JAVA 2 MICRO EDITION

SOPORTE BLUETOOTH

Versión 1.0

Bluetooth

Bluetooth es un standard de facto global que identifica un conjunto de protocolos que facilitan la comunicación inalámbrica entre diferentes tipos de dispositivos electrónicos. Su nombre viene del rey vikingo, Harald Bluetooth (940 A.D.-981A.D.), famoso por su habilidad para la comunicación, y para hacer que la gente hablara entre ella.

Autor: Pedro Daniel Borches Juzgado
e-mail: dani_borches@terra.es

Java 2 Micro Edition: Soporte Bluetooth

AUTOR: Pedro Daniel Borches Juzgado

TUTOR: Celeste Campo Vázquez

Universidad Carlos III de Madrid

20 de Marzo de 2004

ÍNDICE

0 Introducción

Bluetooth es una tecnología de radio de corto alcance, que permite conectividad inalámbrica entre dispositivos remotos. Se diseñó pensando básicamente en tres objetivos: pequeño tamaño, mínimo consumo y bajo precio.

0.1 Nociones sobre Bluetooth

Bluetooth opera en la banda libre de radio ISM[1] a 2.4 Ghz. Su máxima velocidad de transmisión de datos es de 1 Mbps. El rango de alcance Bluetooth depende de la potencia empleada en la transmisión. La mayor parte de los dispositivos que usan Bluetooth transmiten con una potencia nominal de salida de 0 dBm, lo que permite un alcance de unos 10 metros en un ambiente libre de obstáculos.

Salto de frecuencia:

Debido a que la banda ISM está abierta a cualquiera, el sistema de radio Bluetooth deberá estar preparado para evitar las múltiples interferencias que se pudieran producir. Éstas pueden ser evitadas utilizando un sistema que busque una parte no utilizada del espectro o un sistema de salto de frecuencia.

En este caso la técnica de salto de frecuencia es aplicada a una alta velocidad y una corta longitud de los paquetes (1600 saltos/segundo). Con este sistema se divide la banda de frecuencia en varios canales de salto, donde, los transceptores, durante la conexión van cambiando de uno a otro canal de salto de manera pseudo-aleatoria.

Los paquetes de datos están protegido por un esquema ARQ (repetición automática de consulta), en el cual los paquetes perdidos son automáticamente retransmitidos.

El canal:

Bluetooth utiliza un sistema FH/TDD (salto de frecuencia/división de tiempo duplex), en el que el canal queda dividido en intervalos de 625 μs, llamados *slots*, donde cada salto de frecuencia es ocupado por un *slot*.

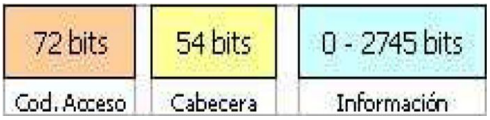
Dos o más unidades Bluetooth pueden compartir el mismo canal dentro de una *piconet* (pequeña red que establecen automáticamente los terminales Bluetooth para comunicarse entre si), donde una unidad actúa como maestra, controlando el tráfico de datos en la *piconet* que se genera entre las demás unidades, donde éstas actúan como esclavas, enviando y recibiendo señales hacia el maestro.

El salto de frecuencia del canal está determinado por la secuencia de la señal, es decir, el orden en que llegan los saltos y por la fase de esta secuencia. En Bluetooth, la secuencia queda fijada por la identidad de la unidad maestra de la *piconet* (un código único para cada equipo), y por su frecuencia de reloj.

1 Banda internacional médico-científica

Datagrama Bluetooth:

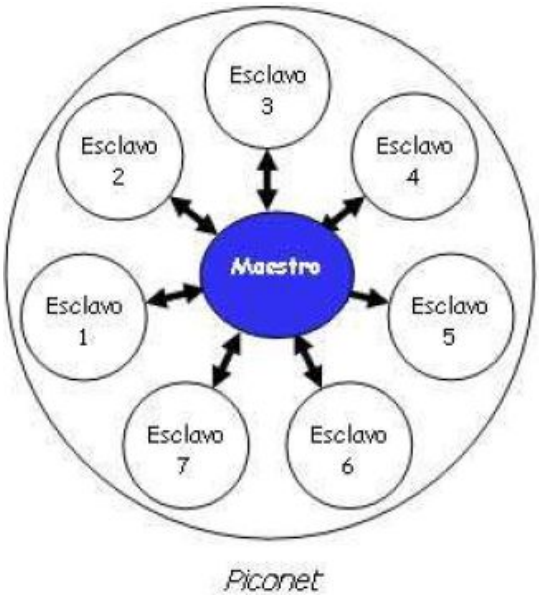
La información que se intercambia entre dos unidades Bluetooth se realiza mediante un conjunto de *slots* que forman un paquete de datos. Cada paquete comienza con un código de acceso de 72 bits, que se deriva de la identidad maestra, seguido de un paquete de datos de cabecera de 54 bits. Éste contiene importante información de control, como tres bits de acceso de dirección, tipo de paquete, bits de control de flujo, bits para la retransmisión automática de la pregunta, y chequeo de errores de campos de cabecera. La dirección del dispositivo es en forma hexadecimal. Finalmente, el paquete que contiene la información, que puede seguir al de la cabecera, tiene una longitud de 0 a 2745 bits.



En cualquier caso, cada paquete que se intercambia en el canal está precedido por el código de acceso. Los receptores de la *piconet* comparan las señales que reciben con el código de acceso, si éstas no coinciden, el paquete recibido no es considerado como válido en el canal y el resto de su contenido es ignorado.

Piconets:

Como hemos citado anteriormente si un equipo se encuentra dentro del radio de cobertura de otro, éstos pueden establecer conexión entre ellos. Cada dispositivo tiene una dirección única de 48 bits, basada en el estándar IEEE 802.11 para WLAN. En principio sólo son necesarias un par de unidades con las mismas características de hardware para establecer un enlace. Dos o más unidades Bluetooth que comparten un mismo canal forman una *piconet*.



Piconet

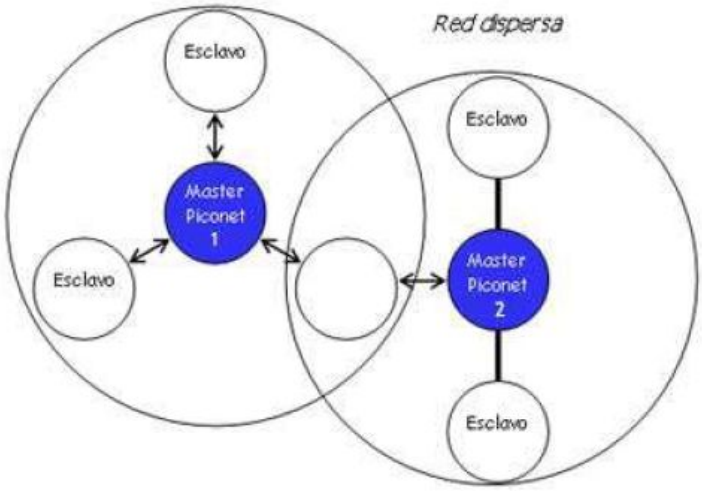
Para regular el tráfico en el canal, una de las unidades participantes se convertirá en maestra, pero por definición, la unidad que establece la *piconet* asume éste papel y todos los demás serán esclavos. Los participantes podrían intercambiar los papeles si una unidad esclava quisiera asumir el papel de maestra. Sin embargo sólo puede haber un maestro en la *piconet* al mismo tiempo. Hasta ocho usuarios o dispositivos pueden formar una *piconet* y hasta diez *piconets* pueden coexistir en una misma área de cobertura.

Medios y velocidades:

Además de los canales de datos, están habilitados tres canales de voz de 64 kbit/s por *piconet*. Las conexiones son uno a uno con un rango máximo de diez metros, aunque utilizando amplificadores se puede llegar hasta los 100 metros, pero en este caso se introduce alguna distorsión. Los datos se pueden intercambiar a velocidades de hasta 1 Mbit/s. El protocolo bandabase que utiliza Bluetooth combina las técnicas de circuitos y paquetes para asegurar que los paquetes llegan en orden.

Scatternet:

Los equipos que comparten un mismo canal sólo pueden utilizar una parte de su capacidad. Aunque los canales tienen un ancho de banda de un 1Mbit, cuantos más usuarios se incorporan a la *piconet*, disminuye la capacidad hasta unos 10 kbit/s más o menos. Para poder solucionar este problema se adoptó una solución de la que nace el concepto de *scatternet*.



Las unidades que se encuentran en el mismo radio de cobertura pueden establecer potencialmente comunicaciones entre ellas. Sin embargo, sólo aquellas unidades que realmente quieran intercambiar información comparten un mismo canal creando la *piconet*. Este hecho permite que se creen varias *piconets* en áreas de cobertura superpuestas.

A un grupo de *piconets* se le llama *scatternet*. El rendimiento, en conjunto e individualmente de los usuarios de una *scatternet* es mayor que el que tiene cada usuario cuando participa en un mismo canal de 1 Mbit. Además, estadísticamente se obtienen ganancias por multiplexación y rechazo de canales salto. Debido a que individualmente cada *piconet* tiene un salto de frecuencia diferente, diferentes *piconets* pueden usar simultáneamente diferentes canales de salto.

