unique Identifier (UUID) `0x1105` is used for searching the standard Inbox service and the short UUID `0x1106` - for Bluetooth OBEX File Transfer Profile services.

Note: Nokia Prototype SDK 3.0 / 4.0 for Java™ Platform, Micro Edition used for developing the example application supports only OBEX over RFCOMM (`btgoep`).

The `Connector.open()` method returns a `ClientSession` object. The `ClientSession` interface provides methods for accessing the client part of the OBEX protocol, which is responsible for sending requests to a server; for example, in order to issue the OBEX CONNECT, call method `connect()` of the `ClientSession` interface. GET and PUT requests are issued by invoking `get()` and `put()` methods respectively.

In order to create headers, the method `createHeaderSet()` of the interface `ClientSession` is used. Using the returned `HeaderSet` object you can define a set of headers, which can be transmitted during such operations as CONNECT, PUT, GET, etc.

Methods `get()` or `put()` return an `Operation` object representing the current PUT or GET operation. In practice, by using an `Operation` object you can access transmitted data via input and output streams. In case of lengthy operations, the `Operation.abort()` method is used to stop the current operation. However, the `Operation.abort()` does not affect the connection session and new operations can be executed after the abortion.

Other OBEX commands are accessible via the `ClientSession` object in a similar manner and are thoroughly described in the JSR-82 specification.

4.2.3.2 OBEX Server connections

In order to create a server connection, the GCF class `Connector` is used as well. The `Connector.open()` method is called with a server URL as a parameter. The server URL for OBEX over RFCOMM may look like:

```
btgoep://localhost:ed495afe28ed11da94d900e08161165f
```

where:

- `btgoep` - is the protocol (Bluetooth Generic Object Exchange Profile)
- `localhost` - identifies the server URL
- `ed495afe28ed11da94d900e08161165f` - is the service UUID

The OBEX server URL is similar to other JSR-82 Bluetooth URL formats and allows using various optional parameters, as defined in the JSR-82 specification. The service UUID is application specific. However, if an application wants to support a particular Bluetooth profile, it is responsible for adding the corresponding profile UUID to the service record. All the UUID values for standard Bluetooth profiles are defined in the Bluetooth Assigned Numbers.

The `Connector.open()` method returns a `SessionNotifier` object. The method `acceptAndOpen()` of the `SessionNotifier` starts waiting for incoming transport layer connections. This follows the JSR-82 server connection scheme and allows modifications in the service records (for details, see JSR-82 specification at http://www.jcp.org/en/jsr/detail?id=82). However, the difference between the OBEX

connection and other types of JSR-82 connections (RFCOMM and L2CAP) is that the `acceptAndOpen()` method takes a `ServerRequestHandler` object as a parameter. The `ServerRequestHandler` class is a callback class that has methods corresponding to incoming OBEX requests.

After the transport layer connection is established, the `acceptAndOpen()` method returns a `Connection` object, which represents the connection to a single client. The server communicates with the remote peer by two means: the `Connection` object and the `ServerRequestHandler` callback methods that are called on incoming OBEX requests. A server application usually creates its own class that extends the `ServerRequestHandler` class and implements the methods corresponding to requests the OBEX server supports. For example, if the server supports only GET requests, it has to implement the `onGet()` method.

Methods `onGet()` and `onPut()` receive an `Operation` object as a parameter. The application uses an `Operation` object to read and write data as described above. After the request is handled, the method must return a response code. All response codes are defined in the `ResponseCode` class.

Note: Note: A server application should not call the `Operation.abort()` method.

4.2.3.3 OBEX Security

The JSR-82 OBEX API complies with the MIDP 2.0 Security model and defines the following permissions and sensitive APIs.

Table 4.6: OBEX-related security permissions and API calls. Source: MIDP 2.0.

Permission	Permitted API calls	Function group
`javax.microedition.io.Connector.obex.client`	Connector.open("btgoep://<server BD_ADDR> ...")Connector.open("irdaobex:// discover ...")Connector.open("irdaobex:// addr ...")Connector.open("irdaobex://conn ...") Connector.open("irdaobex://name ...")	Local connectivity
`javax.microedition.io.Connector.obex.client.tcp`	Connector.open("tcpobex://<server IP_ADDR> ...")	Net access
`javax.microedition.io.Connector.obex.server`	Connector.open("btgoep://localhost: ...") Connector.open("irdaobex://localhost: ...")	Local connectivity
`javax.microedition.io.Connector.obex.server.tcp`	Connector.open("tcpobex://:<PORT>") Connector.open("tcpobex://")	Net access

The JSR-82 specification allows performing the OBEX authentication procedure of the remote peer. The authentication is based on the shared secret or password and follows a challenge-response scheme. The authentication was not used in the Business Card Exchanger example and is not described in this document. For more information on the authentication process, see the JSR-82 specification at http://www.jcp.org/en/jsr/detail?id=82 .

Also basic Bluetooth security mechanisms (authorization, encryption, and authentication) are available for OBEX connections. They are accessible via URL parameters and API calls.

5 Example: Creating a client/server pair using RFCOMM

This tutorial shows you how to create a simple client/server pair using the RFCOMM protocol, where:

- 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.

With the help of this tutorial, you can build the example from scratch and familiarize yourself with some common features related to the Bluetooth API. Alternatively, you can download the example files and run them immediately with your S60 MIDP SDK (3rd Edition or later). For more information on importing, building, and running Java projects, see Getting Started with Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™.

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

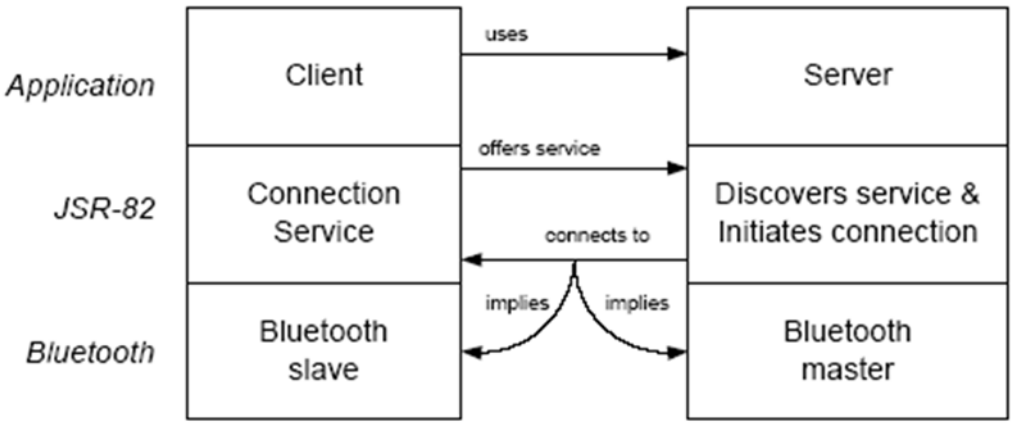

Figure 3: Different roles of the client/server application pair (at various levels)

The example is also provided on Forum Nokia at http://www.forum.nokia.com/info/sw.nokia.com/id/0b51461e-5f77-40f4-b755-7915ad4d0e31/MIDP_Bluetooth_RFCOMM_Example_v1_0_en.zip.html. Instead of creating the example from scratch as shown in this document, you can download the example files and run them immediately with your SDK.

Naming conventions

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.

Required 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.

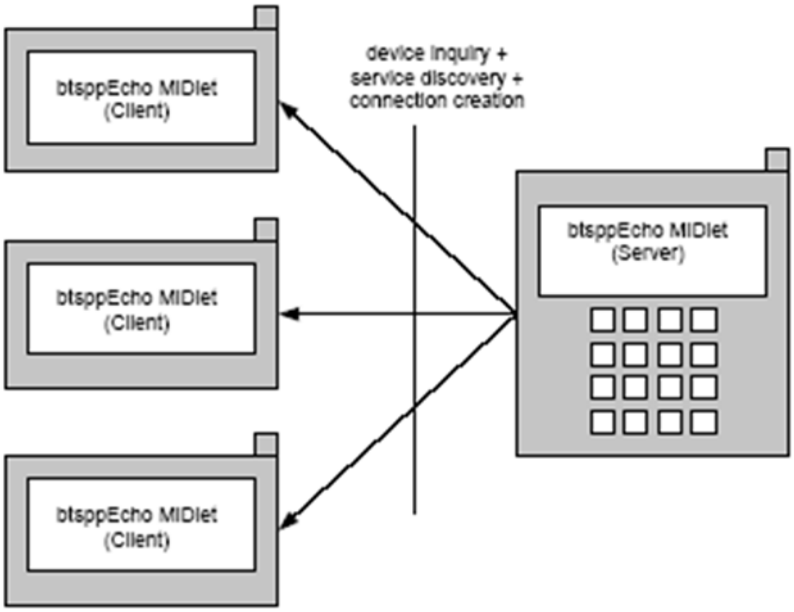

Figure 4: Device inquiry, service discovery, and connection creation

Client/server connections

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.

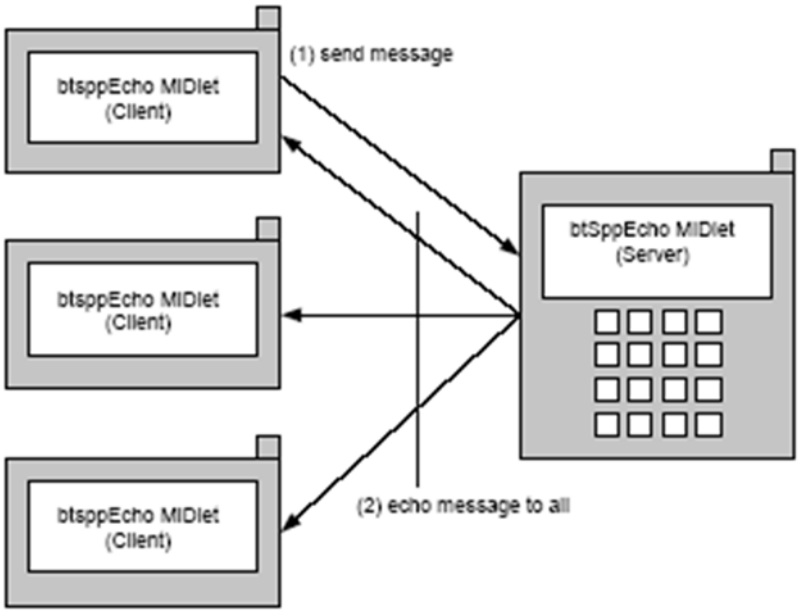

Figure 5: A client sends a message and the server sends that message to all clients

Dropped 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).

5.1 Prerequisites

Note: The S60 3rd Edition FP 1 implementation of the Bluetooth API does not support using LIAC because the Symbian Bluetooth stack does not support the operation. Also changing the discovery mode is not supported at all.

The example MIDlet makes use of the standard MIDP 2.0 application framework. You should also have Series 40 MIDP SDK, 3rd Edition or later. Familiarity with MIDP 2.0 is highly recommended before attempting to understand this example.

Note: For further information about setting up an appropriate environment for the emulators, see the SDK documentation (comes with the SDK, downloadable from the Forum Nokia Web site at http://www.forum.nokia.com/main.html).

5.2 Creating the project environment

The project environment is constructed using the instructions for building MIDlet in IDEs or using command line, that come with the Getting Started with Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library.

5.3 Implementation of btsppEcho MIDlet

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.

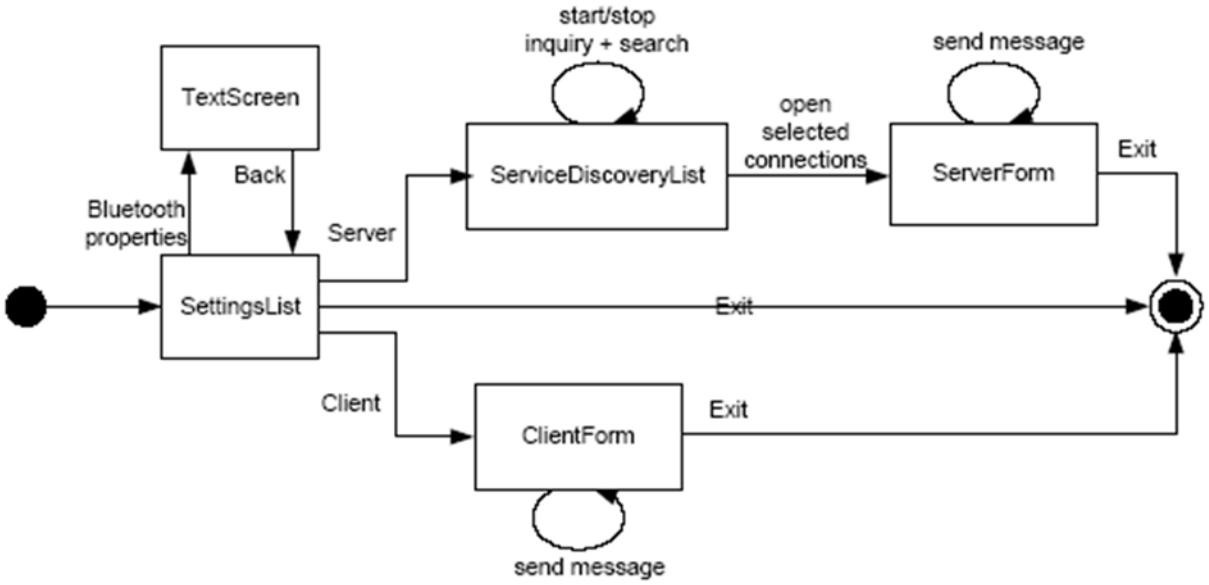

Figure 6: Screen map

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.

