Programación de dispositivos Bluetooth a través de Java

Alberto Gimeno Brieba
gimenete@gimenete.net

Abstract

En este documento se trata la programación de dispositivos Bluetooth con Java mediante el API desarrollada por el JCP y especificada en el JSR-82.

Keywords: Java, Bluetooth, J2ME, JSR-82, móviles.

1 Introducción

Bluetooth es una tecnología de comunicación inalámbrica, al igual que la tecnología Wi-Fi o los infrarrojos. A diferencia de la primera, Bluetooth está diseñada para dispositivos de bajo consumo y para conexiones de corta distancia (10 metros). A diferencia de los infrarrojos, Bluetooth es omnidireccional y tiene un mayor ancho de banda (hasta 11 Mbit/ segundo).

Bluetooth es, pues, una tecnología ideal para la conexión de dispositivos de bajas prestaciones (móviles, cámaras de fotos, auriculares manos libres, impresoras,…).

Uno de los mayores ámbitos de utilización de Bluetooth es sin duda los teléfonos móviles. Cada vez es más común encontrar terminales móviles con soporte para Java y Bluetooth y simplemente es un paso natural que surja la necesidad de programar estos dispositivos a través de Java. Desde el JCP se ha desarrollado un JSR que cubre esta necesidad. Se trata del JSR-82 que será explicado en este documento.

2 El JSR-82

El JSR-82[1] especifica un API de alto nivel para la programación de dispositivos Bluetooth. Depende de la configuración CLDC de J2ME, y se divide en dos paquetes: javax.bluetooth y javax.obex. El primer paquete provee la funcionalidad para la realización de búsquedas de dispositivos, búsquedas de servicios y comunicación mediante flujos de datos (*streams*) o *arrays* de bytes. Por otro lado el paquete javax.obex permite la comunicación mediante el protocolo OBEX (*OBject Exchange*); se trata de un protocolo de alto nivel muy similar a HTTP.

3 El paquete javax.bluetooth

Primero abordaremos la programación de un cliente y más tarde veremos cómo programar un servidor.

3.1 Clientes Bluetooth

Un cliente Bluetooth deberá realizar las siguientes operaciones para comunicarse con un servidor Bluetooth:

- Búsqueda de dispositivos
- Búsqueda de servicios
- Establecimiento de la conexión
- Comunicación

El punto de partida es la clase LocalDevice que representa el dispositivo en el que se está ejecutando la aplicación. Este objeto es un *singleton* y se obtiene mediante LocalDevice.getLocalDevice(). Este objeto permite obtener información sobre el dispositivo: modo de conectividad, dirección bluetooth y nombre del dispositivo.

El primer paso que debe realizar un cliente es realizar una búsqueda de dispositivos. Para ello deberemos obtener un objeto DiscoveryAgent. Este objeto es único y se obtiene a través del objeto LocalDevice.

```
DiscoveryAgent da = LocalDevice.getLocalDevice().getDiscoveryAgent();
```

El objeto DiscoveryAgent nos va a permitir realizar y cancelar búsquedas de dispositivos y de servicios. Y también nos servirá para obtener listas de dispositivos ya conocidos. Esto se lleva a cabo llamando al método retrieveDevices(). A este método se le debe pasar un argumento de tipo entero que puede ser:

- DiscoveryAgent.PREKNOWN. Para obtener una lista de dispositivos encontrados en búsquedas anteriores.
- DiscoveryAgent.CACHED. Para obtener una lista de dispositivos "favoritos".

El método retrieveDevices() devuelve un *array* de objetos RemoteDevice. La clase RemoteDevice representa un dispositivo remoto y tiene métodos similares a LocalDevice que, recordemos, representa al dispositivo en el que se ejecuta la aplicación. Así pues, podemos obtener el nombre del dispositivo mediante getFriendlyName() y su dirección bluetooth mediante getBluetoothAddress().

Podríamos omitir la búsqueda de dispositivos y pasar directamente a la búsqueda de servicios en caso de que deseásemos conectar con alguno de los dispositivos pertenecientes a alguna de estas listas. Sin embargo lo más común será intentar conectar con un dispositivo encontrado en una búsqueda de dispositivos, debido a que obviamente lo tendremos a nuestro alcance.

