

PART 2

Network Programming

- HTTP
- SOCKET
- MESSAGING
- BLUETOOTH
- SERIAL PORT

Handling Media in Mobile Devices

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, HSDPA, HSUPA ...)
- ★ 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, UDP)**, with security/permission

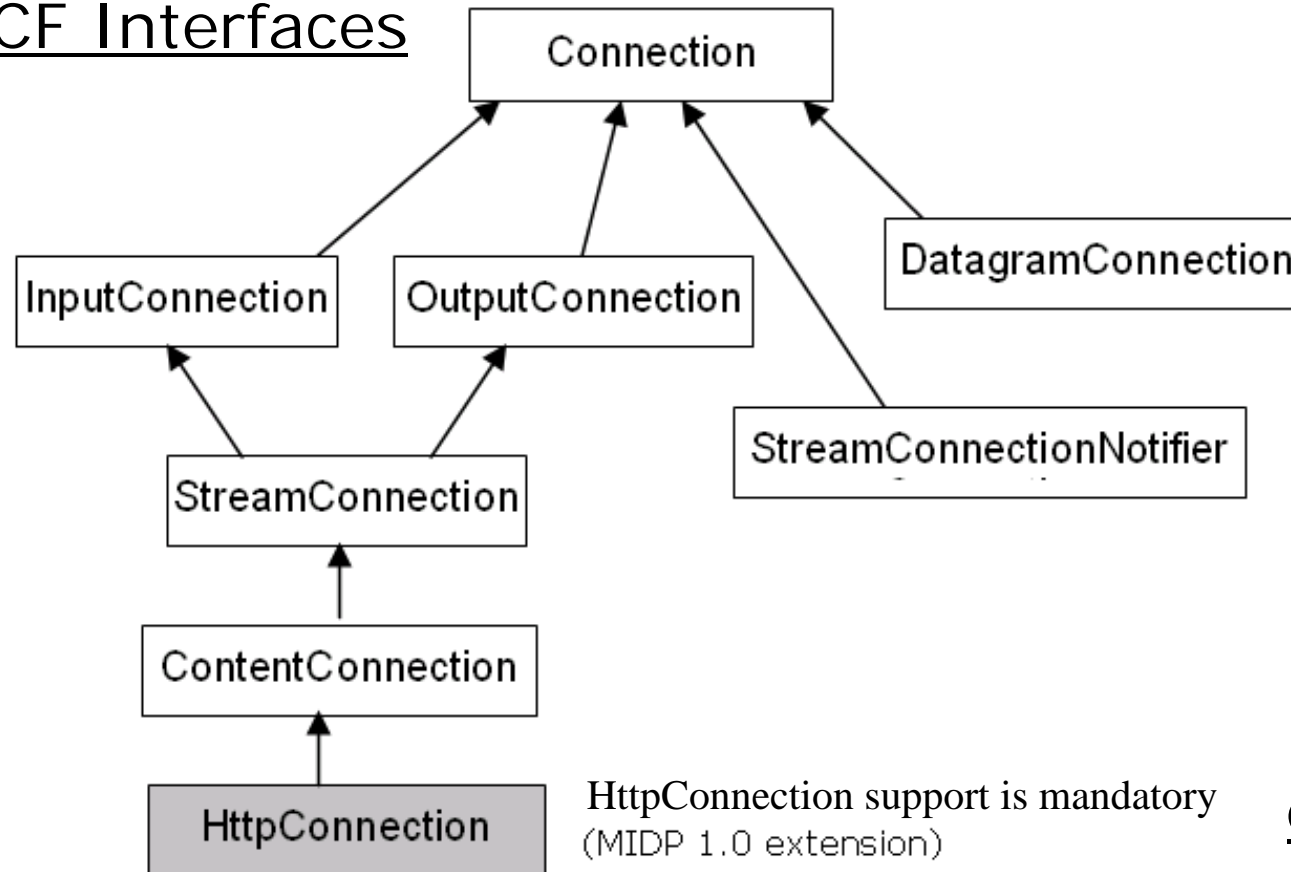
Generic Connection Framework

- ★ All the classes including the common **Connector** class defined in the CLDC specification for networking APIs forms the Generic Connection Framework (GCF).
- ★ Generalization is possible through,
 - An interface hierarchy that is extensible
 - A connection factory [Connector class]– to create any type of connection
 - Standard Uniform Resource Locators (URLs) to indicate the connection types to create [determined by the *protocol* string in the URI parameter]

- ★ Package: **javax.microedition.io**

Q: Why HTTP?

GCF Interfaces



Other Classes in GCF

- Connector
- ConnectionNotFoundException
- Datagram – for packet based connections, represents a packet
- <few more>

Streams from **java.io** [not defined in GCF]

InputStream, DataInputStream, OutputStream, DataOutputStream

Low level IP network APIs in MIDP 2.0

- ★ SocketConnection (TCP)
- ★ ServerSocketConnection (TCP)
- ★ UDPDatagramConnection

- MIDP 2.0 implementers *must* provide support for **HTTP 1.1 servers** as well as **secure HTTP** connections. Support for **low-level IP networking is optional** (left to the handset manufacturers and network operators).
- Re-visit shortly

The GCF - An Extensible Framework

- ★ Connections are not-so-generic Connections
 - HTTP, Bluetooth, Messaging

GCF – Wide Range of Connection Types

URL Scheme	Connectivity	GCF Connection Type
bt12cap	Bluetooth	L2CAPConnection
btsp		StreamConnection
datagram	Datagram	DatagramConnection
file	File Access	FileConnection, (InputConnection in Foundation profile)
http	HyperText Transport Protocol	HttpConnection
https	Secure HTTP	HttpsConnection
comm	Serial I/O	CommConnection
sms	Short Messaging Service	MessageConnection
mms	Multimedia Messaging Service	
cbs	Cell Broadcast SMS	
apdu	Security Element	APDUConnection, JavaCardRMICConnection
jcrmi		
socket	Socket	SocketConnection, ServerSocketConnection
serversocket		
datagram	UDP Datagram	UDPDatagramConnection

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.
 - 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* when timeout occurs.

The Connector class (other methods)

Q: Not recommended?

- ★ `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.
- All these are static methods

★ Eg.