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.

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.

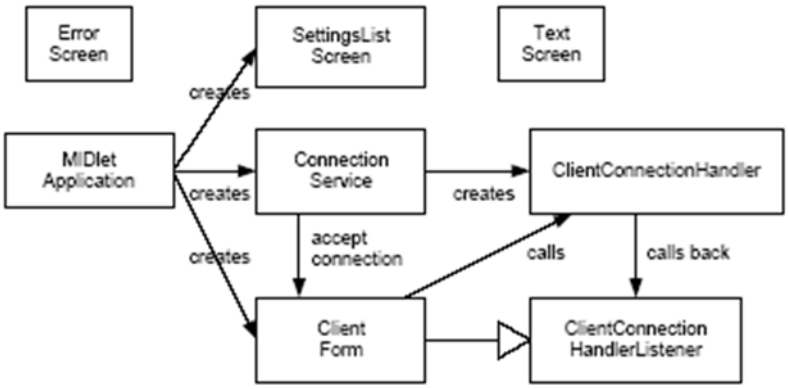

Figure 7: Class diagram of classes used by the client

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.

Server

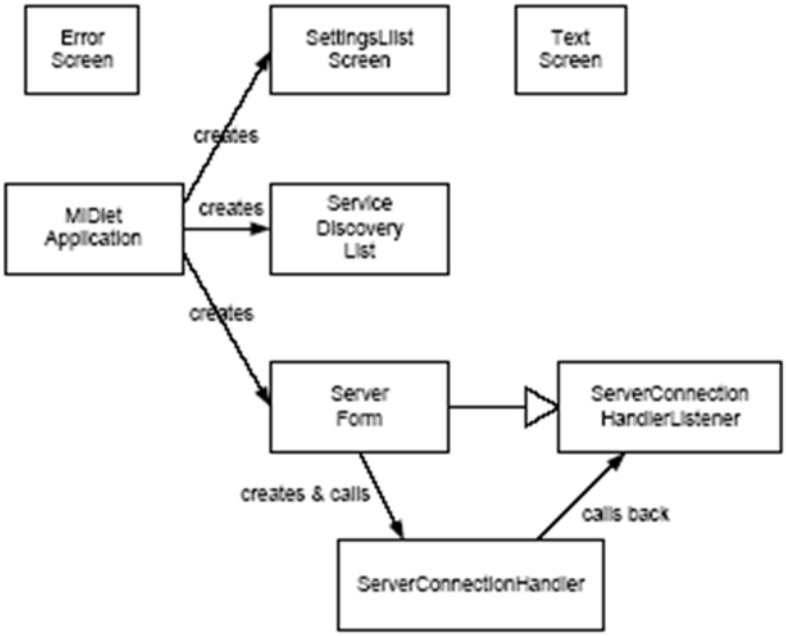

Figure 8: Class diagram of classes use by the server

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.

5.4 Developing btsppEcho MIDLet

5.4.1 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.

To create the class:

1 Create the `MIDletApplication` class file.

2 Assign the class to the package `example.btsppecho` and import the required classes.

```
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;
```

3 Implement the basic functionality of the class.

Use the methods `initialDiscoverableMode` and `restoreDiscoverableModeOnExit` in order to read the discoverable mode of the device at start and to restore the mode on exit.

Implement the `BluetoothStateException` class in order to throw an exception in cases when the system is not in a state where it can support the request.

Utilize the `getLocalDevice` and `getDiscoverable` methods of the `LocalDevice` class for retrieving an object representing the local Bluetooth device and the discoverable mode the device is presently in.

```
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
            }
        }
    }
```

4 Implement screen call backs.

Note the use of the `ServiceRecord` interface and the authentication and encryption choices.

```
    // 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();
        }
```

5 Use the method `serverFormAddConnection` for adding server connections.

```
        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);
        }
    }
```

5.4.2 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 Bluetooth 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.

To create the class:

1 Create the `SettingsList` class file.

2 Assign the class to the package `example.btsppecho` and import the required classes.

```
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;
```

3 Implement the functionality of the class.

For more information on developing user interfaces, see Working with UI section of the Java™ ME Developer's Library at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm.

```
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
```

4 Use the DiscoveryAgent class to set the inquiry type. The inquiry type is used to be discoverable by a client, or when performing discovery by a server.

Note: The S60 3rd Edition FP 1 implementation of the Bluetooth API does not support using LIAC because the Symbian Bluetooth stack does not support the operation. Also changing the discovery mode is not supported at all.

```
        // 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);
            }
        }
    }
```

5.4.3 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.

This class deals mainly with UI related classes. For more information on developing user interfaces, see UI and Graphics section of the Java™ ME Developer's Library at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm.

To create the class:

1 Create the `TextScreen` class file.

2 Assign the class to the package `example.btsppecho` and import the required classes.

```
package example.btsppecho;

import javax.microedition.lcdui.*;
```

3 Implement the functionality of the class.

```
// 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);
        }
    }
```

5.4.4 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.

To create the class:

1 Create the `LogScreen` class file.

2 Assign the class to the package `example.btsppecho` and import the required classes.

```
package example.btsppecho;


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

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

3 Implement the basic functionality of the class.

```
// 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();
        }
    }
```

4 Use DiscoveryAgent and the `DiscoveryListener` classes for the helper methods.

```
    // 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;
    }
}
```

5.4.5 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.

For more information on LCDUI, see UI and Graphics section of the Java™ ME Developer's Library at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm.

To create the class:

1 Create the `ClientForm` class file.

2 Assign the class to the package `example.btsppecho` and import the required classes.

```
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;
```

3 Implement the functionality of the class.

```
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 + "\"");
        }
    }
```

5.4.6 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).

To create the class:

1 Create the `ServiceDiscoveryList` class file.

2 Assign the class to the package `example.btsppecho` and import the required classes.

```
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;
```

3 Use the `getProperty` method of the `LocalDevice` class with the property `bluetooth.sd.trans.max` to request for the maximum number of concurrent service discovery transactions.

To create the discovery agent, use the `getLocalDevice` and getDiscoveryAgent methods of the `LocalDevice` class.

```
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.BACK, 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;
}
```

4 Implement the command action functionality.

```
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);
        }
```

Use the `setDiscoverable(DiscoveryAgent.NOT_DISCOVERABLE)` method to take the device out of discoverable mode. Then disable inquiries.

```
        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);
                }
            }
```

Use the `LocalDevice.getProperty()` method with the `bluetooth.connected.devices.max` property to request the maximum number of connected devices supported.

```
            // 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;
        }
```

5 Implement discovery listener callbacks.

To implement the callbacks, use the following classes:

- The `RemoteDevice` class is used for representing a remote device. The method `getBluetoothAddress` returns the Bluetooth address of the remote device.

- `DeviceClass` class represents the class of device (CoD) record. The method `getMajorDeviceClass` returns the major device class.
- `ServiceRecord` class represents the characteristics of a Bluetooth service.

```
// 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 friendly name and/or bluetooth address
```

```
        // to identify the matching Device + ServiceRecord
        // (Note: Devices with different Bluetooth addresses
        // might have the same friendly name, for example a
        // device's default friendly name.)

        // 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;
                    }
                }
            }
        }
    }
```

5.4.7 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.

To create the class:

1 Create the `ServerForm` class file.

2 Assign the class to the package `example.btsppecho` and import the required classes.

```
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;
```

3 Add `Form` items, create commands, add the commands, and set a command listener.

```
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);
    }
```

4 Implement connections.

```
    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.
    }
```

5 Implement `ServerConnectionHandlerListener` interface methods.

```
    // 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 + "'");
    }
}
```

5.4.8 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.

To create the class:

1 Create the `ConnectionService` class file.

2 Assign the class to the package `example.btsppecho.client` and import the required classes.

```
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;
```

3 Implement the functionality of the class.

```
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);
            }
```

```
                }
            }
        }
    }
```

5.4.9 ClientConnectionHandlerListener

This interface describes the callbacks of a `ClientConnectionHandler`.

To create the class:

1 Create the `ClientConnectionHandlerListener` class file.

2 Assign the class to the package `example.btsppecho.client`.

```
package example.btsppecho.client;
```

3 Implement the functionality of the class.

```
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);

}
```

5.4.10 ClientConnectionHandler

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

To create the class:

1 Create the `ClientConnectionHandler` class file.

2 Assign the class to the package `example.btsppecho.client` and import the required classes.

```
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;
```

3 Implement the functionality of the class.

```
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();
    }
```

4 Implement the `run` method.

```
    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;
    }
```

```
    }
```

5 Create an inner class `Writer` for handling message sending.

```
    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
                        }
                    }
                }
            }
        }
    }
```

5.4.11 ServerConnectionHandlerListener

This interface describes the callbacks of a `ServerConnectionHandler`.

To create the class:

1 Create the `ServerConnectionHandlerListener` class file.

2 Assign the class to the package `example.btsppecho.server`.

```
package example.btsppecho.server;
```

3 Implement the functionality of the class.

```
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);

}
```

5.4.12 ServerConnectionHandler

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

To create the class:

1 Create the `ServerConnectionHandler` class file.

2 Assign the class to the package `example.btsppecho.server` and import the required classes.

```
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;
```

3 Implement the basic functionality of the class.

```
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();
    }
```

4 Implement the `run` method.

```
    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;
    }
}
```

5 Create an inner class `Writer` for handling message sending.

