JSR 82 - Bluetooth

Version 0.9, Draft

API GUIDE

COPYRIGHT

Trademarks and Service Marks

About This Document

This document describes the Bluetooth Wireless Technology (JSR-82). It focuses on Bluetooth connections and protocols. The document also provides sample code snippets for client/server pair using RFCOM.

Scope

This document is intended for users who have knowledge of Java programming language.

Document History:

Date	Version	Comment
11/06/09	0.9	Draft

References:

- JSR 82 specification: http://jcp.org/en/jsr/detail?id=82
- JSR 82 Article: 1) http://developers.sun.com/mobility/midp/articles/bluetooth2/

  2) http://developers.sun.com/mobility/midp/articles/bluetooth1/

Abbreviations:

JSR	Java Specification Request
OBEX	Object Exchange Protocol
RFCOMM	Radio Frequency Communication
CLDC	Connection Limited Device Configuration
L2CAP	Logical Link Control and Adaptation Protocol
LAN	Local Area Network
API	Application Programming Interface
MIDP	Mobile Information Device Profile
PPP	Point-to-Point Protocol

Table of Contents

List of Tables

Introduction

Bluetooth is one of the wireless communication protocols. It is mainly used for short distance and in devices with low power consumption. With the help of Bluetooth, you can communicate in an omni-directional manner of up to 30 feet at 1 Mb/s. JSR 82 is based on version 1.1 of the Bluetooth Specification.

Bluetooth is the first open, non-proprietary standard for developing Bluetooth applications using Java programming language. It hides the complexity of the Bluetooth protocol stack behind a set of Java APIs, that emphasis on application development rather than the low-level details of Bluetooth.

This specification includes basic support for at least, the following Bluetooth protocols:

- RFCOMM
- OBEX
- Service Discovery protocols
- L2CAP (Logical Link Control and Adaptation Protocol)

Additional protocol support may be added in future versions. The specification is primarily targeted at native Bluetooth protocols.

The Java ME APIs for Bluetooth are targeted at devices characterized as follows:

- 512 K minimum total memory available (ROM/Flash and RAM). Application memory requirements are additional.
- Bluetooth network connection.
- Compliant implementation of the Java ME Connected Limited Device Configuration (CLDC).

Overview

Bluetooth devices cannot interact unless they conform to a particular profile. Bluetooth profiles are intended to ensure interoperability among Bluetooth-enabled devices and applications from different manufacturers and vendors. A profile defines the roles and capabilities for specific types of applications.

Bluetooth Profiles

Types of Bluetooth profiles are:

- Generic Access Profile
- Service Discovery Application and Profile
- Serial Port Profile
- LAN Access Profile
- Synchronization Profile
- Basic Imaging Profile
- Basic Printing Profile
- File Transfer Profile

Generic Access Profile: This profile defines connection procedures, device discovery, and link management. It also defines procedures related to use the different security models and common format requirements for parameters accessible on the user interface level. At a minimum, all bluetooth devices must support this profile.

Service Discovery Application and Profile: This profile defines the features and procedures for an application in a Bluetooth device to discover services registered in other Bluetooth devices, and retrieves information related to the services.

Serial Port Profile: This profile defines the requirements for Bluetooth devices that need to set up connections, which emulate serial cables and use the RFCOMM protocol.

LAN Access Profile: This profile defines how Bluetooth devices can access the services of a LAN using PPP (Point-to-Point Protocol), and shows how PPP mechanisms can be used to form a network consisting of Bluetooth devices.

Synchronization Profile: This profile defines the application requirements for Bluetooth devices that need to synchronize data on two or more devices.

Basic Imaging Profile: This profile is designed for sending images between devices and includes the ability to resize, and convert images to make them suitable for the receiving device.

Basic Printing Profile: This allows devices to send text, e-mails, vCards, or other items to printers based on print jobs. This makes it more suitable for embedded devices such as mobile phones and digital cameras, which cannot easily be updated with drivers dependent upon printer vendors.
File Transfer Profile: Provides the capability to browse, manipulate and transfer objects (files and folders) in an object store (file system) of another system.

