Multiplayer Games
### Multiplayer Game Types

<table>
<thead>
<tr>
<th>Real time</th>
<th>Same Place</th>
<th>Bluetooth</th>
<th>Internet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turn based</td>
<td>Pass and play</td>
<td>Messaging</td>
<td></td>
</tr>
</tbody>
</table>
Turn Based Games

- Turn based multiplayer between players in the same place
- One mobile phone, passed back and forth between turns, great for board games (chess, checkers, blackjack etc)
- Easy to develop. (Easier than single player, where the opponent is controlled by AI)
Internet

- **Carriers** (Singtel, Starhub, M1, etc) act as a service provider to the Internet for mobile phones with data access in their service plan
- Speeds vary based on network technology (GSM, CDMA, GPRS, UMTS, ...)
- Connection times and latencies can be high
- Connections dropped quickly after short periods of inactivity to free bandwidth
- All mobile phones and carriers support at least **HyperText Transport Protocol**
  - Connectionless behavior of the HTTP protocol well suited for the unreliability of the network
  - Standard web application programming techniques can be used to implement the server
- Newer handsets can maintain **direct socket connections (TCP)**, with security permission
Internet

- Low latency, real time multiplayer still not practical on a mass market scale
- Internet servers required for games with more than two players, like poker
- Internet servers maintain community high scores, downloadable content
Networking using Generic Connection Framework

★ All the classes including the common Connector class defined in the CLDC specification for networking APIs forms the Generic Connection Framework (GCF).

★ The common Connector class of the GCF can be used to create any type of connection.

★ The type of connection is determined by the protocol string in the URI parameter passed to the open() method of the Connector class.

★ Package: javax.microedition.io
## Generic Connection Framework (GCF)

<table>
<thead>
<tr>
<th>Connection Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.anuflora.com">http://www.anuflora.com</a></td>
<td>for HTTP connection.</td>
</tr>
<tr>
<td>socket://localhost:8000</td>
<td>for connecting to a Socket.</td>
</tr>
<tr>
<td>serversocket://:8001</td>
<td>for connecting to a Server Socket.</td>
</tr>
<tr>
<td>btspp://008003DD8901:1; authenticate=true</td>
<td>for Bluetooth serial port protocol client connection.</td>
</tr>
</tbody>
</table>

- .......
The Generic Connection Framework (GCF) defines

- One Generic class: `Connector`
- One Exception: `ConnectionNotFoundException`
- Eight Interfaces:
  - `Connection`, `ContentConnection`, `Datagram`, `DatagramConnection`, `InputConnection`, `OutputConnection`, `StreamConnection`, `StreamConnectionNotifier`
The Connector class

- The Connector class is a ‘factory’ for creating new Connection objects. The static methods of Connector class return an instance of the Connection interface or one of its descendents.

- Methods
  - `open(String name)`
  - `open(String name, int mode)`
  - `open(String name, int mode, Boolean timeouts)`

- Eg.
  
  ```java
  Connector.open(“socket://127.0.0.1:8080”);
  ```

- Modes
  - READ - read only
  - WRITE - write only
  - READ_WRITE - read and write

- Parameter ‘timeouts’: Indicates whether or not the connection should throw an `InterruptedIOException` is a timeout occurs.
The Connector class (other methods)

- **openInputStream**(String name)
- **openOutputStream**(String name)
- **openDataInputStream**(String name)
- **openDataOutputStream**(String name)

- ‘name’ - URI parameter. The type of connection is determined by the *protocol* string in the URI parameter.

- **Eg.**
  
  ```java
  OutputStream os = Connector.openOutputStream("socket://127.0.0.1:8080");
  ```

**Note:** *The connections must be executed in a separate Thread.*
GCF Interfaces

```
Connection
  "InputConnection"
  "OutputConnection"
  "DatagramConnection"
  "StreamConnection"
  "StreamConnectionNotifier"
  "ContentConnection"
  "HttpConnection"
```

(MIDP 1.0 extension)
GCF Interfaces

- The InputConnection [input stream only]
- The OutputConnection [output stream only]
- The StreamConnection [input and output stream]
- Server socket: StreamConnectionNotifier
- The ContentConnection [input and output stream with content type, content length, content encoding]
- The HttpConnection [input and output stream with most of the http specific methods, Defined in MIDP 1.0]

Note: No TCP Socket, UDP Datagram support in MIDP 1.0
InputConnection interface

🌟 The `InputConnection` interface represents a connection’s stream data as an `InputStream`, that is, a stream of byte-oriented data.

🌟 The `InputConnection` methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>openDataInputStream()</code></td>
<td>Opens and returns a data input stream for a connection.</td>
</tr>
<tr>
<td><code>openInputStream()</code></td>
<td>Opens and returns an input stream for a connection.</td>
</tr>
</tbody>
</table>

🌟 These methods return either an `InputStream` object or `DataInputStream` object.