```
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
                        }
                    }
                }
            }
        }
    }
}
```

5.5 Building and running in an emulator

After you have created all the necessary files for the project, you can build the project and run the application in an emulator.

> **Note:** The MIDlet used in this example was built using the Eclipse IDE. Please see the instructions for building MIDlet projects for Eclipse that come with the Getting Started with Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library.
>
> You can also construct them using command line or other IDEs of your choice.

The example is also provided on Forum Nokia at http://www.forum.nokia.com/info/sw.nokia.com/id/0b51461e-5f77-40f4-b755-7915ad4d0e31/MIDP_Bluetooth_RFCOMM_Example_v1_0_en.zip.html. Instead of creating the example from scratch as shown in this document, you can download the example files and run them immediately with your SDK.

5.6 Deploying to a device

Deploy test packages onto your mobile device, using a USB cable, Bluetooth connection, infrared, or other hardware compatible connection. Alternatively, deploy the application using OTA technology.

For instructions on how to do this, see the Getting Started with Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library.

6 Example: Creating an OBEX connection

This section shows you how to create a simple application for exchanging business cards from scratch and how to run it with an emulator.

A typical scenario of using the application goes as follows: When the MIDlet is started for the first time, the user selects his or her own card from the address book. When two users meet, they start the MIDlet, which exchanges the business cards. New business cards are saved in the phone book on each device.

Note: The device must have at least one contact (business card) in the phone book. Otherwise the MIDlet will not run correctly. The business cards are delivered directly to the application (not to the Inbox). Therefore, sending business cards to a device requires that the BCExchanger MIDlet is running on it.

With the help of this tutorial, you can build the example from scratch and familiarize yourself with some common features related to the OBEX API. Alternatively, you can download the example files and run them immediately with your S60 MIDP SDK (3rd Edition or later). For more information on importing, building, and running Java projects, see Getting Started with Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™.

Note: All files in this example application contain the Nokia copyright statement.

Note: The example is also provided on Forum Nokia at http://www.forum.nokia.com/info/sw.nokia.com/id/b37cb6fa-fa2b-429c-9d1e-3b9728a23ac8/MIDP_Bluetooth_OBEX_Example_Business_Card_Exchange_v1_0_en.zip.html. Instead of creating the example from scratch as shown in this document, you can download the example files and run them immediately with your SDK.

6.1 Implementation of BCExchanger MIDlet

The BCExchanger architecture can be divided into several parts as shown in the figure below. The MIDlet is the central architecture element. It represents the application execution and owns and manages other architecture components.

- The ui module is responsible for the user interface of the application. In practice, it contains classes implementing various MIDlet screens and transitions between screens.
- The comm module is responsible for communication, that is, all operations related to Bluetooth OBEX.
- The Address Book module provides access to the phone book of the device.
- The Storage module is responsible for saving information about the card chosen as the own card.

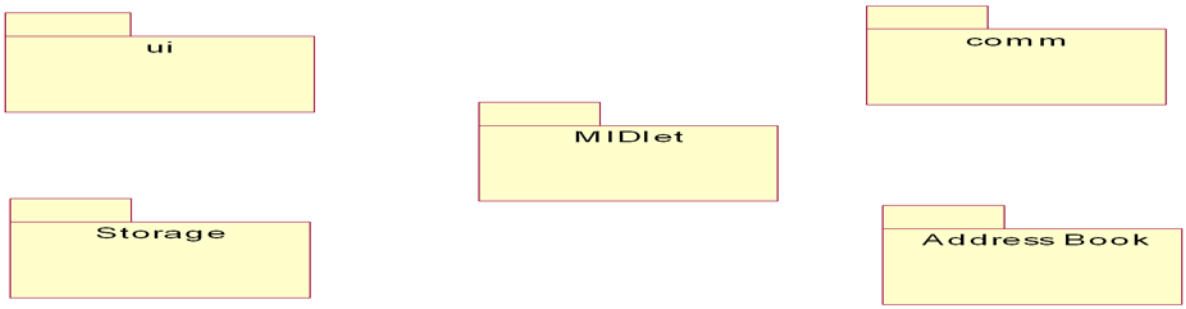

Figure 9: Application architecture

In practice, the Storage module and the Address Book module contain only one class each. The class in the Storage module uses MIDP 2.0 Record Management System (RMS) and the class in the Address Book module uses the PIM API (JSR-75). Both classes are quite straightforward in the implementation.

The modules ui and comm are more sophisticated. The comm module is responsible for various Bluetooth and OBEX operations. The ui module has to represent the sophisticated UI of the application. Each module contains the classes, which implement a state machine. The design pattern State (see Gamma et. all, "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley, 1995) has been used to implement both ui and comm modules.

The figure below illustrates the state machine diagram of the comm module. When the application initiates the exchange of cards, the normal state change flow is IdleState - InquiryState - ServiceDiscoveryState - SendState - ReceiveState - IdleState. However, each operation can fail or be canceled, which leads to the cancellation of the whole exchange process and returning to the IdleState.

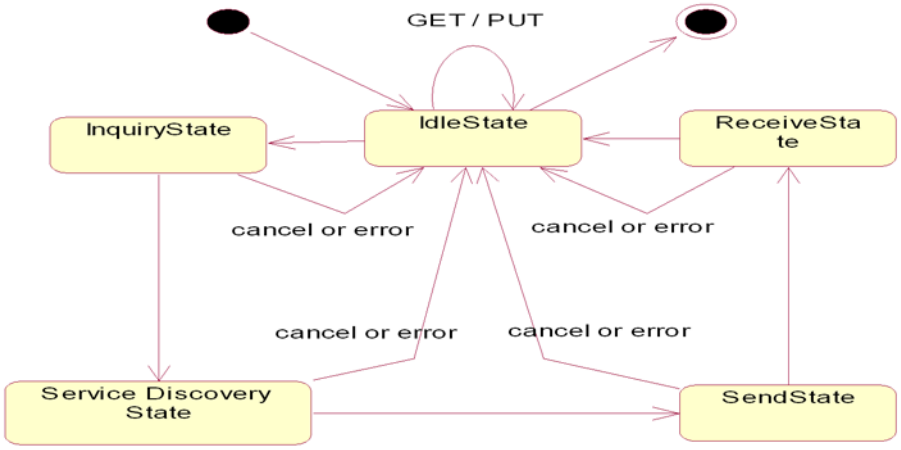

Figure 10: Comm module state diagram

Accepting connection is implemented as serving OBEX GET and PUT requests and therefore it is done without a change of the state. However, note that the state machine has to be in the IdleState to serve the OBEX requests. In other words, the application is always ready to accept the card exchange from a remote device if it is not performing active exchange itself.

The ui module has a design similar to the one shown in the figure below. Each state of the UI state machine represents a separate screen. When starting the application, the initial state is the AddressBookScreen (if it is the first application start) or the MainScreen (if the application has been started before). While the application is in the MainScreen state, it can display the AddressBookScreen if the user needs to change his or her own card, and it then transfers back to the MainScreen after successful selection of the card. Note that the application can exit only if it displays the MainScreen or the AddressBookScreen.

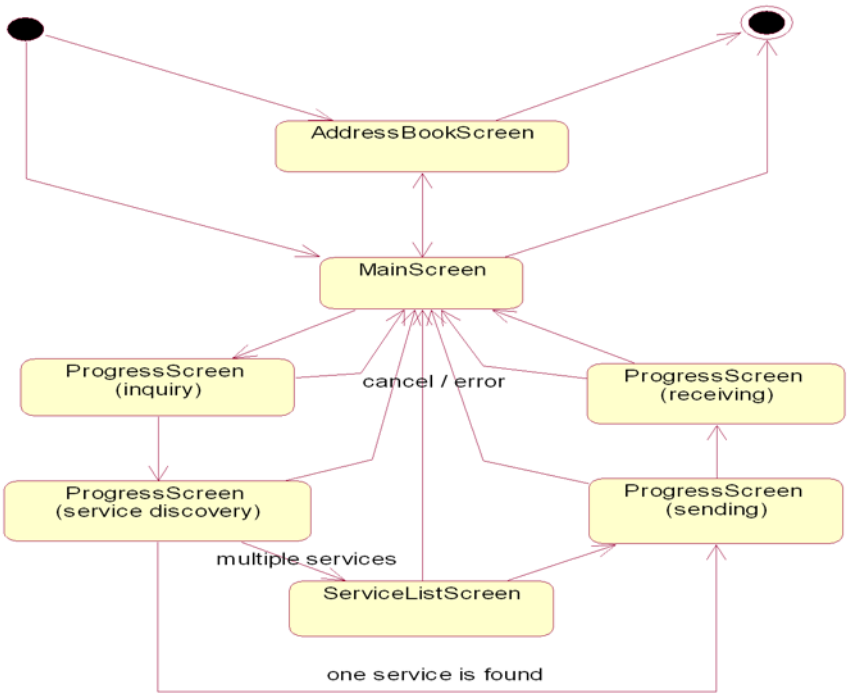

Figure 11: UI module state diagram

When the user initiates the exchange of the cards from the MainScreen, the state machine performs a change of various ProgressScreens as shown in the figure above. Each ProgressScreen is transferred to the next ProgressScreen upon the successful completion, or to the MainScreen in case of errors or cancellations. The only exception is the ProgressScreen (service search) - if more than one service is found, the UI is changed to ServiceListScreen. It allows the user to select one single service.

In practice, one more screen is used in the application - AlertMessageScreen. However, it is used to display the results of an operation (for example, Business card is sent, Inquiry is canceled, etc) and is shown for a limited period of time (using MIDP 2.0 `Alert` class). Therefore is not considered as a state of the UI machine.

Examples of some screens are shown in the figure below.

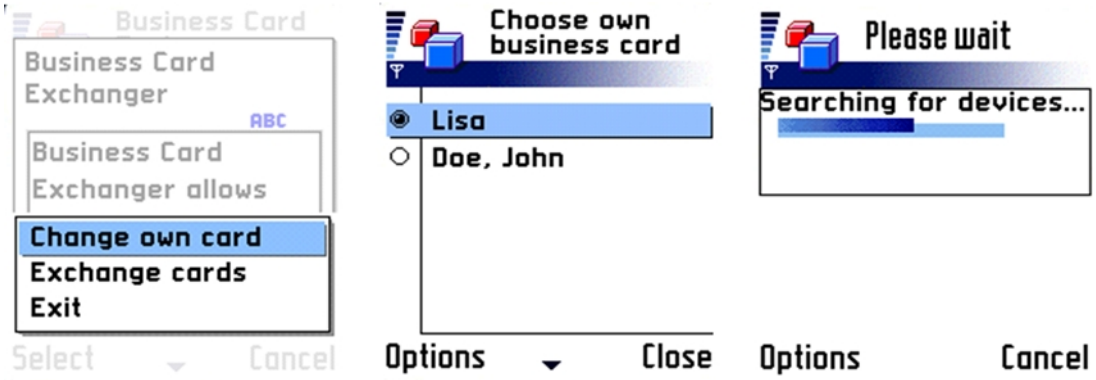

Figure 12: UI screen examples taken from a S60 device

6.1.1 Scenario

A high-level scenario of the BCExchanger application is illustrated in the figure below. Right after the application is started, the user selects his or her own card and proceeds to the main menu. If the card was already selected, the application starts directly from the main menu. While the user is in the main menu, it is possible to select another card, initiate card exchange, accept card exchange, or exit the application.

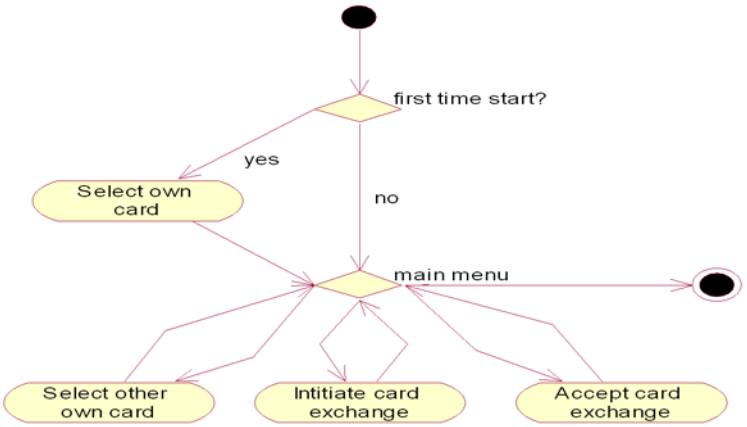

Figure 13: High-level scenario of the BCExchanger application.

The figure below illustrates the "Initiate card exchange" subscenario. When a user initiates the card exchange from the menu, the application starts a series of Bluetooth procedures: inquiry, service discovery, sending own card, and receiving and saving the remote business card. Also if several Bluetooth services declared by the BCExchanger MIDlet are discovered, the application displays the list of found services and asks the user to select one. Each procedure can be canceled or it can end with an error. Both cases should end the execution of the subscenario and return the application to the main menu.

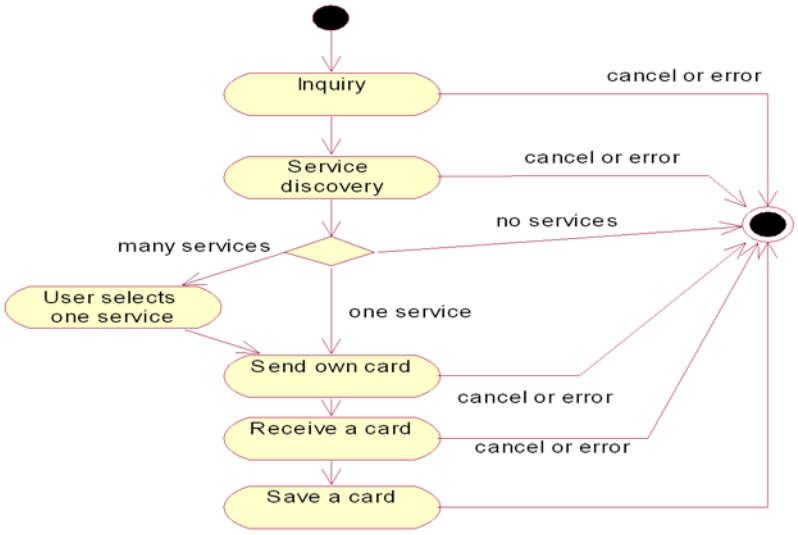

Figure 14: Initiate card exchange subscenario

The figure below shows the Accept card exchange subscenario. When the application acts as an accepting peer, it simply receives the card, saves it, and sends the user's own card. In addition, each activity can be canceled or it can fail. The party that starts the exchange operation receives a notification when the exchange operation proceeds successfully or fails.

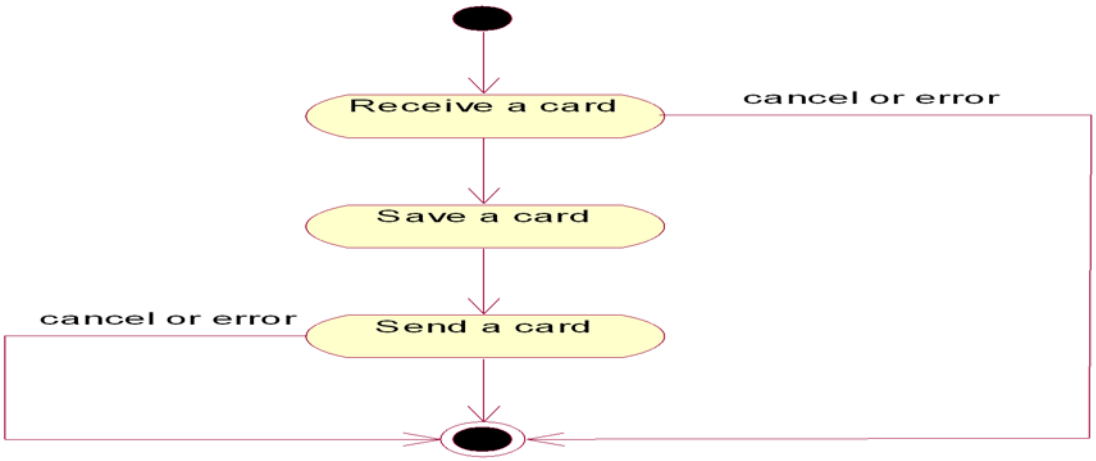

Figure 15: Accept card exchange subscenario

These high-level scenarios are reflected in the design of the application.

6.2 Prerequisites

The example MIDlet makes use of the standard MIDP 2.0 application framework. You should also have S60 MIDP SDK, 3rd Edition or later. Familiarity with MIDP 2.0 is highly recommended before attempting to understand this example.

> **Note:** For further information about setting up an appropriate environment for the emulators, see the S60 MIDP SDK documentation (comes with the SDK, downloadable from the Forum Nokia Web site at http://www.forum.nokia.com/main.html), 3rd Edition or later.

6.3 Creating the project environment

This MIDlet application has been developed using the following tools:

- Eclipse 3.1
- Nokia Prototype SDK 3.0 for Java™ Platform, Micro Edition and Nokia Prototype SDK 4.0 for Java™ Platform, Micro Edition
- Nokia Carbide.j 1.0
- Nokia Connectivity Framework (NCF) 1.2

In order to create the working environment, the applications should be installed in the order they are listed above. Installation and integration instructions are delivered with each product.

Eclipse is used as the main development environment and the Nokia Prototype SDK is used for compilation and execution of the example in the emulator. Nokia Carbide.j provides integration of the Nokia Prototype SDK with Eclipse IDE, which allows executing the application in the emulator directly from the Eclipse IDE. Nokia Connectivity Framework provides an emulation of phone connectivity. It allows emulating Bluetooth connections between several emulators running on the same machine.

> **Note:** Note: It is possible to use the S60 2nd Edition SDK for Symbian OS, Supporting Feature Pack 3, for MIDP as well. However, in this case it will not be possible to use NCF for Bluetooth communication emulation.

Nokia Carbide.j 1.0 contains NCF Lite. It means that, strictly speaking, there is no need to separately install NCF 1.2. However, NCF 1.2, which is available as a separate application, contains a Full version and allows tracing Bluetooth packets sent between applications.

Tip: It is possible to use Bluetooth hardware with NCF 1.2. Please refer to the NCF documentation for more details.

After applications are properly installed, integrated, and configured, the BCExchanger example can be loaded into the Eclipse environment. Start several emulator instances by using the menu command **Run > Run**.

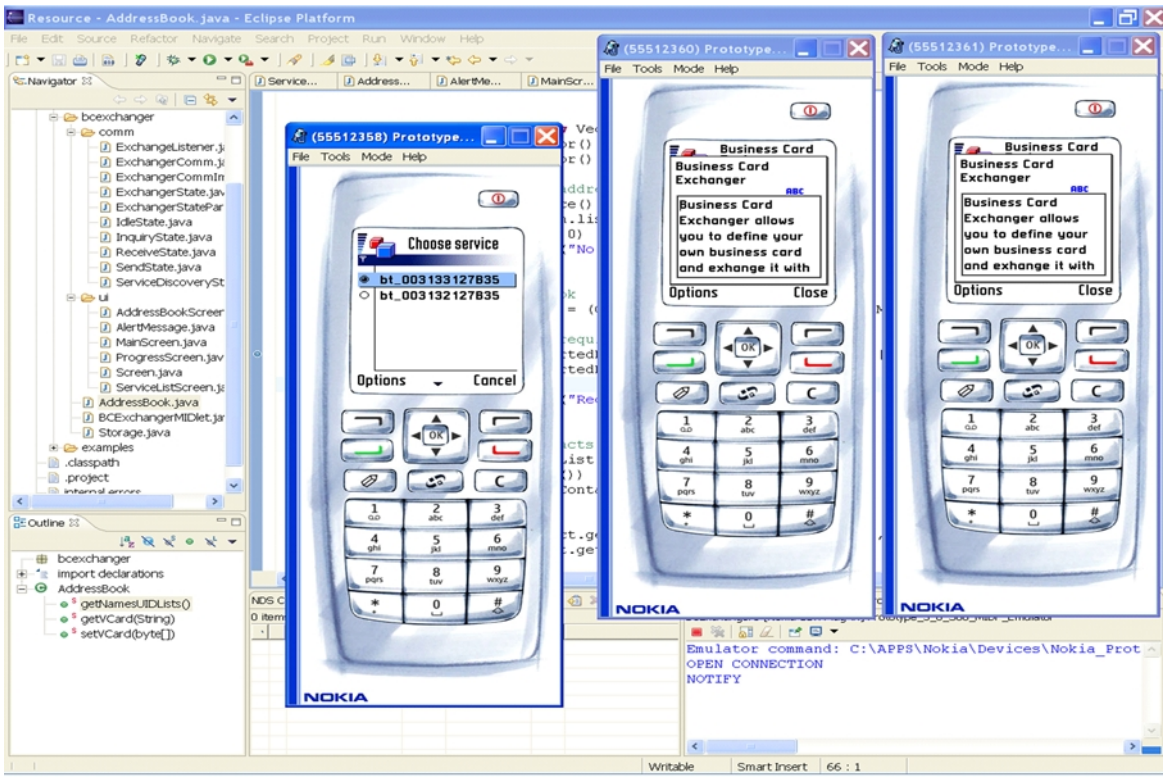

Figure 16: Eclipse development environment and emulators

JAR files can be created and deployed on the device with Carbide.j.

Tip: On some devices (such as the Nokia N90) you need to change the suite settings to allow the application to use PIM data. Otherwise, you will receive an "Error during receiving" message when exchanging business cards. You can change the "Edit user data" value to "Ask every time" from **Tools > Manager > MIDlet suite (BCExchanger) > Options > Suite settings**.

6.4 Creating MIDlet, Storage, and AddressBook classes

This section describes the following classes: `BCExchangerMIDlet` on page 83, `Storage` on page 87, and `AddressBook` on page 88.

6.4.1 BCExchangerMIDlet

This is the main application class which extends a MIDlet class. It serves as a parent to UI states (package `bcexchanger.ui`) and listens to communication events by implementing the `ExchangeListener` interface.

To create the class:

1 Create the `BCExchangerMIDlet` class file.

2 Assign the class to the `bcexchanger` package and import the required classes.