0.2 Establecimiento de la conexión

La conexión con un dispositivo, se hace mediante un mensage *page.* Si la dirección es desconocida, antes del mensaje *page* se necesitara un mensage *inquiry.* Antes de que se produzca ninguna conexión, se dice que todos los dispositivos están en modo *standby.*

Un dispositivo en modo *standby* se despierta cada 1.28 segundos para escuchar posibles mensajes *page/inquiry*. Cada vez que un dispositivo se despierta, escucha una de las 32 frecuencias de salto definidas. Un mensaje de tipo *page*, será enviado en 32 frecuencias diferentes. Primero el mensaje es enviado en las primeras 16 frecuencias (128 veces), y si no se recibe respuesta, el maestro mandará el mensaje *page* en las 16 frecuencias restantes (128 veces). El tiempo máximo de intento de conexión es de 2.56 segundos.

En el estado conectado, el dispositivo Bluetooth puede encontrarse en varios modos de operación:

- *Active mode*: En este modo, el dispositivo Bluetooth participa activamente en el canal.
- *Sniff mode*: En este modo, el tiempo de actividad durante el cual el dispositivo esclavo escucha se reduce. Esto significa que el maestro sólo puede iniciar una transmisión en unos *slots* de tiempo determinados.
- *Hold mode*: En el estado conectado, el enlace con el esclavo puede ponerse en espera. Durante este modo, el esclavo puede hacer otras cosas, como escanear en busca de otros dispositivos, atender otra *piconet*, etc.
- *Park mode*: En este estado, el esclavo no necesita participar en la *piconet*, pero aún quiere seguir sincronizado con el canal. Deja de ser miembro de la *piconet*. Esto es útil por si hay más de siete dispositivos que necesitan participar ocasionalmente en la *piconet*.

0.3 APIs Java para Bluetooth

0.3.1 Introducción:

Mientras que el hardware Bluetooth había avanzado mucho, hasta hace relativamente poco no había manera de desarrollar aplicaciones java Bluetooth – hasta que apareció JSR 82, que estandarizó la forma de desarrollar aplicaciones Bluetooth usando Java. Ésta esconde la complejidad del protocolo Bluetooth detrás de unos APIs que permiten centrarse en el desarrollo en vez de los detalles de bajo nivel del Bluetooth.

Estos APIs para Bluetooth están orientados para dispositivos que cumplan las siguientes características:

- Al menos 512K de memoria libre (ROM y RAM) (las aplicaciones necesitan memoria adicional).
- Conectividad a la red inalámbrica Bluetooth.
- Que tengan una implementación del J2ME CLDC.

0.3.2 JSR 82:

El objetivo de ésta especificación era definir un API estándar abierto, no propietario que pudiera ser usado en todos los dispositivos que implementen J2ME. Por consiguiente fue diseñado usando los APIs J2ME y el entorno de trabajo CLDC/MIDP.

Los APIs JSR 82 son muy flexibles, ya que permiten trabajar tanto con aplicaciones nativas Bluetooth como con aplicaciones Java Bluetooth.

El API intenta ofrecer las siguientes capacidades:

- Registro de servicios.
- Descubrimiento de dispositivos y servicios.
- Establecer conexiones RFCOMM, L2CAP y OBEX entre dispositivos.
- Usar dichas conexiones para mandar y recibir datos (las comunicaciones de voz no están soportadas).
- Manejar y controlar las conexiones de comunicación.
- Ofrecer seguridad a dichas actividades.

Los APIs Java para Bluetooth definen dos paquetes que dependen del paquete CLDC javax.microedition.io:

- javax.bluetooth
- javax.obex

0.3.3 Programación de aplicaciones Bluetooth:

La anatomía de una aplicación Bluetooth está dividida en cuatro partes:

- Inicialización de la pila.
- Descubrimiento de dispositivos y servicios.
- Manejo del dispositivo.
- Comunicación.

I Inicialización

I.1 BCC (Bluetooth Control Center)

Los dispositivos Bluetooth que implementen este API pueden permitir que múltiples aplicaciones se estén ejecutando concurrentemente. El BCC previene que una aplicación pueda perjudicar a otra. El BCC es un conjunto de capacidades que permiten al usuario o al OEM[2] resolver peticiones conflictivas de aplicaciones definiendo unos valores específicos para ciertos parámetros de la pila Bluetooth.

El BCC puede ser una aplicación nativa, una aplicación en un API separado, o sencillamente un grupo de parámetros fijados por el proveedor que no pueden ser cambiados por el usuario. Hay que destacar, que el BCC no es una clase o un interfaz definido en esta especificación, pero es una parte importante de su arquitectura de seguridad.

I.2 Inicialización de la pila

La pila Bluetooth es la responsable de controlar el dispositivo Bluetooth, por lo que es necesario inicializarla antes de hacer cualquier otra cosa. El proceso de inicialización consiste en un número de pasos cuyo propósito es dejar el dispositivo listo para la comunicación inalámbrica.

Desafortunadamente, la especificación deja la implementación del BCC a los vendedores, y cada vendedor maneja la inicialización de una manera diferente. En un dispositivo puede haber una aplicación con un interfaz GUI, y en otra puede ser una serie de configuraciones que no pueden ser cambiados por el usuario. Un ejemplo sería[3]:

```
...
// Configuramos el puerto
BCC.setPortNumber(“COM1”);
// Configuramos la velocidad de la conexión
BCC.setBausRate(50000);
//Configuramos el modo conectable
BCC.setConnectable(true);
//Configuramos el modo discovery a LIAC[4]
BCC.setDiscoverable(DiscoveryAgent.LIAC);
...
```

2 Original Equipment Manufacturer
3 Los APIs invocados aquí no son parte del JSR 82
4 Limited Inquiry Access Code

II Discovery

Dado que los dispositivos inalámbricos son móviles, necesitan un mecanismo que permita encontrar, conectar, y obtener información sobre las características de dichos dispositivos. En este apartado, vamos a tratar como el API de Bluetooth permite realizar todas estas tareas.

II.1 Descubrir Dispositivos (Device discovery)

II.1.1 Introducción:

Cualquier aplicación puede obtener una lista de dispositivos a los que es capaz de encontrar, usando, o bien **startInquiry()** (no bloqueante) o **retrieveDevices()** (bloqueante). **startInquiry()** requiere que la aplicación tenga especificado un *listener,*el cual es notificado cuando un nuevo dispositivo es encontrado después de haber lanzado un proceso de búsqueda. Por otra parte, si la aplicación no quiere esperar a descubrir dispositivos (o a ser descubierta por otro dispositivo) para comenzar, puede utilizar **retrieveDevices()**, que devuelve una lista de dispositivos encontrados en una búsqueda previa o bien unos que ya conozca por defecto.

II.1.2 Clases del Device Discovery:

interface javax.bluetooth.DiscoveryListener

Este interfaz permite a una aplicación especificar un evento en el *listener* que reaccione ante eventos de búsqueda. También se usa para encontrar dispositivos. El método **deviceDiscovered()** se llama cada vez que se encuentra un dispositivo en un proceso de búsqueda. Cuando el proceso de búsqueda se ha completado o cancelado, se llama al método **inquiryCompleted()**. Este método recibe un argumento, que puede ser INQUIRY_COMPLETED, INQUIRY_ERROR o INQUIRY_TERMINATED, dependiendo de cada caso.

interface javax.bluetooth.DiscoveryAgent

Esta interfaz provee métodos para descubrir dispositivos y servicios. Para descubrir dispositivos, esta clase provee del método **startInquiry()** para poner al dispositivo en modo de búsqueda, y el método **retrieveDevices()** para obtener la información de dispositivos previamente encontrados. Además provee del método **cancelInquiry()** que permite cancelar una operación de búsqueda.

II.2 Descubrir Servicios (Service Discovery)

II.2.1 Introducción:

En este capítulo vamos a ver la parte del API que usa el cliente para descubrir servicios disponibles en los dispositivos servidores encontrados. La clase DiscoveryAgent provee de métodos para buscar servicios en un dispositivo servidor Bluetooth e iniciar transacciones entre el dispositivo y el servicio. Este API no da soporte para buscar servicios que estén ubicados en el propio dispositivo.

Para descubrir los servicios disponibles en un dispositivo servidor, el cliente primero debe recuperar un objeto que encapsule funcionalidad SDAP[5] (SDP[6] + GAP[7], se verá más adelante). Este objeto es del tipo DiscoveryAgent, cuyo pseudocódigo viene dado por:

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

II.2.2 Clases del Service Discovery:

class javax.bluetooth.UUID

Esta clase encapsula enteros sin signo que pueden ser de 16, 32 ó 128 bits de longitud. Estos enteros se usan como un identificador universal cuyo valor representa un atributo del servicio. sólo los atributos de un servicio representados con UUID están representados en la Bluetooth SDP.

class javax.bluetooth.DataElement

Esta clase contiene varios tipos de datos que un atributo de servicio Bluetooth puede usar. Algunos de estos son:

- String
- boolean
- UUID
- Enteros con signo y sin signo, de uno, dos, cuatro o seis bytes de longitud
- secuencias de cualquiera de los tipos anteriores.

Esta clase además presenta una interfaz que permite construir y recuperar valores de un atributo de servicio.

class javax.bluetooth.DiscoveryAgent

Esta clase provee métodos para descubrir servicios y dispositivos.