🌟 Tables 6.4 and 6.5 describe the methods of `InputStream` and `DataInputStream` objects to read data.
class readWeb implements Runnable {
    //Runnable class
    public void run() {
        InputStream is = null;
        String b = new String();
        try {
            InputConnection inc = Connector.open("http://books.anuflora.com");
            is = inc.openInputStream();
            int ch;
            while((ch = is.read()) != -1){
                b.append((char)ch);
            }
            strItm.setText(new String(b));
        } catch (IOException e) {
        } finally {
            if (is != null) try { is.close(); } catch (Exception e) {}
            if (inc != null) try { inc.close(); } catch (Exception e) {}
        }
    }
    readWeb r = new readWeb(); //Running in new Thread
    new Thread(r).start();
}
OutputConnection Interface

* The *OutputConnection* interface is another subinterface of *Connection*. The *OutputConnection* interface represents a connection’s stream data as an *OutputStream*.

* OutputConnection methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>openDataOutputStream()</td>
<td>Opens and returns a data output stream for a connection.</td>
</tr>
<tr>
<td>openOutputStream()</td>
<td>Opens and returns an output stream for a connection.</td>
</tr>
</tbody>
</table>

* Tables 6.7 and 6.8 describe the methods of *OutputStream* and *DataOutputStream* objects to write data. (attached)
StreamConnection

StringBuffer b=new StringBuffer();
try {
    stc = (StreamConnection)
        Connector.open("http://www.anuflora.com/index.html");
    is = stc.openInputStream();
    int ch;
    while((ch = is.read())!= -1){
        b.append((char)ch);
    }
} ....

Can both READ and WRITE.
StreamConnectionNotifier

★ Represents the server socket.
★ The StreamConnectionNotifier defines only one method, which returns a StreamConnection interface representing the client.
★ `acceptAndOpen()`
  - Returns a StreamConnection that represents a server side socket connection.
Content Connection Interface

- *ContentConnection* knows how to extract encoding, length and content type of the data received.

```
try {
    cc = (ContentConnection) Connector.open("http://localhost/anuflora/index.htm");
    stream = cc.openDataInputStream();
    byte[] buffer = new byte[1000];
    stream.readFully(buffer);
    strItm.setText("Length: " + cc.getLength() + 
                   " Encoding: " + cc.getEncoding() + " Type: " +
                   cc.getType());
}
```

```java
}catch(IOException e){ .......
```
HttpConnection interface

* The *HttpConnection* interface adds a more complete set of HTTP handling methods including the ability to extract the host name, url, query string, port, get and set request methods (GET, HEAD, POST), response content and return codes. [MIDP 1.0 – SOAP method is not supported]

* Implementations should support HTTP 1.1
HttpConnection States

- Connection creation using `Connector.open('http://...')` method.
- Setup
- Extract data from connection object
- Closed
  - Close the connection using `conn.Close()` method
- Connected

HttpInterface methods and Error codes.
Table 6.10 to 6.14 (Attached)
```java
HttpConnection

HttpConnection c = (HttpConnection)Connector.open(url);
c.setRequestMethod(HttpConnection.POST);
c.setRequestProperty("User-Agent", "Profile/ MIDP-2.0 Configuration / CLDC-1.0");
c.setRequestProperty("Content-Language", "en-US");
os = c.openOutputStream();
os.write("LIST games\n".getBytes());

rc = c.getResponseCode();
if (rc != HttpConnection.HTTP_OK) {
    throw new IOException("HTTP response code: " + rc);
}

is = c.openInputStream();  // Get the ContentType
String type = c.getType();
int len = (int)c.getLength();
if (len > 0) {
    int actual = 0;
    int bytesRead = 0;
    byte[] data = new byte[len];
    while (((bytesRead != len) && (actual != -1)) {
        actual = is.read(data, bytesRead, len - bytesRead);
        bytesRead += actual;
    }
```

**MIDP 2.0 Extensions to GCF**

- **CommConnection**
  - This interface defines a logical serial port connection.

- **HttpsConnection**
  - This interface defines the necessary methods and constants to establish a secure network.

- **SecureConnection**
  - This interface defines the secure socket stream connection.

- **SecurityInfo**
  - This interface defines methods to access information about a secure network connection.

- **ServerSocketConnection**
  - This interface defines the server socket stream connection.

- **SocketConnection**
  - This interface defines the socket stream connection.

- **UDPDatagramConnection**
  - This interface defines a datagram connection which knows its local endpoint address.
ServerSocketConnection (Echo Server)

```java
public void run() {
    try {
        mServerSocketConnection = (ServerSocketConnection)
            Connector.open("socket://:80");
        SocketConnection sc = null;
        sc = (SocketConnection)
            mServerSocketConnection.acceptAndOpen();
        Reader in = new InputStreamReader(
            sc.openInputStream());
        PrintStream out = new PrintStream(sc.openOutputStream());
        out.print("HTTP/1.1 200 OK\r\n\r\n");
        String line;
        while ((line = readLine(in)) != null) { //Echo line by line
            out.print(line);  }
        out.close();
        in.close();
        sc.close();
    } catch (Exception ex) {......
```
Secure Networking

- An HttpsConnection is returned from Connector.open() when an “https://” connection string is accessed. A SecureConnection is returned from Connector.open() when an “ssl://” connection string is accessed. [Both provides secured networking connections (with/without Http).]