```
package bcexchanger;

import java.util.Vector;

import javax.microedition.midlet.MIDlet;
```

```
import javax.microedition.midlet.MIDletStateChangeException;

import bcexchanger.comm.ExchangeListener;
import bcexchanger.comm.ExchangerComm;
import bcexchanger.comm.ExchangerCommImpl;
import bcexchanger.ui.AddressBookScreen;
import bcexchanger.ui.AlertMessage;
import bcexchanger.ui.MainScreen;
import bcexchanger.ui.ProgressScreen;
import bcexchanger.ui.Screen;
import bcexchanger.ui.ServiceListScreen
```

3 Set the class to extend `MIDlet` and to implement `ExchangeListener`. Activate and display the UI screen.

```
/**
 *
 * This is the main application class which extends MIDlet class. It
 * serves as a parent to UI states (package example.BCExchanger.ui)
 * and listens to communicational events by implementing
 * ExchangeListener interface
 *
 * @see example.BCExchanger.comm.ExchangeListener
 */
public class BCExchangerMIDlet extends MIDlet
    implements ExchangeListener {

  private boolean commIsInitialized = false;
  private ExchangerComm comm;
  private int selectedServiceIndex;
  private MainScreen scr = new MainScreen(this);

  /**
   * Constructor
   */
  public BCExchangerMIDlet() { }

  /**
   * Changes the current UI screen
   * <p>
   * This method is used to display and make active a UI screen
   *
   * @param screen -
   *          UI screen to display
   * @see example.BCExchanger.ui.Screen
   */
  public void changeScreen(Screen screen) {
    screen.makeActive();
  }

  /**
   * Called to force termination of MIDlet
   */
  public void quit() {
    if (comm != null) {
      comm.cancelWaiting();
    }

    try {
      destroyApp(true);
      notifyDestroyed();
    } catch (MIDletStateChangeException e) {
      // Ignore, we are closing anyways
    }
  }
```

4 Implement a method for saving the index of a chosen service.

Initialize the communication module.

```
  /**
   * Saves the index of the service of choice
   * <p>
```

```
     * This method is used in case several services are found during
     * service discovery In this case the ServiceListScreen is displayed
     * and on user selection this method is called to save the index of
     * the service of choice.
     *
     * @param index -
     *          index of the serivice chosen by the user
     */
    public void serviceSelected(int index) {
      selectedServiceIndex = index;
    }

    /**
     * Intitializes the communication module
     */
    public void initComm()
    {
      if (!commIsInitialized) { // avoid double initializing
        comm = new ExchangerCommImpl(this);
        commIsInitialized = true;
      }
    }

    /**
     * Getter method for Exchanger - class representing communication
     * module
     *
     * @return a class implementing ExchangerComm interface,
     *         representing communication module
     */
    public ExchangerComm getExchanger() {
      return comm;
    }

    protected void destroyApp(boolean arg0)
        throws MIDletStateChangeException {}

    protected void startApp() throws MIDletStateChangeException {
      // ensure that we have own business card selected
      String uid = null;
      try {
        uid = Storage.getBCUID();
      } catch (Exception e) {
        changeScreen(new AlertMessage(this,
            "Cannot use the persistent storage. Application will exit"));
        return;
      }

      if (uid == null) {
        // if the MIDlet has not been started before
        // and own business card has not been chosen
        changeScreen(new AddressBookScreen(this));
      } else {
        changeScreen(scr);
      }
    }

    protected void pauseApp() {}

    public void onInquiryComplete(int code) {
      if (code == ExchangerComm.DONE) {
        String text = "Searching for Business Card Exchange service...";
        changeScreen(new ProgressScreen(this, text));
      } else if (code == ExchangerComm.ERROR) {
        changeScreen(new AlertMessage(this, "Errors during inquiry",
           scr));
      } else if (code == ExchangerComm.CANCELED) {
        changeScreen(new AlertMessage(this, "Inquiry is canceled",
            scr));
      } else {
        // unknwon error
        throw new RuntimeException("Internal error #26");
      }
```

```
  }

  public void onServiceDiscoveryComplete(int code)
  {
    if (code == ExchangerComm.DONE)
    {
      String text = "Sending own business card...";
      changeScreen(new ProgressScreen(this, text));
    }
    else if (code == ExchangerComm.NO_RECORDS) {
      changeScreen(new AlertMessage(
          this,
          "Cannot find Business card exchange service on the devices",
          scr));
    } else if (code == ExchangerComm.ERROR) {
      changeScreen(new AlertMessage(this,
          "Errors during service discovery", scr));
    } else if (code == ExchangerComm.CANCELED) {
      changeScreen(new AlertMessage(this,
          "Service discovery is canceled",scr));
    } else {
      // unknown error
      throw new RuntimeException("Internal error #27");
    }
  }

  public synchronized int resolveMultipleServices(Vector friendlyNames) throws
InterruptedException {
    changeScreen(new ServiceListScreen(this, friendlyNames));

    wait(); // wait until the choise is made

    return selectedServiceIndex;

  }

  public byte[] getOwnBC() throws Exception {
    byte[] vCard = null;
    String uid;
    try {
      uid = Storage.getBCUID();
      vCard = AddressBook.getVCard(uid);
    } catch (Exception e)
    {
      throw new Exception("getOwnBC did not succeed");
    }
    return vCard;
  }

  public void onSendComplete(int code) {

    if (code == ExchangerComm.DONE) {
      String text = "Send was successful!";
      changeScreen(new AlertMessage(this, text,
               scr));
    } else if (code == ExchangerComm.CANCELED) {
      changeScreen(new AlertMessage(this, "Send is canceled ",
          scr));
    } else { // error

      changeScreen(new AlertMessage(this, "Errors during sending ",
             scr ));
    }
  }


  public void onReceiveComplete(int code) {
    if (code == ExchangerComm.DONE) {
      changeScreen(new AlertMessage(this, "Receiving was successful!",
          scr));
    } else if (code == ExchangerComm.CANCELED) {
      changeScreen(new AlertMessage(this, "Receiving is canceled ",
```

```
            scr));
      } else { // error
        changeScreen(new AlertMessage(this, "Errors during receiving ",
            scr));
      }
    }

    public void onReceiveBC(byte[] vCard) throws Exception {
     AddressBook.setVCard(vCard);
    }
```

5 Implement a helper method for unblocking any waiting code.

```
    /**
     * This is a helper method which unblocks waiting code in
     * resolveMultipleServices()
     */
    public synchronized void unblock() {
      notify(); // let resolveMultipleServices complete
    }

    public void onGetComplete(int code) {

    }

    public void onPutComplete(int code) {

    }

    /**
     * Called in case of problems with Bluetooth OBEX server accepting connection
     *
     */
    public void onServerError() {
      changeScreen(new AlertMessage(this,
      "Fatal Bluetooth error. Application will exit"));
      return;
    }

  }
```

6.4.2 Storage

This class contains methods for abstracting access to Record Management System (RMS). It is needed for saving the ID of the business card that the user has selected as his or her own card.

To create the class:

1 Create the `Storage` class file.

2 Assign the class to the `bcexchanger` package and import the required classes.

```
package bcexchanger;

import javax.microedition.rms.RecordEnumeration;
import javax.microedition.rms.RecordStore;
```

3 Obtain the UID of the contact entry from the address book.

```
/**
 *
 * This class abstracts an RMS system for storing the uid of the
 * contact entry from the address book, selected by user as own entry.
 *
 * @version 1.0 29.09.2005
 */
public class Storage {

  /**
   * Gets the stored uid
   *
   * @return string containing uid of the the contact entry from the
   *         address book, selected by user as own entry or null if no
```

```
     *         contact is previously selected
     * @exception Exception -
     *              in case of errors occur
     */
    static public String getBCUID() throws Exception {
      String uid = null;
      RecordStore db = RecordStore.openRecordStore("uid", true);

      RecordEnumeration e = db.enumerateRecords(null, null, false);

      if (!e.hasNextElement()) {
        uid = null;
      } else {
        byte[] record = e.nextRecord();
        if (record == null) {
          uid = null;
        } else {
          uid = new String(record);
        }
      }
      db.closeRecordStore();

      return uid;
    }
```

4 Store the UID.

```
    /**
     * Stores the uid
     *
     * @param uid -
     *          string containing uid of the the contact entry from the
     *          address book, selected by user as own entry
     * @exception Exception -
     *              in case of errors occur
     */
    static public void setBCUID(String uid) throws Exception {
      RecordStore db = RecordStore.openRecordStore("uid", true);

      // delete all records
      if (db.getNumRecords() != 0) {
        db.closeRecordStore();
        RecordStore.deleteRecordStore("uid");
        db = RecordStore.openRecordStore("uid", true);
      }

      byte[] record = uid.getBytes();
      db.addRecord(record, 0, record.length);

      db.closeRecordStore();
    }

  }
```

6.4.3 AddressBook

This class provides an abstraction of the phone book to application. It implements access to the phone book of the device and provides information in the form the application needs.

The class makes use of the PIM API (JSR-75) at http://www.jcp.org/en/jsr/detail?id=75 . For more information on using this API, see PIM API Implementation Notes and MIDP: PIM API Developer's Guide section of the Java Developer's Library at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm.

To create the class:

1 Create the `AddressBook` class file.

2 Assign the class to the `bcexchanger` package and import the required classes.

