

---

# Bluetooth

## *Part 9: Service Discovery*

**Kjell Jørgen Hole**

UiB



Last updated 01.04.08

Mail: [Kjell.Hole@ii.uib.no](mailto:Kjell.Hole@ii.uib.no)

URL: [www.kjhole.com](http://www.kjhole.com)

- Service discovery overview
- JSR-82 API capabilities
- Programming with the API

**Remark:** This lecture contains large code examples from the text-book *Bluetooth Application Programming with the Java APIs* by T. J. Thompson *et al.*

- Unlike device discovery, the service discovery process involves only *two* devices
- Service discovery follows the [client-server model](#):
  1. The client asks the server whether it has a service defined by a **service record** that has a specific set of attributes
  2. If the server has a service record with the requested attributes, the service record is returned to the client
- Devices can simultaneously be both clients and servers

- Service records contain additional service information not found in the class of device records
- A service record answers the questions:
  - What kind of service does this server app. offer?
  - How does a client connect to this service?
- Service records for standard services are defined in Bluetooth profiles. Examples of standard services are LAN access, file transfer, business-card exchange, synchronization, and printing
- It is also possible to define service records for custom services

- A server utilizes a Service Discovery DataBase (**SDDB**) to store service records
- When a server receives a service search request, it searches the SDDB for a service record with the specified attributes
- The Bluetooth standard does not specify how the SDDB, as well as the operations on the SDDB, should be implemented

1. Creating a service record describing the service offered by an app.
2. Adding the service record to the SDDB
3. Registering the security measures associated with the service
4. Accepting connections from clients requesting the offered service
5. Updating the SDDB if the service changes
6. Removing or disabling the service record in the SDDB

- The JSR-82 API is the first standard API for Bluetooth service registration and service discovery
- The application programmer need not know how (operations on) service records are implemented
- The API provides a *nonblocking* way to retrieve all service records that meet a specific set of requirements on a remote device
- When a client searches for a service, it provides a set of UUIDs, each uniquely defining an attribute of a service

## Comment on `DiscoveryAgent.selectService()`

- The `selectService()` provides a simple way for a `client` app. to perform both device and service discovery (see example in Part 7)
- The method takes a single UUID that defines the requested service on a remote device, and returns a connection string used by `Connector.open()` to connect to the remote device
  - `Connector.open()` is a *blocking* method. It can take more than 10 seconds to connect in some situations
  - `Connector.open()` should run in a separate thread

- If the parameter `url` starts with `btsp://localhost`, then a [server](#) app. can call `Connector.open(String url)` to create a service record automatically
- The first time an `acceptAndOpen()` message is sent to the notifier, a copy of the service record is stored in the SDDB
  - once the service record is in the SDDB, a [client](#) app. can discover the record
- When a `close()` message is sent to the notifier, the service record is removed from the SDDB or disabled

# Automatic Service Record Generation

---

- Note that service records are automatically created for server applications by the JSR-82 API implementation
- These automatically generated records are often sufficient, and a server application does not need to take any other action

**Remark:** Only the automatically generated service records are used in this course. We will, however, indicate how to modify these records

## More on Default Service Records

---

- A service record contains (*attribute ID, attribute value*) pairs. Each pair describes one attribute of the service
  - the attribute ID is an integer between 0 and 65535. The most common IDs are defined by the “Bluetooth Assigned Numbers” document
  - the attribute value is a **DataElement** of one of the following types:
    - null, integer, unsigned integer, URL, UUID, string, boolean, DATALT (DATA element ALTERNative), DATSEQ (DATA element SEQUENCE)

- We'll consider the service record automatically generated from the statement

```
Connector.open("btspp://localhost:68EE141812D211D78EE" +  
"D00B0D03D76EC; name=SPPEX");
```

- The JSR-82 implementation uses the template in the Serial Port Profile to create the following service record

# Service Record Example

---

The service record contains four (*attribute ID, attribute value*) pairs. We'll only study three. Here is the first pair:

ServiceClassIDList< 0x0001 >      —attribute ID

DataElement(type=DATSEQ      —attribute value

    DataElement(type=UUID,

        UUID(68EE141812D211D78EED00B0D03D76EC)

        —from the connection string)

DataElement(type=UUID,

    UUID(SerialPort< 0x1101 >)))

    —server channel identifier)

# Explanation

---

KJhole.com

- ServiceClassIDList—defines the service classes describing the service. The service classes are defined by the Bluetooth SIG
- UUID—represents one of the ServiceClasses in ServiceClassIDList
- Note that the two DataElements defining the two service classes are wrapped in another DataElement of the type DATSEQ

