# *4. Java y Bluetooth*

El lenguaje Java posee una serie de ventajas, que hacen que sea muy adecuado para el desarrollo de aplicaciones que hagan uso de la tecnología Bluetooth de los dispositivos. Entre estas podemos citar la portabilidad, el rápido desarrollo por ser un lenguaje de alto nivel con gran abstracción, verificación de clases, seguridad, gran comunidad de desarrolladores, entre otras. Por ello se creó una API estándar que permite usar la tecnología Bluetooth, de forma que sea uno de los paquetes opcionales para dispositivos J2ME basados en CLDC. El nombre que adopta es *Java APIs for Bluetooth Wireless Technology* (JABWT). La especificación correspondiente desarrollada por el JCP es la JSR-82 [E7].

## 4.1. Descripción de JABWT

### 4.1.1. Arquitectura de JABWT

La funcionalidad que proporciona JABWT se puede dividir en tres categorías: descubrimiento, comunicación y gestión de dispositivos. En la primera se incluye el descubrimiento de dispositivos y servicios, así como el registro de servicios. La comunicación permite la conexión entre dispositivos para la comunicación Bluetooth entre aplicaciones. Esta comunicación se puede realizar sobre diferentes protocolos, ya sea RFCOMM, L2CAP o el protocolo adoptado para intercambio de objetos OBEX. Por último, la gestión de dispositivos permite el control y gestión de estas conexiones, ocupándose de los estados y propiedades tanto del dispositivo local como del remoto. Además facilita los mecanismos de seguridad para las conexiones.

#### a) CLDC, MIDP y JABWT

Los dispositivos MIDP son los primeros en incorporar JABWT. En la figura 4.1 se muestra como encaja JABWT dentro de una arquitectura CLDC + MIDP. Pero aunque aquí hablemos de JABWT sobre un dispositivo MIDP, no quiere decir que la API de JABWT forme parte de MIDP. De hecho, JABWT puede ser utilizado en otros tipos de dispositivos.

En la parte más baja del diagrama se encuentra el sistema operativo del dispositivo, que contiene la pila de protocolos de Bluetooth, así como otras librerías usadas internamente o por aplicaciones nativas del sistema. Vemos que las aplicaciones Bluetooth nativas del sistema interactúan directamente con este *software* del sistema. La implementación de CLDC/KVM, que se encuentra directamente sobre esta capa, proporciona el entorno de ejecución sobre el que se apoyarán el resto de APIs de Java. Dos de estas APIs serán las que ahora nos interesan: JABWT especificada en la JSR-82 y MIDP definida en JSR-37 y JSR-118.

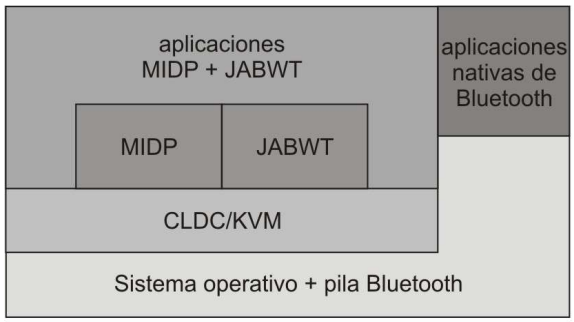


*Figura 4.1 Arquitectura CLDC + MIDP + Bluetooth*

## b) Paquetes

JABWT consta básicamente de dos APIs. Una para el manejo de la tecnología Bluetooth en general. Pero además incluye otra para el protocolo OBEX. Esta API viene por separado debido a que el protocolo OBEX no es específico de Bluetooth, sino que también se puede utilizar sobre otros tipos de transportes como IrDA, USB o TCP. Por este motivo en JABWT se definen dos paquetes diferentes:

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

Ambos se encuentran dentro del paquete `javax.microedition.io`. Cada uno representa un paquete opcional separado. Eso quiere decir que una aplicación que represente una implementación de CLDC puede que no incluya a ninguno de los dos, a uno solo de ellos o los dos.

## c) Modelo Cliente – Servidor

Un servicio Bluetooth es una aplicación que proporciona algún tipo de asistencia a los clientes mediante la comunicación por Bluetooth. Esta asistencia, por llamarlo de alguna manera, suele ser alguna funcionalidad de la que el dispositivo cliente no dispone. Algunos ejemplos de estos servicios son el de impresión, servidor de acceso a LAN, servidor de archivos, etc. Todos

estos servicios entre otros muchos más se encuentran descritos en la especificación de perfiles de Bluetooth. JABWT permite desarrollar aplicaciones que implementen estos perfiles o cualquier otro servicio propio. En cualquier caso, el servidor pone estos servicios a disposición del cliente definiendo lo que se denomina *service record*, donde se describe el servicio, y añadiéndolo a la base de datos de descubrimiento de servicios (SDDB) del dispositivo local.

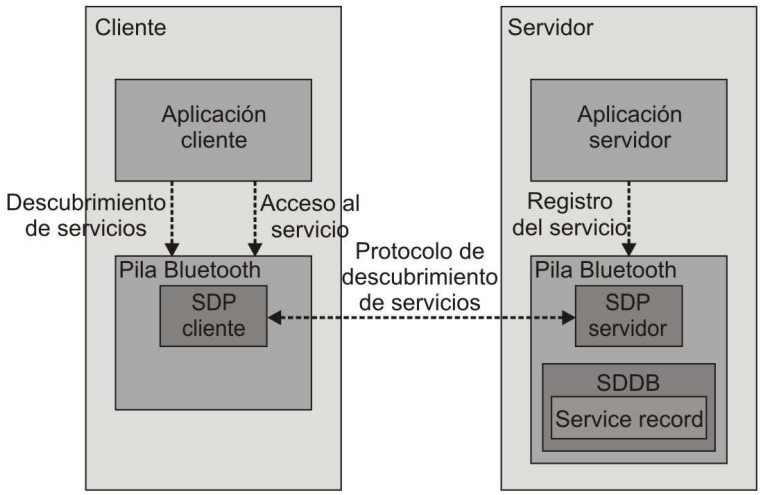


*Figura 4.2 Descubrimiento de servicios*

La figura 4.2 muestra los componentes Bluetooth implicados en el registro y descubrimiento de servicios. Los servicios se descubren con ayuda del protocolo SDP. La aplicación del servidor almacenará un *service record* en la SDDB, y a partir de entonces comenzará a esperar a que la aplicación del cliente contacte con él para acceder al servicio. La pila Bluetooth proporciona un servidor SDP, que es el que se encargará de mantener la base de datos. Por otro lado la aplicación del cliente podrá hacer uso del SDP para buscar algún servidor SDP que le interese. El *service record* del servidor le proporcionará al cliente la información suficiente para conectarse al servicio. Una vez encontrado el servicio, cliente y servidor establecerán una conexión para hacer uso de él.

