Bluetooth

Part 7: RFCOMM

Kjell Jørgen Hole
UiB

Last updated 15.03.09
Mail: Kjell.Hole@ii.uib.no
URL: www.kjhole.com

Outline

- RFCOMM definition
- RFCOMM devices and frame types
- Connecting and disconnecting
- Java RFCOMM API
- Introduction to device and service discovery

RFCOMM Definition (1)

- The name RFCOMM comes from a Radio Frequency (RF)-oriented emulation of the serial COM ports on a PC
- RFCOMM emulates full 9-pin RS-232 (EIATIA-232-E) serial communication over an L2CAP channel
- RFCOMM is based on a subset of the ETSI TS 07.10 standard for software emulation of the RS-232 hardware interface. TS 07.10 is used by GSM cellular phones to multiplex several streams of data onto one physical serial cable
- It can be used to connect to legacy applications

RFCOMM Definition (2)

- RFCOMM provides multiple concurrent connections by relying on L2CAP to handle multiplexing over a single connection
- RFCOMM supports flow control on individual channels
- No error control, assumed that L2CAP provides an error free channel
- The *Serial Port Profile* (**SPP**) defines how an RFCOMM connection should be established between two devices

GAP and SPP

- The Generic Access Profile (GAP) is the most basic Bluetooth profile. All other profiles, including SPP, are built upon GAP
  - the purpose of GAP is to make sure that all devices can successfully establish a baseband link
- Other profiles are built upon SPP: Dial Up Networking (DUN), FAX, Headset, Hands-free, SIM access, Generic object exchange

Types of RFCOMM Devices

- **Type 1**—Internal emulated serial port. An emulation entity is used to map a system specific communication interface (API) to the RFCOMM services, i.e., a type 1 device supports an application on top of RFCOMM
- **Type 2**—Intermediate device with physical serial port. A port proxy entity relays data from RFCOMM to an external RS-232 interface linked to another device

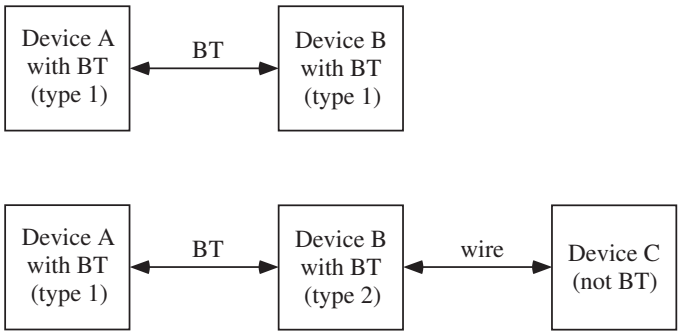

Figure 7-1 RFCOMM direct connect and RFCOMM used with legacy COMM device

RFCOMM Frames

- RFCOMM is based on GSM TS 07.10 and use TS 07.10 frames to communicate
  - not all features of TS 07.10 are adapted by RFCOMM
- The RFCOMM frames become the data payload in L2CAP packets

RFCOMM Frame Types (1)

- SABM—Set Asynchronous Balanced Mode (startup command)
- UA—Unnumbered Acknowledgement (response when connected)
- DISC—Disconnect (disconnect command)
- DM—Disconnected Mode (response to a command when disconnected)
- UIH—Unnumbered Information with Header check

RFCOMM Frame Types (2)

- SABM, UA, DM, and DISC are control frames
- each RFCOMM channel has a Data Link Connection Identifier (DLCI). UIH frames on DLCI $= 0$ are used for control messages, while UIH frames on DLCI $\neq 0$ are used data

Connecting (1)

- An L2CAP connection must be set up before an RCOMM connection can be set up
- RFCOMM has a reserved Protocol and Service Multiplexer (PSM) value used by L2CAP to identify RFCOMM traffic (PSM=0x0003)
  - the PSM is similar in function to the port number on an IP network
  - the PSM is used by a client to connect to a server

Connecting (2)

- Connection procedure:
  1. Initiating device sends SABM (Set Asynchronous Balanced Mode) frame over L2CAP channel
  2. Responding device goes to Asynchronous Balanced Mode (ABM) and sends back an UA (Unnumbered Acknowledgement) frame
  3. Initiator sends a Parameter Negotiation (PN) command and responding device answers with a PN response frame

Disconnecting

- A device not willing to connect sends back a DM (Disconnected Mode) frame when it receives the initial SABM frame
- RFCOMM has a 60s timer which is started when a command is sent. If an acknowledgement isn't received when the timer elapses, the connection will be shut down
- if RFCOMM times out, it must send a DISC (Disconnect) command frame on the same DLCI as the original SABM frame

Java ME

Why Use the RFCOMM API?