- The example contains both a 16-bit UUID and a 128-bit UUID
- A Bluetooth UUID always represents a 128-bit value. A 16-bit UUID is converted to a 128-bit UUID using the formula:

$$\text{UUID}_{128} = (\text{UUID}_{16} * 2^{96}) \\ + 0x00000000000001000800000805F9B34FB$$

- The short UUIDs are defined by the “Bluetooth Assigned Numbers” document

## Service Record Example (Cont.)

---

KJhole.com

ServiceName< 0x0100 >      —attribute ID

DataElement(type=STRING,      —attribute value  
    "SPPex"      —from "name=SPPex" in the connection string)

ServiceRecordHandle< 0x0000 >      —attribute ID  
    DataElement(type=U\_INT\_4,      —attribute value  
        12345      —value assigned by the SDP server)

# The ServiceRecordHandle Attribute

---

- A 32-bit number uniquely identifying a service record within a server. Contained in all service records
- Internal bookkeeping, irrelevant to JSR-82
- JSR-82 apps. may not modify attribute

# Modification to Service Records

---

KJhole.com

- The `javax.bluetooth.LocalDevice` class contains a `getRecord()` method that a `server` app. can use to obtain its `ServiceRecord`
- The server app. can modify the `ServiceRecord` object using `ServiceRecord.setAttributeValue()`
- The changes to a `ServiceRecord` should be carried out before calling `acceptAndOpen()` for the first time
- The method `LocalDevice.updateRecord(serviceRecord)` can be used to modify records already in the SDDB

- As outlined in Part 8, a client discovering a device can consult the `DeviceClass` to determine what kind of device was found
- The `DeviceClass` also indicates the major service classes offered by the discovered device
- Consequently, there are two different ways a server app. can describe the offered services:
  - by adding a service record to the SDDB
  - by activating major service class bits in the `DeviceClass`

# Programming Examples

---

KJhole.com

- During the rest of the lecture we will study two programming examples:
  1. showing how a server app. registers a service in the SDDB (no modification of the service record), and
  2. outlining how a client app. discovers a service

- The following `server` app displays the connection string (example: `btsp://0080375a0000:1`) that the clients must use to connect to this server
  - server channel identifier is assigned by the JSR-82 implementation
- It also displays the service record's attribute IDs as hex numbers
- The app implements the `Runnable` interface to create a separate thread

# DefaultBtsppRecordMIDlet

---

KJhole.com

```
import java.util.*;
import javax.microedition.io.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;

public class DefaultBtsppRecordMIDlet // Code from Thompson et al.
    extends MIDlet implements Runnable, CommandListener {

    StreamConnectionNotifier notifier; // Server side socket connection
```

```
// DefaultBtsppRecordServer instance provides the server capabilities
DefaultBtsppRecordServer server;    //To be defined

// The form displayed to the user
private Form output;

// Creates a DefaultBtsppRecordMIDlet object and
// starts the server thread
public DefaultBtsppRecordMIDlet () {
    output = new Form("Default Record");
    output.addCommand(new Command("Exit",Command.EXIT, 1));
    output.setCommandListener(this);
    new Thread(this).start();        // Calls run(), bad coding practice?
}
```

```
// Called each time the MIDlet is started. This
// method sets the current display
public void startApp () throws MIDletStateChangeException {
    Display currentDisplay = Display.getDisplay(this); // this MIDlet
    currentDisplay.setCurrent(output);
}

public void pauseApp () {}

public void destroyApp (boolean unconditional) {}
```

```
// The server will wait and accept connections from clients
public void run() {
    LocalDevice theRadio;
    // Define the serial port service and create the notifier
    try {
        theRadio = LocalDevice.getLocalDevice();
        server = new DefaultBtsppRecordServer(); // To be defined
        server.askToBeGeneralDiscoverable(theRadio);
        // Create service record
        notifier = server.defineDefaultBtsppService();
    } catch (IOException e) { output.append("Unable to start server"
        + " (IOException: " + e.getMessage() + ")");
        return;
    }
}
```

```
if (notifier != null) {
    ServiceRecord record = theRadio.getRecord(notifier);
    // Print description of service record
    output.append("URL=" + server.getURL(record));
    output.append(server.describeAttributes(record));
} else {
    output.append("Unable to start server");
    return;
}
// Use the notifier to establish serial port connections
server.acceptClientConnections(notifier);
}
```

```
// Called each time a command occurs. The only command is
// the Exit command. This method will destroy the MIDlet.
public void commandAction(Command c, Displayable d) {
    try {
        server.shutdown(notifier);
    } catch (Exception e) { }
    notifyDestroyed();
}
}
```

```
public class DefaultBtsppRecordServer {
    boolean stop = false;

    public StreamConnectionNotifier defineDefaultBtsppService () {
        StreamConnectionNotifier notifier;
        String connString = "btspp://localhost:" +
            "68EE141812D211D78EED00B0D03D76EC;name=SPPEX";
        try { // Create a default service record
            notifier = (StreamConnectionNotifier) Connector.open(connString);
        } catch (IOException e){ return null; }
        return notifier;
    }
}
```

```
public void acceptClientConnections (StreamConnectionNotifier notifier)
    if (notifier == null){ return; }
    try {
        while (!stop){
            StreamConnection clientConn = null;
            /*
            * acceptAndOpen() waits for the next client to connect to this
            * service. The first time through the loop, acceptAndOpen()
            * adds the service record to the SDDB and updates the
            * service class bits of the device.
            */
```

```
try {    // Blocks!
    clientConn = (StreamConnection) notifier.acceptAndOpen();
} catch (ServiceRegistrationException e1) {
} catch (IOException e) { continue; }
}
} finally {    // Always executed
    try {
        shutdown(notifier);
    } catch (IOException ignore) { }
}
}
```

```

void askToBeGeneralDiscoverable (LocalDevice dev) {
    try {
        // Request that the device be made discoverable
        dev.setDiscoverable(DiscoveryAgent.GIAC);
    } catch(BluetoothStateException ignore) {
        // Discoverable is not an absolute requirement }
    }
    // Return connection string for clients
public String getUrl (ServiceRecord record) {
    String url = record.getConnectionURL(
        ServiceRecord.NOAUTHENTICATE_NOENCRYPT,false);
    if (url != null) return url.substring(0, url.indexOf(";"));
    else { return "getConnectionURL()=null";
}
}

