symbian

An Introduction to Programming the Java APIs for Bluetooth Wireless Technology (JSR 82) on Symbian OS

Author: Martin de Jode
Data: 4/5/2004
Version: 1.0

1. Introducing the Java Bluetooth APIs

In this paper we provide an introduction to programming the Java APIs for Bluetooth Wireless Technology (JABWT). JABWT is an optional J2ME package defined by the Java Community Process under Java Specification Request 82 (JSR 82).

The Bluetooth specification (as defined by the Bluetooth Special Interest Group - SIG) is more than 1500 pages long and covers many layers and profiles. It is not the intention of the JABWT to include them all. Instead JSR 82 implements a core subset of the Bluetooth specification consisting of two packages:

```
javax.bluetooth
javax.obex
```

The Object Exchange (OBEX) protocol is a communication protocol originally defined by the Infrared Data Association (IrDA) that has been adopted by the Bluetooth SIG. Since OBEX is an independent protocol, it is supplied in a separate package. Applications may use the OBEX API independent of the Bluetooth API. The current implementation of JABWT available on Symbian OS (Versions 7.0s, 8.0 and UIQ 2.1) only implements the `javax.bluetooth` package. Support for `javax.obex` will be added in a future release.

The `javax.bluetooth` packages provides APIs for the following:

- registering services
- discovering devices and services
- establishing connections
- providing support for secure connections.

and it is these we will concentrate on in this paper. We shall start off by discussing how to register a Bluetooth service as a server, so as to be able to accept incoming connections from clients. With the server side in place we'll then proceed to look at the client side. First we shall discuss how a client may discover active devices in the neighbourhood, then we'll look at searching devices for a specific service and finally how to connect to that service. We also discuss related issues such as connection security and the available communications protocol in JSR 82.

We shall illustrate these concepts with code highlights from a simple Chat MIDlet written to accompany this paper. This MIDlet allows two JSR 82 enabled devices (or emulators) to exchange short messages over L2CAP.

2. Offering a Bluetooth Service

The steps involved in registering a service are defined in the Service Discovery Protocol (SDP), which is part of the Bluetooth Specification, and are as follows:

- create a service record
- add the service record to the Service Discovery Database
- set security measures associated with connections to clients
- accept connections from clients.

This will be the topic of this section.

2.1. Service Records

Each Bluetooth service offered by a host is represented by a Service Record in the Service Discovery Database (SDDB). To connect to a service a client obtains a Service Record from the server and uses the information therein to connect to the service.

The Java APIs provide an abstraction of the Bluetooth Service Record in the `ServiceRecord` interface. A `ServiceRecord` contains information about the service in a set of attributes in the form of (ID, value) pairs, where the ID is a 16-bit unsigned integer, and the value is an instance of a `DataElement`. A `ServiceRecord` may contain many attributes to fully describe the service being offered. However, only two attributes are required to be present in a service record, the `ServiceRecordHandle` and the `ServiceClassIDList`. Each Bluetooth device offering a service represents an instance of an SDP server. The `ServiceRecordHandle` identifies this `ServiceRecord` within the current instance of an SDP server. Otherwise identical service records, running in different SDP instances, will have different values for the `ServiceRecordHandle`. The `ServiceClassIDList` is a list of UUIDs associated with the service being offered.

2.2. Registering a Service

Sample code to register a service over L2CAP is listed below.

```
private static final String UUID_STRING = "00112233445566778899AABBCCDDEEFF";
…
UUID uuid = new UUID(UUID_STRING, false);
…
try {
    LocaDevice device = LocalDevice.getLocalDevice();
    device.setDiscoverable(DiscoveryAgent.GIAC);
    String URL="btl2cap://localhost:" + UUID_STRING + ";name=L2CAPChat";
    L2CAPConnectionNotifier service =
(L2CAPConnectionNotifier)Connector.open(URL);
    L2CAPConnection conn = service.acceptAndOpen();
    …

} catch(BluetoothStateException bse) {
    bse.printStackTrace();
}catch(InputOutputException ioe){
    ioe.printStackTrace();
}
```

Let's look at this code in more detail. Bluetooth services are uniquely identified by a 128-bit Universally Unique Identifier (UUID). Every service has a UUID including generic low-level protocols and profiles. For convenience the Service Discovery Protocol allows the use of 16-bit or 32-bit alias ("short") UUIDs, with the Bluetooth specification providing an algorithm describing how a 16-bit or 32-bit alias can be promoted to the actual 128-bit UUID.