Bluetooth API Architecture

Purpose of the specification was to define an open, non-proprietary standard API that can be used by all Java ME enabled devices. So Standard Java ME APIs and CLDC/MIDP's Generic Connection Framework were used for designing.

Some important features:

Bluetooth protocols and profiles were supported by these specifications. It does not include specific APIs for all Bluetooth profiles simply because the number of profiles is growing.

Specification includes the RFCOMM, OBEX, and Service Discovery protocols, L2CAP communication protocols in the JSR 82 APIs, primarily because all current Bluetooth profiles are designed to use these communication protocols.

The Service Discovery protocol is also supported. JSR 82 defines service registration in detail in order to standardize the registration process for the application programmer.

JSR 82 requires the Bluetooth stack underlying a JSR 82 implementation to be qualified for the Generic Access Profile, the Service Discovery Application Profile, and the Serial Port Profile. The stack must also provide access to its Service Discovery Protocol, and to the RFCOMM and L2CAP layers.

This API is user friendly. So developers can use the Java programming language to build new Bluetooth profiles on top of this API as long as the core layer specification does not change.

Specification is not restricted to promote this flexibility and extensibility to APIs that implement Bluetooth profiles. This JSR also includes APIs for OBEX and L2CAP. This will help for future Bluetooth profiles, which can be implemented in Java. Figure 1 shows where the APIs defined in this specification fit in CLDC/MIDP architecture.

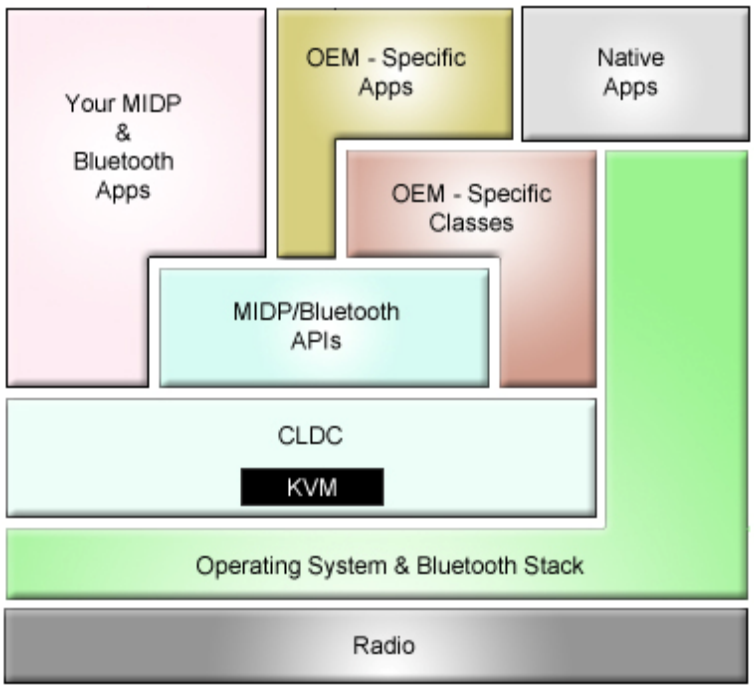

Figure 1: High-level Architecture of J2ME CDLC/MIDP and Bluetooth

API Description:

Java APIs for Bluetooth define two packages that depend on the CLDC *javax.microedition.io.*;* package:

- javax.bluetooth: core Bluetooth API
- javax.obex: APIs for the Object Exchange (OBEX) protocol

OBEX APIs are defined independently of the Bluetooth transport layer and packaged separately. Each of the above packages represents a separate optional package, which means that a CLDC implementation can include either package or both. MIDP enabled devices are expected to be the kind of devices to incorporate this specification. This document focuses on core Bluetooth API (javax.bluetooth).

The javax.bluetooth API is intended to provide the following capabilities:

- Register services
- Discover devices and services
- Establish RFCOMM, L2CAP connections between devices
- Using those connections, send and receive data.
- Manage and control the communication connections.
- Provide security for these activities.