5 SDAP=Service Discovery Application Profile
6 SDP=Service Discovery Profile
7 GAP=Generic Access Profile

interface javax.bluetooth.ServiceRecord

Este interfaz define el Service Record de Bluetooth, que contiene los pares (atributo ID, valor). El atributo ID es un entero sin signo de 16 bits, y valor es de tipo DataElement. Además, este interfaz tiene un método **populateRecord()** para recuperar los atributos de servicio deseados (pasando como parámetro al método el ID del atributo deseado).

interface javax.bluetooth.DiscoveryListener

Este interfaz permite a una aplicación especificar un *listener* que responda a un evento del tipo Service Discovery o Device Discovery. Cuando un nuevo servicio es descubierto, se llama al método **servicesDiscovered()**, y cuando la transacción ha sido completada o cancelada se llama a **serviceSearchCompleted()**.

II.3 Registro del Servicio (Service Registration)

II.3.1 Introducción:

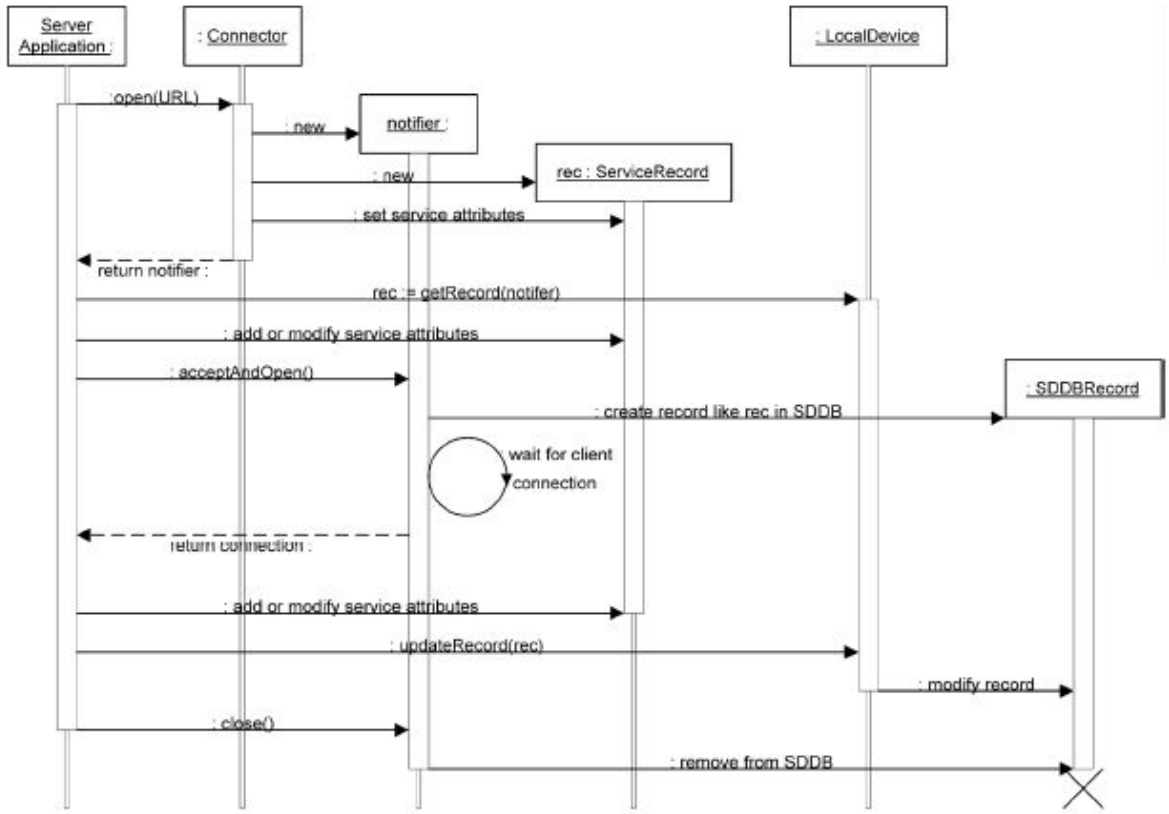
Las responsabilidades de una aplicación servidora de Bluetooth son:

1. Crear un Service Record que describa el servicio ofrecido por la aplicación.
2. Añadir el Service Record al SDDB del servidor para avisar a los clientes potenciales de este servicio.
3. Registrar las medidas de seguridad Bluetooth asociadas a un servicio.
4. Aceptar conexiones de clientes que requieran el servicio ofrecido por la aplicación.
5. Actualizar el Service Record en el SDDB del servidor si las características del servicio cambian.
6. Quitar o deshabilitar el Service Record en el SDDB del servidor cuando el servicio no está disponible.

A las tareas 1,2,5 y 6 se las denominan registro del servicio (Service Registration), que comprenden unas tareas relacionadas con advertir al cliente de los servicios disponibles.

II.3.2 Responsabilidades del Registro de Servicio:

En la figura vemos, que cuando la aplicación llama a **Connector.open()** con un String conexión URL, la implementación crea un ServiceRecord. El correspondiente registro del servicio es añadido a la SDDB por la implementación cuando la aplicación servidora llama a **acceptAndOpen()**. La aplicación servidora puede acceder a dicho ServiceRecord llamando a **getRecord()** y hacer las modificaciones pertinentes. Las modificaciones se hacen también en el ServiceRecord de la SDDB cuando la aplicación llama a **updateRecord()**. Finalmente el ServiceRecord es eliminado de la SDDB cuando la aplicación servidora manda un close al *notifier*.



Colaboracion entre la implementación y la aplicación servidora para el registro del servicio

II.3.3 Modos conectable y no conectable:

- Modo conectable: un dispositivo en este modo escucha periódicamente intentos de iniciar una conexión de un dispositivo remoto.
- Modo no-conectable: un dispositivo en este modo no escucha intentos de iniciar una conexión de un dispositivo remoto.

Para el correcto funcionamiento de una aplicación servidora, es necesario que el dispositivo servidor esté en modo conectable. Es por esto, que en la implementación de **acceptAndOpen()**, ésta debe asegurarse que el dispositivo local esté en modo conectable (dado que depende del usuario el tener o no el dispositivo en un modo u otro). La implementación hace una petición al BCC para hacer al dispositivo local conectable, si ésta no es posible, se lanzará una excepción BluetoothStateException.

Cuando todos los servicios en la SDDB han sido eliminados o deshabilitados, la implementación puede decidir opcionalmente pedir al dispositivo servidor que pase al modo no-conectable.

Aunque un dispositivo esté en el modo no-conectable (no responde a intentos de conexión), puede iniciar un intento de conexión. Por esto, un dispositivo en modo no-conectable puede ser un cliente, pero no un servidor. Por lo tanto la implementación no necesita pedir al dispositivo que se ponga en modo conectable si no tiene ningún ServiceRecord en su SDDB.

II.3.4 Clases del Service Registration:

interfaz javax.bluetooth.ServiceRecord

Un ServiceRecord describe un servicio Bluetooth a los clientes. Está compuesto de unos "cuantos" atributos de servicio. El SDP del servidor mantiene una base de datos de los ServiceRecords. Un servicio *run-before-connect* (la aplicación se ejecuta sin esperar a que haya una conexión establecida) añade su ServiceRecord a la SDDB llamando a **acceptAndOpen()**. El ServiceRecord provee suficiente información a un cliente SDP para poder conectarse al servicio Bluetooth del dispositivo servidor.

La aplicación servidora puede usar el método **setDeviceClasses()** para activar alguno de los bits de la clase servidora para reflejar la incorporación de un nuevo servicio.

class javax.bluetooth.LocalDevice

Esta clase provee un método **getRecord()** que la aplicación servidora puede usar para obtener el ServiceRecord. Una vez modificado, puede ser puesto en la SDDB llamando al método **notifier.acceptAndOpen()** o **updateRecord()** de LocalDevice.

class javax.bluetooth.ServiceRegistrationException extends java.io.IOException

La excepción ServiceRegistrationException se lanza cuando se intenta añadir o modificar un ServiceRecord en la SDDB y hay algún error. Estos errores pueden ocurrir:

- Durante la ejecución de Connector.open().
- Cuando un servicio *run-before-connect* invoca a **acceptAndOpen()** y la implementación intenta añadir el ServiceRecord asociado al *notifier* en la SDDB.
- Después de la creación del ServiceRecord, cuando la aplicación servidora intenta modificar el ServiceRecord en la SDDB usando **updateRecord()**.

II.4 Ejemplo

II.4.1 Introducción:

A continuación vamos a desarrollar un ejemplo que nos permita descubrir dispositivos (device discovery), y, una vez descubiertos, descubrir qué servicios nos ofrecen dichos dispositivos (service discovery).