```
package bcexchanger;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.util.Enumeration;
import java.util.Vector;

import javax.microedition.pim.Contact;
import javax.microedition.pim.ContactList;
import javax.microedition.pim.PIM;
import javax.microedition.pim.PIMItem;
```

3 Obtain all names from the address book.

```
/**
 *
 * This class provides an abstraction of the address book to
 * application. It implements the access to the phone address book and
 * provides information in the form the application needs
 */
public class AddressBook {

  /**
   * This method gets all names from the address book and
   * corresponding uids.
   * <p>
   * This method retrieves full names from the address book and
   * corresponding uinque identifiers (uids)
   *
   * @return two vectors: first is vector of names, second is the
   *         vector of corresponding UIDs
   * @exception Exception -
   *              in case any errors occur while working with address
   *              book
   */
  static public Vector[] getNamesUIDLists() throws Exception {
      Vector[] namesUIDs = new Vector[2];
    namesUIDs[0] = new Vector();
    namesUIDs[1] = new Vector();

    // get the name of the address book
    PIM pim = PIM.getInstance();
    String listNames[] = pim.listPIMLists(PIM.CONTACT_LIST);
    if (listNames.length == 0) {
      throw new RuntimeException("No available Contact lists");
    }

    // opent the address book
    ContactList contactList = (ContactList) pim.openPIMList(
        PIM.CONTACT_LIST, PIM.READ_ONLY, listNames[0]);

    // veryfy the that the required list are supported
    if (!contactList.isSupportedField(Contact.UID)) {
      throw new RuntimeException(
          "Required field (UID) is not supported by the address book");
    }

    // list of preferd order of the name fields
    Vector fieldOrder = new Vector();
    fieldOrder.addElement(new Integer(Contact.FORMATTED_NAME)); // For Series40
    fieldOrder.addElement(new Integer(Contact.NAME_FAMILY));
    fieldOrder.addElement(new Integer(Contact.NAME_GIVEN));
    fieldOrder.addElement(new Integer(Contact.NAME)); // fix

    Vector supportedFieldOrder = new Vector();
    // create the supported list
    for(int i=0; i < fieldOrder.size(); i++) {

        Integer field = (Integer) fieldOrder.elementAt(i);
        if(contactList.isSupportedField(field.intValue())) {
            supportedFieldOrder.addElement(field);
        }
    }
```

```
  // in case none of the names is supported
  if(supportedFieldOrder.size() == 0) {
      throw new RuntimeException(
      "Required name field is not supported by the address book");
  }

  // get the list of contacts and uids
  Enumeration e = contactList.items();
  while (e.hasMoreElements()) {
      Contact contact = (Contact) e.nextElement();
    try {

        // try to obtain any supported name
        String name = null;
        Enumeration e2 = supportedFieldOrder.elements();
        while(e2.hasMoreElements()) {
          try {
              int field = ((Integer) e2.nextElement()).intValue();

              if (field == Contact.NAME)
              {
                  String[] nameArray = contact.getStringArray(field,0);
                  if (nameArray != null) name = nameArray[0];
              }
          else
          {
                  name = contact.getString(field, 0);
          }

              if(name != null && name != "") {
                  break;
              } else {
                  name = null;
              }
          } catch (Exception exception) {
              name = null;
          }
        }

        // this contact does not have a suitable name, continue with the next
        if(name == null) {
            continue;
        }
       String uid = contact.getString(Contact.UID, 0);

       namesUIDs[0].addElement(name);
       namesUIDs[1].addElement(uid);
    } catch (IndexOutOfBoundsException ex) {
      // Ignore, this is a malformed contact and we'll omit it
    }
  }

  return namesUIDs;
}
```

4 Search for vCards in an address book entry.

```
/**
 * Get vCard of the address book entry with the particular uid
 * <p>
 * Longer description
 *
 * @param uid -
 *          unique identifier of the requested address book entry
 * @return byte array which contains a vCard
 * @exception Exception -
 *              in case any error occur
 */
static public byte[] getVCard(String uid) throws Exception {
  ByteArrayOutputStream os = new ByteArrayOutputStream();
```

```
        // create a search pattern (based on uid)
        ContactList contactList = (ContactList) PIM.getInstance()
            .openPIMList(PIM.CONTACT_LIST, PIM.READ_ONLY);
        Contact searchPattern = contactList.createContact();
        searchPattern.addString(Contact.UID, PIMItem.ATTR_NONE, uid);

        Enumeration e = contactList.items(searchPattern);

        // get the first found contact
        if (e.hasMoreElements())
        {
          Contact contact = (Contact) e.nextElement();
          PIM.getInstance().toSerialFormat(contact, os, "UTF-8",
              "VCARD/2.1");
        } else
        {
          throw new RuntimeException("No contacts found");
        }

        return os.toByteArray();
      }

      /**
       * Saves a vCard to the address book
       * <p>
       * Creates a new entry based on the infromation from the vCard in
       * the phones address book
       *
       * @param vCard -
       *          byte array which contains a vCard
       * @exception Exception -
       *              in case any errors occur
       */
      static public void setVCard(byte[] vCard) throws Exception {
        ByteArrayInputStream in = new ByteArrayInputStream(vCard);

        // get address book entry from the input stream
        PIMItem[] items = PIM.getInstance().fromSerialFormat(in, "UTF-8");

        // takes the first one discards the others
        ContactList contactList = (ContactList) PIM.getInstance()
            .openPIMList(PIM.CONTACT_LIST, /* PIM.WRITE_ONLY */
            PIM.READ_WRITE);
        Contact newContact = contactList
            .importContact((Contact) items[0]);

        newContact.commit();

      }

    }
```

6.5 Creating Comm package

This package contains classes which are related to Bluetooth OBEX communication. The classes from the `comm` package include classes realizing the communication state machine and other auxiliary classes and interfaces. The package `comm` contains the interfaces `ExchangeListener` on page 91, `ExchangerComm` on page 93, `ExchangeStateParent` on page 100, classes `ExchangerCommImpl` on page 95, `IdleState` on page 101, `InquiryState` on page 102, `ReceiveState` on page 111, `SendState` on page 114, `ServiceDiscoveryState` on page 105, and the abstract class `ExchangerState` on page 98.

6.5.1 ExchangeListener

The `ExchangeListener` is a callback interface. By means of the interface methods, the OBEX communication module signals to a listener about various networking events. This interface is implemented by the `BCExchangerMIDlet`

To create the class:

1 Create the `ExchangeListener` class file.

2 Assign the class to the `Comm` package and import the required classes.

```
package bcexchanger.comm;

import java.util.Vector;
```

3 Implement the `ExchangeListener` class functionality.

```
/**
 *
 * ExchangeListner is an callback interface. By means of the interface
 * function OBEX communication module signal to a listener about
 * various networking events.
 *
 * This interface is implemented by the BCExchangerMIDlet and OBEX
 * communication classes located in the package bcexchanger.comm use
 * it for signaling
 *
 * @version 1.0 27.09.2005
 */
public interface ExchangeListener {

  /**
   * Called when the OBEX communication module completes the inquiry
   *
   * @param code -
   *          inquiry completion code
   */
  public void onInquiryComplete(int code);

  /**
   * Called when the OBEX communication module completes service
   * discoveries
   * <p>
   * This method is called when service discovery on all devices is
   * complete
   *
   * @param
   * @param code -
   *          service discovery completion code
   */
  public void onServiceDiscoveryComplete(int code);

  /**
   * This method is called when OBEX communication module needs to
   * resolve services
   * <p>
   * If several services are found, the OBEX communcation module asks
   * the MIDlet to resolve the services and choose the only one to
   * connect to
   *
   * @param friendlyNames -
   *          list of friendly names of devices which has the service
   * @return index of the chosen device/service from the vector
   * @throws InterruptedException
   */
  public int resolveMultipleServices(Vector friendlyNames) throws
InterruptedException;

  /**
   * This method returns own business card serialized to vCard/2.1
   * format
   * <p>
   * When OBEX module needs to send the own business card to a remote
   * device it uses this method to request own business card from the
   * MIDlet
   *
   * @return byte array containing own business card serialized to
   *         vCard/2.1 format null if there were errors or cancelation
```

```
     * @throws Exception
     */
    public byte[] getOwnBC() throws Exception;

    /**
     * Called when business card send (Client OBEX PUT) is finished
     *
     * @param code -
     *          completion code
     */
    public void onSendComplete(int code);

    /**
     * Called when business card receive (Client OBEX GET) is finished
     *
     * @param code -
     *          completion code
     */
    public void onReceiveComplete(int code);

    /**
     * Called when Server OBEX GET operation is finished
     *
     * @param code -
     *          completion code
     */
    public void onGetComplete(int code);

    /**
     * Called when Server OBEX PUT operation is finished
     *
     * @param code -
     *          completion code
     */
    public void onPutComplete(int code);

    /**
     * Called when a business card from a remote device is received
     * <p>
     * This method allows MIDlet to save received business card.
     *
     * @param vCard -
     *          byte array containing receive business card serialized
     *          to vCard/2.1 format
     * @throws Exception -
     *           if saving of the business card fails
     */
    public void onReceiveBC(byte[] vCard) throws Exception;

    /**
     * Called in case of problems with Bluetooth OBEX server accepting connection
     *
     */
    public void onServerError();


}
```

6.5.2 ExchangerComm

This interface contains methods representing the functionality of the OBEX communication module to other classes of the application. The OBEX communication module implemented by the classes from the `bcexchanger.comm` package realizes the exchange of the business card over OBEX API (JSR-82). The control of the module is abstracted in this interface.

The `BCExchangerMIDlet` calls the methods of this interface implemented by the `ExchangerCommImpl` class to control OBEX communication.

To create the class:

1 Create the `ExchangerComm` class file.

2 Assign the class to the `Comm` package.

```
package bcexchanger.comm;
```

3 Implement the interface.

```
/**
 *
 * This is an interface to OBEX communication module. OBEX
 * communication module implemented by the classes from
 * bcexchanger.comm package realize exhange of the business card over
 * JSR82 OBEX. The controls of the module is abstracted in this
 * interface.
 *
 * BCExchanger is calling the methods of this interface implemented by
 * ExchangerCommImpl class to control OBEX communication.
 *
 * @version 1.0 27.09.2005
 *
 */
public interface ExchangerComm {

  // Constant field declaration
  static final int DONE = 0;
  static final int CANCELED = 1;
  static final int ERROR = 2;
  static final int NO_RECORDS = 3;
```

4 Initiate the process of sending a business card. Set methods that start and stop listening for incoming connections.

```
  /**
   * Intitiate whole procedure of sending business card to a remote
   * device
   * <p>
   * The method start the procedure of inquiry, service discovery,
   * tranfering own card and receiving the card from the remote
   * device. This method is asynchrounous and it returns immediately
   * after operation is started.
   *
   * @exception Exception -
   *              if any immediate errors occur while the process is
   *              started. In practice it is thrown when inquiry
   *              cannot start or if this method is called while
   *              sending process is in progress
   */
  public void startSending() throws Exception;

  /**
   * Cancels the process of sending
   * <p>
   * The method cancels the current operation of the sending: inquiry,
   * service discovery, OBEX send or receive and returns to the idle
   * state.
   *
   */
  public void cancelSending();

  /**
   * Starts to listen for incomming connections
   * <p>
   * The method start listening for incomming connections. This method
   * is asynchronous and it returns immediately after the connection
   * listening is started.
   *
   */
  public void startWaiting();

  /**
   * This method stops listening for incomming connection
   * <p>
   * The method stops listening for incomming connection if
```

```
     * startWaiting() was called before. Otherwise method does nothing.
     *
     */
    public void cancelWaiting();

}
```

6.5.3 ExchangerCommImpl

This class is the central class in the OBEX communication module (package `bcexchanger.comm`). The class implements the interface `ExchangerComm` and realizes the methods for controlling the process of sending and receiving business cards. It is also a parent for states of the communication state machine. It keeps the current state of the state machine and implements the interface `ExchangerStateParent`. Using this interface, state classes can access the required functionality of the parent.

This class waits for an incoming Bluetooth connection in a separate thread and therefore implements the `Runnable` interface. The class also works as an OBEX server. In order to serve OBEX requests, it extends the `ServerRequestHandler` class and overrides some of its methods.

To create the class:

1 Create the `ExchangerCommImpl` class file.

2 Assign the class to the `Comm` package and import the required classes.

```
package bcexchanger.comm;

import java.io.IOException;

import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.LocalDevice;
import javax.microedition.io.Connection;
import javax.microedition.io.Connector;
import javax.obex.HeaderSet;
import javax.obex.Operation;
import javax.obex.ServerRequestHandler;
import javax.obex.SessionNotifier;
```

3 Set the class to extend the `ServerRequestHandler` class of the `javax.bluetooth` package in order to serve OBEX requests and override some of its methods.

In order to create a server connection, set instance variables for this purpose. This is done by identifying the `btgoep` protocol, the server URL, and the service UUID. This will be used later in this class when the `Connector.open()` method is called.