  - javax.microedition.io.HttpsConnection
  - javax.microedition.io.SecureConnection
  - javax.microedition.io.SecurityInfo
  - javax.microedition.pki.Certificate
  - javax.microedition.pki.CertificateException
Low level network API

- A SocketConnection is returned from Connector.open() when a "socket://host:port" connection string is accessed.
- A ServerSocketConnection is returned from Connector.open() when a "socket://:port" connection string is accessed.
- A UDPDatagramConnection is returned from Connector.open() when a "datagram://host:port" connection string is accessed.

- javax.microedition.io.SocketConnection
- javax.microedition.io.ServerSocketConnection
- javax.microedition.io.DatagramConnection
- javax.microedition.io.Datagram
- javax.microedition.io.UDPDatagramConnection

Question to ponder: What is push registry (javax.microedition.io.PushRegistry)? Is it useful for Games.
Multiplayer Games

⭐ Design Issue
- Network Architecture
- Effects of Latency in real-time networking games

⭐ Design Requirements
- scalability, consistency, good responsiveness, security, cheat prevention, ability to maintain player’s interest

⭐ Design Techniques
- Dead Reckoning – static state based on PDU (protocol data unit), extrapolate using velocity, extrapolate using velocity and acceleration, extrapolate based on orientation (roll, pitch and heading), extrapolate the moving parts of the entities.
- Partitioning
- Interest Filtering
**Messaging**

- Text messages can be used as a carrier of small amounts of data between phones.
- Applications do not need to be running in order to receive specially coded text messages, they will be launched when the message is viewed by the user.
- Allows direct mobile-to-mobile turn based multiplayer without a server, but 1-to-1 only!
- Access to the phone contact/address book key to make it easy to initiate communication.
Wireless Messaging API (WMA)

- Wireless Messaging API (WMA) is the first optional package defined for J2ME, which the applications can use to send and receive short text or binary messages over wireless connections.
- WMA is based on the Generic Connection Framework (GCF) defined for the Connected Limited Device Configuration.
- WMA defines a set of interfaces in the `javax.wireless.messaging` package for sending and receiving short messages through the wireless network such as Global System for Mobile Communication (GSM), Code-Division Multiple Access (CDMA), General Packet Radio Services (GPRS), etc.
Creating a Connection

- **Connector** class factory of Generic Connection Framework (GCF) is used to create a `MessageConnection` interface for sending and receiving messages.

- **Eg.**

  ```java
  // To Create a connection
  conn = (MessageConnection) Connector.open(uri);
  
  // To Close the connection
  Conn.close();
  ```

- The uri passed to the `Connector.open` method is used to identify the protocol (sms or cbs in WMA 1.1 and mms in WMA 2.0).
**URI for SMS and CBS**

- Protocol (sms or cbs)
- Phone number (for receiving messages: optional)
- Port number (for sending messages: optional, if not specified the default text messaging port will be used)

**Examples:**
- sms://+6596709800
- sms://+6596709800:5670
- sms://5670
- Cbs://5070
Sending a Message

**STEPS**

- Create a MessageConnection interface.
- Use the MessageConnection’s `newMessage()` method to create a message object.
  - `newMessage()` method will take a parameter which indicates the message type (TEXT_MESSAGE or BINARY_MESSAGE)
- Use the Message object’s
  - `setPayloadText(text)` - to set message text if TEXT_MESSAGE
  - `setPayloadData(data)` - to set data if BINARY_MESSAGE
- Use the MessageConnection’s `send()` method to send the Message. `send()` method takes a message object as a parameter.
Example: Sending Text Message

```java
public void sendText(MessageConnection conn, String text) throws IOException, InterruptedIOException {
    TextMessage txtMsg =
        conn.newMessage(conn.TEXT_MESSAGE);
    txtMsg.setPayloadText( text );
    conn.send( txtMsg );
}
```
Example: Sending Binary Data

```java
public void sendBinary(MessageConnection conn, byte[] data)
    throws IOException, InterruptedIOException {
    BinaryMessage txtMsg =
        conn.newMessage(conn.BINARY_MESSAGE);
    txtMsg.setPayloadData( data );
    conn.send( txtMsg );
}
```
Receiving a Message

★ To receive a message, open a server connection and then call the connection’s `receive()` method to receive the next available message on the specified port.

★ If no message is available,
  - the method blocks until a new message arrives,
  - or until a different thread closes the connection.
Example: Receiving a Message

MessageConnection conn = null;
String loc = "sms://5070";

try {
    conn = (MessageConnection) Connector.open(loc);
    while (true) {
        Message msg = conn.receive();
        if (msg instanceof TextMessage) {
            String text = ((TextMessage) msg).getPayloadText();
            // Display the text or do some actions
        }
    }
}
Testing the Messaging Application

★ SMS applications are best experienced with the **Over-The-Air (OTA)** provisioning mode of the J2ME Wireless Toolkit.

★ Open the SMS application in the J2ME wireless Toolkit. Build and package it (create the JAD/JAR files).