II.4.2 Clase DiscoveryMIDlet

```
package discoveryBluetooth;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;
import java.util.*;

public class DiscoveryMIDlet extends MIDlet implements CommandListener{

   //Creamos las variables necesarias
   public static DiscoveryMIDlet dm;//Instancia de nuestro objeto
   private static Display display;

   //Objetos que representan a los dispositivos y a los servicios
   private Dispositivo dispositivo = null;
   private Servicio servicio = null;

   //Objetos Bluetooth necesarios
   public LocalDevice dispositivoLocal;
   public DiscoveryAgent da;

   //Lista de servicios y dispositivos
   public static Vector dispositivos_encontrados = new Vector();
   public static Vector servicios_encontrados = new Vector();

   //Instancia del dispositivo remoro seleccionado
   public static int dispositivo_seleccionado = -1;//-1 indica ninguno seleccionado

   //Constructor
   public DiscoveryMIDlet(){
      dm = this;
   }

   //Ciclo de vida del MIDlet
   public void startApp() {
      display = Display.getDisplay(this);
      dispositivo = new Dispositivo();
      servicio = new Servicio();

      //Mostramos la lista de dispositivos(vacia al principio)
      dispositivo.mostrarDispositivos();
      display.setCurrent(dispositivo);
    }

   public void pauseApp() {
   }

   public void destroyApp(boolean unconditional) {
   }

   //Este metodo se encarga de las tareas necesarias para salir del MIDlet
   public static void salir(){
      dm.destroyApp(true);
      dm.notifyDestroyed();
      dm = null;
   }

   //Este metodo se encarga de dar un aviso de alarma cuando se produce una excepcion
   public void mostrarAlarma(Exception e, Screen s, int tipo){
      Alert alerta;
```

```
        if(tipo == 0){
        alerta = new Alert("Excepcion","Se ha producido la excepcion "+e.getClass().getName(), null,
                 AlertType.ERROR);
        }else{
        alerta = new Alert("Error","No ha seleccionado un dispositivo ", null, AlertType.ERROR);
        }
        alerta.setTimeout(Alert.FOREVER);
        display.setCurrent(alerta,s);
    }

    //Este metodo se encarga de buscar dispositivos remotos Bluetoth
    public void buscarDispositivos(){
        try{
            dispositivoLocal = LocalDevice.getLocalDevice();
            //Ponemos el dispositivo en modo General Discoverable Mode
            //General/Unlimited Inquiry Access Code (GIAC)
            dispositivoLocal.setDiscoverable(DiscoveryAgent.GIAC);
            da = dispositivoLocal.getDiscoveryAgent();
            da.startInquiry(DiscoveryAgent.GIAC,new Listener());

        }catch(BluetoothStateException e){
            mostrarAlarma(e, dispositivo,0);
        }
    }

    //Este metodo se encaga de buscar servicios en un dispositivo remoto
    public void buscarServicios(RemoteDevice dispositivo_remoto){
        try{
            //Los servicios posibles vienen identificados por un UUID
            int[] servicios = new int[]{0x0001,0x0003,0x0008,0x000C,0x0100,0x000F,
                                        0x1101,0x1000,0x1001,0x1002,0x1115,0x1116,0x1117};[8]
            UUID[] uuid = new UUID[]{new UUID(0x0100)};//Servicios L2CAP
            da.searchServices(servicios,uuid,dispositivo_remoto,new Listener());
        }catch(BluetoothStateException e){
            mostrarAlarma(e, dispositivo,0);
        }
    }

    //Manejamos la accion del usuario
    public void commandAction(Command c, Displayable d){
        if(d==dispositivo && c.getLabel().equals("Descubrir dispositivos")){
            //Limpiamos la lista anterior
            dispositivos_encontrados.removeAllElements();

            //Buscamos dispositivos
            buscarDispositivos();

            //Escribimos un mensaje al usuario pidiendole que espere
            dispositivo.escribirMensaje("Por favor, espere...");
        }
        else if(d==dispositivo && c.getLabel().equals("Descubrir Servicios")){
            //Limpiamos la lista anterior
            servicios_encontrados.removeAllElements();

            //Leemos el dispositivo seleccionado
            dispositivo_seleccionado = dispositivo.getSelectedIndex();
            //Nos aseguramos que hay un disposotivo seleccionado
            if(dispositivo_seleccionado == -1) {
                mostrarAlarma(null, dispositivo,1);
                return;
            }
```

8 Consultar anexo V.2

```
            RemoteDevice dispostivoRemoto = (RemoteDevice)dispositivos_encontrados.elementAt
                                                    (dispositivo_seleccionado);
            buscarServicios(dispostivoRemoto);
        }
        else if(d==dispositivo && c.getLabel().equals("Salir")){
            salir();
        }
        else if(d==servicio && c.getLabel().equals("Atras")){
            display.setCurrent(dispositivo);
        }
    }

    //Implementamos el DiscoveryListener
    public class Listener implements DiscoveryListener{

        //Implementamos los metodos del interfaz DiscoveryListener

        public void deviceDiscovered(RemoteDevice dispositivoRemoto, DeviceClass clase){
            System.out.println("Se ha encontrado un dspositivo Bluetooth");

            dispositivos_encontrados.addElement(dispositivoRemoto);
        }

        public void inquiryCompleted(int completado){
            System.out.println("Se ha completado la busqueda de servicios");

            if(dispositivos_encontrados.size()==0){
                Alert alerta = new Alert("Problema","No se ha encontrado dispositivos",null, AlertType.INFO);
                alerta.setTimeout(3000);
                dispositivo.escribirMensaje("Presione descubrir dispositivos");
                display.setCurrent(alerta,dispositivo);
            }
            else{
                dispositivo.mostrarDispositivos();
                display.setCurrent(dispositivo);
            }
        }

        public void servicesDiscovered(int transID, ServiceRecord[] servRecord){
            System.out.println("Se ha encontrado un servicio remoto");
            for(int i=0;i<servRecord.length;i++){
                ServiceRecord record = servRecord[i];
                servicios_encontrados.addElement(servRecord);
            }
        }

        public void serviceSearchCompleted(int transID, int respCode){
        System.out.println("Terminada la busqueda de servicios");

        if(respCode==SERVICE_SEARCH_COMPLETED ||
                respCode==SERVICE_SEARCH_NO_RECORDS){
            servicio.mostrarServicios();
            display.setCurrent(servicio);
            }
            else{
                Alert alerta = new Alert("Problema","No se ha completado la busqueda",null, AlertType.INFO);
                alerta.setTimeout(Alert.FOREVER);
                display.setCurrent(alerta,dispositivo);
            }
        }
    }
}
```

II.4.3 Clase Dispositivo

```
package discoveryBluetooth;

import javax.microedition.lcdui.*;
import javax.bluetooth.*;

public class Dispositivo extends List{

   //Constructor
   public Dispositivo(){
      super("Lista de dispositivos",List.IMPLICIT);

      addCommand(new Command("Descubrir dispositivos",Command.SCREEN,1));
      addCommand(new Command("Descubrir servicios",Command.SCREEN,2));
      addCommand(new Command("Salir",Command.SCREEN,3));

      this.setCommandListener(DiscoveryMIDlet.dm);
   }

   //Este metodo se encarga de limpiar la pantalla
   public void limpiar(){
      int s = this.size();
      for(int i=0;i<0;i++){
         delete(i);
      }
   }

   //Este metodo se encarga de mostrar mensajes
   public void escribirMensaje(String str){
      limpiar();
      append(str,null);
   }

   //Este metodo muestra los "friendly names" de los dispositivos remotos
   public void mostrarDispositivos(){
      limpiar();

      if(DiscoveryMIDlet.dispositivos_encontrados.size()>0){
         for(int i=0;i<DiscoveryMIDlet.dispositivos_encontrados.size();i++){
         try{
       RemoteDevice dispositivoRemoto = (RemoteDevice)
                                              DiscoveryMIDlet.dispositivos_encontrados.elementAt(i);
            append(dispositivoRemoto.getFriendlyName(false),null);
         }catch(Exception e){
            System.out.println("Se ha producido una excepcion");
         }
         }
      }
      else append("Pulse descubrir dispositivos",null);
   }
}
```

II.4.4 Clase Servicio

```
package discoveryBluetooth;

import javax.microedition.lcdui.*;
import javax.bluetooth.*;
import java.util.*;

public class Servicio extends List{

   //Constructor
   public Servicio(){
      super("Lista de servicios",List.IMPLICIT);

      addCommand(new Command("Atras",Command.BACK,2));

      this.setCommandListener(DiscoveryMIDlet.dm);
   }

   //Mostramos la lista de servicios provinientes del record
   public void mostrarServicios(){

      int s = this.size();
      for(int i=0;i<s;i++) delete(0);
      for(int j=0;j<DiscoveryMIDlet.servicios_encontrados.size();j++){

         try{
            ServiceRecord rec = (ServiceRecord)DiscoveryMIDlet.servicios_encontrados.elementAt(j);
            DataElement e = rec.getAttributeValue(0x0001);
            Enumeration enum = (Enumeration)e.getValue();
            DataElement e2 = (DataElement)enum.nextElement();
            Object v = e2.getValue();

            String name = "#"+j+" "+uuidToName((UUID) v)[9];
            append(name, null);

         }catch(Exception e){
            System.out.println("Se ha producido una excepcion");
         }
      }

   }
```

9 Este método no está definido en la API Bluetooth. Este método se encarga de convertir el valor hexadecimal que representa a un servicio a su nombe correpondiente. Consultar el Anexo 2.

III Manejo del Dispositivo

Los dispositivos inalámbricos, son más vulnerables a ataques del tipo spoofing y eavesdropping que los demás dispositivos. Es por ello, que la tecnología Bluetooth incluye una serie de medidas para evitar estas vulnerabilidades, como es por ejemplo el salto de frecuencia; más aún, Bluetooth provee además de otros mecanismos opcionales como son la encriptación y autenticación.

Vamos a ver a continuación, las diferentes maneras en las que un dispositivo local responde a un dispositivo remoto. También veremos como usar estos mecanismos de seguridad.