### 4.1.2. BCC

El Centro de Control de Bluetooth (BCC) es parte de la especificación JABWT, pero no hay ninguna API de Java que proporcione acceso directo a él. La función del BCC es la de controlar las distintas aplicaciones Bluetooth del dispositivo evitando que haya conflictos entre ellas. El BCC es la autoridad central del dispositivo en cuanto a la configuración local del dispositivo se refiere. Los detalles del BCC dependen de la implementación, pudiendo dejar mayor o menor acceso al usuario. En general, las tareas específicas del BCC son las siguientes:

- Resolver conflictos entre aplicaciones: Varias aplicaciones se pueden estar ejecutando al mismo tiempo, pudiendo acceder al dispositivo local a la vez de forma que se produzcan

conflictos. En el momento que dos aplicaciones hacen uso de JABWT, puede haber conflictos por el hecho de acceder al mismo dispositivo Bluetooth. Por ejemplo puede que ambas aplicaciones pretendan establecer distintas configuraciones de seguridad o visibilidad. El BCC será el que se encargue de resolver todos estos conflictos.

- Facilitar la modificación de propiedades del dispositivo local: Aunque JABWT permita acceder a algunas propiedades del dispositivo local, no proporciona ningún método para modificarlas directamente. En particular el BCC es el que controla la configuración del nombre, clase de dispositivo, lista de dispositivos "preconocidos", lista de dispositivos de confianza, requerimientos mínimos de seguridad, y soporte para los distintos modos de conectabilidad y visibilidad.
- Tratamiento de operaciones de seguridad: Normalmente estas operaciones requieren una interacción con el usuario, por ejemplo cuando se requiere que se introduzca un código PIN. El BCC es el responsable de adquirir esa información del usuario e introducirla en el proceso de seguridad de Bluetooth.

## 4.2. Programación con JABWT

En este apartado se explicará como realizar una aplicación con JABWT. Se introducirán gran parte de las clases incluídas en el paquete `javax.bluetooth` y se verá cómo se usan. Previamente se planterá cuáles son los componentes básicos de una aplicación Bluetooth. A continuación, en lugar de ver cada clase o interfaz individualmente y por separado, se explicará como desarrollar aplicaciones realizando estos componentes básicos con ayuda de la API. Así se irán presentando las diferentes clases e interfaces de ésta.

De este modo en una aplicación Bluetooth se podrán realizar las siguientes operaciones básicas:

- Inicialización
- Gestión de dispositivos
- Descubrimiento de dispositivos
- Descubrimiento y registro de servicios
- Comunicación

### 4.2.1. Inicialización

Lo primero a realizar es obtener una referencia del gestor de Bluetooth mediante el objeto `LocalDevice`, que proporciona métodos para obtener información del dispositivo local y permitirá

iniciar los diferentes procesos de descubrimiento. Este objeto tiene un constructor privado, evitando así que una aplicación cree un objeto `LocalDevice` por su cuenta, por lo que la forma de obtenerlo será mediante el método `LocalDevice.getLocalDevice()`. Este método lanzará la excepción `BluetoothStateException` si el módulo Bluetooth no se puede iniciar.

Como decíamos `LocalDevice` permite obtener información acerca del dispositivo local. Así por ejemplo los métodos `getBluetoothAddress()` y `getFriendlyName()`, devolverán la dirección y el nombre del módulo Bluetooth respectivamente. Pero no sólo eso, sino que además nos permitirá configurar la visibilidad del dispositivo, útil en el caso de la aplicación del servidor. Esto es posible mediante el método `setDiscoverable()`. Este método tómara como argumento alguna de las constantes de la tabla 4.1 que hacen referencia a los diferentes estados de visibilidad que puede adoptar el dispositivo. Estas constantes se definen en la clase `DiscoveryAgent` que veremos a continuación. El método podrá lanzar la excepción `BluetoothStateException` en el caso de que no se pueda establecer el modo deseado.

| *Modo* | *Nombre completo* | *Descripción* |
|---|---|---|
| `NOT_DISCOVERABLE` | Not Discoverable | No será visible para ningún otro dispositivo |
| `GIAC` | General Inquiry | Visible para todos los demás dispositivos |
| `LIAC` | Limited Inquiry | Visible por un corto período de tiempo, típicamente 1 minuto. Después pasará a estado invisible |

*Tabla 4.1 Modos de visibilidad*

Por otro lado, en el caso del cliente, será necesario obtener una referencia al objeto `DiscoveryAgent`, que será el que le propocione los métodos necesarios para realizar descubrimientos de dispositivos y servicios. Cada dispositivo tiene un único objeto de este tipo, que podrá ser obtenido mediante el método `LocalDevice.getDiscoveryAgent()`. Este método podrá lanzar la excepción `BluetoothStateException`.

Veamos como quedaría el código que inicializaría la aplicación Bluetooth:

```
private LocalDevice local;
private DiscoveryAgent discAgent;
...
// inicializa Bluetooth
try {
    // obtiene referencia al dispositivo local
    local = LocalDevice.getLocalDevice();
    // Establece modo de visibilidad General para el servidor
    local.setDiscoverable(DiscoveryAgent.GIAC);
    // Se obtiene el objeto DiscoveryAgent para el cliente
    discAgent = local.getDiscoveryAgent();
} catch(BluetoothStateException e) {
...
}
```

### 4.2.2. Gestión de dispositivos

Como hemos visto en el apartado anterior la clase `LocalDevice` permite obtener información del dispositivo local gracias a los distintos métodos que proporciona. Análogamente existe la clase `RemoteDevice` que ofrece una serie de métodos para recuperar información de algún dispositivo remoto. Una referencia a este objeto se obtiene en general durante el proceso de descubrimiento. También se puede obtener mediante el método `getRemoteDevice()` que ofrece la clase `RemoteDevice`, que toma como parámetro la conexión con el dispositivo remoto. El método podrá lanzar `IOException` si la conexión está cerrada, o `IllegalArgumentException` si el parámetro no es una conexión Bluetooth.

Una vez que se tenga el objeto `RemoteDevice` se pueden invocar toda una variedad de métodos para obtener información del dispositivo. Por ejemplo `getBluetoothAddress()` devolverá la dirección Bluetooth del dispositivo, y `getFriendlyName()` dará el nombre del dispositivo.

La clase `RemoteDevice` proporciona además métodos para configurar distintos aspectos de seguridad con el dispositivo remoto como autentificación, encriptado y autorización. Esto se podrá realizar mediante los métodos `authenticate()`, `encrypt()` y `authorize()` respectivamente, una vez que la conexión se haya establecido.

Otra clase bastante útil a la hora de tratar con los distintos dispositivos que puede haber en el entorno es `DeviceClass`. Esta clase nos permitirá saber de qué tipo de dispositivo se trata. Existe toda una clasificación en la que se incluyen entre muchos otros ordenadores portátiles, PDAs, teléfonos o puntos de acceso.

Para conocer más acerca de las clases `LocalDevice`, `RemoteDevice` y `DeviceClass` se puede consultar la especificación JSR-82 [E7].

### 4.2.3. Descubrimiento de dispositivos

JABWT permite el descubrimiento de dispositivos y servicios mediante la clase `DiscoveryAgent` y la interfaz `DiscoveryListener`. Como ya hemos dicho cada dispositivo tiene un único objeto `DiscoveryAgent`, que es el que provee los métodos para realizar la búsqueda. Anteriormente ya vimos cómo se obtenía el objeto `DiscoveryAgent` mediante el método `getDiscoveryAgent()`. La interfaz `DiscoveryListener` se usa para devolver a la aplicación los dispositivos y servicios que van siendo encontrados por el `DiscoveryAgent`.