Una búsqueda de dispositivos se inicia llamando al método startInquiry(). Este método requiere un argumento de tipo DiscoveryListener. DiscoveryListener es una interfaz que implementaremos a nuestra conveniencia y que será usada para que el dispositivo notifique eventos a la aplicación cada vez que se descubre un dispositivo, un servicio, o se finaliza una búsqueda. Estos son los cuatro métodos de la interfaz DiscoveryListener:

- deviceDiscovered()
- inquiryCompleted()
- servicesDiscovered()
- serviceSearchCompleted()

Los dos primeros métodos son llamados en el proceso de búsqueda de dispositivos. Los otros dos son llamados en procesos de búsqueda de servicios.

Cada vez que un dispositivo es encontrado se llama al método deviceDiscovered() pasando un argumento de tipo RemoteDevice.

Una vez que la búsqueda de dispositivos ha concluido se llama al método inquiryCompleted() pasando como argumento un entero que indica el motivo de la finalización. Este entero puede valer:

- DiscoveryListener.INQUIRY_COMPLETED si la búsqueda concluyó con normalidad,
- DiscoveryListener.INQUIRY_TERMINATED si la búsqueda ha sido cancelada manualmente o
- DiscoveryListener.INQUIRY_ERROR si se produjo un error en el proceso de búsqueda.

Ya hemos conseguido dar el primer paso para realizar una conexión cliente. El siguiente paso es realizar una búsqueda de servicios. Antes de seguir deberemos comprender ciertos conceptos.

Una aplicación cliente es una aplicación que requiere un servidor para que le ofrezca un servicio. Este servicio puede ser: un servicio de impresión, un servicio de videoconferencia, un servicio de transferencia de archivos, etc. En una comunicación TCP-IP un cliente se conecta directamente a un servidor del que conoce el servicio que ofrece, es decir, conocemos *a priori* la localización del servidor y el servicio que nos ofrecerá; sin embargo un cliente Bluetooth no conoce de antemano qué dispositivos tiene a su alcance ni cuáles de ellos pueden ofrecerle el servicio que necesita. De modo que un cliente Bluetooth necesita primero buscar los dispositivos que tiene a su alcance y posteriormente les preguntará si ofrecen el servicio en el que está interesado. Este último proceso se denomina búsqueda de servicios y es el siguiente paso que un cliente debe realizar.

Cada servicio es identificado numéricamente. Es decir, a cada servicio le asignamos un número y para referirnos a dicho servicio usaremos su número asociado. Este identificador se denomina UUID (Universal Unique IDentifier). Adicionalmente, cada servicio tiene ciertos atributos que lo describen. Por ejemplo un servicio de impresión podría describirse por diversos atributos como: tipo de papel (dinA4, US-letter,…), tipo de tinta (color, blanco y negro), etc. Los atributos también están identificados numéricamente, es decir, para referirnos a un atributo usaremos su número asociado.

Las búsquedas de dispositivos también se realizan mediante el objeto DiscoveryAgent. Concretamente usaremos el método searchServices() al que le tendremos que pasar un objeto DiscoveryListener que recibirá los eventos de la búsqueda, el dispositivo en el que realizar la búsqueda (un objeto RemoteDevice que normalmente obtendremos en la búsqueda de dispositivos), los servicios en los que estamos interesados, y los atributos que queremos conocer sobre

dichos servicios (tipo de papel, tipo de tinta, etc). Por ejemplo un cliente que esté interesado en un servicio de impresión, para imprimir un texto probablemente sólo le interese conocer el tipo de papel, sin embargo si queremos imprimir una imagen estaremos también interesados en si soporta o no tinta de color.

Si se encuentra algún servicio se nos notificará a través del objeto DiscoveryListener mediante el método servicesDiscovered(). Se nos pasará un *array* de objetos ServiceRecord que encapsulan los atributos de servicio que solicitamos al invocar la búsqueda. Los valores de estos atributos de servicio son objetos DataElement.

Un objeto DataElement encapsula los tipos de datos en los que puede ser representado un atributo de servicio. Estos pueden ser: números enteros de diferente longitud con o sin signo, cadenas de texto, URLs, booleanos, o colecciones de DataElements.

Un ServiceRecord es, pues, como una tabla que relaciona los identificadores de los atributos con sus valores (objetos DataElement).

Cuando finalice la búsqueda de servicios se nos notificará mediante una llamada al método serviceSearchCompleted() de la interfaz DiscoveryListener. Se nos pasará un argumento de tipo entero indicando el motivo de la finalización. Este entero puede valer:

- SERVICE_SEARCH_COMPLETED: la búsqueda ha finalizado con normalidad.
- SERVICE_SEARCH_TERMINATED: la búsqueda ha sido cancelada manualmente.
- SERVICE_SEARCH_NO_RECORDS: no existe la información solicitada.
- SERVICE_SEARCH_ERROR: finalizó por un error.
- SERVICE_SEARCH_DEVICE_NOT_REACHABLE: el dispositivo no está a nuestro alcance.