First, we create a UUID to uniquely identify our service. Here we use a 32 character long `String` representing a 128-bit UUID.

```
private final String UUID_STRING = "00112233445566778899AABBCCDDEEFF";
```

and use it to create a UUID instance using the appropriate constructor.

```
UUID uuid = new UUID(UUID_STRING, false);
```

Note that we pass in false as the second argument as we are creating a long UUID.
Next we use the static factory method getLocalDevice to obtain an instance of the LocalDevice representing the host device.

```
LocaDevice device = LocalDevice.getLocalDevice();
device.setDiscoverable(DiscoveryAgent.GIAC);
```

We use this to make the Bluetooth device discoverable using the setDiscoverable method. The pre-defined public static final int GIAC value represents the General/Unlimited Inquiry Access Code, meaning there are no restrictions placed on which remote devices may discover the host.

To establish a L2CAP server connection we use the "btl2cap://…" URI syntax

```
String URL = "btl2cap://localhost:" + UUID_STRING + ";name=L2CAPChat";
```

The localhost identifier refers to the host device and is the L2CAP server channel identifier. This is added to the ServiceRecord as the ProtocolDescriptorList attribute. It is also necessary to append a String representation of the UUID. This is added to the ServiceRecord in the ServiceClassIDList. An optional name for the service can also be appended to the URL which is added to the ServiceRecord as the ServiceName attribute.

JSR 82 also allows various optional security parameters to be added to connection URLs providing for authentication of remote clients, authorizing access to the service and encryption of data traffic.

Authentication refers to the process of verifying the identity of a remote device. The authentication mechanism in Bluetooth is based on a PIN number shared between devices. A Bluetooth server can require authentication of clients by adding the optional authenticate=true parameter to the connection URL as shown below.

```
String URL =
"btl2cap://localhost:" + UUID_STRING + ";name=L2CAPService;authenticate=true";
```

Bluetooth authorisation is the process by which a server device grants access to a specific service it offers to a specific client. A server can require clients be authorized by adding the authorize=true parameter to the connection URL. Note authorisation implicitly requires authentication so some parameter combinations (e.g. authenticate=false;authorize=true) are forbidden and will result in a BluetoothConnectionException being thrown. On Symbian OS dynamic authorisation is granted to a specific remote device by the user on a per connection request basis via a pop up dialog, which prompts the user to accept or reject the connection. In addition, current Symbian OS devices allows static authorisation by the user of a specific *paired* remote device via the system Bluetooth Control Panel. The remote device becomes *trusted* and all future incoming connections by the trusted device are authorized until static authorisation is revoked via the Bluetooth Control Panel.

A server can require a connection be encrypted by adding the encrypt=true parameter to the connection URL. Note encryption implicitly requires the previous authentication of the remote device.

Once we have constructed an appropriate URL we establish a L2CAP server connection by passing the URL into the Connector.open method as follows:

```
L2CAPConnectionNotifier service = (L2CAPConnectionNotifier)Connector.open(URL);
L2CAPConnection conn = service.acceptAndOpen();
```

The `open` method returns an object of type `L2CAPConnectionNotifier`. Calling `acceptAndOpen` on the on the `L2CAPConnectionNotifier` indicates the server is ready to accept client connections and adds the `ServiceRecord` to the SDDB.

The `acceptAndOpen` method blocks until the server accepts a connection request, returning an `L2CAPConnection` object to enable communication between the client and server.

3. Device Discovery

Often we are interested in connecting to a specific remote device, for instance to play a game against a friend or send a message to a colleague. In this section we show how to perform device discovery.

Device discovery is in fact very simple. First we need to implement a `DiscoveryListener`. The `DiscoveryListener` interface mandates the four call-back methods shown below.

```
public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod)

public void inquiryCompleted(int discType)

public void servicesDiscovered(int transID, ServiceRecord[] servRecord)

public void serviceSearchCompleted(int transID, int respCode)
```

To implement a device `DiscoveryListener` we need to provide non-trivial implementations for the first two methods.

The `deviceDiscovered` method is called by the implementation when a device is discovered. It may be called many times. The implementation passes in a `RemoteDevice` instance representing the remote device just discovered and also a `DeviceClass` instance that provides information about the type of device just discovered, allowing filtering of unwanted devices.

The `inquiryCompleted` method is called by the implementation when the device inquiry has completed. The value of `discType` can have one of three values predefined in the `DiscoveryListener` interface.

```
public static int INQUIRY_COMPLETED
public static int INQUIRY_TERMINATED
public static int INQUIRY_ERROR
```

The first indicates the inquiry terminated normally, the second indicates the inquiry was terminated by the application (via the `cancelInquiry` method of `DiscoveryAgent`) and the last value indicates the inquiry failed to complete normally but was not terminated.