★ Choose Project menu and select **Run via OTA**

**WMA 2.0**

★ Adds support for MMS.
Bluetooth

- Bluetooth devices can broadcast their identity to be discovered by others
- Bluetooth is commonly used to emulate a direct serial cable connection
- De facto standard for low-cost and low-power short-range radio links between mobile devices, PCs, headsets, GPS receivers, peripherals and consumer electronics
- Bluetooth Special Interest Group (SIG) releases specifications.
- IEEE 802.15 WPAN
  - 2.4 Ghz ISM band, 1 Mbps (within piconet - gross)
  - Ver1: 10 meters, Ver2: 100 meters
Bluetooth

- Logically Bluetooth belongs to, connection-free token-based multi-access network
- 1 Master and up to 7 Slave
- Shared channel. Master decides which slave has access to the channel.
- “Piconet” - Slaves are synchronised to the same master.
- “Scatternet” – Independent piconets that have overlapping coverage. Time-multiplex mode to communicate with multiple piconets. (Synchronization parameters need to be changed)

Comparison with Wi-Fi
- The cost of Bluetooth chips is under $3
- Bluetooth technology costs a third of Wi-Fi to implement
- Bluetooth technology uses a fifth of the power of Wi-Fi

Compare with other wireless standards
Bluetooth

- Non-game entertainment possibilities: eg. viral social networking applications
- Other Applications: Automation industry, security industry, logistics, construction (more applications when combined with RFID (eg. IDBlue) www.baracoda.com)
- Mobile phone viruses now possible
- “Bluejacking”
  - Sending unexpected messages or files
- “Bluesnarfing”
  - Stealing data from Bluetooth devices
- Bluetooth-enabled kiosks may make retail software distribution a reality
Common Bluetooth protocols

<table>
<thead>
<tr>
<th>OBEX</th>
<th>SDP</th>
<th>TCS Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/UDP</td>
<td>IP</td>
<td></td>
</tr>
<tr>
<td>RFCOMM</td>
<td>BNEP</td>
<td></td>
</tr>
<tr>
<td>L2CAP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Host Controller Interface (HCI)

Firmware/Hardware

Baseband and link control

Bluetooth Radio
Java API for Bluetooth wireless technology (JABWT)

Packages

- javax.bluetooth
- javax.obex

Image Source: Bluetooth Application Programming with the Java APIs (book) mkp.com
Bluetooth Application Activities

Image Source: http://developers.sun.com/, by C. Enrique Ortiz,
© Bhojan ANAND. SoC, NUS
Service Discovery Database (SDDB)

- Database of registered services

Service Records

<table>
<thead>
<tr>
<th>AttributeID</th>
<th>Attrib.Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x1101</td>
</tr>
<tr>
<td>0x0001</td>
<td></td>
</tr>
</tbody>
</table>

Eg. ProtocolDescriptorList attribute
ServiceClassIDList attribute
ServiceName attribute

Attribute List: https://www.bluetooth.org/foundry/assignnumb/document/service_discovery
## Frequently used service record attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute ID</th>
<th>Attribute Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceRecordHandle</td>
<td>0x0000</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td>ServiceClassIDList</td>
<td>0x0001</td>
<td>DATSEQ of UUIDs</td>
</tr>
<tr>
<td>ServiceRecordState</td>
<td>0x0002</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td>ServiceID</td>
<td>0x0003</td>
<td>UUID</td>
</tr>
<tr>
<td>ProtocolDescriptorList</td>
<td>0x0004</td>
<td>DATSEQ of DATSEQ of UUID and optional parameters</td>
</tr>
<tr>
<td>BrowseGroupList</td>
<td>0x0005</td>
<td>DATSEQ of UUIDs</td>
</tr>
<tr>
<td>LanguageBasedAttributeIDList</td>
<td>0x0006</td>
<td>DATSEQ of DATSEQ triples</td>
</tr>
<tr>
<td>ServiceInfoTimeToLive</td>
<td>0x0007</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td>ServiceAvailability</td>
<td>0x0008</td>
<td>8-bit unsigned integer</td>
</tr>
</tbody>
</table>
Frequently used service record attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute ID Offset</th>
<th>Attribute Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BluetoothProfileDescriptor List</td>
<td>0x0009</td>
<td>DATSEQ of DATSEQ pairs</td>
</tr>
<tr>
<td>DocumentationURL</td>
<td>0x000A</td>
<td>URL</td>
</tr>
<tr>
<td>ClientExecutableURL</td>
<td>0x000B</td>
<td>URL</td>
</tr>
<tr>
<td>IconURL</td>
<td>0x000C</td>
<td>URL</td>
</tr>
<tr>
<td>VersionNumberList</td>
<td>0x0200</td>
<td>DATSEQ of 16-bit unsigned integers</td>
</tr>
<tr>
<td>ServiceDatabaseState</td>
<td>0x0201</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td>ServiceName</td>
<td>0x0000</td>
<td>String</td>
</tr>
<tr>
<td>ServiceDescription</td>
<td>0x0001</td>
<td>String</td>
</tr>
<tr>
<td>ProviderName</td>
<td>0x0002</td>
<td>String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>UUID size</th>
<th>Short UUID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDP</td>
<td>uuid16</td>
<td>0x0001</td>
<td>bt-sdp</td>
</tr>
<tr>
<td>UDP</td>
<td>uuid16</td>
<td>0x0002</td>
<td></td>
</tr>
<tr>
<td>RFCOMM</td>
<td>uuid16</td>
<td>0x0003</td>
<td>bt-fcomm</td>
</tr>
<tr>
<td>TCP</td>
<td>uuid16</td>
<td>0x0004</td>
<td></td>
</tr>
<tr>
<td>OBEX</td>
<td>uuid16</td>
<td>0x0008</td>
<td>obex</td>
</tr>
<tr>
<td>IP</td>
<td>uuid16</td>
<td>0x0009</td>
<td></td>
</tr>
<tr>
<td>FTP</td>
<td>uuid16</td>
<td>0x000A</td>
<td>ftp</td>
</tr>
<tr>
<td>HTTP</td>
<td>uuid16</td>
<td>0x000C</td>
<td>http</td>
</tr>
<tr>
<td>L2CAP</td>
<td>uuid16</td>
<td>0x0100</td>
<td>bt-l2cap</td>
</tr>
</tbody>
</table>