III.1 Perfil de acceso genérico (GAP)

III.1.1 Introducción:

En este capítulo vamos a ver las clases que representan los objetos Bluetooth esenciales, como son LocalDevice y RemoteDevice. Las clases DeviceClass y BluetoothStateException dan soporte a la clase LocalDevice. La clase RemoteDevice representa un dispositivo remoto y provee métodos para obtener información del dispositivo remoto.

III.1.2 Clases del GAP:

class javax.bluetooth.LocalDevice

Esta clase provee acceso y control sobre el dispositivo local Bluetooth. Esta diseñada para cumplir con los requerimientos del GAP definidos para Bluetooth.

class javax.bluetooth.RemoteDevice

Esta clase representa al dispositivo Bluetooth remoto. De ella se obtiene la información basica acerca de un dispositivo remoto incluyendo su dirección Bluetooth y su *friendly name* (nombre Bluetooth del dispositivo).

class javax.bluetooth.BluetoothStateException extends java.io.IOException

Esta excepción ocurre cuando un dispositivo no puede atender una petición que normalmente atendería por culpa de las características de la conexión radio(Ej: en ocasiones algunos dispositivos no permiten a otro conectarse cuando ya están conectados a otro dispositivo).

class javax.bluetooth.DeviceClass

Esta clase define los valores del tipo de dispositivo y los tipos de servicios de un dispositivo (https://www.bluetooth.org/foundry/assignnumb/document/baseband).

III.2 Seguridad

III.2.1 Introducción:

Vamos a ver los diferentes mecanismos de seguridad implementados. Tanto el cliente como el servidor pueden incluir opcionalmente parámetros al argumento *conecction string* de **Connector.open()** para especificar la seguridad requerida para la conexión.

III.2.2 Peticiones de seguridad en el Connection String:

Las aplicaciones servidoras usan uno de los métodos open de la clase javax.microedition.io.Connector del CLDC para crear un objeto *notifier* que puede ser usado para esperar a que un cliente se conecte. Los parámetros normales del *connection string* son suficientes para crear un objeto *notifier* y el apropiado ServiceRecord, sin embargo, añadiendo ciertos parámetros, se pueden requerir autenticación, encriptación, autorización y cambios en el rol maestro/esclavo.

Peticiones de autenticación por parte del servidor:

La autenticación consiste en verificar la identidad del dispositivo remoto. La autenticación implica un reto que se lanza entre los dispositivos, que requiere una clave compartida de 128 bits derivada de un PIN compartido por ambos dispositivos. Si el PIN en ambos dispositivos falla, falla el proceso de autenticación.

El parámetro authenticate tiene la siguiente interpretación cuando se usa en el *connection string* de una aplicación servidora:

- Si authenticate=true, la implementación intenta identificar a cada cliente que intente conectarse al servicio.
- Si authenticate=false, la implementación no intenta identificar a cada cliente que intente conectarse al servicio.
- Si el parámetro authenticate no está en el *connection string*, por defecto está a *false*.

No todos los dispositivos Bluetooth soportan autenticación. Si éste es el caso, y authenticate=true en el **Connector.open()** se lanza una excepción BluetoothConnectionException. En algunos casos, puede haber conflictos entre las necesidades de seguridad de una aplicación y las medidas de seguridad tomadas por el dispositivo; algunas implementaciones de la BCC intentan evitar el conflicto preguntando al usuario si quiere cambiar la configuración del dispositivo.

Peticiones de encriptación por parte del servidor:

Cuando está activada, todos los datos transmitidos en ambas direcciones se encriptan. El parámetro encrypt tiene la siguiente interpretación cuando se usa en el *connection string* de una aplicación servidora:

- Si encrypt=true, la implementación encripta todas las comunicaciones de éste y hacia este servicio.
- Si encrypt=false, no se usa encriptación, pero ésta puede ser requerida por el cliente.
- Si encrypt no está presente en el *connection string*, por defecto está a *false*.

Dado que la encriptación requiere una clave compartida, esto significa que la encriptación requiere autenticación. Es decir, sólo ciertas combinaciones de estos parámetros están permitidas. En el caso authenticate=false y encrypt=true provocará una BluetoothConnectionException. Si encrypt=true y el parámetro authenticate no está en el *connection string*, se considerará que su valor es *true*.

Al igual que en la autenticación, no todos los dispositivos soportan encriptación. Si encrypt=true y la encriptación no está soportada, se lanzará una BluetoothConnectionException desde el **Connector.open()**.

Peticiones de autorización por parte del servidor:

La autorización Bluetooth es un procedimiento por el cual un usuario de un dispositivo servidor garantiza el acceso a un servicio específico a un cliente específico. La implementación de la autorización puede implicar preguntar al usuario del dispositivo servidor si el dispositivo cliente está autorizado a acceder a dicho servicio.

El parámetro authorize tiene la siguiente interpretación cuando se usa en el *connection string* de una aplicación servidora:

- Si authorize=true, la implementación consulta con la BCC para determinar si la petición de conexión del cliente se autoriza o no.
- Si authorize=false, todos los clientes tienen acceso al servicio.
- Si no está presente en el connection string, por defecto está a false.

Como en la encriptación, la autorización implica que la identidad del dispositivo cliente debe ser verificada mediante autenticación. Por esto, sólo ciertas combinaciones son válidas. Si authenticate=false y authorize=true se lanzará una BluetoothConnectionException. Si authorize=true y el parámetro authenticate no está presente, se considera que es *true*.

Al igual que en la autenticación y encriptación, no todos los dispositivos soportan autorización. Si authorize=true y la autorización no está soportada, se lanzará una BluetoothConnectionException desde el **Connector.open()**.

Peticiones de cambio de rol maestro/esclavo por parte del servidor:

Cada red Bluetooth tiene un dispositivo maestro cuya secuencia de salto de frecuencia (*frecuency hopping*) se usa para sincronizar de uno a siete esclavos. El dispositivo que inicia la formación del canal de comunicaciones es el maestro. A continuación vamos a ver como un esclavo pide el cambio de rol maestro/esclavo.

El parámetro master tiene la siguiente interpretación cuando se usa en el *connection string* de una aplicación servidora:

- Si master=true, tan pronto como la conexión esté establecida, la implementación hace una petición de que el cliente y el servidor cambien sus papeles.
- Si master=false, el servidor está dispuesto a ser tanto maestro como esclavo.
- Si el parámetro no está definido, por defecto esta a *false*.

No todos los dispositivos soportan cambios de rol. Si master=true y el cambio de rol no está soportado, se lanzará una BluetoothConnectionException desde el **Connector.open()**.

Peticiones por parte del cliente:

Las aplicaciones cliente también pueden usar los parámetros authenticate, encrypt y master en el connection string. Estos tienen los siguientes significados:

- Si authenticate=true la implementación intenta verificar la identidad del servicio.
- Si encrypt=true, la implementación encripta todas las comunicaciones, encrypt=true implica que authenticate=true.
- Si master=true, el cliente debe jugar el papel de maestro en la comunicación con el servidor.

Con este API, el único dispositivo que necesita garantizar permiso para usar un servicio es el que ofrece dicho servicio. Por eso, el parámetro authorize no está permitido en conexiones del cliente. Si aparece el parámetro authorize en el connection string del cliente se lanzara una BluetoothConnectionException.

Muchas de las diferentes combinaciones del *connection string* del cliente y servidor son válidas. La única excepción que no se puede dar es la de que ambos tengan master=true. En este caso el intento de conexión falla. El **Connector.open()** del cliente lanza una BluetoothConnectionException, mientras que el servidor no tiene conocimiento de este fallo ya que la implementación simplemente rechaza la conexión.

III.2.3 Clases de seguridad:

class javax.bluetooth.RemoteDevice

Esta clase contiene los métodos que pueden ser usados en cualquier momento para hacer una petición de cambio en las configuraciones de seguridad, o de averiguar la configuración actual de seguridad en la conexión. Algunos de estos métodos toman instancias de javax.microedition.io.Connector como argumento. Este tipo de argumento genérico se usa en aquellos métodos que son aplicados en conexiones a puertos serie, conexiones L2CAP o conexiones OBEX.

IV Comunicación

Para usar un servicio en un dispositivo Bluetooth remoto, el dispositivo local debe comunicarse usando el mismo protocolo que el servicio remoto. Los APIs permiten usar RFCOMM, L2CAP u OBEX como protocolo de nivel superior

El Generic Connection Framework (GFC) del CLDC provee la conexión base para la implementación de protocolos de comunicación. CLDC provee de los siguientes métodos para abrir una conexión:

```
Connection Connector.open(String name);
Connection Connector.open(String name, int mode);
Connection Connector.open(String name, int mode, boolean timeouts);
```

La implementación debe soportar abrir una conexión con una conexión URL servidora o con una conexión URL cliente, con el modo por defecto READ_WRITE.

IV.1 Perfil del puerto serie (SPP)

IV.1.1 Introducción:

El protocolo RFCOMM provee múltiples emulaciones de los puertos serie RS-232 entre dos dispositivos Bluetooth. Las direcciones Bluetooth de los dos puntos terminales definen una sesión RFCOMM. Una sesión puede tener más de una conexión, el número de conexiones dependerá de la implementación. Un dispositivo podrá tener más de una sesión RFCOMM en tanto que esté conectado a más de un dispositivo.

IV.1.2 Un vistazo al API:

Una aplicación que ofrezca un servicio basado en el perfil de puerto serie (SPP) es un servidor SPP. Una aplicacion que inicie una conexión a un servicio SPP es un cliente SPP. Cliente y servidor residen en los extremos de una sesión RFCOMM. El servidor SPP registra su servicio en el SDDB, y como parte del proceso de registro, se añade un identificador de canal (*channel identifier*) al ServiceRecord por la implementación.