```
/**
 *
 * This class is the central class in the OBEX communication module
 * (package bcexchanger.comm).
 *
 * Class implements interface ExchangerComm and realizes the methods
 * for controling process of sending and receiving business cards. It
 * also is a parent for states of the BCExchanger communication state
 * machine. It keeps the current state of the state machine and
 * implements interface ExchangerStateParent. Using this interface
 * state classes access to the required functionality of the parent.
 *
 * This class waits for Bluetooth incomming connection in the separate
 * thread and therefore it implments Runnable interface.
 *
 * This class also works as a OBEX server. In order to serve OBEX
 * requests it extends the ServerRequestHandler class and overides
 * some of its methods.
 *
 * @version 1.0 29.09.2005
 * @see bcexchanger.comm.ExchangerStateParent
 *      bcexchanger.comm.ExchangerComm javax.obex.ServerRequestHandler
 *      Design patterns: State
 *
```

```
 */
public class ExchangerCommImpl extends ServerRequestHandler
    implements ExchangerStateParent, ExchangerComm, Runnable {

  // Instance variables
  final private String uuid = "ed495afe28ed11da94d900e08161165f";
  final private String serverURL = "btgoep://localhost:" + uuid;

  private boolean cancelWaitingInvoked = false; // becomes true if
                                                // cancelWaiting() is
                                                // called

  private Thread waitingThread;

  private ExchangeListener listener;
  private ExchangerState currentState;

  private SessionNotifier notifier = null;
  private Connection con = null;
```

4 Set a listener for the module event communication.

```
  /**
   * Constructor
   * <p>
   * description
   *
   * @param _listener -
   *          listener of the communication module events
   * @exception
   * @see
   */
  public ExchangerCommImpl(ExchangeListener _listener)
  {
    listener = _listener;

    startWaiting();
    setState(new IdleState(this));
  }

  public void setState(ExchangerState state) {

    currentState = state;
  }

  public ExchangeListener getListener() {
    return listener;
  }

  public ExchangerState getState() {
    return currentState;
  }

  public void startSending(int oper) throws Exception {
    getState().startSending(oper);
  }

  public void startSending() throws Exception {
        getState().startSending(0);
      }


  public void cancelSending()
  {
    getState().cancelSending();
  }

  public void startWaiting() {
    cancelWaitingInvoked = false;

    waitingThread = new Thread(this);
    waitingThread.start();
  }
```

```
public void cancelWaiting()
{
  cancelWaitingInvoked = true;

  try
  {
    notifier.close(); // indicate to acceptAndOpen that it is
                      // canceled
  } catch (IOException e) {
    // Ignore, we're closing anyways
  }
}
```

5 Use the run method to initialize the stack. Make the device discoverable by using the `setDiscoverable()` method of the `LocalDevice` class in the `javax.bluetooth` package.

Call the `Connector.open()` method to return a `SessionNotifier` object. Use the method `acceptAndOpen()` of the `SessionNotifier` interface to start waiting for incoming transport layer connections.

After the transport layer connection is established, call the `acceptAndOpen()` method to return a `Connection` object, which represents the connection to a single client.

The server communicates with the remote peer by two means: the `Connection` object and the `ServerRequestHandler` callback methods that are called on incoming OBEX requests.

```
public synchronized void run() {

    // initialize stack and make the device discoverable
  try
  {
    LocalDevice local = LocalDevice.getLocalDevice();
    local.setDiscoverable(DiscoveryAgent.GIAC);
  } catch (Exception e)
  {
    // catching notifier exception
    listener.onServerError();
    return;
  }
  try
  {
      notifier = (SessionNotifier) Connector.open(serverURL);
  }
  catch (IOException e2)
  {

  }
  // the cycle stops only if cancelWaiting() was called
  while (!cancelWaitingInvoked)
  {
    try
    {

      try
      {
        con = notifier.acceptAndOpen(this);
        wait(); // wait until the remote peer disconnects
        try
        {
          con.close();
        }
        catch (IOException e0)
        { }
      }
      catch (Exception e1)
      {
          listener.onServerError();
          return;
      }
```

```
        }
        catch (Exception e)
        {
          listener.onServerError();
          return;
        }

      }

    }
```

6 Implement the OBEX server related methods.

The `ServerRequestHandler` class methods `onGet()` and `onPut()` are called when a GET or PUT method is received. These methods receive an `Operation` object as a parameter. The application uses the `Operation` object to read and write data. After the request is handled, the method must return a response code. To see all the available response codes, see the `ResponseCodes` class.

```
  /*
   * This method is related to OBEX server functionality. This method
   * is delegating this execution to the current state
   *
   * @see javax.obex.ServerRequestHandler#onGet()
   */
  public int onGet(Operation op) {
    return getState().onGet(op);
  }

  /*
   * This method is related to OBEX server functionality. This method
   * is delegating this execution to the current state
   *
   * @see javax.obex.ServerRequestHandler#onPut()
   */
  public int onPut(Operation op) {
    return getState().onPut(op);

  }

  /*
   * This method is related to OBEX server functionality. This method
   * handles OBEX DISCONNECT command from the remote device.
   *
   * @see javax.obex.ServerRequestHandler#onDisconnect()
   */
  public synchronized void onDisconnect(HeaderSet request,
      HeaderSet reply) {
    super.onDisconnect(request, reply);
    notify();// stops waiting in run()

  }

  public String getUUID() {
    return uuid;
  }

}
```

6.5.4 ExchangerState

This is the abstract base class for all the states of the communication state machine. Each state implements only those methods which can be executed in that particular case.

Classes `IdleState`, `InquiryState`, `ServiceDiscoveryState`, `ReceiveState`, and `SendState` inherit from this class.

To create the class:

1 Create the `ExchangerState` class file.

2 Assign the class to the Comm package and import the required class.

```
package bcexchanger.comm;

import javax.obex.Operation;
```

3 Implement the class. Implement the way the server handles the OBEX GET and PUT methods.

```
/**
 *
 * This is the abstract base class for the all states of the
 * communication state machine. Each state is implementing methods
 * which are can be executed in that particular case.
 *
 * Classes IdleState, InquiryState, ServiceDiscoveryState,
 * ReceiveState and SendState inherit from this class
 *
 * @see Design patterns: State
 *
 */
public abstract class ExchangerState {

  protected ExchangerStateParent parent;

  /**
   * Implements OBEX GET command
   * <p>
   * Implements server handling of the OBEX GET command
   *
   * @param op -
   *           OBEX operation class
   * @return - OBEX response code
   * @see javax.obex.ServerRequestHandler
   */
  abstract public int onGet(Operation op);

  /**
   * Implements OBEX PUT command
   * <p>
   * Implements server handling of the OBEX PUT command
   *
   * @param op -
   *           OBEX operation class
   * @return - OBEX response code
   * @see javax.obex.ServerRequestHandler
   */
  abstract public int onPut(Operation op);

  /**
   * Implements actions related to starting sending process
   * <p>
   * Implements reaction of the state on command to start sending
   * process
   *
   * @exception Exception -
   *               if any immediate error occur
   * @see example.BCExchanger.comm.ExchangerComm
   */
  abstract public void startSending(int oper) throws Exception;

  /**
   * Implements actions related to canceling sending process
   * <p>
   * Implements reaction of the state on command to cancel sending
   * process
   *
   * @see example.BCExchanger.comm.ExchangerComm
   */
  abstract public void cancelSending();

  /**
   * Constructor
   *
```

```
     * @param -
     *          _parent - the class which nests the current state of the
     *          communication machine
     */
    public ExchangerState(ExchangerStateParent _parent) {
      parent = _parent;
    }

  }
```

6.5.5 ExchangerStateParent

This interface contains the function used by the state to address the functionality of the parent class `ExchangerCommImpl`. It contains the methods which the states of the communication state machine use to get certain functionality from the parent.

This interface is implemented by the `ExchangerCommImpl` class.

To create the class:

1 Create the `ExchangerStateParent` class file.

2 Assign the class to the `Comm` package.

```
package bcexchanger.comm;
```

3 Implement the methods to be used by the communication state machine.

```
/**
 *
 * This interface contains the methods which the states of the
 * communication state machine use to get needed functionality from
 * the parent
 *
 * This interface is implemented by ExchangerCommImpl class.
 *
 * @version 1.0 29.09.2005
 *
 */
public interface ExchangerStateParent {

  /**
   * Current state setter
   * <p>
   * Sets the current state in the parent
   *
   * @param state -
   *          state of communication machine
   */
  public void setState(ExchangerState state);

  /**
   * Listener getter
   * <p>
   * Returns the communication event listener which is stored in the
   * parent
   *
   * @return communication event listener interface
   */
  public ExchangeListener getListener();

  /**
   * Communication state getter
   * <p>
   * Returns current state of the communication machine which is kept
   * in the parent
   *
   * @return current state of the communication machine
   */
  public ExchangerState getState();

  /**
   * Returns "Business Card Exchanger Service" UUID
```

```
     *
     * @return string containing UUID of the Bluetooth OBEX server
     *         connection
     */
    public String getUUID();

}
```

6.5.6 IdleState

This class implements the idle state of the communication state machine. In idle state, the machine waits for external events, such as incoming Bluetooth connections or a user command to start business card exchange. The `IdleState` class extends the `ExchangerState` abstract class.

To create the class:

1 Create the `IdleState` class file.

2 Assign the class to the `Comm` package and import the required classes.

```
package bcexchanger.comm;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.InputStream;
import java.io.OutputStream;
import javax.obex.Operation;
import javax.obex.ResponseCodes;
```

3 Implement the idle state. Set up a listener for external events.

The `ServerRequestHandler` class methods `onGet()` and `onPut()` are called when a GET or PUT method is received. These methods receive an `Operation` object as a parameter. The application uses the `Operation` object to read and write data. After the request is handled, the method must return a response code. All response codes are defined in the `ResponseCodes` class.

```
/**
 *
 * This class implements idle state of the communication state machine. In idle
 * state machine is waiting for external events (like incomming Bluetooth
 * connections or user command to start business card exchange)
 *
 * This class extends ExchangerState abstract class.
 *
 * @version 1.0 29.09.2005
 * @see bcexchanger.comm.ExchangerState Design Patterns: State
 *
 */
public class IdleState extends ExchangerState {
    /**
     * Constructor
     *
     * @param _parent -
     *            the class which is nesting the current state of the state
     *            machine
     */
    public IdleState(ExchangerStateParent _parent) {
        super(_parent);

    }

    public int onGet(Operation op) {

        try {

            OutputStream out = op.openOutputStream();
            byte[] vCard = parent.getListener().getOwnBC(); // getting the
            // own card
```

```
                int vlen = vCard.length;
                byte[] tmpBuf = new byte[vlen + 4];
                System.arraycopy(vCard, 0, tmpBuf, 4, vlen);
                tmpBuf[0] = (byte) ((vlen >>> 24) & 0xff);
                tmpBuf[1] = (byte) ((vlen >>> 16) & 0xff);
                tmpBuf[2] = (byte) ((vlen >>> 8) & 0xff);
                tmpBuf[3] = (byte) ((vlen >>> 0) & 0xff);

                out.write(tmpBuf); // sending data

                op.close();

                return ResponseCodes.OBEX_HTTP_OK;

            } catch (Exception e) {

                return ResponseCodes.OBEX_HTTP_INTERNAL_ERROR;
            }

        }

        public int onPut(Operation op) {

            try
            {
                InputStream in = op.openInputStream();
                byte[] fullResult = null;
                {
                    byte[] buf = new byte[256];
                    ByteArrayOutputStream bout = new ByteArrayOutputStream(2048);
                    for (int len = in.read(buf); len >= 0; len = in.read(buf))
                        bout.write(buf, 0, len);
                    fullResult = bout.toByteArray();
                }
                ByteArrayInputStream bin = new ByteArrayInputStream(fullResult);
                DataInputStream din = new DataInputStream(bin);
                int size = din.readInt();
                byte[] vCard = new byte[size];
                din.read(vCard);
                // card is received

                op.close();
                parent.getListener().onReceiveBC(vCard);

                return ResponseCodes.OBEX_HTTP_OK;

            }
            catch (Exception e) {

                return ResponseCodes.OBEX_HTTP_INTERNAL_ERROR;
            }

        }

        public void startSending(int oper) throws Exception {
            parent.setState(new InquiryState(parent, oper));

        }

        public void cancelSending() {
            // Internal error, but not fatal
        }

    }
```

6.5.7 InquiryState

This class represents the state of running inquiry. In this state an active device discovery is performed. No OBEX server commands are handled. This class extends the `ExchangerState` abstract class. This class also implements the `DiscoveryListener` interface to receive Bluetooth inquiry callbacks.

To create the class:

1 Create the `InquiryState` class file.

2 Assign the class to the `Comm` package and import the required classes.

```
package bcexchanger.comm;

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

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.obex.Operation;
import javax.obex.ResponseCodes;
```

3 Set the class to extend `ExchangerState` and to implement `DiscoveryListener` and implement the constructor.

Use the method `startInquiry()` to start an inquiry for devices. Devices that respond to the inquiry are returned to the application via the method `deviceDiscovered()` of the interface `DiscoveryListener`. Call the `cancelInquiry()` method to stop the inquiry.