La manera más fácil y rápida de recuperar una lista de dispositivos (`RemoteDevice`) es mediante el método `retrieveDevices()`, que devolverá una lista de dispositivos preconocidos o almacenados en *caché* según el parámetro pasado sea `DiscoveryAgent.PREKNOWN` o `DiscoveryAgent.CACHED` repectivamente. Dispositivos preconocidos son aquellos con los que

comúnmente se interactúa. Los dispositivos almacenados en *caché* son aquellos que se han encontrado en una búsqueda (*inquiry*) anterior. Los dispositivos obtenidos mediante este método puede que no estén en el entorno y que, por lo tanto, no se pueda establecer una conexión con ellos.

Los dispositivos que realmente se encuentran en el entorno se podrán obtener realizando una búsqueda o *inquiry*. Para que una aplicación pueda realizar un *inquiry* lo primero necesario es que implemente la interfaz `DiscoveryListener`, y en particular sus métodos `deviceDiscovered()` e `inquiryCompleted()` que veremos más adelante. Para comenzar la búsqueda la clase `DiscoveryAgent` proporciona el método `startInquiry()`, que tomará como parámetros el tipo de *inquiry* y una implementación de la interfaz `DiscoveryListener`.

El tipo de *inquiry* `DiscoveryAgent.GIAC` localizará a todos los dispositivos de la zona independientemente de si tienen modo de visibilidad general o limitada. Un *inquiry* limitado se realizará pasando `DiscoveryAgent.LIAC` como parámetro, y sólo localizará a dispositivos cuya visibilidad sea limitada. El método `startInquiry()` devolverá `true` si se pudo iniciar el *inquiry*, y `false` si el código de tipo es correcto pero no es soportado por el dispositivo. Además podrá lanzar la excepción `IllegalArgumentException` si este código no es correcto, o `BluetoothStateException` si el *inquiry* no se pudo llevar a cabo debido a que el dispositivo no lo permite en ese momento.

Una vez comenzado el proceso de búsqueda con `startInquiry()`, el método `deviceDiscovered()` de la interfaz `DiscoveryListener` será llamado cada vez que se encuentre un dispositivo. El evento `deviceDiscovered()` proporciona el objeto `RemoteDevice` y `DeviceClass` correspondiente al dispositivo encontrado. Estos objetos nos permitirán, como ya hemos visto, obtener información acerca del dispositivo, que nos será útil por ejemplo, si después se desea realizar una conexión con él.

JABWT provee el método `cancelInquiry()` para detener el proceso de *inquiry*. Esto se suele llevar a cabo normalmente porque se haya encontrado un dispositivo particular, o cuando la aplicación se va a suspender o terminar. Este método tomará como parámetro el objeto `DiscoveryListener` utilizado también para comenzar el *inquiry*, lo que evitará que una aplicación pueda cancelar la búsqueda comenzada por otra. Los valores devueltos serán `true` si se pudo cancelar, y `false` si no fue posible debido a que no se encontró *inquiry* asociado al `DiscoveryListener` pasado como parámetro.

Cuando termina el *inquiry* se comunica mediante el evento `inquiryCompleted()`. Este evento proporciona la razón de que acabase el *inquiry* como parámetro del método. Entre otros, este parámetro podrá ser `INQUIRY_COMPLETED` si terminó correctamente, `INQUIRY_TERMINATED` si acabó porque fue cancelado mediante el método `cancelInquiry()`, o `INQUIRY_ERROR` si ocurrió algún error durante el proceso.

A continuación se expone un ejemplo que realiza una búsqueda de dispositivos. Primero obtiene los dispositivos preconocidos y de la *caché* mediante el método `retrieveDevices()`, y después realiza un *inquiry* para buscar nuevos dispositivos. A medida que se van encontrando se van mostrando las direcciones Bluetooth en una lista. Cuando termina el *inquiry* se muestra una alerta que lo indica. En el ejemplo sólo se muestran las porciones de código que resulten relevantes para este apartado. Se omitirá lo anteriormente visto, como por ejemplo la inicialización de los objetos Bluetooth.

```
public class BuscadorDispositivos
       extends MIDlet
       implements DiscoveryListener, CommandListener {

    ...
    // constructor
    public BuscadorDispositivos() {
        ...
        listaDispositivos = new List("Dispositivos", List.IMPLICIT);
        display.setCurrent(listaDispositivos);
    }


    public void startApp() {
        ...
        // recupera dispositivos preconocidos y de la caché
        anadeDispositivos();

        // inicia inquiry
        try {
            discAgent.startInquiry(DiscoveryAgent.GIAC, this);
        } catch (BluetoothStateException e) {
            ...
        }
    }

    // Al encontrar un dispositivo se anade a la lista
    public void deviceDiscovered(RemoteDevice remoto, DeviceClass clase) {
        listaDispositivos.insert(0, remoto.getBluetoothAddress(), null);
    }

    // cuando termina inquiry se muestra alerta que lo indica
    public void inquiryCompleted(int param) {
        Alert info = new Alert(null, "Inquiry completado", null, AlertType.INFO);
        info.setTimeout(2000);
        display.setCurrent(info, listaDispositivos);
    }
    ...

    // recupera dispositivos preconocidos y de cache
    private void anadeDispositivos() {
        RemoteDevice[] listaRemotos =
               discAgent.retrieveDevices(DiscoveryAgent.PREKNOWN);

        if(listaRemotos != null) {
            for(int i = 0; i < listaRemotos.length; i++) {
                listaDispositivos.insert(0, listaRemotos[i].getBluetoothAddress(),
                                         null);
            }
        }

        listaRemotos = discAgent.retrieveDevices(DiscoveryAgent.CACHED);
        if(listaRemotos != null) {
            for(int i = 0; i < listaRemotos.length; i++) {
                listaDispositivos.insert(0, listaRemotos[i].getBluetoothAddress(),
                                         null);
            }
```

```
            }
        }
    }
```

### 4.2.4. Descubrimiento de servicios

Una vez que se han encontrado los dispositivos que hay en el entorno, habrá que descubrir cuáles son los servicios que es capaz de ofrecer un dispositivo remoto. El proceso de búsqueda de servicios se llevará a cabo entre dos dispositivos en concreto. Para que un dispositivo pueda saber si otro ofrece un servicio determinado, le tendrá que preguntar si tiene un servicio definido por un *Service Record* con un conjunto de atributos específicos. Si el dispositivo remoto lo posee, le devolverá el *Service Record* que lo describe.

El descubrimiento de servicios sigue el modelo Cliente-Servidor, de forma que el cliente enviará una petición de búsqueda de servicios al servidor. Éste determinará si tiene algún servicio con las características solicitadas. Para esto buscará en su Base de Datos de Descubrimiento de Servicios (SDDB) que forma parte de la pila de Bluetooth. Esto ya se ilustró en la figura 4.2.

El *Service Record* consta de múltiples atributos de diferentes tipos, que proporcionan la información de un servicio específico. Los atributos especifican entre otras cosas el nombre, una descripción, o indicación de cómo conectarse al servicio. Cuando una aplicación busca un servicio, tendrá que proporcionar el conjunto de UUIDs que hay que buscar. Un UUID es una secuencia de bits que identifica a un atributo determinado. Hay algunos UUID definidos en la especificación Bluetooth, pero también se pueden utilizar UUIDs propios.