Bluetooth protocols

Bluetooth API javax.bluetooth defines the following protocol for communication:

- L2CAP
- RFCOMM

Logical Link Control and Adaptation Protocol (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.

Radio Frequency Communication (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.

Bluetooth Communication:

Bluetooth application has to carry out the following parts for communication:
stack initialization, setting discovery mode, device discovery, service discovery, and connection.

- **Stack initialization**

In Bluetooth device, the Bluetooth stack is used for controlling the device. So this should be initialized first before starting. After the initialization, device will be ready for use.

In JSR 82, specification consists of LocalDevice class. This is also called as basic entry point. LocalDevice class helps in initializing the JSR 82 stack. This is also used to control the local Bluetooth settings. LocalDevice class provides the lowest level of access to the Bluetooth stack. LocalDevice class provides with the ability to query information about your own Bluetooth device.

To initialize Bluetooth stack, get the LocalDevice reference by calling static method *LocalDevice.getLocalDevice ()*

```
localDevice = LocalDevice.getLocalDevice();
```

- **Discovery Mode**

Now set the device's discoverable mode by calling *LocalDevice.setDiscoverable(int mode)*. JSR 82 supports three access modes:

- *DiscoveryAgent.GIAC*
- *DiscoveryAgent.LIAC*
- *DiscoveryAgent.NOT_DISCOVERABLE*

A Bluetooth device can have various settings for its "discoverable mode." For example, a device may be configured, not to be discoverable. In this case, other Bluetooth devices that are within range cannot 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. To take out the device from the discoverable mode, set mode to NOT_DISCOVERABLE.

It will return true or false after setting the discoverable mode, if it returns false, that mean the Bluetooth device does not support the access mode.

```
/* Retrieve the local device to get to the Bluetooth Manager*/
   localDevice = LocalDevice.getLocalDevice();
   /* Set the discoverable mode to GIAC*/
   localDevice.setDiscoverable(DiscoveryAgent.GIAC);
```

- **Device discovery**

The core Bluetooth API's ***DiscoveryAgent*** class and ***DiscoveryListener*** interface provide the necessary discovery services.
After getting a LocalDevice object, instantiate a DiscoveryAgent by calling LocalDevice.getDiscoveryAgent().

```
/* Retrieve the local device to get to the Bluetooth Manager*/
   localDevice = LocalDevice.getLocalDevice();
   /* set the discoverable mode to GIAC*/
   localDevice.setDiscoverable(DiscoveryAgent.GIAC);
   /*retrieve the discovery agent*/
   discoveryAgent = localDevice.getDiscoveryAgent();
```

There are two ways to obtain a list of accessible devices (RemoteDevice) using DiscoveryAgent.

1) The *DiscoveryAgent.startInquiry(int accessCode,DiscoveryListener listener)* method places the device into an inquiry mode. Application must specify a DiscoveryListener that will respond to inquiry-related events to take advantage of this mode. *DiscoveryListener.deviceDiscovered()* is called each time an inquiry finds a device. *DiscoveryListener.inquiryCompleted()* is invoked when the inquiry is completed or canceled.

```
public class BTDiscovery implements DiscoveryListener{
...
/* retrieve the discovery agent */
DiscoveryAgent agent = local.getDiscoveryAgent();

/* place the device in inquiry mode*/
boolean complete =
agent.startInquiry(DiscoveryAgent.GIAC, this);
...
public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {

form.append("Device discovered: "+btDevice.getBluetoothAddress());
}

public void inquiryCompleted(int discType)

switch (discType) {

case DiscoveryListener.INQUIRY_COMPLETED :
        form.append("INQUIRY_COMPLETED");
break;

case DiscoveryListener.INQUIRY_TERMINATED :

form.append("INQUIRY_TERMINATED");

break;

case DiscoveryListener.INQUIRY_ERROR :

form.append ("INQUIRY_ERROR");

break;

default :
```

```
form.append ("Unknown Response Code");

break;

}

}

}
```

2) DiscoveryAgent.retrieveDevices(int retrieveOption) method is used to retrieve an existing list if the device does not wait for other devices to be discovered. Depending on the parameter passed, this method will return either a list of devices that were found in a previous inquiry (CACHED), or a list of pre-known.

```
/* Retrieve PREKNOWN devices*/
RemoteDevice[] devices = agent.retrieveDevices(
DiscoveryAgent.PREKNOWN);

/*Retrieve CACHED devices */
RemoteDevice[] devices = agent.retrieveDevices(
DiscoveryAgent.CACHED);
```

RemoteDevice class represents a remote device (i.e., a device within a range of reach) and provides methods to retrieve information like Bluetooth address and name about the device.