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

Note: The connections must be opened in a separate Thread.

GCF Examples- FYORP

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:

Method	Description
<u>openDataInputStream()</u>	Opens and returns a data input stream for a connection.
<u>openInputStream()</u>	Opens and returns an input stream for a connection.

- ★ 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.

InputConnection

```

Class readWeb implements Runnable { //Runnable class
public void run() {
    InputConnection inc = null;    InputStream is = null;
    StringBuffer b=new StringBuffer();
    try {
        inc = (InputConnection)
                Connector.open("http://books.anuflora.com");
        is = inc.openInputStream();
        int ch;
        while((ch = is.read())!= -1){
            b.append((char)ch);
        }
        strI tm.setText(new String(b));
    }catch(IOException e){
    } finally {
        if (is!=null) try { is.close();} catch (Exception e) {}
        if (stc!=null) try { stc.close();} catch (Exception e) {}
    }
}
}
}
        readWeb r = new readWeb(); //Running in new Thread
        new Thread(r).start();
}
}

```

GCF Examples- FYORP

OutputConnection Interface

- ★ The *OutputConnection* interface is another subinterface of *Connection*. The *OutputConnection* interface represents a connection's stream data as an *OutputStream*.
- ★ *OutputConnection* methods:

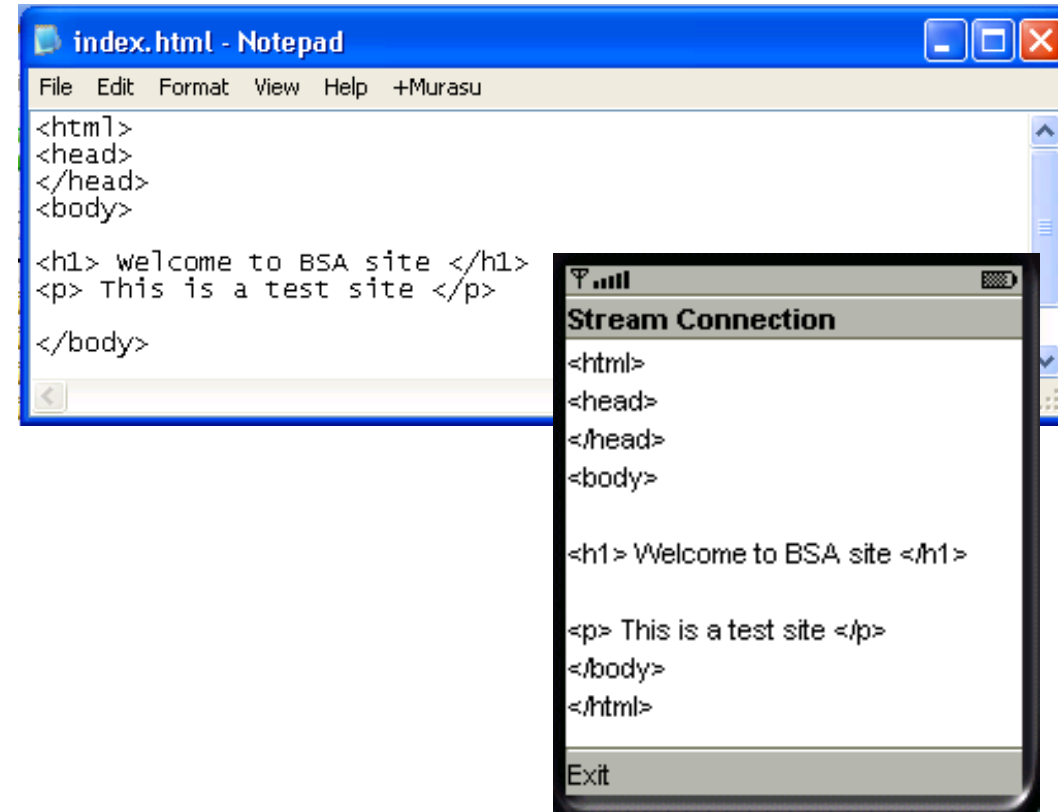
Method	Description
<u>openDataOutputStream()</u>	Opens and returns a data output stream for a connection.
<u>openOutputStream()</u>	Opens and returns an output stream for a connection.

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

GCF Examples- FYORP

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.