JABWT proporciona la API necesaria para llevar a cabo dos procesos claramente diferenciados que forman parte del descubrimiento de servicios. Por un lado permitirá al cliente buscarlos, y por otro proporcionará los métodos necesarios para que el servidor realice el registro del servicio. Además de esto, se verá a continuación otra forma más simple de ralizar la búsqueda de dispositivos y servicios al mismo tiempo.

#### a) Registro de un servicio

Esta sección describe como el servidor debe crear el registro de un servicio mediante JABWT. En la realización del registro se engloban las siguientes tareas que la API de Java será capaz de cubrir:

- Creación del *Service Record* que describe un servicio.
- Adición del *Service Record* a la SDDB para que sea visible a los clientes.
- Actualización del *Service Record* si cambian las características.
- Eliminación o deshabilitación del *Service Record* cuando el servicio ya no esté disponible.

Además de los métodos pertenecientes a JABWT, tendrán especial importancia los métodos `open()`, `acceptAndOpen()`, y `close()`. Estos métodos forman parte del GFC definido en CLDC y se usan para las operaciones de E/S. Se verán más en detalle en el capítulo 4.2.5 donde se explica como se establece una conexión.

- **Creación del Service Record:**

La creación del *Service Record* se produce directamente al utilizar el método `open()` de la clase `Connector` cuando el parámetro que se le pasa, que no es más que la cadena de conexión, comienza por `btspp://localhost:`, `btgoep://localhost:`, o `btl2cap://localhost:`. Estas cadenas de conexión son las que utiliza un servidor Bluetooth para abrir una conexión RFCOMM, OBEX o L2CAP respectivamente. De este modo al ejecutar la línea siguiente se creará un *Service Record* para un servicio que utiliza conexión RFCOMM:

```
Connector.open("btspp://localhost:123456789ABC;name=MyService");
```

El *Service Record* generado al ejecutar este método con estos parámetros tendrá cuatro atributos. Cada atributo se compone de un ID de atributo y de su valor. Hay IDs definidas por la especificación del protocolo SDP, pero la mayoría de los valores están libres para definir atributos propios. Cada valor de atributo es un DataElement, que es una estructura de datos que contiene el tipo y el valor del dato. El primer atributo, que toma el nombre de ServiceClassIDList, es el UUID del servicio (`123456789ABC` en este ejemplo). El segundo es el ProtocolDescriptorList e indica al cliente cómo conectarse al servicio. El tercero es el ServiceName que toma el valor del nombre que le hemos dado al servicio en la cadena de conexión. El último es el ServiceRecordHandle, que tiene una gran importancia para la implementación del descubrimiento de servicios, pero que el programador debe ignorar.

Las cadenas correspondientes a los protocolos L2CAP y OBEX crearán *Service Records* similares pero con algunas diferencias. Información más detallada acerca del *Service Record* se puede encontrar en la especificación de Bluetooth [E1].

- **Adición del Service Record a la SDDB:**

El método `Connector.open()` ha generado un *Service Record*, pero éste todavía no será visible para los clientes. El servidor que lo ha creado puede acceder a él y modificarlo antes de hacerlo visible. Por esta razón, la implementación de JABWT no añadirá el *Service Record* a la SDDB hasta que la aplicación llame a `acceptAndOpen()`. A partir de ese momento formará parte de la SDDB y será visible para los clientes.

Un servicio RFCOMM puede aceptar múltiples conexiones de diferentes clientes llamando repetidas veces a `acceptAndOpen()` del mismo objeto `Notifier`. Todos los clientes acceden al mismo *Service Record* y se conectan usando el mismo canal RFCOMM. Si el sistema Bluetooth

del dispositivo no acepta conexiones múltiples, `acceptAndOpen()` lanzará una excepción del tipo `BluetoothStateException`.

El método `acceptAndOpen()` lanzará `ServiceRegistrationException` si existe un error al intentar añadir el *Service Record* a la SDDB.

- **Eliminación del Service Record de la SDDB:**

El *Service Record* será directamente eliminado al llamar al método `close()` del objeto `StreamConnectionNotifier`, `L2CAPConnectionNotifier`, o `SessionNotifier` según corresponda.

- **Modificación del Service Record:**

En muchos casos será deseable modificar el *Service Record* generado automáticamente al llamar a `Connector.open()`. Para ello la clase `LocalDevice` proporciona el método `getRecord()` que devuelve el objeto `ServiceRecord` correspondiente a una conexión. Este método tomará como parámetro el objeto *notifier* correspondiente. Una vez obtenido el `ServiceRecord` se podrán añadir o modificar atributos mediante su método `setAttributeValue()`. Si esto se realiza antes de llamar a `acceptAndOpen()` por primera vez los cambios se verán reflejados en el *Service Record* de la SDDB una vez que se llame a `acceptAndOpen()`.

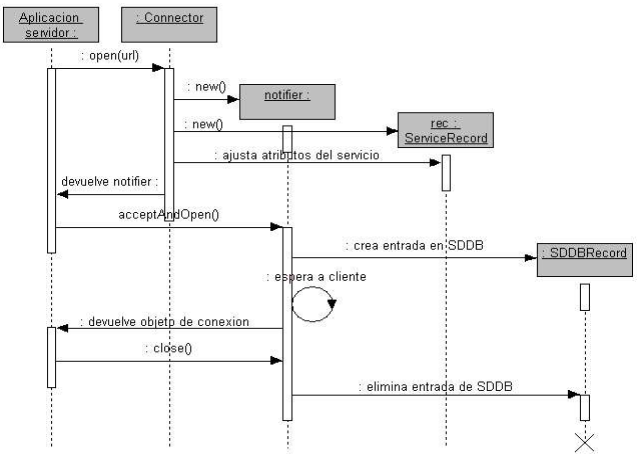


*Figura 4.3 Ciclo de vida del Service Record*

Una vez que se haya llamado a `acceptAndOpen()`, los cambios que se realicen en el objeto `ServiceRecord` de la forma anteriormente descrita, no se reflejarán en el *Service Record* de la SDDB. Para actualizarlo, de forma que los cambios sean visibles a los clientes se utiliza el método `updateRecord()` del objeto `LocalDevice`. Como parámetro tomará el `ServiceRecord` anteriormente obtenido.

En el diagrama de la figura 4.3 se ilustra lo anteriormente explicado, mostrando como es el ciclo de vida del *Service Record*.