- **Service discovery:**

Once the local device has discovered at least one remote device, it can begin to search for available services - Bluetooth applications can use to accomplish useful tasks. Because service discovery is much like device discovery, *DiscoveryAgent* also provides methods to discover services on a Bluetooth server device, and to initiate service-discovery transactions. Note that the API provides mechanisms to search for services on remote devices, but not for services on the local device.

The *servicesDiscovered()* and *serviceSearchCompleted()* methods of *DiscoveryAgent* must be implemented. They will handle the events occurring when services are found or when the service discovery completes.

```
public class BTDiscovery implements DiscoveryListener{
...
/* retrieve the discovery agent */
DiscoveryAgent agent = local.getDiscoveryAgent();
```

```
/* place the device in inquiry mode*/
boolean complete =
agent.startInquiry(DiscoveryAgent.GIAC, this);
...

public void deviceDiscovered(RemoteDevice remoteDevice, DeviceClass cod) {

int[] attributes = {0x100,0x102,0x102};
/*
* Supplying UUIDs in an UUID array enables searching for
* specific services.
*/
UUID[] uuids = new UUID[1];
uuids[0] = new UUID(0x1002);
try {
agent.searchServices(attributes,uuids, remoteDevice,this);
} catch (BluetoothStateException e) {
/* Error handling code here*/
}
}

public void inquiryCompleted(int discType)
{
}
public void servicesDiscovered(int transID,
ServiceRecord [] serviceRecord) {
/* Services discovered, keep reference to the ServiceRecord
* array
*/
servicesFound = serviceRecord;
}
public void serviceSearchCompleted(int transID, int respCode) {
switch(respCode) {
case DiscoveryListener.SERVICE_SEARCH_COMPLETED:
/*
* Service search completed successfully
* Add appropriate code here
*/
break;
case
DiscoveryListener.SERVICE_SEARCH_DEVICE_NOT_REACHABLE:
/* device not reachable, add appropriate code here*/
break;
case DiscoveryListener.SERVICE_SEARCH_ERROR:
/* Service search error, add appropriate code here*/
break;
case DiscoveryListener.SERVICE_SEARCH_NO_RECORDS:
```

```
/* No records found, add appropriate code here*/
break;
case DiscoveryListener.SERVICE_SEARCH_TERMINATED:
/*
* Search terminated by agent.cancelServiceSearch()
* Add appropriate code here
*/
break;

}
```

Service should first be registered or advertised on a Bluetooth server device, before it is discovered. Server performs many things like creating a service record that describes the service offered, accepting connections from clients, and adding a service record to the server's Service Discovery Database (SDDB).

- **Connection**

As mentioned in the Bluetooth protocol section, Bluetooth connections are based on the Logical Link Control and Adaptation Layer Protocol (L2CAP), a low-level data-packet protocol, and a serial emulation protocol over L2CAP that is supported by the Serial Port Profile (SPP) RFCOMM. Bluetooth connections are based on the Generic Connection Framework (GCF) factory method *javax.microedition.io.Connector.open()* and are represented by the L2CAPConnection and *StreamConnection* types respectively.

The connection URL scheme determines the connection type to create:

The URL format for an L2CAPConnection:

```
btl2cap://hostname:[PSM | UUID];parameters
```

The URL format for an RFCOMM StreamConnection:

```
btspp://hostname:[CN | UUID];parameters
```

hostname is either localhost to set up a server connection, or the Bluetooth address to create a client connection.

PSM is the Protocol/Service Multiplexer value, used by a client connecting to a server.