Estas constantes son miembros de la interfaz DiscoveryListener.

Si hemos encontrado algún servicio que nos interesa pasaremos al siguiente paso: abrir la conexión.

Abrir una conexión Bluetooth se lleva a cabo de la misma forma que se abre cualquier otro tipo de conexión en CLDC: a través de la clase javax.microedition.Connector. Usaremos su método open() y le pasaremos una URL que contendrá los datos necesarios para realizar la conexión.

No necesitaremos construir la URL a mano ya que el objeto ServiceRecord posee un método que nos ahorra esta tarea: getConnectionURL().

Llegados a este punto debemos saber que tenemos dos formas diferentes de comunicación: a través de flujos de datos utilizando el protocolo SPP (*Serial Port Profile*) , o bien a través de L2CAP enviando y recibiendo *arrays* de *bytes*. La forma más sencilla es mediante SPP.

Si el servidor utiliza SPP el método Connector.open() nos devolverá un objeto de tipo javax.microedition.io.StreamConnection. A través de este objeto podemos obtener un (Data)InputStream y un (Data)OutputStream. Por lo tanto ya tenemos un flujo de lectura y un flujo de escritura por lo que estamso en condiciones de leer y escribir datos.

En caso de que el servidor utilice L2CAP el método Connector.open() nos devolverá un objeto del tipo javax.bluetooth.L2CAPConnection. Con este objeto leeremos *bytes* con receive() y escribiremos *bytes* con send().

3.2 Servidores Bluetooth

La creación de un servidor Bluetooth es más sencilla que la programación de un cliente ya que no necesitamos realizar ningún tipo de búsqueda. Concretamente los pasos que debe realizar un servidor Bluetooth son los siguientes:

- Crear una conexión servidora
- Especificar los atributos de servicio
- Abrir las conexiones cliente

Crear la conexión servidora es relativamente simple. Sencillamente debemos llamar al método Connector.open() pasándole una URL con una sintaxis determinada. En caso de querer comunicarnos mediante SPP la URL comenzará por "btspp://" y en caso de querer comunicarnos mediante L2CAP la URL comenzará por "btl2cap://". A continuación deberemos indicar "localhost/" como *host*. Esto determina que no queremos conectarnos a nadie, sino que queremos ser servidores. Seguidamente sólo nos queda concatenar a la URL el identificador del servicio (UUID) que vamos a ofrecer.

A continuación llamaremos al método Connector.open() pasando la URL como argumento. Si la URL comienza por "btspp://" nos devolverá un objeto del tipo javax.microedition.StreamConnectionNotifier y en caso de que la URL comience por "btl2cap://" nos devolverá un objeto javax.bluetooth.L2CAPConnectionNotifier.

El siguiente paso es especificar los atributos de servicio. Por ejemplo si vamos a ofrecer un hipotético servicio de impresión podríamos indicar qué tipo de papel y de tinta ofrecemos. Los atributos de servicio se almacenan en un objeto ServiceRecord. Cada conexión servidora tiene un ServiceRecord asociado que se obtiene a través del LocalDevice.

Establecer los atributos de servicio es sencillo, simplemente tenemos que crear objetos DataElement y añadirlos al ServiceRecord.

Una vez establecidos los atributos de servicio ya estamos en condiciones de escuchar y procesar las conexiones cliente. Para ello usaremos el método acceptAndOpen(). En una conexión servidora SPP este método devuelve un javax.microedition.StreamConnection, y en una conexión servidora L2CAP devuelve un objeto del tipo javax.bluetooth.L2CAPConnection. En este punto ya podemos leer y escribir datos del mismo modo que lo hace un cliente.

4 El paquete javax.obex

El paquete javax.obex permite manejar el protocolo de alto nivel OBEX (*OBject Exchange*). Se trata de un protocolo muy similar a HTTP. Al igual que este último, OBEX se basa en mensajes compuestos por cabeceras de mensaje y opcionalmente de un cuerpo de mensaje. Adicionalmente los mensajes de respuesta del servidor poseen un código de respuesta indicando éxito o error.

Al igual que en HTTP, los mensajes de petición del cliente al servidor en OBEX se clasifican por métodos. Estos son los métodos que existen.

- CONNECT. Inicia la sesión.
- PUT. Envía un archivo al servidor.
- GET. Solicita un archivo al servidor.
- DELETE. Solicita la eliminación de un archivo.
- SETPATH. El cliente desea cambiar el directorio actual dentro del sistema de archivos del servidor.
- DISCONNECT. Usado para finalizar la sesión.