El siguiente código de ejemplo ilustra algunos de los conceptos vistos hasta ahora. La aplicación consiste en un servidor que crea un *Service Record* y lo modifica. Primero se obtiene el objeto `LocalDevice` que da acceso al sistema Bluetooth. Seguidamente se llama al método `Connector.open()` que abrirá la conexión. Con el objeto *notifier* devuelto ya se puede obtener el `ServiceRecord` con el método `getRecord()`. Antes de poder crear el nuevo atributo en el `ServiceRecord`, será necesario crear el `DataElement` que formará parte del atributo. El nuevo atributo será una cadena por lo que el tipo del `DataElement` será `DataElement.STRING`. Una vez que se tiene el `DataElement` podremos crear el atributo con `setAttributeValue()`. Como ID de atributo se ha utilizado 0x2222 que corresponde un atributo no estandarizado. Una vez creado, se hará uso de `getAttributeValue` para leer el valor del atributo. Éste método lo que devuelve es el `DataElement` que forma parte del atributo. Para leer su contenido, es decir la cadena que queríamos como atributo haremos uso de `getValue()`. Por último, se llama a `acceptAndOpen()` con lo que se almacenará el *Service Record* en la SDDB y empezará a ser visible para los posibles cientes. No hará falta utilizar el método `updateRecord()` pues la modificación se ha realizado antes de llamar a `acceptAndOpen()`.

```
public class ServidorServiceRecord extends MIDlet
    implements Runnable, CommandListener {
...
    private LocalDevice local;
    private StreamConnectionNotifier notifier;
    private StreamConnection conn;
    private ServiceRecord record;
    ...
    public void run() {
        ...
        connectionUrl = "btspp://localhost:" + serviceUUID + ";name=MiServicio";

        try {
            // obtiene LocalDevice y Notifier
            local = LocalDevice.getLocalDevice();
            notifier = (StreamConnectionNotifier)Connector.open(connectionUrl);

            // obtiene Service Record
            record = local.getRecord(notifier);

            // crea nuevo DataElement para el atributo
            DataElement atributo=new DataElement(DataElement.STRING, "MiAtributo");

            // crea el nuevo atributo en el Service Record
            record.setAttributeValue(0x2222, atributo);

            //imprime el atributo por pantalla
            serverForm.append(record.getAttributeValue(0x2222).getValue());

            // acceptAndOpen. El ServiceRecord se crea en la SDDB
            conn = (StreamConnection)notifier.acceptAndOpen();

        } catch(IOException e) {
            serverForm.append("IOException");
        } catch(SecurityException e) {
            serverForm.append("SecurityException");
        }
```

```
    }
...
}
```

## b) Descubrimiento de servicios

El descubrimiento de servicios se realiza, al igual que el de dispositivos, de forma que no se bloquea la aplicación. De este modo, cuando un servicio es encontrado durante la búsqueda se pasa a la aplicación como un evento. Además también se indicará como evento el fin del proceso de búsqueda. Pero a diferencia del descubrimiento de dispositivos, se pueden llevar a cabo varias búsquedas de servicios al mismo tiempo.

Lo que se debe hacer para realizar una búsqueda de servicios en un dispositivo remoto, es enviarle una lista de UUIDs a buscar. Además se le indicará los atributos de los *Service Records* correspondientes a los servicios encontrados que se quieran obtener. El dispositivo remoto comprobará entonces todos sus *Service Records* para ver si alguno de ellos se corresponde con alguno de los UUID enviados. Para todos los *Service Records* que contengan algun UUID de la lista, el dispositivo remoto devolverá el ServiceRecordHandle y los atributos solicitados para ese *ServiceRecord.*

Para el descubrimiento de servicios se emplearán la clase `DiscoveryAgent` y la interfaz `DiscoveryListener`. La búsqueda de servicios se iniciará con el método `searchServices()` de la clase `DiscoveryAgent`, que tómara cuatro argumentos. El primero será la lista de atributos a obtener de los *Service Records* encontrados. Por defecto, si la lista esta vacía, se obtendrán los siguientes: ServiceRecordHandle, ServiceClassIDList, ServiceRecordState, ServiceID y ProtocolDescriptorList. Estos atributos son los que se necesitan para establecer la conexión. Si en la lista se especifica alguno más, se obtendrá además de estos. El segundo parámetro es la lista de UUIDs que debe contener el *Service Record*. El tercero es el objeto `RemoteDevice` correspondiente al dispositivo en el que se ha de buscar. El último argumento es el objeto `DiscoveryListener` que será notificado de los distintos eventos, como puede ser al descubrir un servicio o cuando la búsqueda haya finalizado.

El método `searchServices()` devolverá la ID de transacción, que le puede ser útil a la aplicación para cancelar una búsqueda determinada, identificar qué búsqueda fue la que encontró un servicio o determinar cuando una búsqueda ha terminado. El sistema podrá lanzar una excepción `BluetoothStateException` al usar este método, si se supera el número máximo de búsquedas simultáneas permitidas, o simplemente si no se puede iniciar.

Una vez que ha comenzado la búsqueda, los servicios encontrados se irán pasando al objeto `DiscoveryListener` a través de su método `servicesDiscovered()`, al que se le pasarán el ID de transacción y los distintos *Service Records* correspondientes al servicio como una tabla de `ServiceRecord`. Cada uno de estos `ServiceRecord` contendrá los atributos especificados al llamar a `searchServices()` además, claro está, de los atributos por defecto.

Cuando la búsqueda de servicios finaliza es notificado mediante una llamada al método `serviceSearchCompleted()`, que devolverá el ID de transacción de la búsqueda y un código que indica como ha finalizado. En la tabla 4.2 se especifican los posibles códigos y su significado.

| *Código de finalización* | *Razón* |
|---|---|
| `SERVICE_SEARCH_COMPLETED` | La búsqueda terminó normalmente. Al menos un Service Record encontrado. |
| `SERVICE_SEARCH_TERMINATED` | La búsqueda fue cancelada mediante `cancelServiceSearch()` |
| `SERVICE_SEARCH_ERROR` | Ocurrio un error durante la búsqueda. |
| `SERVICE_SEARCH_NO_RECORDS` | No se encontro ningún Service Record. |
| `SERVICE_SEARCH_DEVICE_NOT_REACHABLE` | No se pudo establecer con el dispositivo remoto |

*Tabla 4.2 Códigos de respuesta de la búsqueda de servicios*

La búsqueda se puede cancelar llamando al método `cancelServiceSearch()` que toma como argumento el ID de transacción. Como respuesta se devolverá `true` si se pudo detener con éxito, lo que causará una llamada al método `serviceSearchCompleted()` con el código `SERVICE_SEARCH_TERMINATED`. Si el método devuelve `false` es porque la búsqueda ya había acabado o porque el ID de transacción no es válido.

A continuación se expone un código de ejemplo en el que se emplean los conceptos básicos para la búsqueda de servicios. El ejemplo viene a ser una continuación del que se ha propuesto para la búsqueda de dispositivos, en el que los que se iban encontrando se añadían a una lista. Lo que se realiza ahora es seleccionar un dispositivo de la lista para buscar un servicio determinado.