**BASE_UUID:** 00000000-0000-1000-8000-00805F9B34FB (16 bytes, 128 bit)
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>UUID size</th>
<th>UUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>SerialPort</td>
<td>uuid16</td>
<td>0x1101</td>
</tr>
<tr>
<td>LANAccessUsingPPP</td>
<td>uuid16</td>
<td>0x1102</td>
</tr>
<tr>
<td>DialupNetworking</td>
<td>uuid16</td>
<td>0x1103</td>
</tr>
<tr>
<td>OBEXObjectPush</td>
<td>uuid16</td>
<td>0x1105</td>
</tr>
<tr>
<td>OBEXFileTransfer</td>
<td>uuid16</td>
<td>0x1106</td>
</tr>
<tr>
<td>Headset</td>
<td>uuid16</td>
<td>0x1108</td>
</tr>
<tr>
<td>CordlessTelephony</td>
<td>uuid16</td>
<td>0x1109</td>
</tr>
<tr>
<td>AudioSource</td>
<td>uuid16</td>
<td>0x110A</td>
</tr>
<tr>
<td>AudioSink</td>
<td>uuid16</td>
<td>0x110B</td>
</tr>
<tr>
<td>A/V_RemoteControlTarget</td>
<td>uuid16</td>
<td>0x110C</td>
</tr>
<tr>
<td>A/V_RemoteControl</td>
<td>uuid16</td>
<td>0x110E</td>
</tr>
<tr>
<td>Intercom</td>
<td>uuid16</td>
<td>0x1110</td>
</tr>
<tr>
<td>Fax</td>
<td>uuid16</td>
<td>0x1111</td>
</tr>
<tr>
<td>WAP</td>
<td>uuid16</td>
<td>0x1113</td>
</tr>
<tr>
<td>WAP_CLIENT</td>
<td>uuid16</td>
<td>0x1114</td>
</tr>
</tbody>
</table>

### UUIDs for common Bluetooth profiles

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>UUID size</th>
<th>UUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectPrinting</td>
<td>uuid16</td>
<td>0x1118</td>
</tr>
<tr>
<td>Imaging</td>
<td>uuid16</td>
<td>0x111A</td>
</tr>
<tr>
<td>Handsfree</td>
<td>uuid16</td>
<td>0x111E</td>
</tr>
<tr>
<td>HandsfreeAudioGateway</td>
<td>uuid16</td>
<td>0x111F</td>
</tr>
<tr>
<td>DirectPrintingReferenceObjectsService</td>
<td>uuid16</td>
<td>0x1120</td>
</tr>
<tr>
<td>SIM_Access</td>
<td>uuid16</td>
<td>0x112D</td>
</tr>
<tr>
<td>Phonebook Access</td>
<td>uuid16</td>
<td>0x1130</td>
</tr>
</tbody>
</table>

Full List:  
Connection String

Client
- StreamConnection con = (StreamConnection) Connector.open("btspp://0050C000321B:5");
- L2CAPConnection con = (L2CAPConnection) Connector.open("btl2cap://0050C000321B:1000");

Server
- StreamConnectionNotifier cn = (StreamConnectionNotifier) Connector.open("btspp://localhost:" + MY_SERVICE_NUMBER);
- L2CAPConnectionNotifier cn= (L2CAPConnectionNotifier)Connector.open("btl2cap://localhost:" + MY_SERVICE_NUMBER);
Optional Parameters in URI

scheme://host:port;parameters - clients
scheme://localhost:UUID;parameters - server

★ String URL = “btl2cap://localhost:UUID_STRING
;name=L2CAPService;authenticate=true; authorize=true;
master=true”;
  - Master/slave – for piconet and scatternets. Note in
    scatternet: one device in each piconet should play dual
    role (both master and slave)