CN is the Channel Number value, used by a client connecting to a server.

UUID is the Universally Unique Identifier used when setting up a service on a server. Each UUID is guaranteed to be unique across all time and space.

Parameters include name to describe the service name, and the security parameters authenticate, authorize, and encrypt.

Example : Server RFCOMM URL:

```
btspp://localhost:2D26618601FB47C28D9F10B8EC891363;authenticate=false;
   encrypt=false;name= rfcommtest
```

A client RFCOMM URL:

```
btspp://0123456789AF:1;master=false;encrypt=false;authenticate=false
```

Using localhost as a hostname indicates you want a server connection. To create a client connection to a known device and service, use the service's connection URL, found in its ServiceRecord.

The following code snippet shows how to create an RFCOMM server connection:

```
...
/* Bluetooth Service name*/
private static final String myServiceName = " rfcommtest ";

/* Bluetooth Service UUID of interest*/
private static final String myServiceUUID = "2d26618601fb47c28d9f10b8ec891363";
private UUID MYSERVICEUUID_UUID = new UUID(myServiceUUID, false);
...
/* Define the server connection URL*/
String                                    connURL                                    =
"btspp://localhost:"+MYSERVICEUUID_UUID.toString()+";"+name="+myServiceName;
...
/* Create a server connection (a notifier)*/
StreamConnectionNotifier scn = (StreamConnectionNotifier) Connector.open(connURL);
...
/*Accept a new client connection*/
StreamConnection sc = scn.acceptAndOpen();
...
```

The following code snippet shows how to create a client connection for a given service of interest, using its service record:

```
...
/* Given a service of interest, get its service record*/
ServiceRecord sr = (ServiceRecord)discoveredServices.elementAt(i);
/* Then get the service's connection URL*/
String connURL = sr.getConnectionURL(ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
false);
/* Open connection*/
StreamConnection sc = (StreamConnection) Connector.open(connURL);
...
```

- **Service Registration**

Before a service can be discovered, it must first be registered - advertised on a Bluetooth server device. The server is responsible for:

1. **Creating a service record that describes the service offered.**

The Bluetooth implementation automatically creates a service record when your application creates a connection notifier, either a *StreamConnectionNotifier* or an L2CAPConnectionNotifier.

The following code snippet defines and instantiates an RFCOMM connection notifier, resulting in the creation of the service record:

```
...
StreamConnectionNotifier streamConnectionNotifier;
/* Create notifier (and service record)*/
streamConnectionNotifier = (StreamConnectionNotifier)
   Connector.open(connectionURL);
...
```

2. **Registering the services and Adding the service record to the server's Service Discovery Database (SDDB), so it is visible and available to potential clients and wait for client.**

Once you have created the connection notifier and the service record, the server is ready to register the service and wait for clients. Invoking the notifier *acceptAndOpen()* method, causes the Bluetooth implementation to insert the service record for the associated connection into the SDDB, making the service visible to clients.

```
...
/* Insert service record into SDDB and wait for an incoming client*/
StreamConnection conn = streamConnectionNotifier.acceptAndOpen();
...
```

3. **Updating the service record in the SDDB whenever the service's attributes change**.

You can retrieve the record from the SDDB by calling *LocalDevice.getRecord()*, add or change attributes of interest by calling *ServiceRecord.setAttributeValue()*, and write the service record back to the SDDB with a call to *LocalDevice.updateRecord()*:

```
...
try {
   /* Retrieve service record and set/update optional attributes,
    for example, ServiceAvailability, indicating service is available*/
   sr = localDevice.getRecord(streamConnectionNotifier);
   sr.setAttributeValue(SDP_SERVICEAVAILABILITY,
      new DataElement(DataElement.U_INT_1, 0xFF));
   localDevice.updateRecord(sr);
} catch (IOException ioe) {
   /* Catch exception, display error*/
}
...
```

4. **Removing or disabling the service record in the SDDB when the service is no longer available**

When the service is no longer needed, remove it from the SDDB by closing the connection notifier:

```
...
streamConnectionNotifier.close();
...
```

The following steps demonstrate the server using RFCOMM:

1. Take the example of ***LocalDevice*** reference, which represents the basic functions of Bluetooth manager. It provides the lowest level of interface possible into the Bluetooth stack. It is a singleton object.

```
localDevice = LocalDevice.getLocalDevice();
```

2. To put device in discoverable mode call *setDiscoverable(int inputMode)* method with input mode.

```
localDevice.setDiscoverable(DiscoveryAgent.GIAC);
```

DiscoveryAgent.GIAC = the inquiry access code for General/Unlimited Inquiry Access Code (GIAC). This is used to specify the type of inquiry to complete or respond.