Como vemos la clase implementa la interfaz `DiscoveryListener`, al igual que en el descubrimiento de dispositivos. Una vez que se selecciona un dispositivo de la lista se iniciará la búsqueda de servicios. Para ello, antes que nada se crearán la lista de UUIDs y atributos. El atributo del *Service Record* que se recuperará será el nombre del servicio que después se mostrará por pantalla. Para ello se pasa su identificador (0x0100). Por otro lado hará falta el objeto `RemoteDevice` que se habrá obtenido durante el proceso de *inquiry*. Una vez que se tienen todos los parámetros, se inicia la búsqueda llamando a `searchServices()`. A partir de entonces cada vez que se encuentre un servicio el método `servicesDiscovered()` será llamado. Lo que se hará en este metodo es obtener el nombre del servicio del `ServiceRecord` que se recibe como parámetro e imprimirlo. Por último, al terminar la búsqueda, se llama a `serviceSearchCompleted()`.

```
public class BuscadorServicios implements DiscoveryListener, CommandListener {

    ...

    public void commandAction(Command command, Displayable displayable) {
        ...
        if(command == List.SELECT_COMMAND) {
```

```
            ...
            iniciaBusquedaServicios()
        }
    }

    // Configura e inicia la búsqueda. Para obtener el nombre de los servicios
    // encontrados se utilizara el identificador del atributo que es 0x0100
    public void iniciaBusquedaServicios() {
        ...
        try{
            UUID[] uuidList = new UUID[1];
            uuidList[0] = new UUID(serviceUUID, false);
            attrList = new int[1];
            attrList[0] = 0x0100;

            index = deviceList.getSelectedIndex();
            remoto = (RemoteDevice)tablaDispositivos.elementAt(index);

            id = agent.searchServices(attrList, uuidList, remoto, this);
        } catch (BluetoothStateException e) {
            ...
        }
    }

    // Llamado al finalizar la búsqueda
    public void serviceSearchCompleted(int transID, int type) {
        if(type != SERVICE_SEARCH_COMPLETED) {
            form.append("Service not found\n");
        }
    }

    // Llamado cuando se encuentra el servicio. Muestra el nombre por pantalla
    public void servicesDiscovered(int transID, ServiceRecord[] record) {
        if(record[0] != null) {
            form.append((String)record[0].getAttributeValue(0x0100).getValue())
        }
    }
}
```

Como hemos visto en este ejemplo se ha podido hacer uso de los atributos del `ServiceRecord` obtenido durante el proceso de búsqueda. Esto se realiza como vemos, al igual que al accecer al objeto `ServiceRecord` local, mediante su método `getAttributeValue()`. Como vimos esto nos devolvía un objeto `DataElement` que es el tipo de datos que encapsula al atributo. Para obtener el valor del atributo verdaderamente `DataElement` proporciona el método `getValue()`.

Asi es como accedemos a un atributo que ha sido recuperado durante el proceso de búsqueda. Pero hay algunos a los que no se podrá acceder porque no han sido recuperados, es decir, aquellos que no se hayan especificado al llamar a `searchServices()` .

Puede ser que además se quiera disponer de otros atributos. Se podría pensar que lo adecuado sería obtenerlos todos cuando se hace la búsqueda. Pero esto se intenta evitar para reducir el volumen de datos por el aire. Además el número de atributos a recuperar durante la búsqueda está limitado. Por lo tanto es necesario disponer de un método para poder obtener atributos adicionales. Este método es `populateRecord()` que toma como parámetros la lista de identificadores de los atributos que se quiere recuperar. Devolerá `true` si alguno o todos se han podido recibir y además podrá lanzar una `IOException` si no se puede alcanzar el dispositivo remoto

o si el servicio no está disponible. Este método no funciona con eventos como `searchServices()`, sino que bloquea la aplicación hasta que se obtienen los atributos especificados, por ello es bueno utilizarlo en un hilo independiente.

#### c) Descubrimiento simple de dispositivos y servicios

Para hacer la búsqueda de dispositivos y servicios de una forma más sencilla, existe el método `selectService()` que combina ambos procesos. Este método devuelve directamente una cadena de conexión que se puede utilizar para conectarse al servicio. Si el servicio no se encuentra en ningún dispositivo del entorno devolverá `null`.

Los parámetros de `selectService()` son tres: El primero es el UUID a buscar en el atributo ServiceClassIDList de los distintos servicios. El segundo indica los requisitos mínimos de seguridad a adoptar por la conexión, y el tercero sirve para forzar que el dispositivo local sea el maestro en la conexión.

En el siguiente código de ejemplo se hace uso de `selectService()` para obtener la cadena de conexión. Ahora no se supone que los dispositivos hayan sido anteriormente buscados como en el ejemplo anterior, ya que `selectService()` se encargará también de eso. Por esto también hay que tener en cuenta que `selectService()` puede tener la aplicación hasta varios segundos bloqueada.

```
String cadenaConexion;
UUID servUUID = new UUID("1234567890ABC", false);
...
try {
    cadenaConexion = discAgent.selectService(servUUID,
        ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);
} catch(BluetoothStateException e) {
    ...
}
...
```

### 4.2.5. Comunicación

En esta sección veremos como se lleva a cabo la comunicación entre dos dispositivos. Para ello explicaremos los dos protocolos básicos que proporciona Bluetooth para esto: RFCOMM y L2CAP. Además de estos, se suele utilizar OBEX para el intercambio de objetos, aunque éste no pertenezca a la especificación Bluetooth.

#### a) RFCOMM

Para las comunicaciones RFCOMM no se han definido nuevos métodos o clases, sino que se usan las clases e interfaces que proporciona GCF. De este modo toda comunicación RFCOMM comienza con la llamada a `Connector.open()` y una cadena de conexión válida. Como se vio la cadena de conexión es de la forma

{protocolo}:{dirección}:{parámetros}

Para usar RFCOMM el {protocolo} usado tanto para cliente como para servidor es `btspp`. La {dirección} y los {parámetros} sin embargo, serán diferentes según se trate de uno u otro. En cualquier caso los {parámetros} que se usan en los dos son similares. La tabla 4.3 muestra los distintos {parámetros} que se pueden usar en una cadena de conexión RFCOMM, L2CAP u OBEX junto con los valores que pueden tomar. Si se usa algún otro tipo de valor `Connector.open()` lanzará una `IllegalArgumentException`. Todos estos {parámetros} son opcionales por lo que no tendrán por que aparecer en la cadena de conexión.

| *Parámetro* | *Descripción* | *Valores válidos* | *Cliente/Servidor* |
|---|---|---|---|
| `master` | Especifica si el dispositivo tiene que ser el master de la conexión | `true/false` | Ambos |
| `authenticate` | Especifica si el dispositivo remoto tienen que ser autentificado antes de establecer la conexión | `true/false` | Ambos |
| `encrypt` | Especifica si el enlace debe ser encriptado | `true/false` | Ambos |
| `authorize` | Especifica si las conexiones con este dispositivo deben recibir una autorización para usar el servicio | `true/false` | Servidor |
| `name` | Especifica el atributo ServiceName del *Service Record* | Cualquier cadena válida | Servidor |

*Tabla 4.3 Parámetros válidos para cadenas de conexión RFCOMM*

Los parámetros que determinan los niveles de seguridad de la comunicación son `authenticate`, `encrypt` y `authorize`. Los valores que pueden adoptar son `true` o `false`. Si no se especifica en la cadena de conexión, la implementación lo interprétara como `false` a no ser que otro parámetro lo requiera como `true`. Por ejemplo si `encrypt` es `true`, `authenticate` será también `true` aunque no se especifique en la cadena de conexión, porque el encriptado requiere autentificación.

Algunas combinaciones de parámetros no son válidas. Por ejemplo, `authenticate` no puede ser `false` si `encrypt` o `authorize` es `true`. Si se utiliza una combinación de parámetros que no es válida `Connector.open()` lanzará una `BluetoothConnectionException`. También se lanzará si el proceso de autentificación, encriptado o autorización falla durante el establecimiento de la conexión.