To initiate a device inquiry we need to obtain a `DiscoveryAgent` using the `LocalDevice getDiscoveryAgent` method and invoke the

```
public boolean startInquiry(int accessCode, DiscoveryListener listener)
```

method on it. The `startInquiry` method takes an appropriately implemented `DiscoveryListener` and also an `accessCode` that which can have one of two values pre-defined in the `DiscoveryAgent` class.

```
public static final GIAC
public static final LIAC
```

The first refers to the General/Unlimited Inquiry Access Code, indicates an unlimited search returning all devices found in the vicinity. LIAC refers to the Limited Dedicated Inquiry Access Code. Only remote devices in LIAC discovery mode will be discovered. Since our server is using GIAC we use this value in our device inquiry

```
DeviceDiscoverer deviceDiscoverer = new DeviceDiscoverer(this);
try {
    agent.startInquiry(DiscoveryAgent.GIAC, deviceDiscoverer);
} catch(BluetoothStateException bse){
    status.setText("BSException: " + bse.getMessage());
}
```

A simple implementation of a `DiscoveryListener` for device discovery from our Chat MIDlet example application is listed below.

```
import javax.bluetooth.*;
import java.util.*;

public class DeviceDiscoverer implements DiscoveryListener {

    private ChatController controller;
    private Vector remoteDevices;

    public DeviceDiscoverer(ChatController controller) {
        this.controller = controller;
        remoteDevices = new Vector();
    }

    public void servicesDiscovered(int transID, ServiceRecord[] servRecord){}

    public void serviceSearchCompleted(int transID, int respCode) {}

    public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {
        remoteDevices.addElement(btDevice);
    }

    public void inquiryCompleted(int discType) {
        String message = null;

        if (discType == INQUIRY_COMPLETED) {
            message = "Inquiry completed";
        } else if (discType == INQUIRY_TERMINATED) {
            message = "Inquiry terminated";
        } else if (discType == INQUIRY_ERROR) {
            message = "Inquiry error";
        }

        RemoteDevice[] devices = new RemoteDevice[remoteDevices.size()];
        for(int i = 0; i < remoteDevices.size(); i++) {
            devices[i] = (RemoteDevice)remoteDevices.elementAt(i);
        }
        controller.deviceInquiryFinished(devices, message);

        controller = null;
        remoteDevices = null;
```

```
    }

}
```

For device discovery we need to provide non-trivial implementations for the `deviceDiscovered` and `inquiryCompleted` call-back methods.

Every time a device is discovered the `deviceDiscovered` method is called by the system. There may be many devices in range and we may not be interested in them all so we can use the `DeviceClass` value to filter out unwanted devices (as shown below).

```
public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {
    // The major device class of 0x600 is an imaging device
    if ((cod.getMajorDeviceClass() == 0x600) {
        // The minor device class of 0x80 is a printer
        if ((cod.getMinorDeviceClass() & 0x80) != 0) {
            // The service class of 0x40000 is a rendering service
              if ((cod.getServiceClasses() & 0x40000) != 0) {
                remoteDevices.addElement(btDevice);
          }
        }
    }
}
```

The `DeviceClass` class provides an abstraction of the Class of Device/Service (CoD) record, as defined in the Bluetooth specification Assigned Numbers document (https://www.bluetooth.org/foundry/assignnumb/document/baseband).

When the inquiry is finished the system invokes the `inquiryCompleted` method. Assuming the inquiry completed successfully, we return our filtered `RemoteDevice`s for further processing. The steps involved in a typical device inquiry are shown in the sequence diagram in Figure 1 below.