3. Invoke ***Connector.open*** with a server connection URL argument to create a new service record that represents the service, and cast the result to a ***StreamConnectionNotifier*** that represents the service:

```
StreamConnectionNotifier
                    notifier= (StreamConnectionNotifier)Connector.open(SERVERURL);
```

It pauses thread until Transmission occurs:

```
conn = notifier.acceptAndOpen();
```

The following steps snippet demonstrates the Client using RFCOMM:

1. The Client has the different process to initiate the Bluetooth stack as compare to server. Take the ***LocalDevice*** reference and get ***DiscoveryAgent*** object for placing the device in inquiry mode.

```
LocalDevice localDevice = LocalDevice.getLocalDevice();

discoveryAgent = localDevice.getDiscoveryAgent();

discoveryAgent.startInquiry(DiscoveryAgent.GIAC, this);
```

2. The Client must implement the ***DiscoveryListener.*** This allows an application to receive device discovery and service discovery events. This interface provides four methods, two for discovering devices and two for discovering services.

- *deviceDiscovered(RemoteDevice btDevice, DeviceClass cod)*

  Called when a device is found during an inquiry.

- *inquiryCompleted(int discType)*

  Called when an inquiry is completed.

- *servicesDiscovered(int transID, ServiceRecord[] servRecord)*

  Called when service(s) are found during a service search.

- *serviceSearchCompleted(int transID, int respCode)*

  Called when a service search is completed or was terminated because of an error.

3. To take the service URL:

```
public void servicesDiscovered(int transID, ServiceRecord[] servRecord) {
   /* in this example there is only one service*/
   for(int i=0;i<servRecord.length;i++) {
      serviceUrl = servRecord[i].getConnectionURL(0,false);
   }
 }
```

4. To take the server response:

```
public void serviceSearchCompleted(int transID, int responseCode) {

……
      conn = (StreamConnection)Connector.open(serviceUrl);
…..
      OutputStream output = conn.openOutputStream();
…..
      InputStream inputStream = conn.openInputStream();
….

}
```

5. Once the communication is over then close all connection:

```
Conn.close();
Output.close();
inputStream.close();
```

- **Communication:**

Two devices must share a common communication protocol to use a service on a remote device in a local device. This sharing allows applications to access a wide variety of Bluetooth services; the Java APIs for Bluetooth provide mechanisms that allow connections to any service that uses RFCOMM, L2CAP, or OBEX as its protocol.

If the service uses another protocol (such as TCP/IP) layered above one of these protocols, the application can access the service, but only if it implements the additional protocol in the application, using the CLDC Generic Connection Framework.

URL used as a service record consists of digits and symbols. It looks something like: btspp://AAAAAAAAA11111111111111:8. This means that a client should use a Bluetooth Serial Port Profile (btspp://) to establish a connection with server channel 8 on a device with address AAAAAAAAA11111111111111. Device addresses are similar to physical addresses of computers.

JSR 82 - Bluetooth Example:

This sample example shows how to use Bluetooth API.