```
/**
 *
 * This class implements inquiry state. In inquiry state an active
 * device discovery is peformed. No OBEX server commands are served.
 *
 * This class extends ExchangerState abstract class. This class also
 * implments DiscoveryListener interface to receive Bluetooth inquiry
 * callbacks.
 *
 * @version 1.0 29.09.2005
 * @see bcexchanger.comm.ExchangerState
 *      javax.bluetooth.DiscoveryListener Design Patterns: State
 *
 */
public class InquiryState extends ExchangerState
    implements DiscoveryListener {

  private DiscoveryAgent agent;
  private Vector remoteDevices; // vector of found devices
  private int operation = ServiceDiscoveryState.GET;

  /**
   * Constructor
   *
   * @param _parent -
   *          the class which is nesting the current state of the
   *          state machine
   */
  public InquiryState(ExchangerStateParent _parent,int oper)
      throws IOException {
    super(_parent);
    operation = oper;
    remoteDevices = new Vector();
    // initiate Bluetooth
    LocalDevice local = LocalDevice.getLocalDevice();
    agent = local.getDiscoveryAgent();
    // start Bluetooth inquiry
    agent.startInquiry(DiscoveryAgent.GIAC, this);

  }

  /*
   * Inquiry state does not allow to start any other business card
   * exchange process.
   *
```

```
     * @see bcexchanger.comm.ExchangerState#startSending()
     */
    public void startSending(int op) throws IOException {
      throw new IOException(
          "Inquiry is in progress. Inquiry has to be canceled before starting new
sending process");
    }

    /*
     * InquiryState allows to cancel inquiry process.
     *
     * @see bcexchanger.comm.ExchangerState#cancelSending()
     */
    public void cancelSending() {
      agent.cancelInquiry(this);
    }
```

4 The method `deviceDiscovered()` is called when a device is found during an inquiry.

The method `inquiryCompleted()` is called when the inquiry is completed. After that, the inquiry completion codes are converted into application completion codes. For more information on inquiry completion, see the interface `DiscoveryListener`.

```
    public void deviceDiscovered(RemoteDevice dev, DeviceClass devClass)
    {
      remoteDevices.addElement(dev);
    }

    public void inquiryCompleted(int code) {
      try
      {
          int completionCode = ExchangerComm.ERROR;

        // convert the inquiry completion code to application completion
        // code
        switch (code) {
          case DiscoveryListener.INQUIRY_COMPLETED:
            completionCode = ExchangerComm.DONE;
            break;
          case DiscoveryListener.INQUIRY_TERMINATED:
            completionCode = ExchangerComm.CANCELED;
            break;
          case DiscoveryListener.INQUIRY_ERROR:
            completionCode = ExchangerComm.ERROR;
            break;
        }
        parent.getListener().onInquiryComplete(completionCode); // signal
                                                                // that
                                                                // inquiry
                                                                // is
                                                                // done

        if (code == DiscoveryListener.INQUIRY_COMPLETED) { // no errors
                                                           // or
                                                           // cancelations
          parent.setState(new ServiceDiscoveryState(parent,
              remoteDevices,operation));

        } else {
          parent.setState(new IdleState(parent));
        }

      } catch (Exception e) {
          parent.setState(new IdleState(parent));
      }
    }

    /*
     * Service discovery callbacks are not handled and not supposed to
     * occur, since service discovery process is not started
     *
     * @see javax.bluetooth.DiscoveryListener#servicesDiscovered(int,
```

```
     *      javax.bluetooth.ServiceRecord[])
     */
    public void servicesDiscovered(int arg0, ServiceRecord[] arg1) {
      throw new RuntimeException(
          "Internal error #4: InquiryState.servicesDiscovered() should not be
called");

    }

    /*
     * Service discovery callbacks are not handled and not supposed to
     * occur, since service discovery process is not started
     *
     * @see javax.bluetooth.DiscoveryListener#serviceSearchCompleted(int,
     *      int)
     */
    public void serviceSearchCompleted(int arg0, int arg1) {
      throw new RuntimeException(
          "Internal error #5: InquiryState.serviceSearchCompleted() should not be
called");

    }
```

5 The `ServerRequestHandler` class methods `onGet()` and `onPut()` are called when a GET or PUT method is received. These methods receive an `Operation` object as a parameter. The application uses the `Operation` object to read and write data. After the request is handled, the method must return a response code. All response codes are defined in the `ResponseCodes` class.

```
    /*
     * Serving OBEX GET operation is supported only in IdleState
     *
     * @see bcexchanger.comm.ExchangerState#onGet(javax.obex.Operation)
     */
    public int onGet(Operation op) {

      return ResponseCodes.OBEX_HTTP_CONFLICT;
    }

    /*
     * Serving OBEX GET operation is supported only in
     * IdleState
     *
     * @see bcexchanger.comm.ExchangerState#onPut(javax.obex.Operation)
     */
    public int onPut(Operation op) {
      // onPut is supported only in IdleState
      return ResponseCodes.OBEX_HTTP_CONFLICT;
    }

  }
```

6.5.8 ServiceDiscoveryState

This class represents the state of the running service discovery. In this state an active service discovery is performed and no OBEX server commands are handled.

This class extends the `ExchangerState` abstract class. It also implements the `DiscoveryListener` interface to receive Bluetooth service discovery callbacks. Service discovery procedures are run from a separate thread and therefore the class implements the `Runnable` interface.

To create the class:

1 Create the `InquiryState` class file.

2 Assign the class to the `Comm` package and import the required classes.

```
package bcexchanger.comm;

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

```
import java.util.Vector;

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.obex.Operation;
import javax.obex.ResponseCodes;
```

3 Set the class to extend `ExchangerState` and to implement `DiscoveryListener` and `Runnable`. Set up instance variables.

Initiate a separate thread for starting service discovery.

```
/**
 *
 * This class implements service discovery state. In this state an active
 * service discovery is peformed. No OBEX server commands are served.
 *
 * This class extends ExchangerState abstract class. This class also implments
 * DiscoveryListener interface to receive Bluetooth service discovery callbacks.
 * Service discoveries are run from the separate thread therefore the class
 * implements Runnable interface
 *
 * @version 1.0 29.09.2005
 * @see bcexchanger.comm.ExchangerState javax.bluetooth.DiscoveryListener Design
 *      Patterns: State
 *
 */
public class ServiceDiscoveryState extends ExchangerState implements
        DiscoveryListener, Runnable {

    // Instance variables
    private Thread serviceDiscoveryThread;

    private Vector services;

    private DiscoveryAgent agent;

    private int serviceDiscoveryID;

    private Vector devices;

    private boolean canceled = false;

    public static int GET = 0;
    public static int PUT = 1;

    /**
     * Constructor
     *
     * @param _parent -
     *            the class which is nesting the current state of the state
     *            machine
     * @param _devices -
     *            a vecotor of RemoteDevice object representing devices found
     *            during inquiry
     */
    public ServiceDiscoveryState(ExchangerStateParent _parent, Vector _devices,int
oper)
            throws IOException {
        super(_parent);
        canceled = false;

        services = new Vector();
        devices = _devices;

        // initiate Bluetooth
        LocalDevice local = LocalDevice.getLocalDevice();
```

```
            agent = local.getDiscoveryAgent();

            serviceDiscoveryThread = new Thread(this);
            serviceDiscoveryThread.start();

        }
```

4 Set the functionality for canceling the service discovery process and handling inquiry callbacks at this state.

The method `servicesDiscovered()` is called when service(s) are found during a service search. The method `serviceSearchCompleted()` is called when a service search is completed or was terminated because of an error.

```
        /*
         * ServiceDiscoveryState does not allow to start any other business card
         * exchange process.
         *
         * @see bcexchanger.comm.ExchangerState#startSending()
         */
        public void startSending(int op) throws Exception {
            throw new IOException(
                    "Service discovery is in progress. Service discovery has to be
    canceled before starting new sending process");

        }

        /*
         * ServiceDiscoveryState allows to cancel discovery process.
         *
         * @see bcexchanger.comm.ExchangerState#cancelSending()
         */
        public void cancelSending() {

            canceled = true;
            agent.cancelServiceSearch(serviceDiscoveryID);

        }

        /*
         *
         * Inquiry callbacks are not handled and not supposed to occur, since
         * inquiry process is not started
         *
         * @see javax.bluetooth.DiscoveryListener#deviceDiscovered
    (javax.bluetooth.RemoteDevice,
         *      javax.bluetooth.DeviceClass)
         */
        public void deviceDiscovered(RemoteDevice arg0, DeviceClass arg1) {

            throw new RuntimeException(
                    "Internal error #8: ServiceDiscoveryState.deviceDiscovered() should
    not be called");
        }

        /*
         *
         * Inquiry callbacks are not handled and not supposed to occur, since
         * inquiry process is not started
         *
         * @see javax.bluetooth.DiscoveryListener#inquiryCompleted(int)
         */
        public void inquiryCompleted(int arg0) {

            throw new RuntimeException(
                    "Internal error #9: ServiceDiscoveryState.inquiryCompleted() should
    not be called");

        }

        public void servicesDiscovered(int id, ServiceRecord[] _services) {
```

```
        for (int i = 0; i < _services.length; i++) {
            services.addElement(_services[i]);
        }

    }

    public synchronized void serviceSearchCompleted(int arg0, int arg1) {
        notify();

    }
```

5 Use the `Run()` method to implement the logic of the service discovery process. Set the search for all available services unless the search is canceled. Use the method `searchServices()` to look for services on remote devices that have all the UUIDs specified in uuidSet. The method returns the transaction ID of the service search. For more information on UUIDs, see the class UUID.

If one service is found, connect to that service. However, if more than one service is found, display a list of those services and let user choose the service to connect to. Set listeners for handling user actions.

```
    /*
     * (non-Javadoc)
     *
     * This methods implements logic of the service discovery process
     *
     * @see java.lang.Runnable#run()
     */
    public synchronized void run() {

        Enumeration e = devices.elements();
        // Business card exchanger service UUID
        UUID[] uuids = new UUID[1];
        uuids[0] = new UUID(parent.getUUID(), false);

        // proceeded with all devices if not canceled
        while (e.hasMoreElements() && !canceled) {

            RemoteDevice device = (RemoteDevice) e.nextElement();
            try {
                serviceDiscoveryID = agent.searchServices(null, uuids, device,
                        this);
            } catch (Exception e0) {
                // signal error to the MIDlet
                parent.getListener().onServiceDiscoveryComplete(
                        ExchangerComm.ERROR);
                parent.setState(new IdleState(parent));
                return;
            }

            try {
                wait(); // wait until service search is done on this device
            } catch (Exception e1) {
                // Ignore
            }
        }

        if (canceled) { // handle user's cancelation
            try {
                parent.getListener().onServiceDiscoveryComplete(
                        ExchangerComm.CANCELED);
                parent.setState(new IdleState(parent));
            } catch (Exception e1) {
                throw new RuntimeException(
                        "Internal error #10: ServiceDicoveryState.run()");
            }
        } else
        { // not canceled
            if (services.isEmpty()) { // no services on devices
                try {
                    parent.getListener().onServiceDiscoveryComplete(
                            ExchangerComm.NO_RECORDS);
                    parent.setState(new IdleState(parent));
```

```
                    } catch (Exception e1) {

                    }
                }
                else if (services.size() == 1)
                { // one service found,
                    // connect to it
                    try {
                        ServiceRecord serviceRecord = (ServiceRecord) services
                                .firstElement();
                        parent.getListener().onServiceDiscoveryComplete(
                                ExchangerComm.DONE);
                            SendState state = new SendState(parent, serviceRecord
                                    .getConnectionURL(
                                            ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
                                            false));

                            parent.setState(state);
                            state.doSend();
                            state = null;
                            Thread.sleep(new Long(5000).longValue());

                            ReceiveState rstate = new ReceiveState(parent,
serviceRecord
                                    .getConnectionURL(
                                            ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
                                            false));
                            parent.setState(rstate);
                            rstate.doGet();
                            rstate = null;
                            Thread.sleep(new Long(5000).longValue());

                    }
                    catch (Exception e2)
                    {

                    }
                }
                else
                { // several services found, let user choose
                    try {
                        // list of friendly names of devices which contain services
                        Vector friendlyNames = createFriendlyNamesList(services);

                        int index = parent.getListener().resolveMultipleServices(
                                friendlyNames);

                        if (!canceled)
                        { // if not canceled during resolving
                            ServiceRecord serviceRecord = (ServiceRecord) services
                                    .elementAt(index);

                            parent.getListener().onServiceDiscoveryComplete(
                                    ExchangerComm.DONE);

                                SendState state = new SendState(parent, serviceRecord
                                        .getConnectionURL(

ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
                                                false));
                                parent.setState(state);
                                state.doSend();
                                state = null;
                                Thread.sleep(new Long(5000).longValue());

                                ReceiveState rstate = new ReceiveState(parent,
serviceRecord
                                        .getConnectionURL(

ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
                                                false));
                                parent.setState(rstate);
```

```
                            rstate.doGet();

                            rstate = null;
                            Thread.sleep(new Long(5000).longValue());

                        }
                        else
                        { // if canceled during resolving
                            parent.getListener().onServiceDiscoveryComplete(
                                    ExchangerComm.CANCELED);
                            parent.setState(new IdleState(parent));
                        }

                        }
                        catch (Exception e1) {
                        }

                }

            }
        }
```

6 Implement the method, used in the previous step, for creating a list of devices that contains the searched services. In case of an exception in getting the name of a device, use the method `getBluetoothAddress()` to retrieve the Bluetooth address of the device.

```
    /*
     * This method creates a list of the friendly names of devices that contain
     * the servicies
     *
     * @param _services - vector of ServiceRecord objects representing found
     * services @return - vectors of strings containing friendly names of
     * devices
     */
    private Vector createFriendlyNamesList(Vector _services) {

        Vector friendlyNames = new Vector();
        Enumeration e = _services.elements();
        while (e.hasMoreElements()) {
            ServiceRecord serviceRecord = (ServiceRecord) e.nextElement();
            RemoteDevice device = serviceRecord.getHostDevice();

            try {
                friendlyNames.addElement(device.getFriendlyName(false));
            } catch (IOException e1) {
                // If there is an excption getting the friendly name
                // use the address
                friendlyNames.addElement(device.getBluetoothAddress());
            }
        }
        return friendlyNames;
    }

    /*
     * Server OBEX GET command is supported only in IdleState
     *
     * @see bcexchanger.comm.ExchangerState#onGet(javax.obex.Operation)
     */
    public int onGet(Operation op) {
        return ResponseCodes.OBEX_HTTP_CONFLICT;
    }

    /*
     * Server OBEX PUT command is supported only in IdleState
     *
     * @see bcexchanger.comm.ExchangerState#onPut(javax.obex.Operation)
     */
    public int onPut(Operation op) {
        return ResponseCodes.OBEX_HTTP_CONFLICT;
    }
}
```

6.5.9 ReceiveState

This class represents the state when a business card is received from the remote device. In this state the client OBEX GET operation is performed. No OBEX server commands are handled.