GCF Examples- FYORP

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

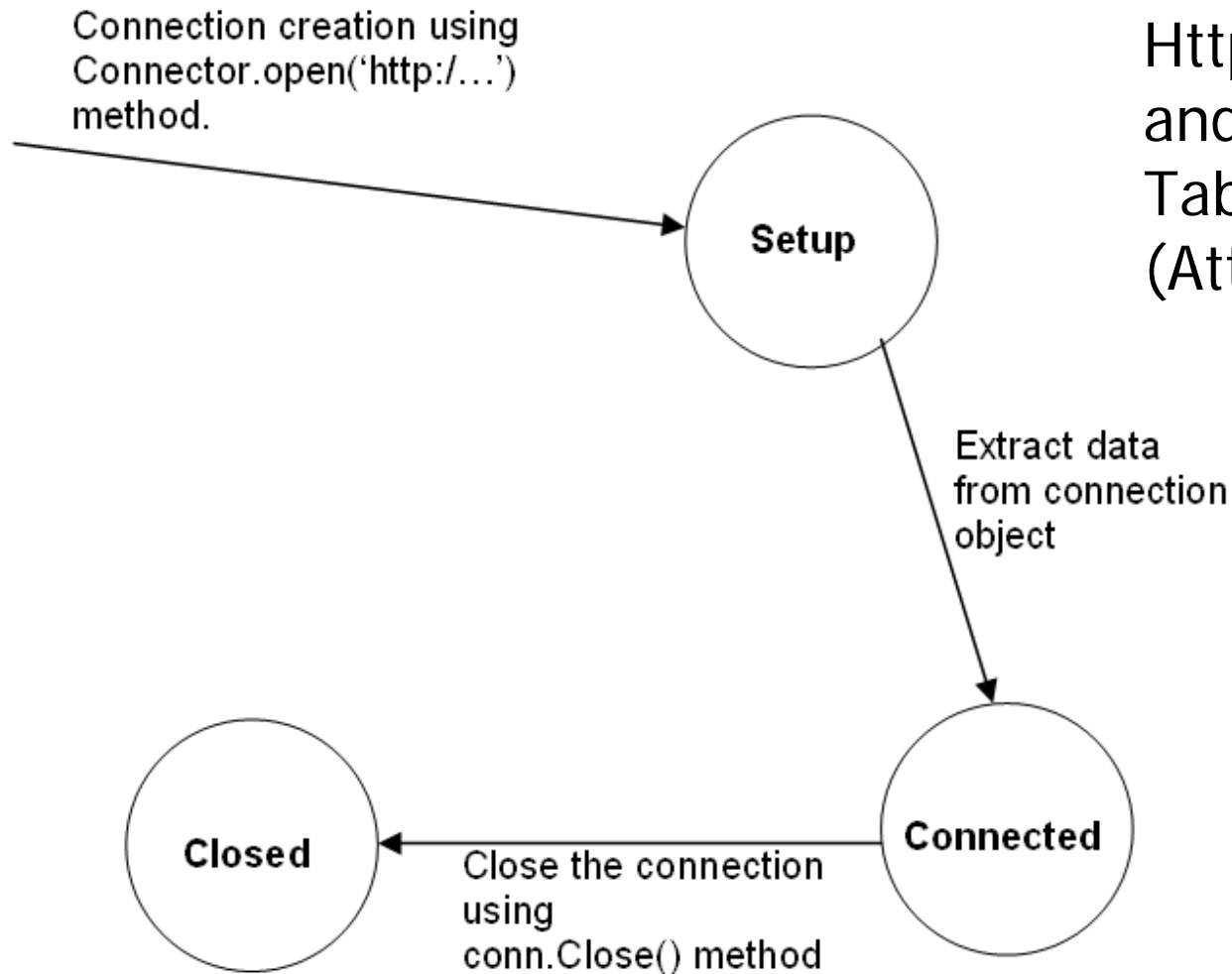
- ★ *ContentConnection* knows how to extract encoding, length and content type of the data received.

```
try {  
    cc = (ContentConnection)  
        Connector.open("http://localhost/anuflora/index.htm");  
    stream = cc.openDataInputStream();  
    byte[] buffer = new byte[1000];  
    stream.readFully(buffer);  
    strItn.setText("Length: " + cc.getLength() +  
        " Encoding: " + cc.getEncoding() + " Type: " +  
        cc.getType());  
} 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]
- ★ MIDP 2.0 Implementations must should support HTTP 1.1

HttpConnection States

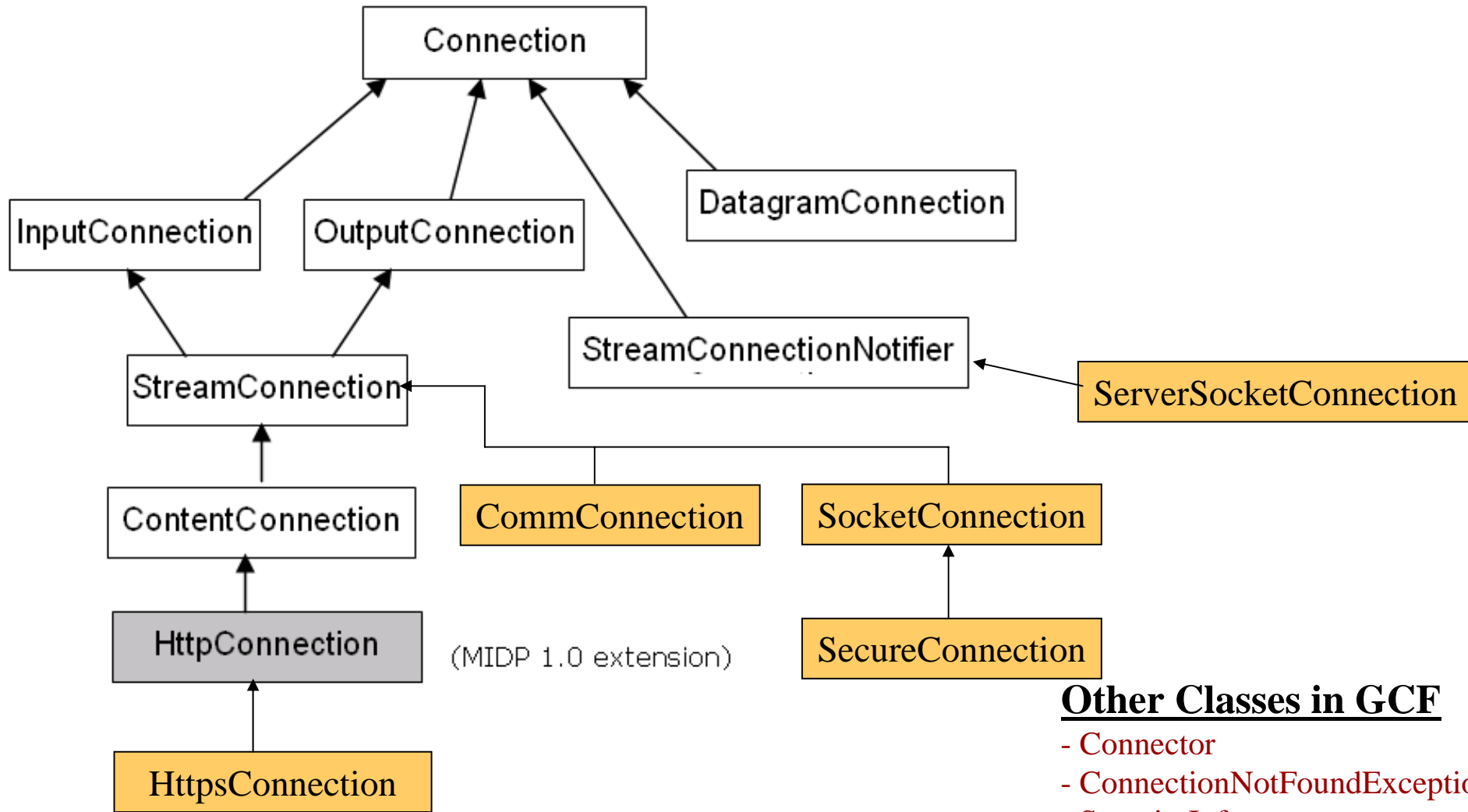


HttpInterface methods and Error codes.
Table 6.10 to 6.14
(Attached)

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;    } .....
```



Other Classes in GCF

- Connector
- ConnectionNotFoundException
- SecurityInfo
- Datagram – for packet based connections, represents a packet
- <few more>

Streams from **java.io** [not defined in GCF]

InputStream, DataInputStream, OutputStream, DataOutputStream

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 it's local end point address.