Exception

★ BluetoothConnectionException
### RFCOMM (serial Port) and L2CAP connections

<table>
<thead>
<tr>
<th>Bluetooth Connection</th>
<th>URL Scheme</th>
<th>Client Connection</th>
<th>Server Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial Port Profile (RFCOMM)</td>
<td>btspp</td>
<td>StreamConnection</td>
<td>StreamConnectionNotifier StreamConnection</td>
</tr>
<tr>
<td>L2CAP</td>
<td>btl2cap</td>
<td>L2CAPConnection</td>
<td>L2CAPConnectionNotifier L2CAPConnection</td>
</tr>
</tbody>
</table>
BLUETOOTH C/S Application – using Serial Port Profile

Device Discovery – (client)

device = LocalDevice.getLocalDevice(); // obtain reference to singleton
device.setDiscoverable(DiscoveryAgent.GIAC); // set Discover mode to GIAC
agent = device.getDiscoveryAgent(); // obtain reference to singleton
agent.startInquiry( DiscoveryAgent.GIAC, new Listener() );

Other Modes:
- DiscoveryAgent.GIAC
- DiscoveryAgent.LIAC
- DiscoveryAgent.NOT_DISCOVERABLE
  - GIAC – General Inquiry Access Code (general discoverable)
  - LIAC – Limited Inquiry Access Code (limited discoverable)
**Device Discovery** – (client) – **Call back events (DiscoveryListener)**

list of RemoteDevice discovered

```java
public static Vector devices = new Vector();

public void deviceDiscovered(RemoteDevice remoteDevice, DeviceClass deviceClass)
{
   devices.addElement( remoteDevice );
}
```

```java
public void inquiryCompleted(int complete)
{
   if ( devices.size() == 0 )
   {
      Alert alert = new Alert( "Problem!", "No Bluetooth device found", null, AlertType.INFO );
      alert.setTimeout(3000);
      display.setCurrent( alert, deviceDiscoveryScreen );
   } else {
      // update the GUI list to reflect all the found devices
      deviceDiscoveryScreen.showList();
      display.setCurrent( devicediscoveryScreen );
   }
}
```
Service Discovery – (client)

public void doDiscoverService(RemoteDevice remote)
    int[] attr = new int[]{0x0000, 0x0001, 0x0002, 0x0003, 0x0004,
                        0x0005, 0x0006, 0x0007, 0x0008, 0x0009, 0x000A, 0x000B,
                        0x000C, 0x000D, 0x0100, 0x0101, 0x0102, 0x0200, 0x0201,
                        0x0301, 0x0302, 0x0303, 0x0304, 0x0305, 0x0306, 0x0307, 0x0308,
                        0x0309, 0x030A, 0x030B, 0x030C, 0x030D, 0x030E, 0x0310, 0x0311,
                        0x0312, 0x0313};
    try {
        agent.searchServices(attr, // null = just retrieve the default attributes,
                        attr = all L2CAP services
                        new UUID[]{ new UUID(0x1101) }, // 0x1100 - SerialPort Profile
                        remoteDevice,
                        new Listener()); // direct discovery response to Listener object
    }
    catch (BluetoothStateException e) { ......}
**Service Discovery – (client) – Call back events (DiscoveryListener)**

```java
public static Vector services = new Vector();

public void servicesDiscovered(int transId, ServiceRecord[] records) {
    for (int i = 0; i < records.length; i++) {
        ServiceRecord record = records[i];
        services.addElement(record);
    }
}
```

```java
public void serviceSearchCompleted(int transId, int complete) {
    if (services.size() > 0) {
        // found at least one SPP service. We can send a message. If more than one SPP service is found, we send to the first one. (use sppConnection).
        sendData("Hello There");  // sendData -> MAKE CONNECTION and SEND
    } else {
        // no service record found for SerialPort
        Alert alert = new Alert( "Problem!", "no spp", null, AlertType.ERROR );
        alert.setTimeout( Alert.FOREVER );
        display.setCurrent( devicediscoveryScreen );
    }
}
```

```java
........
```
Send Data over Bluetooth – (client)

```java
public void sendData(String msg) {

    ServiceRecord r = (ServiceRecord) services.elementAt(0); // to first spp service
    // obtain the URL reference to this service on remote device
    String url = r.getConnectionURL(ServiceRecord.NOAUTHENTICATE_NOENCRIPT, false);
    try {
    
    // obtain connection and stream to this service
    StreamConnection con = (StreamConnection) Connector.open(url);
    DataOutputStream out = con.openDataOutputStream();

    // write data into serial stream
    out.writeUTF(msg);
    out.flush();

    ..........

    Note: Each connection must be in a New Thread.
```
Bluetooth Server Connection - (server) - Register Service

```java
device = LocalDevice.getLocalDevice(); // obtain reference to singleton
device.setDiscoverable(DiscoveryAgent.GIAC); // set Discover mode to L
String appName = "SSPServer";
// unique UUID for this service. this can be defined by developers
UUID uuid = new UUID(0xABCD);
StreamConnectionNotifier server = null;
StreamConnection c = null;
try {
    server = (StreamConnectionNotifier)Connector.open(
        "btspp://localhost:" + uuid.toString() + ";name="+appName);
    // Retrieve the service record template (empty)
    ServiceRecord rec = device.getRecord( server );

    // set/update optional attributes that are to be added to the service record.
    // populate BluetoothProfileDescriptionList (0x0009) using SerialPort version 1
    DataElement e1 = new DataElement( DataElement.DATSEQ );
    e1.addElement( new DataElement( DataElement.UUID, new UUID(0x1101) )); // add SerialPort
    e1.addElement( new DataElement( DataElement.INT_8, 1) ); // add Version 1
    rec.setAttributeValue( 0x0009, e2 ); // add BluetoothProfileDescriptionList
```