Class: Bluetooth Demo

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.Choice;
import javax.bluetooth.UUID;

public final class BluetoothDemo extends MIDlet implements CommandListener {

 /* Shared UUID by server and client */
 public static final UUID RFCOMM_UUID = new UUID(0x0003);
 /*Soft key names*/
 private static final String SELECT = "Select";
 private static final String EXIT   = "Exit";

 /* Soft-keys*/
 private final Command CMD_EXIT = new Command(EXIT, Command.EXIT, 1);
 private final Command CMD_SLECT = new Command(SELECT, Command.SCREEN, 2);

 /*  String array show on list */
 private static final String[] MENULABLES = {"Server", "Client" };
 /*List*/
 private final List menu = new List("Bluetooth Echo Demo", Choice.IMPLICIT, MENULABLES,
null);
 /*Display*/
 private Display display            = null;

 public BluetoothDemo() {

   menu.addCommand(CMD_EXIT);
   menu.addCommand(CMD_SLECT);
   menu.setCommandListener(this);

 }

 public void startApp() {
```

```
  display = Display.getDisplay(this);
  display.setCurrent(menu);

}

protected void destroyApp(boolean unconditional) {
}

protected void pauseApp() {
}

public void commandAction(Command cmd, Displayable d) {
                if (cmd == CMD_EXIT) {

                        destroyApp(true);
                        notifyDestroyed();
                        return;
                }

                else if (cmd == CMD_SLECT) {

                        switch (menu.getSelectedIndex()) {

                        case 0:
          /*Create the Server Object*/
                                new Server(display);
                                break;
                        case 1:
          /*Create the Client Object*/
                                new Client(display);
                                break;
                        }

                }
        }
}
```

Class: Server

```
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.TextBox;
import javax.microedition.lcdui.TextField;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.BluetoothStateException;
import javax.microedition.io.Connector;
import javax.microedition.io.StreamConnection;
import javax.microedition.io.StreamConnectionNotifier;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;

public class Server implements Runnable{

        private StreamConnectionNotifier notifier;
        private StreamConnection conn;
        private LocalDevice localDevice;
        private boolean isInit;
        private static final String SERVERURL = "btspp://localhost:"
                        + BluetoothDemo.RFCOMM_UUID +
                                                ";name=rfcommtest;authorize=false";
        private Display display = null;
        private final Form form = new Form("Server...");//,
                        //"Searching for Client...", 50, TextField.ANY);
        private static final String SERVERMSG = "\nHello Form Server...";


 public Server(Display dis) {
        display = dis;
        display.setCurrent(form);
         isInit = false;
         Thread thread = new Thread(this);
        thread.start();
 }

 public void run() {
        if (!isInit) {
                /* Initialization is done in the thread to avoid dead lock 'isInit'*/
                 /*ensures it is done once only*/
                try {
                        conn = null;
                        form.append("Searching for Client...");
                        localDevice = LocalDevice.getLocalDevice();
                        localDevice.setDiscoverable(DiscoveryAgent.GIAC);
```

```
                    notifier = (StreamConnectionNotifier)
                                        Connector.open(SERVERURL);
                } catch (BluetoothStateException e) {
                        form.append("BluetoothStateException: " +e.getMessage());
                } catch (IOException e) {
                        form.append("IOException: " + e.getMessage());
                }
                        isInit = true;
                }
                try {
                        /* Pauses thread until Transmission occurs*/
                        conn = notifier.acceptAndOpen();
                        OutputStream output = conn.openOutputStream();
                        output.write(SERVERMSG.length()); // length is 1 byte
                        output.write(SERVERMSG.getBytes());
                        output.close();
                        InputStream inputStream = conn.openInputStream();
                        int length = inputStream.read();
                        byte[] data = null;
                        data = new byte[length];
                        length = 0;
                        /* Assemble data*/
                        while (length != data.length) {
                                int ch = inputStream.read(data, length, data.length - length);
                                if (ch == -1) {
                                        throw new IOException("Can't read data");
                                }
                                length += ch;
                        }
                        String msg = new String(data);
                        form.append(msg);
                        inputStream.close();
                        conn.close();
                        notifier.close();

                } catch (Exception ex) {
                        form.append("Bluetooth Server Running Error: " + ex);
                }
        }
}
```

Class: Client