Las cabeceras de un mensaje OBEX son encapsuladas por un objeto HeaderSet. Existen cabeceras de uso común como COUNT, NAME, LENGTH,… Sin embargo podremos crear cabeceras personalizadas.

La clase Operation provee la funcionalidad para leer y enviar mensajes que no sólo tienen cabeceras sino que también tienen un cuerpo de mensaje. Esta clase permite obtener un (Data)InputStream y un (Data)OutputStream para leer o escribir el cuerpo del mensaje.

Ahora que conocemos las clases básicas pasemos a ver cómo programar un cliente OBEX.

4.1 Un cliente OBEX

La programación de un cliente OBEX es relativamente simple. Debemos abrir la conexión, como siempre en CLDC con el objeto Connector. Deberemos pasarle una URL que comience por "irdaobex://" y nos devolverá un objeto de tipo javax.obex.ClientSession. Lo primero que deberemos hacer será ejecutar el método connect() para iniciar la sesión.

A partir de aquí ya podemos realizar peticiones al servidor a través de los métodos put(), delete(), get() y setPath(). Todos los métodos requieren un objeto HeaderSet como parámetro. Los métodos put() y get() adicionalmente devuelven un objeto Operation que permite escribir o leer el cuerpo del mensaje respectivamente.

Para cerrar la sesión llamaremos al método disconnect().

4.2 Un servidor OBEX

Crear una conexión servidora OBEX es también muy simple. Lo primero de todo es crear un SessionNotifier llamando al método Connector.open(). La URL debe comenzar por "irdaobex://localhost". Ahora simplemente escucharemos las conexiones cliente llamando al método acceptAndOpen(). Este método requiere un argumento de tipo ServerRequestHandler. El ServerRequestHandler es un objeto que deberemos implementar nosotros. Se implementa de forma muy similar a un *servlet*: por cada método del protocolo OBEX tiene un método asociado al que se le pasan los datos de la petición. Así pues tenemos los métodos onConnect(), onGet(), onPut(), onDelete() y onDisconnect(). Todos los métodos tienen como argumento un objeto HeaderSet que encapsula las cabeceras de los mensajes, exceptuando los métodos

onPut() y onGet() que requieren un cuerpo de mensaje y por ello su argumento es de tipo Operation.

Adicionalmente todos los métodos a excepción de onDisconnect() deben devolver un entero que será el código de respuesta indicando el éxito o no de la petición y su motivo.

5 Implementaciones del JSR-82

Existen dispositivos móviles que soportan Java y tienen Bluetooth, pero sin embargo no soportan el API JSR-82. Esto quiere decir que no tenemos posibilidad de acceder al dispositivo Bluetooth a través de Java. Por ello habrá que acudir a las especificaciones del fabricante para cerciorarnos de que las APIs están soportadas.

A pesar de que el JSR-82 se especificó pensando en la plataforma J2ME. No sólo existen implementaciones y emuladores para J2ME. Debido a que J2ME es una versión reducida de J2SE, es perfectamente factible crear una implementación que pueda ser usada desde J2SE. De hecho existen implementaciones y emuladores. La mayoría de estas implementaciones son libres y suelen soportar dispositivos Bluetooth conectados al puerto serie. Otras implementaciones son *bindings* para Java de las APIs Bluetooth que ofrece el sistema operativo.

Ciertos emuladores y entornos de desarrollo también implementan estas APIs simulando dispositivos Bluetooth, es decir, permiten realizar aplicaciones que usen las APIs JSR-82 sin necesidad de tener físicamente un dispositivo Bluetooth.

6 Documentación

La documentación sobre las APIs definidas en el JSR-82 es muy escasa y mucho más escasa es en español. Sin embargo en javaHispano se publicó un tutorial[2] al respecto en el que se puede encontrar mas información y enlaces a otros documentos.

Por último añadir que siempre es fundamental tener a mano la documentación javadoc de las APIs, la cual se puede descargar desde la página del JSR-82[1] junto con la especificación.

Agradecimientos

Agradezco su apoyo a la Escuela Universitaira Politécnica de La Almunia, a Ángel Blesa y a todos mis compañeros de carrera. Pero sobre todo a todos los miembros de javaHispano por hacer esto posible.

Referencias

[1] Especificación del JSR-82: Bluetooth desde Java. *http://jcp.org/en/jsr/detail?id=82*.

[2] Alberto Gimeno Brieba. JSR-82: Bluetooth desde Java. *http://www.javahispano.org/tutorials.item.action?id=49*.