ServerSocketConnection (Echo Server)

```
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`)?

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.

– 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

★ URI for SMS and CBS has three parts:

- 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

Q: Can only receive SMS from other J2ME SMS clients. Why?

Sending a Message

★ STEPS

- Create a *MessageConnection* interface.
- Use the *MessageConnection*'s *newMessage()* method to create a message object.
 - *newMessage()* method will takes 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

```
public void sendText(MessageConnection conn, String text)
    throws IOException, InterruptedException {
    TextMessage txtMsg =
        conn.newMessage(conn.TEXT_MESSAGE);
    txtMsg.setPayloadText( text );
    conn.send( txtMsg );
}
```

Example : Sending Binary Data

```
public void sendBinary(MessageConnection conn, byte[] data)
    throws IOException, InterruptedException {
    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

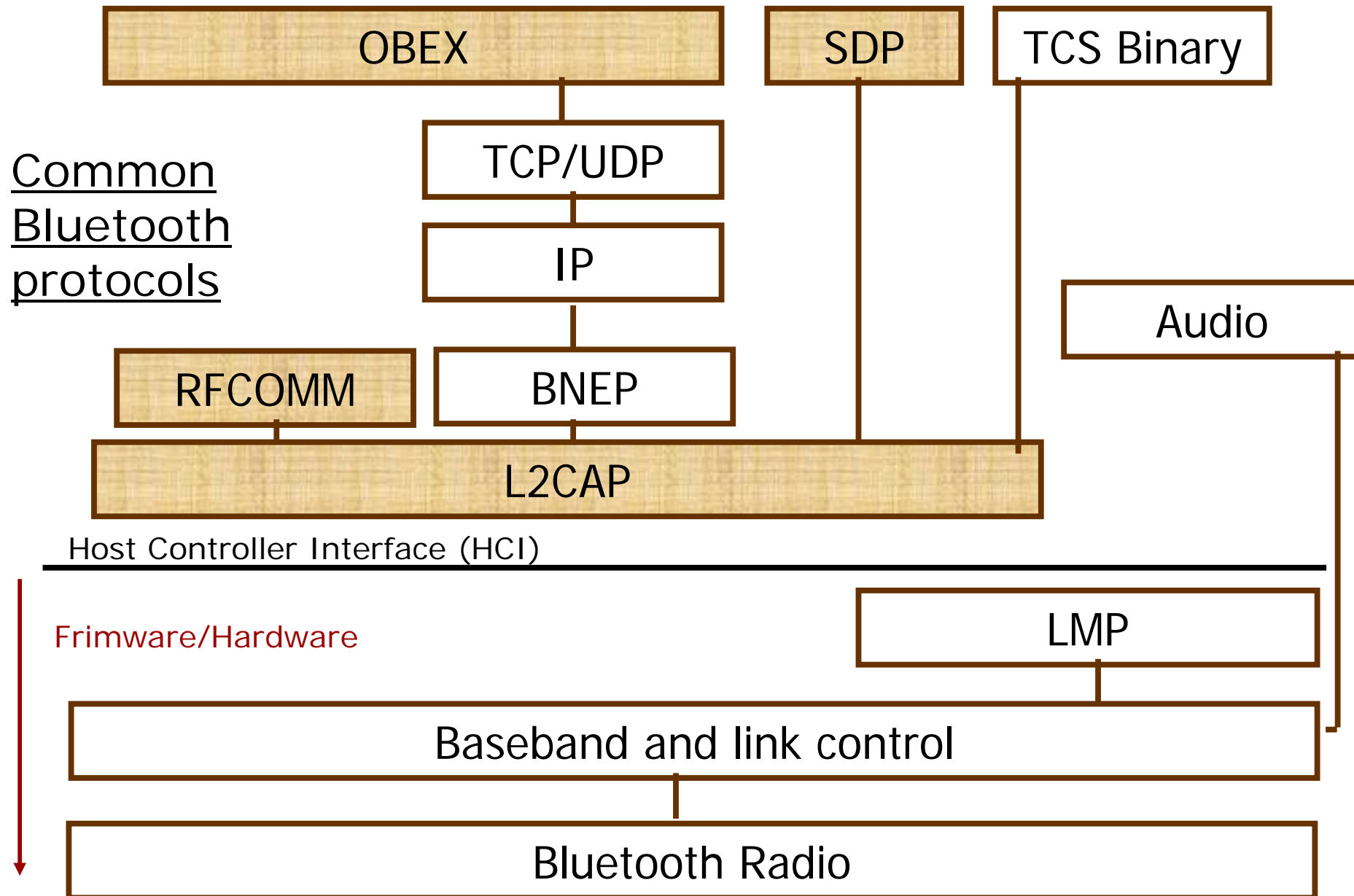
- ★ Bluetooth devices can broadcast their identity to be discovered by others
- ★ Bluetooth is commonly used to emulate a direct serial cable connection
- ★ De facto standard for low-cost and low-power short-range radio links between mobile devices, PCs, headsets, GPS receivers, peripherals and consumer 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
 - <http://bluetooth.com/Bluetooth/Learn/Technology/Compare/>

Bluetooth

- ★ Mobile entertainment applications (game)
- ★ 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



Java API for Bluetooth wireless technology (JABWT)