- Communicating with a remote device using the RFCOMM API is similar to communicating over a socket connection, i.e., data is sent between devices via *streams*
- The RFCOMM API is simpler to use than the packet-oriented L2CAP API
- The API supports multiple Bluetooth connections over a single Bluetooth link
- Supports different levels of security (to be discussed in Part 10)

RFCOMM API

- No new methods or classes are defined for the RFCOMM API. It utilizes existing classes and interfaces from the GCF (Generic Connection Framework) in CLDC
- All RFCOMM connections are initiated with `javax.microedition.io.Connector.open(String s)` where `s` is a valid URL of the form {scheme}:{target}{params}
  - {scheme} is the "btspp" protocol for both Slave and Master
  - {target} is the network address starting with "//"
  - {params} are formed as a series of equates of the form ";x=y"

Parameters for RFCOMM Connection Strings

Name	Description	Values	Client or Server
master	this device must be the Master	true, false	both
authenticate	remote device must be authenticated	true, false	both
encrypt	link must be encrypted	true, false	both
authorize	all connections to this device must receive authorization	true, false	server
name	ServiceName attribute in service record	any valid string	server

- All parameters in the table are optional, i.e., they need not be included in the connection string
- All other parameter values will cause an `IllegalArgumentException` to be thrown by `Connector.open()`

Why Master?

- Usually, the device initiating a connection, i.e. the client, becomes the Master
- Sometimes, a particular device is configured to be the Master because
  - it is required by a certain profile, or
  - the device must be able to form a piconet

Security Parameters

- The security parameters "authenticate," "encrypt," and "authorize" do not have to be set
- A security parameter that is not set has the value false unless another parameter requires this parameter to be true
  - if "encrypt=true" and "authenticate" is not part of the connection string, then "authenticate" is set to true because encryption requires authentication
  - "authenticate=false" and "authorize=true" is an invalid combination

Client and Server Connections

- For a client connection, the method `Connector.open()` returns a `StreamConnection` object (interface in `javax.microedition.io`)
- `Connector.open()` returns a `StreamConnectionNotifier` object for a server connection (interface in `javax.microedition.io`)
  - the `acceptAndOpen()` method must be called after `Connector.open()`
  - it blocks until a client connects to the server, then the method returns a `StreamConnection` object
- A client and a server communicate via `InputStream`s and `OutputStream`s returned by a `StreamConnection` object

See Java ME code examples `HelloClient` and `HelloServer` in Part 4

BluetoothConnectionException

- A `BluetoothConnectionException` is thrown if
  - an invalid combination of security parameters is passed to `Connector.open()`
  - an authentication, encryption, or authorization request fails during establishment of a connection
- Extends `java.io.IOException`

More on `BluetoothConnectionException`

- When a server makes the request to `Connector.open()` to retrieve a notifier object, it can set "master=true" in the connection string to request that a client using this service is the Slave

- If the client device initiating a connection to the server device does not support a Master/Slave change, then an exception is thrown

Server Connections

- The {target} for server connections is "//localhost" followed by a colon and the Universally Unique IDentifier (UUID), 128-bit unsigned integer, for the service to add to the service record. See examples below

1. UUID is 0x10203040506070809OA1B1C1D1E10073 and device can be either Master or Slave:

   "btspp://localhost:10203040506070809OA1B1C1D1E10073; name=Print_Server; master=false"

More Server Connection Strings

2. All communication to the server must be authenticated and authorized:

   "btspp://localhost:1231242432434A4AA3B056104AC0CD5F; authenticate=true; authorize=true; name=Echo"

3. The server must be the Master of the link and the link must be authenticated, encrypted, and authorized:

   "btspp://localhost:AB93248543812312312312ADEFE49A02F; encrypt=true; authorize=true; master=true"

Server Connection Example (1)

KJhole.*com*

```
// Create a notifier object
StreamConnectionNotifier notifier = (StreamConnectionNotifier)
  Connector.open("btspp://localhost:123456789ABCDE;name=Echo_Server");

// Create I/O streams
StreamConnection conn = notifier.acceptAndOpen();     // Blocks!
OutputStream output = conn.openOutputStream();
InputStream input = conn.openInputStream();
```

Server Connection Example (2)


```
    ⋮
// Close the streams and the connection
output.close()
input.close()
conn.close()
```


How to Generate UUIDs

- The Linux `uuidgen -t` command generates a 128-bit UUID based on
  - 80 bits obtained from the current system time
  - 48-bit hardware address of NIC
- The command will generate UUIDS on the form feb7b8d9-c9a0-11d8-a536-000a95a4ddae
- The hypens "-" must be deleted before the UUID can be used by JSR-82

Client Connections

- {scheme} is still given by the "btspp" protocol
- {target} starts with "//" followed by the Bluetooth address of the server device and the server channel identifier
  - The server channel identifier is a number between 0 and 31 assigned by the JSR-82 implementation. The number is similar to a TCP port and identifies a service on a device
- {params} are "master," "authenticate," and "encrypt"