En este capítulo vamos a ver las capacidades que una implementación SPP tiene que tener para los interfaces StreamConnection y StreamConnectionNotifier del CLDC.

IV.1.3 Conexiones URL de un cliente y servidor SPP :

A continuacion vamos a mostrar algunos de los argumentos (ABNF) necesarios para la conexión URL entre clientes y servidores.

```
srvString =  protocol colon slashes srvHost 0*5(srvParams)
cliString  =  protocol colon slashes cliHost 0*3(cliParams)

protocol = btspp

btspp = %d98.116.115.112.112      // define el literal btspp

cliHost = address colon channel
srvHost = "localhost" colon uuid

channel = %d1 -30
uidd = 1*32(HEXDIG)

colon = ":"
slashes = "//"
bool = "true" / "false"
address = 12*12(HEXDIG)
text = 1*( ALPHA/ DIGIT / SP / "-" / "_" )

name = ";name=" text
master = ";master=" bool
encrypt = ";encrypt=" bool
authorize = ";authorize=" bool
authenticate = ";authenticate=" bool
cliParams = master / encrypt / authenticate
servParams = name / master / encrypt / authorize / authenticate
```

SP se usa para espacios, ALPHA para letras alfabéticas mayúsculas y minúsculas, DIGIT se usa para números de cero a nueve y HEXDIG para números hexadecimales (0-9, a-f,A-F).

IV.1.4 Registro del servicio del puerto serie :

Un SPP debe inicializar los servicios que ofrece y registrarlos en el SDDB. Un servicio de puerto serie viene representado por un par de objetos emparentados:

1. Un objeto que implementa el interfaz javax.microedition.io.StreamConnectorNotifier Este objeto escucha conexiones clientes que demanden este servicio.
2. Un objeto que implemente el interfaz javax.bluetooth.ServiceRecord. Este objeto describe el servicio y como puede ser accedido por dispositivos remotos.

Para crear estos objetos la aplicación servidora usa el método **Connector.open()** con un argumento de conexión URL, del siguiente modo:

```
StreamConnectionNotifier service = (StreamConnectionNotifier)Connector.open
("btspp://localhost:102030405060708090A1B1C1D1D1E100;name=SPPEx");
```

Invocando **Connector.open()** con un argumento conexión URL, éste devuelve un StreamConnectionNotifier que representa el servicio SPP. La implementación de **Connector.open() además** crea un nuevo registro de servicio (*service record*) que representa el servicio SPP. Una implementación de un SPP debe realizar los siguientes pasos cuando crea el registro de servicio.

1. Crear un identificador de un canal servidor RFCOMM, chanN y asignarlo.
2. chanN es añadido al ProtocolDescritorList en el registro de servicio.
3. El UUID (102030...) usado en el c*onnection string* para describir el tipo de servicio ofrecido es añadido al ServiceClassIDList.
4. El atributo ServiceName es añadido al registro de servicio con el valor "SPPEx".

En el caso de un servicio *run-before-connect*, el registro de servicio es añadido a la SDDB la primera vez que la aplicación servidora llama a **acceptAndOpen()** en el StreamConnectionNotifier asociado. El registro de servicio se hace visible a potenciales clientes SPP cuando es añadida a la SDDB.

IV.2 Establecimiento de la conexión

IV.2.1 Establecimiento de la conexión del servidor:

Un servidor SPP crea un objeto StreamConnectionNotifier del siguiente modo:

- Usando el apropiado string para un servidor SPP como argumento de **Connector.open()**
- Haciendo un *casting* del resultado de **Connector.open()** al interfaz StreamConnectionNotifier.

```
StreamConnectionNotifier service = (StreamConnectionNotifier)Connector.open
("btspp://localhost:102030405060708090A1B1C1D1D1E100;name=SPPEx");

StreamConnection con = (StreamConnection) service.acceptAndOpen();
```

Un servicio SPP puede aceptar múltiples conexiones de diferentes clientes llamando a **acceptAndOpen()** repetidamente. Cada cliente accede al mismo registro de servicio y se conecta al servicio usando el mismo canal servidor RFCOMM. Si el sistema Bluetooth no soporta múltiples conexiones, el a**cceptAndOpen()** lanzará un BluetoothStateException.

El método **close()** en el objeto StreamConnection representa que se ha usado una conexión SPP servidora para cerrar la conexión.

Cuando un servicio *run-before-connect* manda un mensaje **close()** al StreamConnectionNotifier, el registro de servicio asociado a ese *notifier* se vuelve inaccesible a los clientes que estén usando el servicio *discovery*. La implementación debe eliminar el registro de servicio de la SDDB. El mensaje **close()** además hace que la implementación desactive los bits de clase que fueron activados por **setServiceClasses()** (excepto si otro *notifier* activó esos bits y aún está activo).

IV.2.2 Establecimiento de la conexión del cliente:

Antes de que un cliente SPP pueda establecer una conexión con un servicio SPP, éste debe previamente haber descubierto el servicio mediante el servicio *discovery*. Una conexión URL del cliente incluye la dirección Bluetooth del dispositivo servidor y el identificador de canal del servidor. El método **getConnectionURL()** en el interfaz ServiceRecord se usa para obtener la conexión URL del cliente.

Invocando el método **Connector.open()** con una conexión URL del cliente, devuelve un objeto StreamConnection que representa la conexión SPP del lado del cliente.

```
StreamConnection con =
(StreamConnection) Connector.open("btspp://dirección:identificador_canal");
```

IV.2.3 Registro de servicio del SPP:

Los registros de servicio consisten en una colección de pares (attrID, attrValue). Cada par describe un atributo del servicio. La aplicación servidora puede opcionalmente añadir otros atributos de servicio al ServiceRecord. Es posible incluso añadir atributos definidos por el usuario.

Con el método **updateServiceAvailability()**, la aplicación servidora puede obtener el ServiceRecord que fue creado por:

```
ServiceRecord record = localDev.getRecord(notifier);
```

La aplicación servidora modificará el atributo ServiceAviability basándose en el número de conexiones de cliente actuales. Las modificaciones que la aplicación servidora hace al ServiceRecord no se reflejan inmediatamente en la SDDB, si no que para ello se usará:

```
localDev.updateRecord(record);
```

IV.2.4 Restricciones en la modificación de los registros de servicio:

Las aplicaciones sólo pueden acceder a sus propios *notifiers*, no es posible para una aplicación modificar el ServiceRecord de otra aplicación en el SDDB servidor .

El ProtocolDescriptorList le dice a una aplicación cliente como conectarse al servicio. Protocol0 Protocol1 representan la pila (L2CAP, RFCOMM) que normalmente se usa para conectar a un servicio de puerto serie. Esos atributos son fijos para asegurarse que esa pila esté siempre en el ProtocolDescriptorList.

ProtocolSpecificParameter0 es el identificador del canal servidor. Es un atributo fijo para asegurarse que la implementación de RFCOMM puede manejar la asignación de los valores del canal servidor.

IV.3 L2CAP

IV.3.1 Introducción:

En este capítulo vamos a ver el protocolo de adaptación y control lógico del enlace (*Logical Link Control and Adaptation Protocol*). L2CAP soporta dos tipos de conexiones, orientadas a conexión (bidireccionales) y no orientadas a conexión (unidireccionales). Todas las conexiones hechas usando la primitiva de servicio connect de la capa L2CAP son orientadas a conexión, este API no soporta comunicaciones en grupo, y por lo tanto no soporta canales no orientados a conexión.

IV.3.2 Un vistazo al API:

Este API soporta sólo canales L2CAP orientados a conexión. El L2CAPConnectionNotifier le indica a un servidor L2CAP cuando un cliente inicia una conexión. Una vez que la conexión está establecida, se devuelve un objeto L2CAPConnection. El interfaz L2CAPConnection y L2CAPConnectionNotifier extienden el interfaz Connection. Este interfaz puede ser usado para enviar o recibir datos de un dispositivo remoto usando el protocolo L2CAP.

IV.3.3 Configuración del canal:

Los canales orientados a conexión necesitan ser configurados una vez que se ha establecido la conexión. Los parámetros de configuración del canal que se negocian entre los dispositivos Bluetooth son:

- Unidad máxima de transferencia (MTU): Es el tamaño del *payload* que el que envía la petición es capaz de atender. El valor por defecto es de 672 bytes (DEFAULT_MTU).
- Tiempo de descarte: Es la cantidad de tiempo durante el cual el administrador del canal intenta transmitir satisfactoriamente el paquete antes de descartarlo. El valor 0xFFFF (valor por defecto) indica que el paquete será retransmitido hasta que llegue un asentimiento o hasta que el enlace ACL termine.
- Calidad del servicio (QoS): esta opción describe el flujo de tráfico. Este parámetro no está soportado en el API.

Unidad máxima de transferencia (MTU)

La implementación es responsable de configurar el canal con la MTU pedida o usando la que tiene por defecto, antes de cualquier operación de lectura o escritura. El parámetro ReceiveMTU es el número máximo de bytes que el dispositivo local puede recibir en el *payload*. El parámetro TransmitMTU es el número máximo de bytes que el dispositivo local puede mandar al dispositivo remoto en el *payload*.

Si $ReceiveMTU_A < TransmitMTU_B$ o $TransmitMTU_A < ReceiveMTU_B$ la conexión falla.