Packages

- javax.bluetooth
- javax.obex

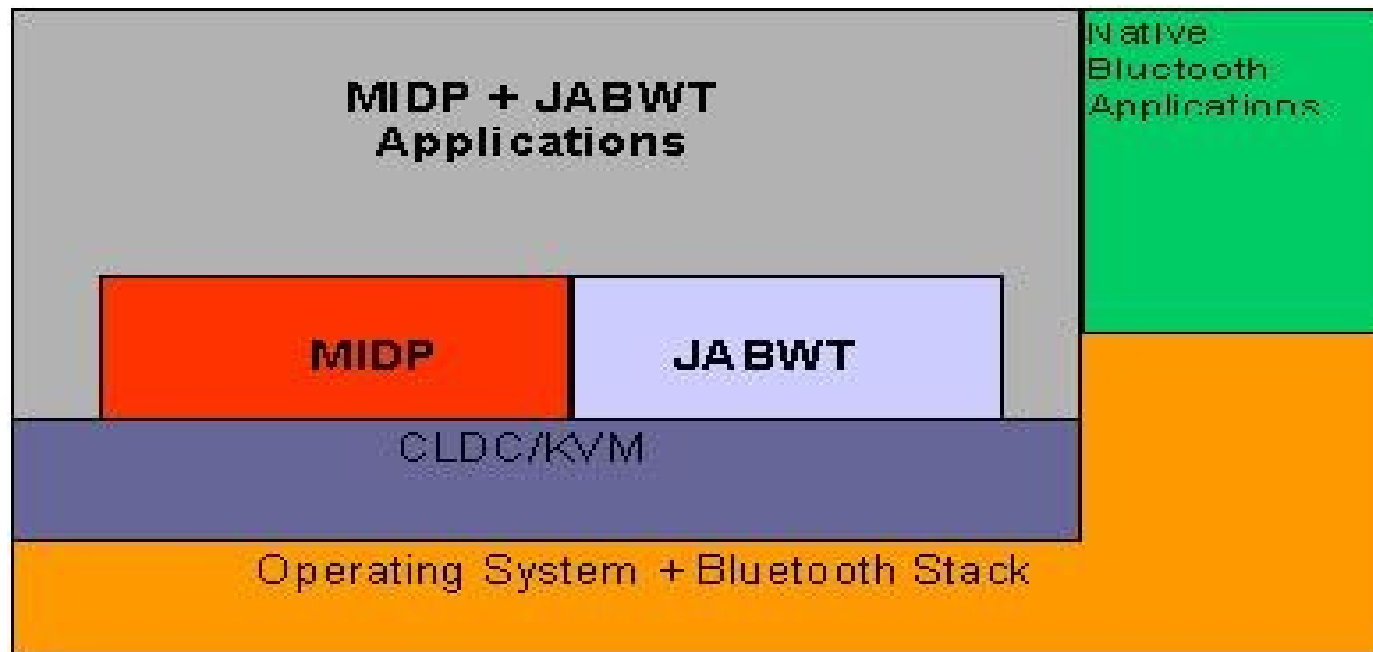


Image Source: Bluetooth Application Programming with the Java APIs (book) mkp.com