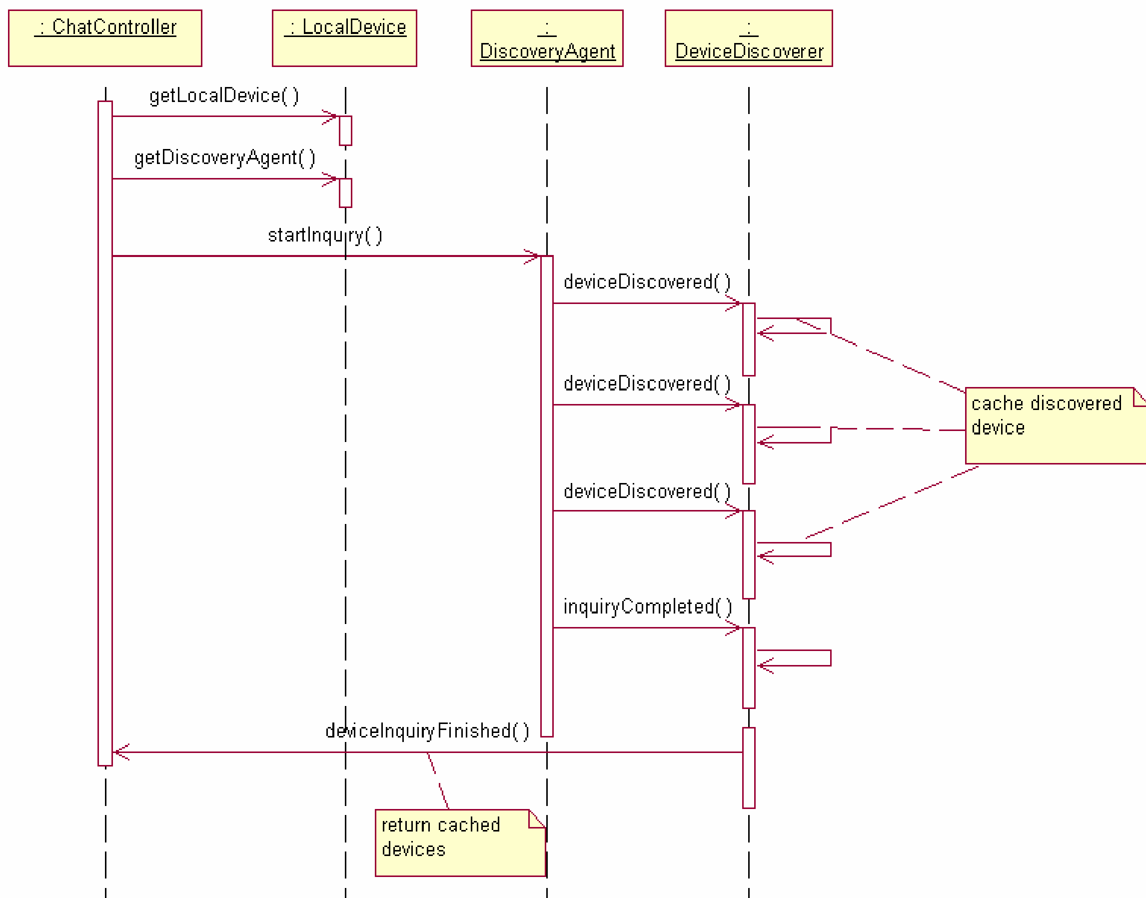


Figure 1 The steps involved in a typical device inquiry

4. Service Discovery

Having discovered the device of interest we will probably want to connect to a service offered by that device. To do this we use the `searchServices` method of `DiscoveryAgent`

```
searchServices(int[] attrSet, UUID[] uuidSet, RemoteDevice btDev,
DiscoveryListener discListener)
```

For example

```
ServiceDiscoverer serviceDiscoverer = new ServiceDiscoverer(this);
int[] attrSet = {0x0100};//return service name attribute
UUID[] uuidSet = new UUID[1];
uuidSet[0] = new UUID(uuidString, false);
try {
    agent.searchServices(attrSet, uuidSet, remoteDevice, serviceDiscoverer);
} catch(BluetoothStateException bse) {
    bse.printStackTrace();
}
```

This takes four parameters. The `int[] attrSet` refers to the attributes to be retrieved from the `ServiceRecord`. If a `null` value is passed then a default attribute set is returned, consisting of the `ServiceRecordHandle` (0x0000), `ServiceClassIDList` (0x0001), `ServiceRecordState` (0x0002), `ServiceID` (0x0003), and `ProtocolDescriptorList` (0x0004). The `uuidSet` is an array of UUIDs that the service that is to be searched for must contain. The `remoteDevice` parameter is the `RemoteDevice` to be searched and the `serviceDiscoverer` is the `DiscoveryListener` that will be notified of the search results. To implement a `DiscoveryListener` for service discovery we need to provide non-trivial implementations for the following call-back methods.

```
public void servicesDiscovered(int transID, ServiceRecord[] servRecord)

public void serviceSearchCompleted(int transID, int respCode)
```