```

```
public String describeAttributes (ServiceRecord record) {
    int[] attributeIDs = record.getAttributeIDs();
    StringBuffer strBuf = new StringBuffer(100);
    strBuf.append("\n").append(Integer.toString(attributeIDs.length));
    strBuf.append(" Attributes: ");
    for (int i = 0; i < attributeIDs.length; i++) {
        strBuf.append("<0x");
        strBuf.append(Integer.toHexString(attributeIDs[i]));
        strBuf.append(">\n");
    }
    return strBuf.toString();
}
}
```

# Device and Service Discovery MIDlet

---

- The DiscoveryMIDlet searches for the Bluetooth game service:
  1. *The MIDlet first finds a set of devices and presents the user with a list (see following code)*
  2. The user selects a device from the list
  3. The MIDlet then searches the selected device for the UUID defining the Bluetooth game service
  4. The MIDlet displays all services using the UUID

```
import java.util.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;

// Code from Thompson et al.
public class DiscoveryMIDlet extends BluetoothMIDlet
    implements DiscoveryListener {

    private List deviceList;    // List of remote devices on the screen
    private DiscoveryAgent agent;    // for local device
    private Vector deviceVector;    // RemoteDevice objects
```

```
private boolean isInInquiry;    // currently occurring
private List serviceRecordList; // service records found
private Vector serviceRecordVector;
    //The service records that were found in the last service search

public void startApp () throws MIDletStateChangeException {
    isInInquiry = false;

    // Create a new List of choices and set it to the current displayable
    deviceList = new List("List of Devices", List.IMPLICIT);
    deviceList.addCommand(new Command("Exit",Command.EXIT, 1));
    deviceList.setCommandListener(this);
    Display.getDisplay(this).setCurrent(deviceList);
}
```

```
// Retrieve the DiscoveryAgent object.  If retrieving the
// local device causes a BluetoothStateException,
// something is wrong so stop the app from running.
```

```
try {
    LocalDevice local = LocalDevice.getLocalDevice();
    agent = local.getDiscoveryAgent();
} catch (BluetoothStateException e) {
    throw new MIDletStateChangeException(
        "Unable to retrieve local Bluetooth device.");
}
```

```
deviceVector = new Vector();    // for remote devices
addDevices();                  // pre-known and cached devices (not shown)
```

```

try {
    agent.startInquiry(DiscoveryAgent.GIAC, this); // device search
} catch (BluetoothStateException e) {
    throw new MIDletStateChangeException(
        "Unable to start the inquiry"); }
isInInquiry = true;
}

// Required by the DiscoveryListener interface for device search
public void deviceDiscovered (RemoteDevice device, DeviceClass cod) {
    String address = device.getBluetoothAddress();
    deviceList.insert(0, address + "-I", null);
    deviceVector.insertElementAt(device, 0);
}