Bluetooth Application Activities

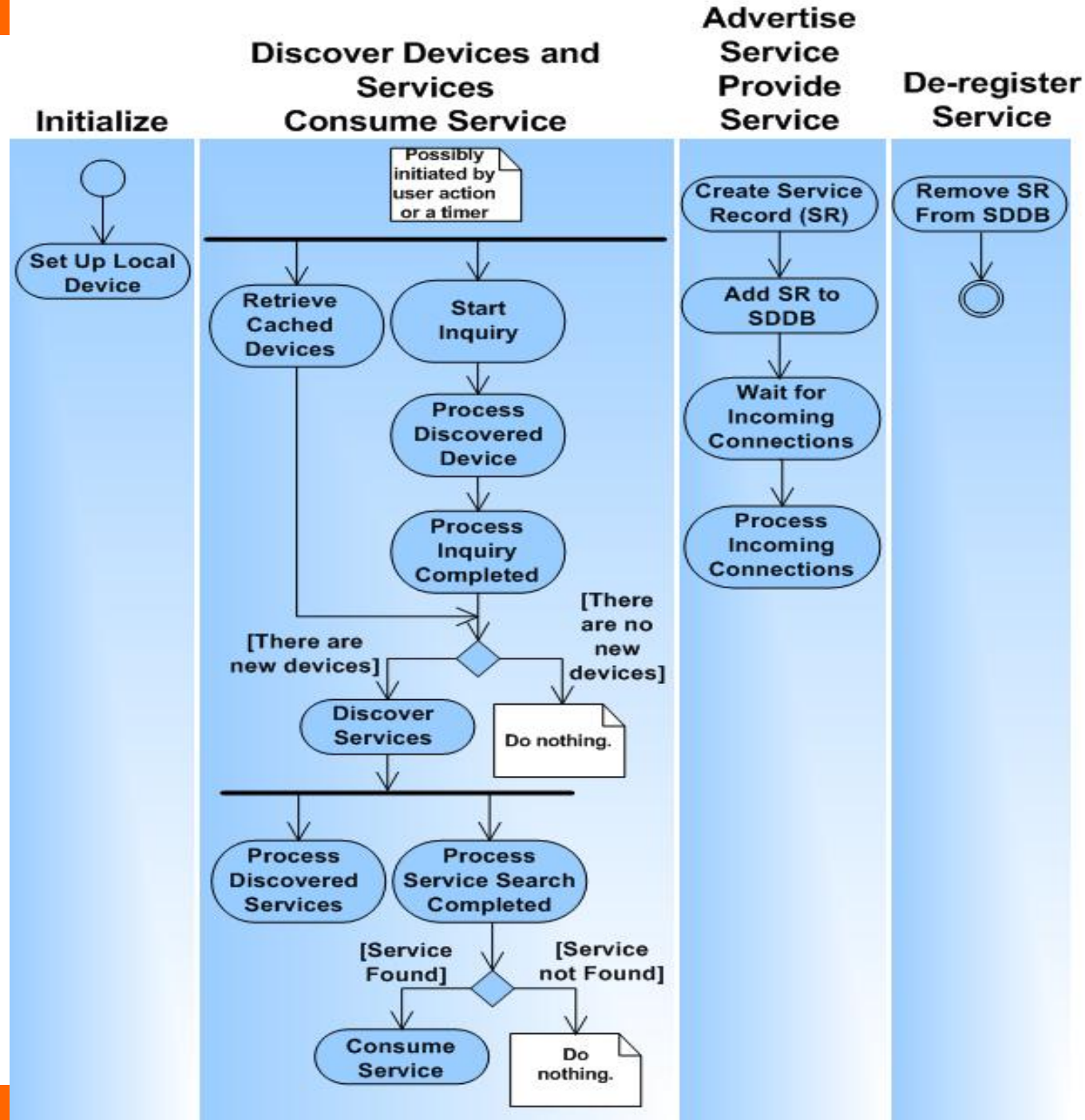
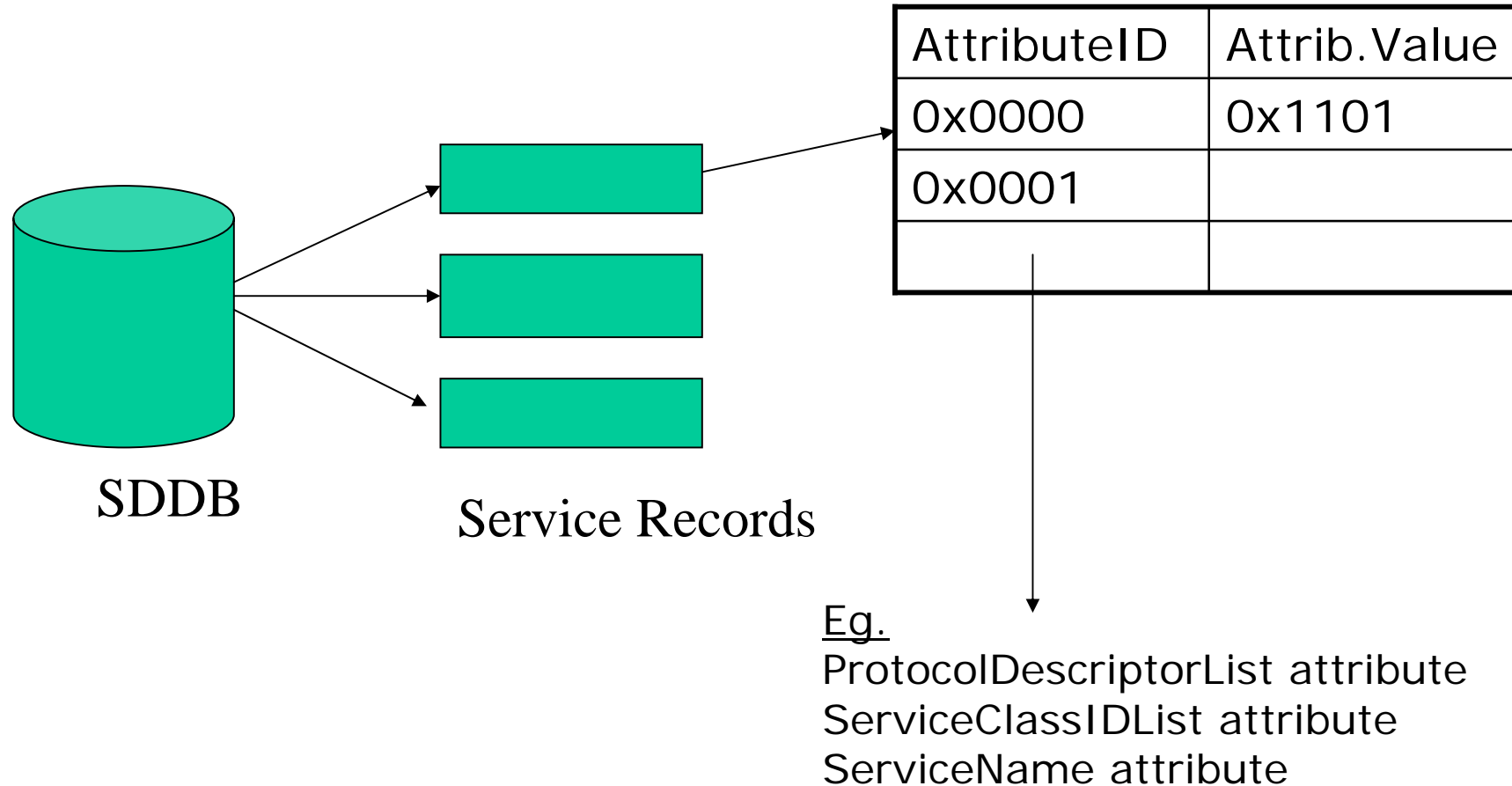


Image Source:
<http://developers.sun.com/>, by C.
Enrique Ortiz,

Service Discovery Database (SDDB)

★ Database of registered services



Attribute List: https://www.bluetooth.org/foundry/assignnumb/document/service_discovery

Frequently used service record attributes

Attribute Name	Attribute ID	Attribute Value Type
ServiceRecordHandle	0x0000	32-bit unsigned integer
ServiceClassIDList	0x0001	DATSEQ of UUIDs
ServiceRecordState	0x0002	32-bit unsigned integer
ServiceID	0x0003	UUID
ProtocolDescriptorList	0x0004	DATSEQ of DATSEQ of UUID and optional parameters
BrowseGroupList	0x0005	DATSEQ of UUIDs
LanguageBasedAttributeIDList	0x0006	DATSEQ of DATSEQ triples
ServiceInfoTimeToLive	0x0007	32-bit unsigned integer
ServiceAvailability	0x0008	8-bit unsigned integer

Frequently used service record attributes

Attribute Name	Attribute ID Offset	Attribute Value Type
BluetoothProfileDescriptorList	0x0009	DATSEQ of DATSEQ pairs
DocumentationURL	0x000A	URL
ClientExecutableURL	0x000B	URL
IconURL	0x000C	URL
VersionNumberList	0x0200	DATSEQ of 16-bit unsigned integers
ServiceDatabaseState	0x0201	32-bit unsigned integer
ServiceName	0x0000	String
ServiceDescription	0x0001	String
ProviderName	0x0002	String

<http://www.bluetooth.com/dev/specifications.asp>) for full list.

UUIDs for common Bluetooth protocols

Mnemonic	UUID size	Short UUID	Name
SDP	uuid16	0x0001	bt-sdp
UDP	uuid16	0x0002	
RFCOMM	uuid16	0x0003	bt-fcomm
TCP	uuid16	0x0004	
OBEX	uuid16	0x0008	obex
IP	uuid16	0x0009	
FTP	uuid16	0x000A	ftp
HTTP	uuid16	0x000C	http
L2CAP	uuid16	0x0100	bt-l2cap

The Base UUID is used for calculating 128-bit UUIDs from 'short UUIDs' (uuid16 and uuid32)

Base UUID *296 + Short UUID32

Uuid32 = uuid16 + 32 bit zeros

Need to promote small size to big size before comparing 2 uuids.