An implementation of a `DiscoveryListener` class for service discovery, taken from the Chat MIDlet accompanying this paper, is listed below.

```
import javax.bluetooth.*;
import java.io.*;

public class ServiceDiscoverer implements DiscoveryListener {

    private static final String SERVICE_NAME = "L2CAPchat";
    private ChatController controller;
    private ServiceRecord serviceRecord;

    public ServiceDiscoverer(ChatController controller) {
        this.controller = controller;
    }

    public void servicesDiscovered(int transID, ServiceRecord[] servRecord) {
        for(int i = 0; i < servRecord.length; i++) {
            DataElement serviceNameElement =
servRecord[i].getAttributeValue(0x0100);
            String serviceName = (String)serviceNameElement.getValue();
            if(serviceName.equals(SERVICE_NAME)){
                serviceRecord = servRecord[i];
                break;
            }
        }
    }

    public void serviceSearchCompleted(int transID, int respCode) {
        String message = null;

        if (respCode == DiscoveryListener.SERVICE_SEARCH_DEVICE_NOT_REACHABLE) {
            message = "Device not reachable";
        }
        else if (respCode == DiscoveryListener.SERVICE_SEARCH_NO_RECORDS) {
            message = "Service not available";
        }
        else if (respCode == DiscoveryListener.SERVICE_SEARCH_COMPLETED) {
            message = "Service search completed";
        }
        else if (respCode == DiscoveryListener.SERVICE_SEARCH_TERMINATED) {
            message = "Service search terminated";
        }
```

```
        else if (respCode == DiscoveryListener.SERVICE_SEARCH_ERROR) {
            message = "Service search error";
        }

        controller.serviceSearchFinished(serviceRecord, message);
        controller = null;
        serviceRecord = null;
    }

    public void inquiryCompleted(int discType){}

    public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod){}

}
```

If the required service is found, the `servicesDiscovered` method will be invoked by the implementation. We can cache the service records corresponding to all services offered by the device corresponding to our requirements. In the case of searches for generic services, such as the Serial Port Profile, there may be more than one instance being offered by the device. In the example code above we filter the discovered service according to the name of the service prior to caching. To ascertain the service name we use the `getAttributeValue` method of `ServiceRecord`. Note the service name attribute (0x0100 for the primary language) is not returned in the default attribute set, so we must have requested it specifically in the attribute set passed into the `searchServices` method.

When the service search is complete (whether successful or not) the `searchCompleted` method is invoked by the system. We can interrogate the `respCode` value to ascertain the success of search, as in the code shown above. The steps involved in a typical service search are shown in the sequence diagram shown in Figure 2

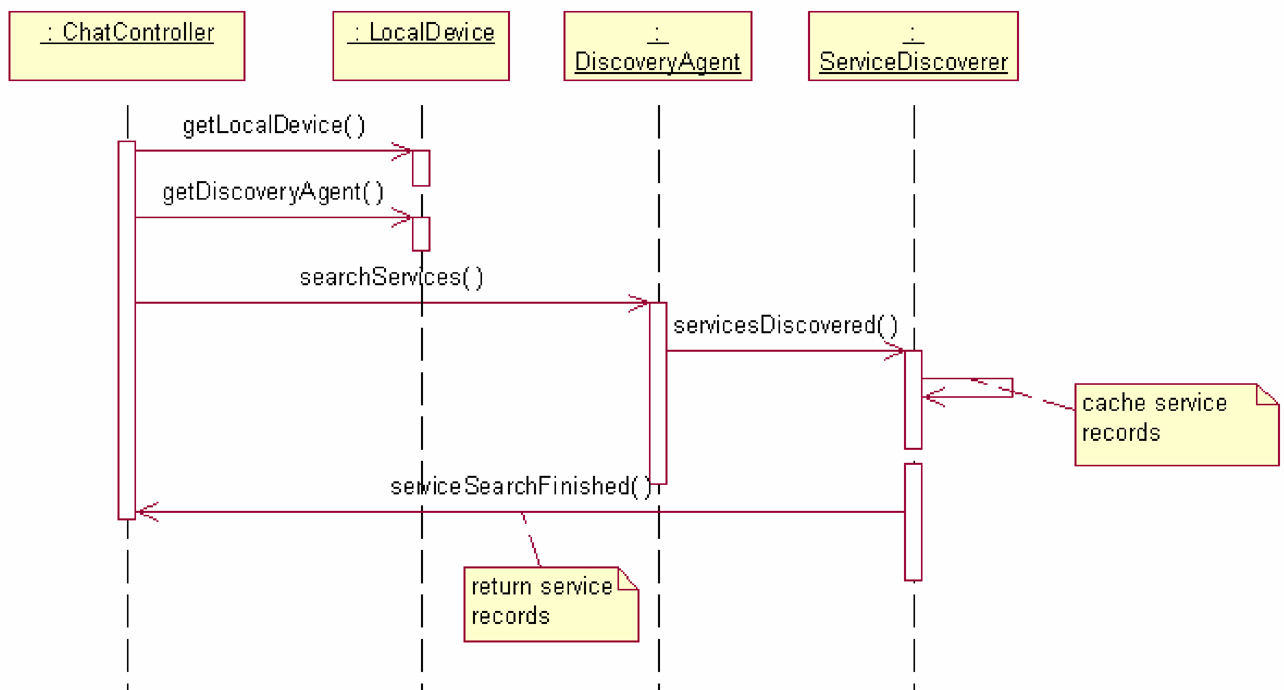


Figure 2 The steps involved in a typical service search

Assuming we successfully discovered our service, we use its `ServiceRecord` to obtain a connection URL, enabling a connection to be established with that service.