Vamos a ver como la implementación configura la MTU de acuerdo a la petición hecha. Hay diferentes posibilidades:

1. La aplicación especifica el ReceiveMTU y el TransmitMTU. En este caso la implementación advierte del valor del ReceiveMTU al dispositivo remoto. Si el dispositivo remoto responde con una respuesta de configuración negativa la conexión falla, si es positivo, la implementación espera a la petición de configuración del dispositivo remoto. Cuando la recibe compara el valor que le ha venido de ReceiveMTU con el TransmitMTU que ha mandado, si TransmitMTU es menor o igual, la conexión es posible, sino la conexión falla.
2. La aplicación especifica ReceiveMTU pero no TransmitMTU. La aplicación puede usar el método **getTransmitMTU()** de la clase L2CAPConnection para obtener la MTU de salida para evitar enviar muchos datos.
3. La aplicación especifica TransmitMTU. En este caso la aplicación avisa al dispositivo remoto que el ReceiveMTU va a ser el DEFAULT_MTU (672bytes).
4. Si la aplicación no especifica ninguno de los dos parámetros, estamos en un caso similar al caso 2 y 3.

IV.3.4 Interfaz de conexión L2CAP:

Conexiones URL de un cliente y servidor L2CAP:

```
srvString =  protocol colon slashes srvHost 0*7(srvParams)
cliString  =  protocol colon slashes cliHost 0*5(cliParams)

protocol = btl2cap
btspp = %d98.116.108.50.99.97.112              // define el literal l2cap

cliHost = address colon psm
srvHost = “localhost” colon uuid

psm = 4*4(HEXDIG)
uidd = 1*32(HEXDIG)

colon = “:”
slashes = “//”
bool = “true” / “false”
address = 12*12(HEXDIG)
text = 1*( ALPHA/ DIGIT / SP / “-” / “_” )

name = “;name=” text
master = “;master=” bool
encrypt = “;encrypt=” bool
authorize = “;authorize=” bool
authenticate = “;authenticate=” bool
receiveMTU = “;receiveMTU=” 1*(DIGIT)
transmitMTU = “transmitMTU=” 1*(DIGTI)
cliParams = master / encrypt / authenticate / receiveMTU / transmitMTU
servParams = name / master / encrypt / authorize / authenticate/ receiveMTU /
                transmitMTU
```

SP se usa para espacios, ALPHA para letras alfabéticas mayúsculas y minúsculas, DIGIT se usa para números de cero a nueve y HEXDIG para números hexadecimales (0-9, a-f,A-F).

El string psm es un descriptor de la conexión, y representa el *Protocol Service Multiplexor*. De este modo las aplicaciones servidoras L2CAP se pueden diferenciar entre si. Este valor está comprendido entre (0x1001..0xFFFF), siendo el bit menos significativo impar y todos los demás pares.

El pseudocódigo para abrir una conexión cliente L2CAP es:

```
try{
    L2CAPConnection client = (L2CAPConnection)
    Connctor.open("btl2cap://...:...;ReceiveMTU=xxx;TransmitMTU=yyy");
}catch(...)
```

El pseudocódigo para abrir una conexión servidor L2CAP es:

```
try{
    L2CAPConnectionNotifier server = (L2CAPConnectionNotifier)
    Connctor.open("btl2cap://localhost:...;name=L2CAPEx");
    L2CAPConnection con = (L2CAPConnection) server.acceptAndOpen();
}catch(...)
```

En el caso de que haya habido algún fallo durante la conexión y se haya lanzado alguna excepción, se puede obtener el motivo del fallo usando el método **getStatus()**.

Registro de servicio L2CAP:

Cuando una aplicación servidora L2CAP llama a **Connector.open()**, se crea un registro de servicio de manera similar a como se hacía en los servicios de puerto serie

IV.3.5 Clases de conexión L2CAP:

interface javax.bluetooth.L2CAPConnection extends javax.microedition.io.Connection

Este interfaz representa las conexiones L2CAP. Contiene métodos para obtener las MTUs usadas en la conexión y métodos para enviar y recibir datos.

interface javax.bluetooth.L2CAPConnectionNotifier extends javax.microedition.io.Connection

El único método de este interfaz es **acceptAndOpen()**, que es usado por los servidores L2CAP para escuchar conexiones de clientes.

Class javax.bluetooth.BluetoothConnectionException extends java.io.IOException

Esta excepción se lanza cuando una conexión Bluetooth (RFCOMM o L2CAP) no puede ser establecida satisfactoriamente. El método **getStatus()** de esta clase indicará la razón del fallo de la conexión.

IV.4 Protocolo de intercambio de objetos (OBEX)

IV.4.1 Introducción:

En este apartado vamos a tratar el protocolo de intercambio de objetos. En realidad, este tema aparentemente no se debería tratar, ya que dicho protocolo OBEX no está definido en el API Bluetooth, si no que tiene su propio API.

Este es un protocolo diseñado por el IrDA (Infrared Data Association) para intercambiar objetos entre clientes y servidores, mediante el establecimiento de una sesión OBEX. En vez de incluir esta funcionalidad en el API Bluetooth, se ha optado por extender el API OBEX y dar soporte a Bluetooth. Nosotros vamos a centrarnos únicamente en el soporte Bluetooth.

IV.4.2 Un vistazo al API:

OBEX implementa la transferencia de objetos estableciendo una sesión OBEX, mediante una petición CONNECT. Ésta termina mediante una petición DISCONECT. Entre estas dos peticiones, el cliente puede traer objetos del servidor mediante GET, o enviarlos mediante PUT. Los objetos pueden ser archivos, vCards, arrais de bytes,etc. El cliente puede además cambiar el archivo o carpeta en uso mediante la petición SETPATH. Otras operaciones permitidas son; ABORT, CREATE-EMPTY, PUT-DELETE.

OBEX, como HTTP, tiene métodos que le permiten pasar información adicional entre el cliente y el servidor mediante el uso de cabeceras.

Header Name	How to Manipulate the Header in the API
Count	HeaderSet.getHeader(), HeaderSet.setHeader()
Name	HeaderSet.getHeader(), HeaderSet.setHeader()
Type	HeaderSet.getHeader(), HeaderSet.setHeader()
Length	HeaderSet.getHeader(), HeaderSet.setHeader()
Time	HeaderSet.getHeader(), HeaderSet.setHeader()
Description	HeaderSet.getHeader(), HeaderSet.setHeader()
Target	HeaderSet.getHeader(), HeaderSet.setHeader()
HTTP	HeaderSet.getHeader(), HeaderSet.setHeader()
Body	Operation.openInputStream(), Operation.openDataInputStream(), Operation.openOutputStream(), Operation.openDataOutputStream()
End of Body	Operation.openInputStream(), Operation.openDataInputStream(), Operation.openOutputStream(), Operation.openDataOutputStream()
Who	HeaderSet.getHeader(), HeaderSet.setHeader()
Connection ID	ClientSession.setConnectionID(), ClientSession.getConnectionID(), ServerRequestHandler.setConnectionID(), ServerRequestHandler.getConnectionID()
Application Parameters	HeaderSet.getHeader(), HeaderSet.setHeader()
Authentication Challenge	HeaderSet.createAuthenticationChallenge(), Authenticator.getPasswordAuthentication()
Authentication Response	Authenticator.getPasswordAuthentication(), Authenticator.validatePassword()
Object Class	HeaderSet.getHeader(), HeaderSet.setHeader()
User Defined	HeaderSet.getHeader(), HeaderSet.setHeader()

Cuando se hace la llamada a **Connector.open()** se pueden dar las siguientes excepciones:

- ConnectionNotFoundException : se lanza cuando el entorno usado no es válido o cuando el tipo de protocolo no existe.
- IllegalArgumentException: se lanza cuando los parámetros del *connection string* no se reconocen.
- IOException: se lanza cuando no se puede conectar con el objetivo.

IV.4.3 Conexión del cliente:

Para crear una conexión OBEX, el cliente le debe parar el string apropiado al **Connector.open()**, y este devolverá un objeto javax.obex.ClientSession. Para establecer la conexión OBEX el cliente crea un objeto javax.obex.HeaderSet usando el método **createHeaderSet ()** del interfaz ClientSession. Finalmente el cliente facilita el objeto HeaderSet al método **connect()** de la interfaz ClientSession.

Para determinar si la petición ha tenido éxito o no, se usa el método **getResponseCode()** del interfaz HeaderSet, que devuelve un código de respuesta mandado por el servidor, que viene definido en la clase javax.obex.ResponseCodes.

Para la petición DISCONECT, se procede del mismo modo, excepto que en vez de usar el método **connect()** se usa el método **disconnect()**.

Para completar una operación SETPATH, el cliente llama al método **setPath()** en el objeto ClientSession. Para especificar el nombre del directorio destino, pone el nombre llamando al método **setHeader()** del HeaderSet. Si la cabecera es muy larga se lanzará una excepción java.io.IOException.

Para completar una operación GET o PUT, el cliente crea un objeto javax.obex.HeaderSet con el método **createHeaderSet()**. Después de establecer los valores de cabecera, el cliente llama a los métodos **put()** o **get()** del objeto javax.obex.ClientSession.

Para abortar un PUT o un GET, el cliente llama al método **abort()** del objeto javax.obex.Operation. El método **abort()** llama además al método **close()** del objeto Operation.