Bluetooth ServerConnection – (server)– Listen and Read

\[ c = \text{server.acceptAndOpen();} \] //wait for incoming connection & create service record

// obtain an input stream to the remote service
\[ \text{DataInputStream in = c.openDataInputStream();} \]

// read in a string from the string
\[ \text{String s = in.readUTF();} \]

// display this string on GUI
\[ \text{append(s, null);} \]

// close current connection
\[ \text{c.close();} \]

Further Reading: benhui.net, forum.nokia.com, developers.sonyericsson.com
Bluetooth C/S using L2CAP (Client)

```java
int index = 0;
L2CAPConnection con = null; transmitBuffer[] temp = null; byte[] data = ...;
try {
    con = (L2CAPConnection)Connector.open(url);
    int MTUSize = con.getTransmitMTU();  //Maximum Transmission Unit
    // Allocation a buffer of that (MTU) size
    transmitBuffer = new byte[MaxOutBufSize];
    :::
    while (index < data.length) {
        // Send the data... move MTUSize bytes from data
        // buffer to transmit buffer
        if ((data.length - index) < MTUSize) {
            System.arraycopy(data, index, transmitBuffer, 0, data.length - index);
        } else {
            System.arraycopy(data, index, transmitBuffer, 0, MTUSize);
        }
        con.send(transmitBuffer);
        index += MTUSize;
        // Reset the transmit buffer
        for (int=0; i<MTUSize; i++) transmitBuffer[i] = 0;
    }
    con.close();
} catch (Exception e) {... Handle Exception }
```
Bluetooth C/S using L2CAP (Server)

L2CAPConnectionNotifier server = null; byte[] data = null;
int length;
:.....
try {
LocalDevice local = LocalDevice.getLocalDevice();
local.setDiscoverable(DiscoveryAgent.GIAC);
server = (L2CAPConnectionNotifier)
Connector.open("btl2cap://localhost:1020304050d0708093a1b121d1e1f100 ");
while (!done) {
    L2CAPConnection conn = null;
    conn = server.acceptAndOpen();
    length = conn.getReceiveMTU();
    data = new byte[length];
    length = conn.receive(data);
    :....
}
} catch (Exception e) {
    ... Handle Exception
Multiplayer Games

RFCOMM vs L2CAP

L2CAP

★ 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.

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.
★ 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).
Device classes (DeviceClass class)

DeviceClass represents a class of device (CoD) as specified in the Bluetooth specification.
Devices classes are identified using a major, minor and service class.

- int getMajorDeviceClass() – retrieves the major device class.
- int getMinorDeviceClass() – retrieves the minor device class.
- int getServiceClasses() – retrieves the major service classes.
Device classes (DeviceClass class)

static final NLDMSG = 0x22000; // Networking, Limited Discoverable Major Service Class
static final PHONE_MAJOR_CLASS = 0x200;
static final CELLULAR_MINOR_CLASS = 0x04;

LocalDevice localDevice;
DeviceClass deviceClass;

try {
    localDevice = LocalDevice.getLocalDevice();
    deviceClass = localDevice.getDeviceClass();
    if (deviceClass.getMajorDeviceClass() == PHONE_MAJOR_CLASS) {
        if (deviceClass.getMinorDeviceClass() == CELLULAR_MINOR_CLASS) {
            .....//Do something
        }
    }
}
RemoteDevice class

- static RemoteDevice
  getRemoveDevice(javax.microedition.io.Connection) – static method to retrieve the RemoteDevice object associated with the passed Connection.

- java.lang.String getBluetoothAddress() – retrieves the Bluetooth address of the remote device. java.lang.String
  getFriendlyName() – retrieves the name of the remote device.

- boolean authenticate() – attempts to authenticate the remote device.

- boolean isAuthenticated() – determines if this RemoteDevice has been authenticated.

- boolean isEncrypted() – determines if data exchanges with this RemoteDevice are currently being encrypted.
LocalDevice class

- static LocalDevice getLocalDevice()
- java.lang.String getBluetoothAddress()
- java.lang.String getFriendlyName()
- DiscoveryAgent getDiscoveryAgent() - returns the discovery agent for this device.
- boolean setDiscoverable(int mode) – sets the discoverable mode of the device.
- static java.lang.String getProperty(java.lang.String property) – retrieves Bluetooth system properties. [refer next slide]
- ServiceRecord
  getRecord(javax.microedition.io.Connection notifier) – retrieves the service record corresponding to the passed (btspp, btl2cap, or btgoep) notifier.
Property