5. Connecting to a Service

5.1. Following Service Discovery

Once a `ServiceRecord` has been obtained we use the `getConnectionURL` method to connect to the service represented by that `ServiceRecord`.

```
public String getConnectionURL(int requiredSecurity, boolean mustBeMaster)
```

The `getConnectionURL` method takes two parameters. The first integer value specifies the level of security for the connection. The client can require authentication of the server and if necessary encryption of the connection, using any one of three predefined static integer values defined in the `ServiceRecord` class.

```
public static final int NOAUTHENTICATE_NOENCRYPT
public static final int AUTHENTICATE_NOENCRYPT
public static final int AUTHENTICATE_ENCRYPT
```

This causes the appropriate security parameters (e.g. `authenticate=true; encrypt=true`) to be appended to the returned URL.

The second `boolean mustBeMaster` parameter indicates whether the local device must be *master* in connections to this service. If `false` the local device is willing to be master or slave in the relationship. The current implementation of JSR 82 on Symbian OS only supports a `false` value for the `mustBeMaster` parameter.

To connect to the service we use the URL we have obtained as the argument to the `Connector open` method as shown below.

```
String url =
serviceRecord.getConnectionURL(ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
false);
try{
    L2CAPConnection conn = (L2CAPConnection)Connector.open(url);
    …
}catch(IOException ioe){
    //handle
}
```

So let us recap on the steps involved in connecting to a service. First we perform a device inquiry which returns active Bluetooth devices in the vicinity (possibly filtered). We then perform a service search for a specific service defined by its UUID on a specific device. If successful, the service search will return a `ServiceRecord` which can be used to obtain the connection URL required to open a connection to the service running on that device.

The reader will have observed that JSR 82 takes a “Service Oriented” approach to Bluetooth. Connections are made to services via their service record and UUID. A prerequisite of discovering and connecting to a service is an a-priori knowledge of the UUID of that service. JSR 82 provides no mechanism to browse the available services offered by a device.

5.2. *The Quick Way*

Normally users will want to connect to a specific device so the approach outlined above is the appropriate one. However, JSR 82 does offer an alternative method for connecting to a service that doesn't require a device inquiry or indeed any searches at all (so no `DiscoveryListener`s require implementing), namely the `selectService` method of the `DiscoveryAgent` class (shown below).

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

This method simply takes the `UUID` of the service required, an `int` indicating the level of security for the connection and the master/slave `boolean` indicator. The method will search for the service denoted by the `UUID` on any devices in range. If the service is found, a `String` representing the URL to be used to connect to the service via the `Connector open` method is returned. If no service is found a value of `null` is returned. If there are many devices in the area offering the required service a connection URL may be returned to any one of them (it is not possible to specify which).

Note, as this method only takes a single `UUID,` it is best employed to search for a specific `UUID` created to denote a specific service (rather than generic services such as the Serial Port Profile, which may be offered by many devices). Unfortunately at the time of writing implementations of JSR 82 available on current Symbian OS based phones do not support the `selectService` method, a `null` value always being returned. This will be addressed in a future release.

6. Connection Protocols

JSR 82 specifies support for three communication protocols: L2CAP, RFCOMM and OBEX. L2CAP is a packet-based communications protocol, whereas RFCOMM is a stream-based, wire replacement protocol. OBEX, the Object Exchange protocol, can be used to send and receive complete objects (e.g. files, images etc), but is unsupported on Symbian OS at present. We will look at L2CAP first as it's the protocol used in the sample Chat MIDlet accompanying this paper.

6.1. *Logical Link Control and Adaption Protocol (L2CAP)*

The Logical Link Control and Adaption Protocol (L2CAP) is the lowest level data transmission protocol in the Bluetooth stack. Data is sent and received in packets, whose maximum size is defined by the Maximum Transmission Unit (MTU). The default value for the MTU is 672 bytes, however, the JABWT allows the developer to recommend an alternative value for the connection MTU used for transmitting and receiving data, in the connection URL using the optional `ReceiveMTU` and `TransmitMTU` parameters. For example

```
String url =
"btl2cap://localhost:" + UUID_STRING + ";ReceiveMTU=512;TransmitMTU=512";
```

The actual MTU arrived at for the connection will be the lowest common denominator supported by both the local and remote devices. Symbian's implementation of the JABWT limits the maximum value for the MTU to 672 bytes (same as the default value).