To create the class:

1 Create the `InquiryState` class file.

2 Assign the class to the `Comm` package and import the required classes.

```
package bcexchanger.comm;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.IOException;
import java.io.InputStream;
import javax.microedition.io.Connector;
import javax.obex.ClientSession;
import javax.obex.HeaderSet;
import javax.obex.Operation;
import javax.obex.ResponseCodes;
```

3 Set the class to extend the `ExchangerState` class. Set the instance variables for `ClientSession` and `Operation`.

```
/**
 *
 * This class implements business card receiving state. In this state
 * an client OBEX GET operation is peformed. No OBEX server commands
 * are served.
 *
 * This class extends ExchangerState abstract class.
 *
 * @version 1.0 29.09.2005
 * @see bcexchanger.comm.ExchangerState Design Patterns: State
 */
public class ReceiveState extends ExchangerState{

  // Instance variables
  private ClientSession connection = null;
  private Operation operation = null;
  String url = null;

  boolean cancelInvoked = false; // true after cancelSending is
                                 // called

  /**
   * Constructor
   * <p>
   * description
   *
   * @param _parent -
   *          the class which is nesting the current state of the
   *          state machine
     * @param _url -
     *              a URL of the remote OBEX service. This state is responsible
     *              for establishin the connection
   * @exception
   * @see
   */
  public ReceiveState(ExchangerStateParent _parent,
      String url)
  {
    super(_parent);
    this.url = url;
  }

  /*
   * ReceiveState does not allow to start any other business card
```

```
   * exchange process.
   *
   * @see bcexchanger.comm.ExchangerState#startSending()
   */
  public void startSending(int to) throws IOException {

    throw new IOException(
        "Receiving is in progress. Receiving has to be canceled before starting new
sending process");

  }

  /*
   * ReceiveState allows to cancel receiving process.
   *
   * @see bcexchanger.comm.ExchangerState#cancelSending()
   */
  public void cancelSending() {
    cancelInvoked = true;

    try { // cancel any active operation running
      operation.abort();
    } catch (IOException e) {
      // Ignore, aborting operation anyway
    }

    try { // send DISCONNECT command to the remote peer
      connection.disconnect(null);
    } catch (IOException e1) {
       // Ignore, disconnecting anyway
    }

  }
```

4 Implement the logic for receiving a business card.

The `getResponseCode()` method of the `HeaderSet` interface returns the response code received from the server. To see all the available response codes, see the `ResponseCodes` class.

```
  /*
   * This method implements the logic for
   * receiving a business card
   *
   */
  public void doGet()
  {
      try
      {
          connection = (ClientSession) Connector.open(url);
          HeaderSet response = connection.connect(null);
          operation = connection.get(null);
          InputStream in = operation.openInputStream();
          byte[] fullResult = null;
          {
              byte[] buf = new byte[256];
              ByteArrayOutputStream bout = new ByteArrayOutputStream(2048);
              for (int len = in.read(buf); len >= 0; len = in.read(buf))
              {
                  bout.write(buf, 0, len);
              }
              fullResult = bout.toByteArray();
          }
          ByteArrayInputStream bin = new ByteArrayInputStream(fullResult);
          DataInputStream din = new DataInputStream(bin);
          int size = din.readInt();
          byte[] vCard = new byte[size];
          din.read(vCard);

          parent.getListener().onReceiveBC(vCard);

          // End the transaction
          in.close();
```

```
            int responseCode = operation.getResponseCode();

            operation.close();

            if (cancelInvoked) {
                throw new Exception(
                "Cancel did not invoke any exception; throw exception then");
            }

            // receive is done
            try

            {
                parent.setState(new IdleState(parent));
                parent.getListener().onReceiveComplete(ExchangerComm.DONE);
            }
            catch (Exception e)
            {
            }

      }
      catch (Exception e1)
      {
        if (cancelInvoked)
        { // if exception was caused by canceling
            parent.setState(new IdleState(parent));
          parent.getListener()
              .onReceiveComplete(ExchangerComm.CANCELED);
        }
        else
        { // if exception was caused by error

          parent.setState(new IdleState(parent));
          parent.getListener().onReceiveComplete(ExchangerComm.ERROR);
        }
      }
      finally
      {
        try { // finalizing operation
          operation.close();
        } catch (IOException e3)
        {
        }

        try { // sending DISCONNECT command
          connection.disconnect(null);
        } catch (IOException e2)
        {
          // Ignore, disconnecting anyway
        }


      }
    }

    /*
     * Server OBEX GET command is supported only in
     * IdleState
     *
     * @see bcexchanger.comm.ExchangerState#onGet(javax.obex.Operation)
     */
    public int onGet(Operation op) {
      return ResponseCodes.OBEX_HTTP_CONFLICT;
    }

    /*
     * Server OBEX PUT command is supported only in
     * IdleState
     *
     * @see bcexchanger.comm.ExchangerState#onPut(javax.obex.Operation)
     */
    public int onPut(Operation op) {
      return ResponseCodes.OBEX_HTTP_CONFLICT;
```

```
    }

}
```

6.5.10 SendState

This class represents the state when a business card is sent to the remote phone. In this state the client OBEX PUT operation is performed. No OBEX server commands are handled. This class extends the `ExchangerState` abstract class.

To create the class:

1 Create the `InquiryState` class file.

2 Assign the class to the `Comm` package and import the required classes.

```
package bcexchanger.comm;

import java.io.IOException;
import java.io.OutputStream;

import javax.microedition.io.Connector;
import javax.obex.ClientSession;
import javax.obex.HeaderSet;
import javax.obex.Operation;
import javax.obex.ResponseCodes;
```

3 Set the class to extend the `ExchangerState` class. Set the instance variables.

```
/**
 * This class implements business card sending state. In this state a client
 * OBEX PUT operation is peformed. No OBEX server commands are served.
 *
 * This class extends ExchangerState abstract class.
 *
 * @version 1.0 29.09.2005
 * @see bcexchanger.comm.ExchangerState Design Patterns: State
 */
public class SendState extends ExchangerState {

    // Instance variables

    private ClientSession connection = null;

    private String url = null;

    private Operation operation = null;

    private boolean cancelInvoked = false;

    /**
     * Constructor
     *
     * @param _parent -
     *            the class which is nesting the current state of the state
     *            machine
     * @param _url -
     *            a URL of the remote OBEX service. This state is responsible
     *            for establishin the connection
     */
    public SendState(ExchangerStateParent _parent, String _url) {
        super(_parent);

        url = _url;
    }

    /*
     * SendState does not allow to start any other business card exchange
     * process.
     *
     * @see bcexchanger.comm.ExchangerState#startSending()
```

```
     */
    public void startSending(int o) throws IOException {
        throw new IOException(
                "Sending is in progress. Sending has to be canceled before starting
new sending process");

    }

    /*
     * SendState allows to cancel sending process.
     *
     * @see bcexchanger.comm.ExchangerState#cancelSending()
     */
    public void cancelSending() {

        cancelInvoked = true;

        try { // cancel any active OBEX operation
            operation.abort();
        } catch (Exception e) { // catch NullPointer exception also
            // Ignore, aborting anyway
        }

        try { // send DISCONNECT to the remote peer

            connection.disconnect(null);
        } catch (Exception e1) { // catch NullPointer exception also
            // Ignore, disconnecting anyway
        }

        try { // close the connection
            connection.close(); //

        } catch (Exception e2) { // catch NullPointer exception also
            // Ignore, closing anyway
        }

    }
```

4 Implement the logic for sending a business card.

The `getResponseCode()` method of the `HeaderSet` interface returns the response code received from the server. To see all the available response codes, see the `ResponseCodes` class.

```
    /*
     * This method implements the logic for sending a
     * business card
     *
     */
    public void doSend()
    {
        try
        {
            connection = (ClientSession) Connector.open(url);
            HeaderSet response = connection.connect(null);

            // Initiate the PUT request
            operation = connection.put(null);
            OutputStream out = operation.openOutputStream();
            //getting the own card
            byte[] vCard = parent.getListener().getOwnBC();
            int vlen = vCard.length;
            byte[] tmpBuf = new byte[vlen + 4];
            System.arraycopy(vCard, 0, tmpBuf, 4, vlen);
            tmpBuf[0] = (byte) ((vlen >>> 24) & 0xff);
            tmpBuf[1] = (byte) ((vlen >>> 16) & 0xff);
            tmpBuf[2] = (byte) ((vlen >>> 8) & 0xff);
            tmpBuf[3] = (byte) ((vlen >>> 0) & 0xff);
            //sending data
            out.write(tmpBuf);
            out.close();
            int responseCode = operation.getResponseCode();
```

```
                operation.close();

                if (cancelInvoked) {
                    throw new Exception(
                            "Cancel did not invoke any exception; throw exception
then");
                }

                // send is done
                try {
                    parent.setState(new IdleState(parent));
                } catch (Exception e)
                {
                }
            }
            catch (Exception e1)
            {
                parent.setState(new IdleState(parent));
                if (cancelInvoked)
                { // if exception is caused by cancelation
                    parent.getListener().onSendComplete(ExchangerComm.CANCELED);
                } else
                { // if exception is caused by error
                    parent.setState(new IdleState(parent));
                    parent.getListener().onSendComplete(ExchangerComm.ERROR);
                }
            }
                finally
                {
                  try { // finalizing operation
                    operation.close();
                  }
                  catch (IOException e3) {

                  }

                  try { // sending DISCONNECT command
                    connection.disconnect(null);
                  }
                  catch (IOException e2)
                  {
                    // Ignore, disconnecting anyway
                  }
                }
        }

        /*
         * Server OBEX GET command is supported only in IdleState
         *
         * @see bcexchanger.comm.ExchangerState#onGet(javax.obex.Operation)
         */
        public int onGet(Operation op) {
            return ResponseCodes.OBEX_HTTP_CONFLICT;
        }

        /*
         * Server OBEX PUT command is supported only in IdleState
         *
         * @see bcexchanger.comm.ExchangerState#onPut(javax.obex.Operation)
         */
        public int onPut(Operation op) {
            return ResponseCodes.OBEX_HTTP_CONFLICT;
        }


    }
```

6.6 Creating UI package

The package `bcexchanger.ui` contains the classes which implement the UI of the application. The UI is implemented as a state machine, where each screen is a state. The package `ui` contains the abstract class `Screen` and classes `AddressBookScreen`, `AlertMessage`, `MainScreen`, `ProgressScreen`, and `ServiceListScreen`.

> **Note:** Since this package contains only UI related classes, the code is not described here. To see the code, download the example from Forum Nokia at http://www.forum.nokia.com/info/sw.nokia.com/id/b37cb6fa-fa2b-429c-9d1e-3b9728a23ac8/MIDP_Bluetooth_OBEX_Example_Business_Card_Exchange_v1_0_en.zip.html. For more information on user interfaces, see also UI and Graphics section of the Java™ ME Developer's Library at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm.

6.6.1 AddressBookScreen

This class is responsible for displaying the content of the phone book and allowing user to choose his or her own business card. The user can either choose the contact or exit the application. The class extends the `Screen` (`bcexchanger.ui`) class.

6.6.2 AlertMessage

This class represents a screen displaying an alert message. The class extends the `Screen` (`bcexchanger.ui`) class.

6.6.3 MainScreen

This is the idle state screen. This class implements the main screen of the application. Using this screen the user can initiate an exchange of business cards, change own business card, and exit the application. The class is extends the `Screen` (`bcexchanger.ui`) class.

6.6.4 ProgressScreen

This class implements the screen, which indicates that some operation is in progress. For example, when inquiry, service discovery, or business card sending and receiving is running, this screen is displayed. With this screen the user can cancel the process. The class is extends the `Screen` (`bcexchanger.ui`) class.

6.6.5 Screen

This class is an abstract base class for all the UI screens in this application. This class implements the `CommandListner` interface.

6.6.6 ServiceListScreen

This class is used to display the list of devices which provide services (in case there is more than one "Business Card Exchanger Service" in the vicinity) and to provide the means for choosing only one service or canceling the process.

6.7 Building and running in an emulator

After you have created all the necessary files for the project, you can build the project and run the application in an emulator.

Note: The MIDlet used in this example was built using the Eclipse IDE. Please see the instructions for building MIDlet projects for Eclipse that come with the Getting Started with Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library.

You can also construct them using command line or other IDEs of your choice.

6.8 Deploying to a device

Deploy test packages onto your mobile device, using a USB cable, Bluetooth connection, infrared, or other hardware compatible connection. Alternatively, deploy the application using OTA technology.

For instructions on how to do this, see the Getting Started with Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library.