Por otro lado existe el parámetro `master`, que sirve para forzar al dispositivo local a tomar el papel de dispositivo maestro durante la conexión. Éste puede ser tanto el servidor como el cliente, por lo que este parámetro se podrá utilizar en la cadena de conexión de ambos tipos de aplicaciones. Hay que tener en cuenta que hay dispositivos que no aceptan el cambio de papel de maestro a esclavo o viceversa (*role switch*). En este caso, se lanzará una excepción del tipo `BluetoothConnectionException` .

El parámetro `name` lo usará el servidor, y le sirve para especificar el nombre del servicio. Este nombre será lógicamente el mismo que el del atributo ServiceName del *Service Record*. Su valor podrá ser cualquier cadena válida.

- **Servidor:**

El establecimiento de la conexión comienza por parte del servidor utilizando el método `Connector.open()` con una cadena válida. Como hemos dicho {protocolo} será `btssp`, mientras que la {direccion} que se empleará para el servidor será `localhost`. A continuación se muestran un par de ejemplos de cadenas de conexión que puede usar el servidor:

```
"btssp://localhost:1234567ABCDE12345;name=serviceName;"
"btssp://localhost:1234567ABCDE12345;authenticate=true;authorize=true;master=true"
```

Mediante el método `Connector.open()` se obtendrá el objeto `StreamConnectionNotifier`. Además de esto, se creará un *Service Record* básico. Seguidamente se realiza una llamada al método `acceptAndOpen()` que proporciona `StreamConnectionNotifier`. Este método bloquea el hilo esperando hasta que un cliente se conecte al servidor. Al llamar a `acceptAndOpen()` se registra el *Service Record* anteriormente creado en la SDDB. Lo que devuelve el método es un objeto `StreamConnection` que permitirá a la aplicación intercambiar datos con el cliente. Se aceptarán tantas conexiones como veces se llame a `acceptAndOpen()`.

El servidor esperará en `acceptAndOpen()` hasta que un cliente se conecte. Una vez establecida la conexión se podrán abrir los flujos de entrada y salida. Esto se realiza mediante los métodos `OpenInputStream` y `OpenOutputStream` del objeto `StreamConnection`, que devolverán los objetos `InputStream` y `OutputStream` respectivamente. Estos objetos proporcionan los métodos `read()` y `write()` para finalmente recibir y enviar datos. Por último se utiliza el método `close()` tanto para cerrar la conexión (`StreamConnection`) como los flujos.

A continuación se expone un ejemplo de código que implementa un servidor que emplea una conexión RFCOMM. El servidor aceptará la conexión de un cliente, y le devolverá los datos a medida que los vaya recibiendo. Nótese también que los métodos que se utilizan pueden lanzar una excepción del tipo `IOException`.

```
...
private StreamConnectionNotifier notifier;
private StreamConnection conexion;
OutputStream salida;
InputStream entrada;
byte[] datos;
int tamano;
...
try
{
    String cadenaConexion = "btssp://localhost:12345678ABCE;name=nombreservicio";

    // se obtiene el objeto notifier
```

```
    notifier = (StreamConnectionNotifier)Connector.open(cadenaConexion);
    // se acepta la conexion de un cliente
    conexion = (StreamConnection)notifier.acceptAndOpen();

    // una vez establecida la conexion se abren los flujos
    salida = conexion.openOutputStream();
    entrada = conexion.openInputStream();

    // se reciben y envian los mismos datos
    byte[] datos = new byte[10];
    int tamano = 0;
    while((tamano = entrada.read(datos)) != -1) {
        salida.write(datos, 0, tamano);
    }

    // se cierran los flujos y la conexion
    salida.close();
    entrada.close();
    conexion.close();

} catch(IOException e) {
    // se captura la excepcion
}
...
```

- **Cliente:**

Para establecer la conexión el cliente usa una cadena de conexión también con `btssp` como {protocolo}, seguido de la dirección Bluetooth del dispositivo servidor a conectar y, por último el identificador de canal del servidor. El identificador de canal del servidor es un número entre 0 y 31, que identifica de forma única al servicio en el dispositivo servidor. Este identificador es asignado para cada servicio por la implementación JABWT, y se encuentra almacenado en el *Service Record*. Esto permite al método `getConnectionURL()` de `ServiceRecord` generar la cadena de conexión que debe usar el cliente para conectarse al servicio, que podrá ser algo parecido a las siguientes:

```
"btssp://00082546123A:1;authenticate=true"
"btssp://00082546123A:5;master=true;encrypt=true"
```

Una vez que se le pasa la cadena de conexión adecuada a `Connector.Open()` se obtiene el objeto `StreamConnection` que permite a la aplicación intercambiar datos con el dispositivo remoto. Como vemos el servidor no actúa igual que el cliente, donde la conexión se establece nada más llamar a `Connector.Open()`. De este modo tras invocarlo se podrán obtener los flujos de entrada y salida mediante los métodos `openInputStream()`, `openOutputStream()`, `openDataInputStream()` y `openDataOutputStream`. Al final el método `close()` tendrá que ser utilizado para cerrar el objeto `StreamConnection` y los distintos flujos que se hayan abierto.

A continuación se expone un ejemplo en el que el cliente se conecta al servidor, del que previamente se ha obtenido el `ServiceRecord`, y le envía una cadena de datos. Los parámetros del método `getConnectionURL()` indica si hay que llevar a cabo algún proceso de seguridad durante la conexión, y si el dispositivo debe ser el maestro de la conexión.

```
...
ServiceRecord record;
StreamConnection conexion;
OutputStream salida;
...
try
{
    // cadena a enviar
    String cadenaDatos = "Hola";

    // obtiene la cadena de conexion a partir del Service Record
    String cadenaConexion = record.getConnectionURL(0, false);

    // establece la conexion
    conexion = (StreamConnection)Connector.open(cadenaConexion);

    // abre flujo de salida
    salida = conexion.openOutputStream();

    // obtiene los bytes de la cadena a enviar
    byte[] datos = cadenaDatos.getBytes();

    // envia cadena
    salida.write(datos);

    // cierra flujo y conexion
    salida.close();
    conexion.close();
}
catch( IOException e )
{
    // IOException
}
```

Hasta aquí se ha visto que facilidades proporciona JABWT para establecer una conexión RFCOMM, que será el protocolo que más se use. RFCOMM ofrece una comunicación serie en los dos sentidos que permite utilizar los métodos tradicionales de comunicación utilizados en J2ME.

### b) L2CAP

A continuación se expone la API de Java para establecer una comunicación L2CAP. L2CAP es una capa de multiplexación que permite a aplicaciones o protocolos de capas más altas usar la comunicación Bluetooth. También RFCOMM utiliza L2CAP. Veamos a continuación por qué en ocasiones es necesario usar L2CAP directamente en lugar de RFCOMM.

El protocolo RFCOMM es utilizado en diversos perfiles como SPP, perfil de Fax, o perfil de dispositivo manos libres. Además el protocolo SDP hace uso de RFCOMM para la conexión. Sin embargo existen muchos más perfiles además de los que contínuamente van creándose, que no hacen uso de RFCOMM pero sí de L2CAP. Eso implica que si se quiere implementar algunos de estos perfiles con JABWT, si es que es posible, habrá de hacerse directamente sobre L2CAP. Pero téngase en cuenta que JABWT no proporciona una interfaz a todas las carácterísticas que ofrece L2CAP, por lo que habrá algunos perfiles y protocolos que no se puedan implementar con JABWT.