★ bluetooth.api.version
★ bluetooth.l2cap.receiveMTU.max
★ bluetooth.connected.devices.max
★ bluetooth.connected.inquiry
★ bluetooth.connected.page
★ bluetooth.connected.inquiry.scan
★ bluetooth.connected.page.scan
★ bluetooth.master.switch
★ bluetooth.sd.trans.max
★ bluetooth.sd.attr.retrievable.max
Games Over Bluetooth

★ Bluetooth is suitable for ‘proximity gaming’ – playing games with people around you
★ The low latency makes it suitable for real-time games
  – driving games, shooting games, ...
  – but also card games, etc.
★ Up to 8 players, if master device supports point-to-multipoint
★ Use L2CAP packets or RFCOMM streams
Games over Bluetooth – Best Practices

★ Several Bluetooth actions at the same time does not speed the application.
★ All Bluetooth activities consume bandwidth, which leads to higher latency for the game. All Bluetooth activities that do not belong to the game should be canceled.
★ Then the user should be asked to select a game client or game host role.
★ Bluetooth provides a reliable connection; there is no need to add a custom protocol for data correction or data acknowledgement. Corrupted packets are retransmitted until they are correctly received.
★ Use a protocol with little overhead, such as L2CAP.
Game Update Strategies

- **Frame-based**: clients operate synchronously with server, displaying each frame as they receive its data.
  - **BLUETOOTH**
    - needs latency < 40ms or so
- **Dead reckoning**: clients operate asynchronously from server, predicting action and correcting when the server sends updates.
  - OK for Internet-level latencies, 100-200ms
- **Turn-based**: clients take turn to act, when the server tells them it’s their turn.
  - OK even if latency is several seconds
Example Game Screen Map

Splash

Host game / Join game

Search for players

Choose players

Connect

Wait for connection

Play

hosting game

joining game
Bluetooth Latency

✦ Using JSR-82, we measured round-trip latency of about 30ms
✦ It gets worse if:
  – many other devices are around
  – you send data so fast it must be buffered
  – your packets are bigger than your device’s packet size (MTU)
  – devices are far apart (so poor link quality and re-sends)
✦ For more details see Forum Nokia document *Games Over Bluetooth: Recommendations to Game Developers*
Special Considerations for Nokia Devices

- **Low-Power Mode**: if a Nokia device gets no Bluetooth data for 15 seconds, it enters SNIFF mode, only checking for new data every 0.5 seconds
  - avoid this by sending an empty message every few seconds if necessary
- **Link Loss**: if the device receives no low-level Bluetooth packets for 20 seconds, the link will be dropped
- **Disconnection**: players will often leave a game before it ends
Demo Game: Paintball (Nokia)
★ Simple real-time ‘shooter’ game
★ Motion on a 16x16 grid
★ Master holds the game state
Demo Game: Communications
Problems with Bluetooth connections

★ Device and service discovery sometimes fail
★ Connection setup takes time
★ Connections can drop anytime
★ Latency
★ Threading
★ Testing!
Hints & Tips

★ Pay attention to threading issues
★ Close connections on exit
★ Ignore cached devices, since you can’t find out their Class of Device
★ On Series 60, prefer RFCOMM to L2CAP
★ Test with many devices, different devices, and with other Bluetooth devices (e.g. headsets) in proximity
Other topics

- Use OBEX to exchange images
- How to connect Bluetooth devices of different platforms
- Understand Bluetooth security
- How to develop multi-connection Bluetooth application
- N-Gage Arena (SNAP) – full-scale mobile online game environment
- X-Box Live
Some examples

★ It’s Alive (Swedish):
  - “botfighters” www.botfighters.com
  - Players chase each other to various cellular network locations

★ Jamba (German):
  - “Attack of the Killer Virus”
  - Player shoots viruses/monsters projected to a real-life environment shown through a lens of a camera phone. A player has to move around with the camera to destroy the viruses.

★ Warhol’s 15 minutes
  - Messages -> game actions
  - Shown to large audiences on TVs
SNAP Overview

✦ Package: com.nokia.sm.net
  - Contains classes that support communication with a SNAP Mobile game server.

✦ SnapEventListener
  - Callback interface for asynchronous SNAP Mobile event notification.

✦ ItemList
  - This class implements a container for one or more items of different types.

✦ ServerComm
  - This class facilitates communication between a game client and a SNAP Mobile server.
SNAP Overview

- Communication between a game client and a SNAP Mobile server. It provides methods for accessing online multiplayer game and community features such as
  - instant messaging,
  - chat,
  - presence management,
  - buddy list (or friends list),
  - versatile matchmaking, and
  - ranking.

- Implementation does not depend on the underlying network protocol. At present, HTTP and TCP are the only supported protocols, but other protocols may be added in the future.
SNAP Overview

Server Events:

- SNAP Mobile servers generate events for certain actions that take place, such as creating new lobbies or game rooms, chat messages delivered to a particular user, and so on.
- These events are held on the server until retrieved by the client.
- Retrieving Events:
  - Polling by client using methods such as receiveEvents(int,int,int) and retrieveAllEvents().
  - Client can register as a listener for SNAP events by calling addSnapEventListener(). SNAP server calls back the client when new events are available.
SNAP server

★ Demo Application