```
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.TextBox;
import javax.microedition.lcdui.TextField;
import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.AlertType;
import javax.bluetooth.DiscoveryListener;
import javax.bluetooth.DeviceClass;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.RemoteDevice;
import javax.bluetooth.UUID;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.ServiceRecord;
import javax.microedition.io.StreamConnection;
import javax.microedition.io.Connector;
import java.io.OutputStream;
import java.io.InputStream;
import java.io.IOException;

public class Client implements DiscoveryListener {
          private DiscoveryAgent discoveryAgent;
           private UUID[] uuidSet;
           private String serviceUrl;
           private Display display = null;
           private StreamConnection conn = null;
           private final Form form = new Form("Client...");
           private static final String CLIENTMSG = "\nHello From Client..";

 public Client(Display dis) {
             display = dis;
             display.setCurrent(form);
           try {
                      LocalDevice localDevice = LocalDevice.getLocalDevice();
                      discoveryAgent = localDevice.getDiscoveryAgent();
                      discoveryAgent.startInquiry(DiscoveryAgent.GIAC, this);
                     form.append("\nstart Inquiry...");
                     form.append("\nSearching for device...");
            } catch (Exception e) {
                      form.append("exception "+e);
           }
 }

 public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {
   try {
     /* Get Device Info*/
    form.append("\nDevice Discovered");
```

```
    form.append("\nMajor Device Class: " + cod.getMajorDeviceClass() + " Minor Device Class: "
   +  cod.getMinorDeviceClass());
    form.append("\nBluetooth Address: " + btDevice.getBluetoothAddress());
    form.append("\nBluetooth Friendly Name: " + btDevice.getFriendlyName(true));
    /* Search for Services*/
    uuidSet = new UUID[1];
    uuidSet[0] = BluetoothDemo.RFCOMM_UUID;
    int searchID = discoveryAgent.searchServices(null,uuidSet,btDevice,this);
  } catch (Exception e) {
    form.append("\nDevice Discovered Error: " + e);
  }
}

public void inquiryCompleted(int discType) {
  form.append("\nInquiryCompleted");
}

public void servicesDiscovered(int transID, ServiceRecord[] servRecord) {
    /*handling only one service*/
  form.append("\nServicesDiscovered");
  for(int i=0;i<servRecord.length;i++) {
     serviceUrl = servRecord[i].getConnectionURL(0,false);
  }
}

public void serviceSearchCompleted(int transID, int responseCode) {
  if(responseCode == SERVICE_SEARCH_ERROR)
    form.append("\nSERVICE_SEARCH_ERROR\n");

  if(responseCode == SERVICE_SEARCH_COMPLETED) {
    form.append("\nSERVICE_SEARCH_COMPLETED\n");
    form.append("\nService URL: " + serviceUrl);
    try {
          Alert alert = new Alert("Info...","Server Responding...",null,AlertType.INFO);
          alert.setTimeout(1200);
          conn = (StreamConnection)Connector.open(serviceUrl);
           OutputStream output = conn.openOutputStream();
           InputStream inputStream = conn.openInputStream();
          int length = inputStream.read();
           byte[] data = null;
          data= new byte[length];
          length = 0;
          // Assemble data
          while (length != data.length)
           {
                      int ch = inputStream.read(data, length, data.length - length);
                     if (ch == -1)
```

```
                    {
                            throw new IOException("Can't read data");
                     }
                     length += ch;
            }
            String msg = new String(data);
             form.append(msg);
            display.setCurrent(form);
            output.write(CLIENTMSG.length());
            output.write(CLIENTMSG.getBytes());
            output.close();
    } catch (Exception ex) {
            form.append ("exception "+ex);
    } finally {
            try {
              conn.close();
            } catch (IOException ioe) {
              form.append("Error Closing connection " + ioe);
      }
    }
   }
   if(responseCode == SERVICE_SEARCH_TERMINATED)
    form.append("\nSERVICE_SEARCH_TERMINATED\n");

   if(responseCode == SERVICE_SEARCH_NO_RECORDS)
    form.append("\nSERVICE_SEARCH_NO_RECORDS\n");

   if(responseCode == SERVICE_SEARCH_DEVICE_NOT_REACHABLE)
    form.append("\nSERVICE_SEARCH_DEVICE_NOT_REACHABLE\n");
  }
}
```