Examples of Client Connection Strings

1. Valid client connection string that connects to a Server device with Bluetooth address 008003DD8901, using server channel identifier 1:

   "btspp://008003DD8901:1;authenticate=true"

2. String requiring that the local device is the Master and using server channel identifier 5:

   "btspp://008012973FAE:5;master=true;encrypt=true"

More on Client Connections

- When the connection string is passed to `Connector.open()`, an attempt is made to establish a connection to the server device
  - unlike for a server connection, this connection is established once `Connector.open()` returns

Client Connection Example (1)


```
String connString = "btspp://008012973FAE:5;master=true";

// Establish a connection to the server
StreamConnection conn = (StreamConnection)
                    Connector.open(connString);

// Create I/O streams
InputStream input = conn.openInputStream();
OutputStream output = conn.openOutputStream();
    ⋮
```

Client Connection Example (2)


```
    ⋮
// Close the streams and the connection
input.close()
output.close()
conn.close()
```

Next: The remainder of this lecture discusses classes used by a *client* to carry out a *combined* device and service discovery (first discussed in Part 4). It is also shown how to set some security parameters. We will return to these topics in later lectures

javax.bluetooth.LocalDevice

- The `javax.bluetooth.LocalDevice` class defines the basic functions of the Bluetooth manager
- The Bluetooth manager provides access to and control of the local Bluetooth device

```
1  LocalDevice local = LocalDevice.getLocalDevice();
```

- Note that there is no public constructor, multiple calls to `getLocalDevice()` will return the same object

javax.bluetooth.DiscoveryAgent

- The `javax.bluetooth.DiscoveryAgent` class provides methods to perform device and service discovery
- A local device must have only one `DiscoveryAgent` object

```
2  DiscoveryAgent agent = local.getDiscoveryAgent();
```

- Observe that the `DiscoveryAgent` must be retrieved by a call to `LocalDevice.getDiscoveryAgent()`

DiscoveryAgent.selectService()

- Attempts to locate a service that contains `uuid` in the ServiceClassIDList of its service record

```
String selectService(UUID uuid, int security, boolean master)
throws BluetoothStateException
```

- The method returns a string that may be used in `Connector.open()` to establish a connection to the service
- The `ServiceRecord` interface and the `javax.bluetooth.UUID` class may be used to specify the `selectService()` parameters

ServiceRecord Field Summary

The `ServiceRecord` interface describes characteristics of a Bluetooth service. The following constants are defined:

`static int AUTHENTICATE_ENCRYPT` Authentication and encryption are required for connections to this service

`static int AUTHENTICATE_NOENCRYPT` Authentication is required for connections to this service, but not encryption

`static int NOAUTHENTICATE_NOENCRYPT` Authentication and encryption are not needed on a connection to this service

javax.bluetooth.UUID

- The `UUID` class defines *universally unique identifiers*. These 128-bit unsigned integers are guaranteed to be unique across all time and space
- Constructor `UUID(String uuidValue, booleanshortUUID)` creates a UUID object from the string provided
  - If `shortUUID` is `true`, `uuidValue` represents a 16-bit or 32-bit UUID

javax.bluetooth.DiscoveryAgent.selectService()

- The `selectService()` method does a service search on a set of remote devices
- How the service is selected if there are multiple services with `uuid` and which devices to search is implementation dependent

```
3   String connString = agent.selectService(
          new UUID("86b4d249fb8844d6a756ec265dd1f6a3",false),
          ServiceRecord.NOAUTHENTICATE_NOENCRYPT,false);
```


Complete Example


```
1   LocalDevice local = LocalDevice.getLocalDevice();
2   DiscoveryAgent agent = local.getDiscoveryAgent();
3   String connString = agent.selectService(
      new UUID("86b4d249fb8844d6a756ec265dd1f6a3",false),
      ServiceRecord.NOAUTHENTICATE_NOENCRYPT,false);
4   if (connString != null) {
5     try {
6       StreamConnection conn = (StreamConnection)
            Connector.open(connString);
                    ⋮
```

Summary—RFCOMM

- RFCOMM provides serial port emulation for legacy applications and Bluetooth profiles
- RFCOMM supports end devices with applications on top and intermediate devices with physical RS-232 serial ports on top
- RCFOMM utilizes an L2CAP connection
- RFCOMM frames are sent in the payload field of L2CAP packets

Summary—API

- The JSR-82 API for RFCOMM is the most used API within JSR-82 because
  - RFCOMM provides serial two-way communication
  - the API reuses familiar APIs from Java ME
- To make it easier to start writing MIDlets, we presented a simple technique for carrying out a *combined* device and service discovery