```

```
// Required by the DiscoveryListener interface for device search
public void inquiryCompleted(int type) {
    isInInquiry = false;
    Alert dialog = null;
    // Determine if an error occurred.  If one did occur display
    // an Alert before allowing the application to exit.
    if (type != DiscoveryListener.INQUIRY_COMPLETED) {
        // If the device inquiry was terminated, then the user must have
        // selected a Remote Device to perform a service search on so
        // start the service search.
        if (type == DiscoveryListener.INQUIRY_TERMINATED) {
            startServiceSearch();
            return;
        }
    }
}
```

```
    } else {  
        dialog = new Alert("Bluetooth Error","The inquiry failed",  
            null,AlertType.ERROR);  
    }  
} else {  
    dialog = new Alert("Inquiry Completed","The inquiry completed",  
        null, AlertType.INFO);  
}  
dialog.setTimeout(Alert.FOREVER);  
Display.getDisplay(this).setCurrent(dialog);  
}  
}
```

- The current version of the `DiscoveryMIDlet` can carry out a device search and keep track of the `RemoteDevice` objects found
- The following methods carry out the service search
  - the search starts when the user selects one of the devices listed on the screen
  - because many Bluetooth devices cannot start service searches during inquiry, an active inquiry is canceled before the start of the search (*not shown*)

- a. The local device sends a list of UUIDs to search for a service on a remote device
- b. The remote device checks all service records in the SDDB for all of the received UUIDs
- c. For every service record that has all the UUIDs, the remote device sends back the ServiceRecordHandle and the requested attributes for that service record

```
/**
 * Called when a Command is selected.  If it is an Exit Command,
 * then the MIDlet will be destroyed.
 *
 * @param c the Command that was selected
 * @param d the Displayable that was active when the Command
 *     was selected
 */
public void commandAction (Command c, Displayable d) {
    ...    // details not shown
}
```

```
// Starts the service search
private void startServiceSearch () {
    serviceRecordVector = new Vector();
    try {
        // Search for the Bluetooth Game service record
        // and retrieve the name attribute in addition to
        // the default attributes.
        UUID[] uuidList = new UUID[1];
        uuidList[0] = new UUID(
            "0FA1A7AC16A211D7854400B0D03D76EC", false);
        int[] attrList = new int[1];
        attrList[0] = 0x100;    // ServiceName attribute ID
    }
}
```

```

// The RemoteDevices are in the deviceVector in the same
// order as they are on the screen so getting the index
// allows us to retrieve the correct RemoteDevice object.
int index = deviceList.getSelectedIndex();
RemoteDevice d = (RemoteDevice)deviceVector.elementAt(index);
int id = agent.searchServices(attrList, uuidList, d,this);
} catch (BluetoothStateException e) {
    Alert error = new Alert("Error",
        "Unable to start the service search (" +
        e.getMessage() + ")", null, AlertType.ERROR);
    error.setTimeout(Alert.FOREVER);
    Display.getDisplay(this).setCurrent(error, deviceList);
}
}

```

```
/**
 * Called each time a service is discovered. Retrieve the name
 * attribute from the service record and display it on the screen.
 * Add the service record to the service record Vector for later.
 *
 * @param transID the transaction ID
 * @param record the service records that were found
 */
public void servicesDiscovered(int transID,ServiceRecord[] record) {

    // Process each service record individually
    for (int i = 0; i < record.length; i++) {
```

```
/Retrieve the name attribute from the service record
DataElement nameElement =
    (DataElement)record[i].getAttributeValue(0x100);

// The name attribute is only valid if it exists and is a string
// If either of these conditions fail, move on to the next
// service record.
if ((nameElement != null) &&
    (nameElement.getDataType() == DataElement.STRING)) {

    // Retrieve the name and display it on the screen.
    String name = (String)nameElement.getValue();
    serviceRecordList.insert(0, name, null);
```

```
        serviceRecordVector.insertElementAt(record[i], 0);
    }
}
}
/**
 * Called when the service search has ended. Displays a
 * message to the user that the service search completed
 * and specifies if the search completed normally.
 *
 * @param transID the transaction ID
 * @param type specifies how the service search completed
 */
public void serviceSearchCompleted (int transID, int type) {
```

```
Alert dialog = null;

// Determine if an error occurred.  If one did occur display
// an Alert before allowing the application to exit.
if (type !=
    DiscoveryListener.SERVICE_SEARCH_COMPLETED) {
    dialog = new Alert("Bluetooth Error",
        "The service search failed to complete normally",null,
        AlertType.ERROR);
} else {
    dialog = new Alert("Service Search Completed",
        "The service search completed normally", null,
        AlertType.INFO);
}
```

```
    dialog.setTimeout(Alert.FOREVER);  
    Display.getDisplay(this).setCurrent(dialog);  
}  
}
```