Además de las aplicaciones estandarizadas definidas como perfiles, se pueden desarrollar aplicaciones propias. En este caso se podrá usar RFCOMM, L2CAP y OBEX. En general, las APIs de tipo flujo correspondientes a los protocolos de alto nivel RFCOMM y OBEX serán las adecuadas para la mayoría de las aplicaciones. Pero si se quiere tener un control a más bajo nivel de la comunicación, controlando incluso que bytes se envían juntos en un paquete, habrá que usar la API de L2CAP.

JABWT proporciona las interfaces `L2CAPConnection` y `L2CAPConnectionNotifier` para la comunicación L2CAP. El servidor usa `L2CAPConnectionNotifier` para esperar a que un cliente establezca una conexión. Es entonces cuando se obtiene un objeto `L2CAPConnection` que da acceso al canal L2CAP entre cliente y servidor, y permitirá el intercambio de datos entre ambos.

Análogamente cuando un cliente se conecta a un servidor L2CAP lo que le devuelve `Connector.open()` es un objeto `L2CAPConnection` que le da acceso al canal L2CAP, permitiéndole enviar y recibir datos. El cliente no hará uso de la interfaz `L2CAPConnectionNotifier`.

La interfaz `L2CAPConnection` proporciona los métodos `send()` y `receive()` para enviar y recibir datos a través del canal. El método `receive()` bloquea hasta que un paquete L2CAP haya sido recibido o hasta que se cierre el canal. Por esto existe el método `ready()`, que indica si hay paquetes disponibles para ser leídos de forma inmediata por `receive()` evitando así el bloqueo.

La cadena de conexión empleada en `Connector.open()` será análoga a la que se utiliza en conexiones RFCOMM, siendo {protocolo} `btl2cap`. En el caso del cliente la dirección Bluetooth irá seguida del PSM, que indica cual será la aplicación destino del canal L2CAP que se está creando. Este valor se puede obtener del *Service Record*. En general el PSM se asigna dinámicamente al iniciar la aplicación, aunque hay protocolos de alto nivel como RFCOMM o SDP que tienen un PSM preasignado. Los parámetros de seguridad así como `master` también podrán ser empleados en este caso.

Además existen los parámetros exclusivos de conexiones L2CAP `receiveMTU` y `transmitMTU`, para establecer las MTUs de recepción y transmisión respectivamente. La MTU (Maximum Transmission Unit) es el tamaño máximo que puede adoptar un paquete L2CAP. Puede diferir de un sentido a otro, de forma que el cliente y servidor tengan que enviar paquetes de diferente tamaño máximo.

Vemos que se pueden especificar las MTUs de forma que se indica el tamaño de los paquetes a transmitir. Pero esto habrá que hacerlo siguiendo unas reglas determinadas. De este modo, habrá un tamaño mínimo indicado por `L2CAPConnection.MINIMUM_MTU`, y un tamaño máximo que se puede obtener con `LocalDevice.getProperty("bluetooth.l2cap.receiveMTU.max")`. Además al enviar paquetes con `send(byte[] outBuf)` habrá que tener en cuenta que `outBuf` no puede ser mayor que `transmitMTU`, ya que si es así se descartarán *bytes* antes de que el paquete sea enviado. Análogamente al recibir datos con `receive(byte[] inBuf)`, si el tamaño de `inBuf` es menor que

`transmitMTU`, los *bytes* que no quepan en `inBuf` serán desechados. De esto se deriva que cliente y servidor deben ponerse de acuerdo en cuanto a los MTUs.

En general para obtener la cadena de conexión se actuará igual que en RFCOMM, es decir, usando los métodos `getConnectionURL()` o `selectService()`. Ambos devolverán la cadena de conexión, pero si se quieren especificar los MTUs habrá que añadir los parámetros correspondientes a la cadena. A continuación se exponen un par de posibles cadenas de conexión del cliente:

```
"btl2cap://0080256341A2:1001;receiveMTU=512;transmitMTU=1024"
"btl2cap://0080256341A2:1001;authenticate=true;encrypt=true"
```

En el servidor la cadena será igual excepto que {dirección} será `localhost` al igual que en RFCOMM. Lógicamente no habrá que incluir el PSM. Los parámetros para especificar las MTUs también podrán utilizarse de la misma forma que en el resultando algo como las siguientes:

```
"btl2cap://localhost:1254569598899952ABCD;receiveMTU=512;name=L2CAPservice"
"btl2cap://localhost:1254569598899952ABCD;receiveMTU=512;master=true;encrypt=true"
```

A continuación se expone un listado de código correspondiente a un servidor L2CAP. Lo que hace el servidor del ejemplo, es recibir datos de un cliente. Como vemos es parecido al caso de RFCOMM pero con algunas diferencias importantes. Por un lado al llamar a `Connector.open()` se captura la excepción `IllegalArgumentException`, porque es más probable que se lance que en RFCOMM si se incumple alguna de las reglas a seguir por los MTU. Por otro lado los objetos devueltos por `Connector.open()` y `acceptAndOpen()` son diferentes como hemos comentado anteriormente. Por último vemos como el *buffer* de recepción se crea del tamaño del MTU de recepción.

```
...
private L2CAPConnectionNotifier notifier;
private L2CAPConnection conexion;
...
String cadenaConexion = "btl2cap://localhost:125456959852ABCD;name=L2CAPservice"
try
{

    try
    {
        // se obtiene el objeto notifier
        notifier = (L2CAPConnectionNotifier)Connector.open(cadenaConexion);
    }
    catch(IllegalArgumentException e)
    {
        ...
    }

    // se acepta la conexion de un cliente
    conexion = (L2CAPConnection)notifier.acceptAndOpen();

    // se crea el buffer de entrada de tamano la MTU de recepcion
    byte[] inBuf = new byte[conexion.getReceiveMTU()];

    // recibe los datos del cliente
    for(;;)
    {
```

```
        conexion.receive(inBuf);
    }

    conexión.close();
} catch(IOException e) {
    ...
}
...
```

El código del cliente se realiza de forma similar. En el código propuesto a continuación se envía un paquete de MTU 1024. Vemos como para especificar la MTU hay que añadirle el parámetro apropiado a la cadena de conexión.

```
...
ServiceRecord record;
String cadenaConexion;
L2CAPConnection conexion;
...
try
{
    // se obtiene la cadena de conexion
    cadenaConexion = record.getConnectionURL(0, false) + ";transmitMTU=1024";

    try
    {
        // se obtiene el objeto notifier
        conexion = (L2CAPConnection)Connector.open(cadenaConexion);
    }
    catch(IllegalArgumentException e)
    {
        ...
    }

    // se acepta la conexion de un cliente
    conexion = (L2CAPConnection)notifier.acceptAndOpen();

    // se crea el buffer de entrada de tamano la MTU de recepcion
    byte[] sBuf = new byte[conexion.getTRansmitMTU()];

    // se introducen los datos a enviar en el buffer
    ...

    // envia el paquete
    conexion.send(sBuf);

    // cierra la conexion
    conexion.close();

} catch(IOException e) {
    ...
}
...
```