To establish a server L2CAP connection we would use the following code:

```
L2CAPConnectionNotifier notifier = (L2CAPConnectionNotifier)Connector.open(url);
L2CAPConnection conn = notifier.acceptAndOpen();
```

A client could connect to this service as follows:

```
L2CAPConnection conn = (L2CAPConnection)Connector.open(url);
```

Once a connection has been established we can send and receive packets of data using the `send` and `receive` methods of `L2CAPConnection`. For example

```
if(conn.ready()){
    byte[] data = new byte[conn.getReceiveMTU()];
    int length = conn.receive(data);
    String message = new String(data, 0, length);
    System.out.println(message);
}
```

The `ready` method returns `true` if a call to `receive` will not block. Note if the size of the data buffer is less than the value of `ReceiveMTU` for the connection then any data in the package that exceeds the buffer size will be lost.

Similarly when sending a packet, if the data buffer size exceeds the `TransmitMTU` value, any excess bytes will be lost.

```
byte[] data = message.getBytes();
int transmitMTU = conn.getTransmitMTU();
if(data.length <= transmitMTU){
    conn.send(data);
}
…
```

6.2. RFCOMM and the Serial Port Profile

The RFCOMM layer provides a stream–based connection protocol analogous to serial port communication and is the basis of the Serial Port Profile (SPP). We would set up a server connection using the SPP and RFCOMM as follows:

```
String url = "btspp://localhost:" + UUID_STRING;
StreamConnectionNotifier notifier = (L2CAPConnectionNotifier)Connector.open(url);
StreamConnection conn = notifier.acceptAndOpen();
DataInputStream input = conn.openDataInputStream();
…
DataOutputStream output = conn.openDataOutputStream();
…
input.close();
output.close();
conn.close();
```

Since RFCOMM connections are stream-based, communication is performed in the standard manner using `InputStream`s and `OutputStream`s.

To make a client connection to a SPP service we would obtain a connection URL and use it to open a `StreamConnection` in the following manner.

```
StreamConnection conn = (StreamConnection)Connector.open(url);
DataOutputStream output = conn.openDataOutputStream();
…
output.close();
conn.close();
```

More information on programming RFCOMM from JSR 82 can be found in Programming J2ME for Symbian OS as well as in the additional resources listed at the end of the paper.

6.3. *OBEX*

The final communication protocol supported by JSR 82 is OBEX. OBEX is the highest level communication protocol available in JSR 82, sitting on top of RFCOMMM and L2CAP.

Code for opening establishing an OBEX server is shown below.

```
String url = "btgoep://localhost:" + UUID_STRING;
SessionNotifier notifier = (SessionNotifier)Connector.open(url);
notifier.acceptAndOpen(requestHandler);
```

The `acceptAndOpen` method takes an instance of `javax.obex.ServerRequestHandler` as a parameter. This class defines an event listener that will respond to OBEX requests made to the server. However, we will recall from the introduction that the `javax.obex` package is currently not part of Symbian's JSR 82 implementation. So all attempts to open a connection using the "`btgoep://`…" URI syntax will result in a `ConnectionNotFoundException`.

7. JABWT and the MIDP 2.0 Security Model

All current Symbian OS based phones implementing JSR 82 also support MIDP 2.0. Under the MIDP 2.0 security model access to certain, security sensitive, APIs is restricted (so called *protected APIs*). Signed MIDlet suites containing MIDlets that require access to protected APIs must explicitly request permission to do so in the MIDlet suite's `MIDlet-Permissions` attribute. Under the "Recommended Security Policy for GSM/UMTS Compliant Devices" addendum to the MIDP 2.0 specification client or server Bluetooth connections are deemed as protected API calls. Therefore a signed MIDlet suite which contains MIDlets which open Bluetooth connections must request permission to do so. For example to open client and server connections the following entry is required.

```
MIDlet-Permissions: javax.microedition.io.Connector.bluetooth.client,
javax.microedition.io.Connector.bluetooth.server
```