Full List:

https://www.bluetooth.org/foundry/assignment/document/service_discovery

BASE_UUID: 00000000-0000-1000-8000-00805F9B34FB (16 bytes, 128 bit)

Mnemonic	UUID size	UUID
SerialPort	uuid16	0x1101
LANAccessUsingPPP	uuid16	0x1102
DialupNetworking	uuid16	0x1103
OBEXObjectPush	uuid16	0x1105
OBEXFileTransfer	uuid16	0x1106
Headset	uuid16	0x1108
CordlessTelephony	uuid16	0x1109
AudioSource	uuid16	0x110A
AudioSink	uuid16	0x110B
A/V_RemoteControlTarget	uuid16	0x110C
A/V_RemoteControl	uuid16	0x110E
Intercom	uuid16	0x1110
Fax	uuid16	0x1111
WAP	uuid16	0x1113
WAP_CLIENT	uuid16	0x1114

UUIDs for common Bluetooth profiles

Mnemonic	UUID size	UUID
DirectPrinting	uuid16	0x1118
Imaging	uuid16	0x111A
Handsfree	uuid16	0x111E
HandsfreeAudioGateway	uuid16	0x111F
DirectPrintingReferenceObjectsService	uuid16	0x1120
SIM_Access	uuid16	0x112D
Phonebook Access	uuid16	0x1130

Full List:

https://www.bluetooth.org/foundry/assignment/document/service_discovery

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

Bluetooth Connection	URL Scheme	Client Connection	Server Connection
Serial Port Profile (RFCOMM)	btsp	StreamConnection	StreamConnectionNotifier StreamConnection
L2CAP	btl2cap	L2CAPConnection	L2CAPConnectionNotifier L2CAPConnection

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

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

```
public void deviceDiscovered(RemoteDevice remoteDevice, DeviceClass  
deviceClass)
```

```
{  
    devices.addElement( remoteDevice );  
}
```

```
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)

```
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 ); }
}
```

```
public void serviceSearchCompleted(int transId, int complete)
```

```
{
  if ( services.size() > 0 )
  {
    // found at least one SPP service. We can send a message. If morethan 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 ); }
} .....
```

Send Data over Bluetooth – (client)

```
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_NOENCRYPT,  
    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 ServerConnection – (server)– Register Service

```

device = LocalDevice.getLocalDevice(); // obtain reference to singleton
device.setDiscoverable(DiscoveryAgent.GIAC); // set Discover mode to GIAC
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 = server.acceptAndOpen(); //wait for incoming
                             connection & create service record
// obtain an input stream to the remote service
DataInputStream in = c.openDataInputStream();

// read in a string from the string
String s = in.readUTF();

// display this string on GUI
append( s, null );

// close current connection
c.close();
```

Further Reading: benhui.net, forum.nokia.com, developers.sonyericsson.com

Bluetooth C/S using L2CAP (Client)

```

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
```

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 NLDMSC = 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 getAddress() – 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 (btsp, bt12cap, 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`

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. All Bluetooth activities that do not belong to the application should be canceled.
- ★ 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.

Observations by Nokia- Bluetooth Latency

- ★ Using JSR-82, round-trip latency is 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

Problems with Bluetooth connections

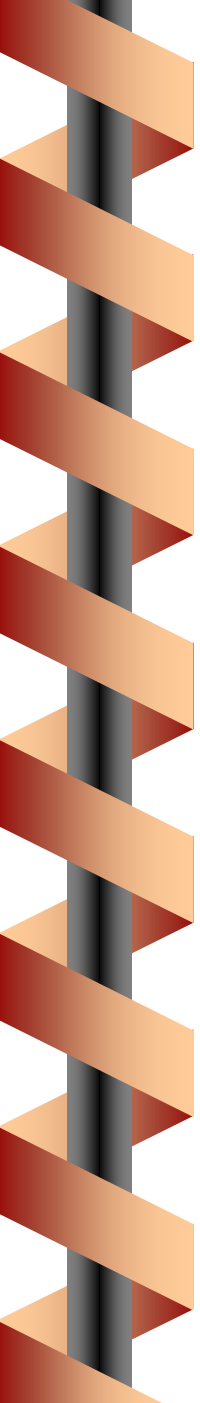
- ★ Device and service discovery sometimes fail
- ★ Connection setup takes time
- ★ Connections can drop anytime
- ★ Latency
- ★ Threading
- ★ Testing!

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



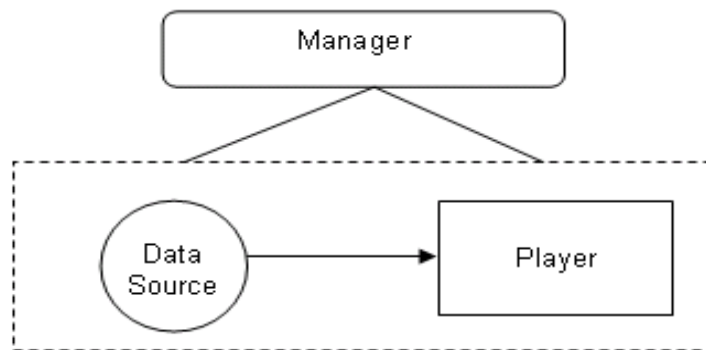
Handling Media in Mobile Devices

Audio and Video

Targets CLDC devices. (For MIDP : A subset of MMAPI called MIDP Media API is defined.)

Four fundamental objects :

- ★ Player : Intercepts media data received from the *DataSource*.
 - javax.microedition.media.Player
- ★ Control : To control the Player. [Play, Stop, ff]
 - javax.microedition.media.Control : all controls
 - javax.microedition.media.controls : specific control
- ★ DataSource : Knows how to get media data from its original location. (HTTP, RMS)
 - javax.microedition.media.protocol.Datasource



Four fundamental objects :

- ★ Manager : Serves as an entry point to the API. Has abstract methods for obtaining playerS and datasourceS.
 - `javax.microedition.media.protocol.Manager`

Extract from Listing 7.1

```
public void playMID() {  
    try {  
        Player p = Manager.createPlayer  
            ("http://localhost:8081/mcom/av/hotel.mid");  
        p.setLoopCount(5);  
        p.start();  
    } catch (IOException ioe) { }  
    catch (MediaException e) { }  
}
```

Data Source at WWW



Playing audio from local file

```
try {  
    InputStream is = getClass().getResourceAsStream("paradise.wav");  
    Player p = Manager.createPlayer(is, "audio/X-wav");  
    p.start();} catch (IOException ioe) {} catch (MediaException me) { }
```

Playnig audio from record store

// set up RecordStore *rs* with the music files.

```
try {  
    InputStream is = new ByteArrayInputStream(rs.getRecord(recordID));  
    Player p = Manager.createPlayer(is, "audio/X-wav");  
    p.start();} catch (IOException ioe) {} catch (MediaException me) { }
```

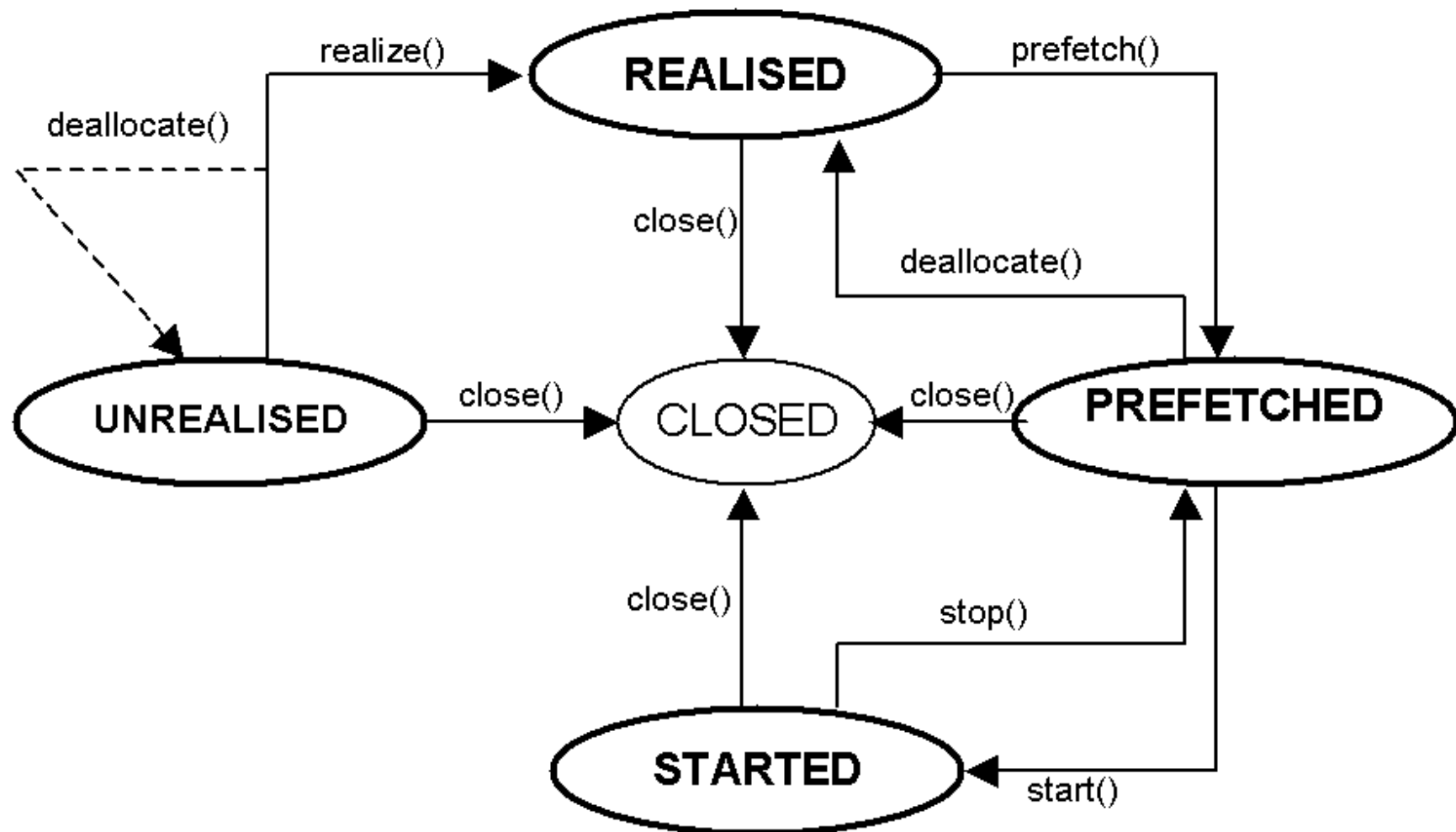


Figure 7.2 : Player state transition

Player Interface Methods

– **Table 8.2**

Listening a Player

- ★ `p.addListener(new Listener());`
- ★ Define Listener class that implements the `PlayerListener` interface by defining its **`playerUpdate()`** method

– Player Events Table 8.3

Listing 8.2a : Video Player demo

```
public void playMPG() {
    Player p;
    VideoControl vc;
    try {
        p = Manager.createPlayer
            ("http://localhost:8081/mcom/av/dancers.mpg");
        p.realize();
        vc = (VideoControl)p.getControl("VideoControl");
        if (vc != null) { Form form = new Form("video");
            form.append((Item)vc.initDisplayMode
                (vc.USE_GUI_PRIMITIVE, null));
            Display.getDisplay(this).setCurrent(form);
        }
        p.start();
    } catch (IOException ioe) { }
        catch (MediaException e) { }
    }
}
```

Audio/Video demo



The following code snippet plays a tone (ToneControl.C4) for 4000 milliseconds with a maximum (100) volume level.

```
try {  
    Manager.playTone(ToneControl.C4, 4000, 100);  
} catch (MediaException e) { }
```

A camera has a special URI string 'capture://video' that is used to create its Player.

★ *p = Manager.createPlayer("capture://video");*

★ *VideoControl.getSnapshot(String imageType);*

– is used to capture a camera's picture.

Demo: MMAPI