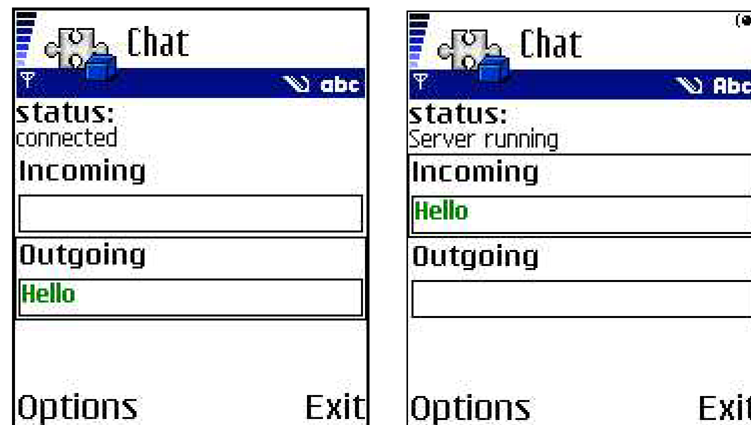
If the security policy relating to the protection domain to which the signed MIDlet suite would be bound grants the requested permissions the MIDlet suite can be installed and the MIDlets it contains will be able to open Bluetooth client and server connections, either automatically (*Allowed* permission), or with explicit user permission (*User* permission), depending upon the security policy in effect on the device. The Bluetooth protected APIs form part of the Local Connectivity function group as defined in the "Recommended Security Policy for GSM/UMTS Compliant Devices". The security policy in effect for MIDlet suites bound to the trusted protection domain on the Sony Ericsson P900/908 (firmware version R2*** of later) grants automatic access to the Local Connectivity function group. In the case of the Nokia 6600 (firmware version 4.09.1 or later) MIDlets bound to the Third Party Trusted protection domain can access the Local Connectivity function group with *User* permission, the default being *Session* (permission is only requested the first time the MIDlet accesses the protected API and, if granted, remains in force until the MIDlet closes), but user configurable to *Blanket* (permission is granted until the MIDlet is uninstalled, or the user revokes it) or disallowed all together.

Whether MIDlets in untrusted MIDlet suites (ie unsigned MIDlets) can open Bluetooth connections depends on the security policy relating to the Local Connectivity function group for the *untrusted* domain in force on the device. On current Symbian OS based phones untrusted MIDlets can access these APIs with *User* permission, the default being set to *Session*.

The MIDP 2.0 security model relates to whether the installed MIDlet is trusted to access the *local* resources of the phone. This is orthogonal to connection security discussed in Sections 2 and 5 which relates to connections to or from remote devices.

8. L2CAP Chat Sample Code

The Chat MIDlet can be run in either client or server mode. In client mode the user can search for a list of active Bluetooth devices in the vicinity. The user can then select a device and search for the L2CAP Chat service on that device. If discovered the user can connect to the Chat service and commence a conversation (see Figure 1).



An L2CAP Chat client chatting to the L2CAP Chat server

The full source code and JAR and JAD files can be downloaded from here. It can be run under simulation in the Sun J2ME Wireless Toolkit using the Nokia Series 60 MIDP Concept SDK Beta 0.3 emulator as the device, or on actual phones such as the Nokia 6600 or the Sony Ericsson P900/908.

9. Discussion

In this paper we have provided an introduction to programming the `javax.bluetooth` package of the JABWT (JSR 82) as implemented on Symbian OS. We have looked at registering a service, performing device and service discovery, opening connections to services and security issues. We have also provided a fully-functioning Chat MIDlet to illustrate the concepts discussed. The aim of this paper is to cover the key features offered by JSR 82 on Symbian OS, giving developers enough information to start programming the JABWT. For a detailed description of the API the reader is directed to the javadoc for JSR 82. For more information on programming JSR 82 see "Programming J2ME on Symbian OS" as well as the other resources listed at the end of the paper.

10. Further Resources

"Programming Java 2 Micro Edition on Symbian OS: A developer's guide to MIDP 2.0", Martin de Jode; Wiley (June 2004), ISBN: 0-470-09223-8.
(http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470092238.html)

The Java APIs for Bluetooth Wireless Technology (JSR 82) specification and javadoc (http://jcp.org).
Specification of the Bluetooth System, Volume 1. Bluetooth SIG (https://www.bluetooth.org/spec).

"The Java APIs for Bluetooth Wireless Technology"
(http://developers.sun.com/techtopics/mobility/midp/articles/bluetooth2/index.html)

"Controlling Bluetooth Mini Race Car from the P900"
(http://developer.sonyericsson.com/site/global/techsupport/tipstrickscode/java/p_tips_java_1202.jsp)

A useful 3rd Party JABWT resource (http://benhui.net/bluetooth).

Sun's J2ME Wireless Toolkit 2.0 (http://java.sun.com/products/j2mewtoolkit).

Series 60 MIDP Concept SDK Beta 0.3.1 (http://forum.nokia.com).

Want to be kept informed of new articles being made available on the Symbian Developer Network?
Subscribe to the Symbian Community Newsletter.
The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.