IV.4.4 Conexión del servidor:

Para crear una conexión servidora, el servidor invoca a **Connector.open()**, que le devuelve un objeto javax.obex.SessionNotifier. Este objeto espera a que el cliente cree una capa de transporte llamando a **acceptAndOpen().**

El servidor debe crear una nueva clase que extienda la clase javax.obex.serverRequestHandler. El servidor deberá implementar aquellos métodos de OBEX a los que da soporte. Las aplicaciones servidoras no deben llamar al método **abort()**, ya que si no el argumento javax.obex.Operation, que es parte de los métodos **onGet()** y **onPut()**, lanzará una java.io.IoException.

IV.4.5 Autenticación:

Para autenticar a un cliente o servidor OBEX, tanto el cliente como el servidor deben compartir un secreto o un password. Éste nunca es intercambiado durante el proceso de autenticación. Si el cliente quiere autenticar al servidor, éste le envía una cabecera con un reto (de 16 bytes). Con ésta, el servidor determina el password o el secreto. Entonces el servidor combina el password con el reto aplicando el algoritmo MD5. El resultado (llamado *response digest*) se le devuelve en la cabecera de autenticación. Entonces el cliente compara el reto que mandó combinado con el password y el algoritmo MD5 con el que ha recibido; si son el mismo, el servidor se ha autenticado.

El proceso de autenticación comienza con la llamada al método **createAuthenticationchallenge()**, el cual le indica a la implementación que incluya un reto de autenticación en la siguiente petición o respuesta.

Para facilitar el proceso de autenticación, el interfaz Authenticator provee de métodos que pueden ser implementados por la aplicación para responder retos. El método **onAuthenticationChallenge()** es invocado cuando una cabecera con un reto de autenticación es recibida. Cuando se recibe la respuesta de una autenticación, se llama al método **onAuthenticationResponse()** con el nombre de usuario (si está incluido en la cabecera de respuesta).

Si el proceso de autenticación falla, cuando el cliente invoque **connect()**, **setPath()**, **delete()**, **get()**, **put()** o **disconnect()** se producirá un IOException lanzado por el método. Si los valores no son iguales en el servidor OBEX, se llamará al método **onAuthenticationFailure()** en el ServerRequestHandler del servidor.

IV.4.6 Clases OBEX:

interface javax.obex.ClientSession extends javax.microedition.io.Connection

Este interfaz representa los objetos de conexión del lado del cliente. Provee de los métodos para las peticiones CONNECT, DISCONNECT, SETPATH, PUTDELETE, CREATE-EMPTY, PUT y GET.

interface javax.obex.HeaderSet

Este interfaz define las cabeceras OBEX que deben ser implementadas en una operación. Provee de los métodos **get()** y **set()**. Los clientes pueden crear una objeto HeaderSet llamado al método **createHeader()** del objeto javax.obex.ClientSession.

class javax.obex.ResponseCodes

Esta clase implementa los códigos de respuesta válidos en un servidor OBEX

class jaxa.obex.ServerRequestHandler

Esta clase define el esquema de como el cliente OBEX maneja las peticiones. La aplicación que extiende esta clase sólo necesita reescribir aquellos métodos que soporta.

interface javax.obex.SessionNotifier extends javax.microedition.io.Connection

Este interfaz define el objeto de sesión *notifier* que es devuelto siguiendo a una llamada a **Connector.open()**. Provee además de métodos para esperar a un cliente para que establezca una conexión.

interface javax.obex.Operation extends javax.microedition.io.ContentConnection

Esta interfaz define un objeto de operación que es usado para las operaciones GET y PUT. Además provee del método ABORT.

interface Authenticator

Este interfaz maneja la autenticación y las cabeceras de respuesta de autenticación

class PasswordAuthentication

Esta clase encapsula el nombre de usuario y el password usados en la autenticación.

IV.5 Ejemplo: “Hola Mundo”

IV.5.1 Introducción:

Para asentar bien los conceptos relacionados con la comunicación bluetooth, vamos a desarrollar un ejemplo de comunicación bluetooth, usando para ello el perfil de puerto serie (SPP).

Este ejemplo va a seguir la estructura clásica cliente-servidor. Vamos a ofrecer un servicio de envío de mensajes. El servidor será el encargado de dar a conocer su servicio, y quedarse a la espera de las conexiones de los clientes[10]. Por su parte, el cliente se encargará de buscar el servidor y el servicio que ofrece, y, una vez encontrado el servidor y su servicio (que debe coincidir con el servicio que estamos buscando) vamos a enviar el mensaje “Hola Mundo”[11] a este.

10 En éste ejemplo, por sencillez, el servidor sólo aceptará una única conexión cliente.
11 Este mensaje es el que estará puesto por defecto, pero se dará al usuario la posibilidad de mandar otro mensaje.

IV.5.2 Clase SPPServidorMIDlet

```
package SPPBluetooth;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class SPPServidorMIDlet extends MIDlet implements CommandListener {

   //Creamos las variables necesarias
   public static SPPServidorMIDlet SPPs= null;
   public static Display display;

   private SPPServidor s =null;

   //Constructor
   public SPPServidorMIDlet() {
      SPPs = this;
   }

   //Implementamos el ciclo de vida del MIDlet
   public void startApp() {
      display = Display.getDisplay(this);
      s = new SPPServidor();
      s.inicializar();

      display.setCurrent(s);
   }


   public void pauseApp() {
   }


   public void destroyApp(boolean unconditional) {
   }

   //Este metodo se encarga de las tareas necesarias para salir del MIDlet
   public void salir(){
      SPPs.destroyApp(true);
      SPPs.notifyDestroyed();
      SPPs = null;
   }

   //Manejamos la accion del usuario
   public void commandAction(Command c, Displayable d) {

      if (d == s && c.getLabel().equals("Salir")) {
         //Salimos de la aplicacion
         try{
            s.fin = true;
            s.servidor.close();
         }
         catch(Exception e){}

         salir();
      }
   }

}
```

IV.5.3 Clase SPPServidor

```
package SPPBluetooth;

import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import javax.bluetooth.*;
import java.io.*;

public class SPPServidor extends List implements Runnable{

   //Objetos Bluetooth necesarios
   public LocalDevice dispositivoLocal;
   public DiscoveryAgent da;

   public boolean fin = false;

   public StreamConnectionNotifier servidor;

   //Constructor
   public SPPServidor(){
      super("Servidor SPP",List.EXCLUSIVE);

      addCommand(new Command("Salir",Command.EXIT,1));

      setCommandListener(SPPServidorMIDlet.SPPs);
   }

   //Este metodo se encarga de dar un aviso de alarma cuando se produce una excepcion
   public void mostrarAlarma(Exception e, Screen s, int tipo){
      Alert alerta;
      if(tipo == 0){
         alerta = new Alert("Excepcion:","Se ha producido la excepcion "+e.getClass().getName(), null,
AlertType.ERROR);
      }else{
      alerta = new Alert("Error:","No ha seleccionado un dispositivo ", null, AlertType.ERROR);
      }
      alerta.setTimeout(Alert.FOREVER);
      SPPServidorMIDlet.display.setCurrent(alerta,s);
   }

   //Este metodo se va a encargar de inicializar el servidor
   public void inicializar(){
      try{
         dispositivoLocal = LocalDevice.getLocalDevice();
         dispositivoLocal.setDiscoverable(DiscoveryAgent.GIAC);

         //Lanzamos un hilo servidor (solo aceptara un cliente)
         Thread hilo = new Thread(this);
         hilo.start();
      }
      catch(BluetoothStateException be){
         System.out.println("Se ha producido un error al inicializar el hilo servidor");
      }
   }

   public void run(){
      //Le damos un nombre a nuestra aplicacion
      String nombre = "Ejemplo SPP";

      //Definimos un UUID unico para este servicio. Elegimos uno a nuestro gusto.
      UUID uuid = new UUID(0xABCD);
```

```
        servidor = null;//Similar a un socket servidor
        StreamConnection sc = null;//Similar a un socket cliente

        RemoteDevice rd = null;

        //Intentamos crear una conexion, usando SPP
        try{
                    servidor  =  (StreamConnectionNotifier)Connector.open("btspp://localhost:"+uuid.toString()
+";name="+nombre);
           //Obtenemos el service record
           ServiceRecord rec = dispositivoLocal.getRecord(servidor);
           //Rellenamos el BluetoothProfileDescriptionList usando el SerialPort version 1
           DataElement e1 = new DataElement(DataElement.DATSEQ);
           DataElement e2 = new DataElement(DataElement.DATSEQ);
          e2.addElement(new DataElement(DataElement.UUID,new UUID(0x1101)));//agregamos el puerto
                                                                                    serie
           e2.addElement(new DataElement(DataElement.INT_8,1));//version 1
           e1.addElement(e2);
           //agregamos al service record el BluetoothProfileDescriptionList
           rec.setAttributeValue(0x0009,e1);
        }
        catch(Exception e){
           System.out.println("Se ha producido un error al lanzar el hilo servidor");
           mostrarAlarma(e, this,0);
           return;
        }

        while(!fin){
           try{
              append("Esperando mensaje:",null);
              //Aceptamos conexiones del cliente y obtenemos el objeto remoto
              sc = servidor.acceptAndOpen();
              rd = rd.getRemoteDevice(sc);
              //Obtenemos el input stream del objeto remoto
              DataInputStream in = sc.openDataInputStream();
              //Leemos el mensaje y lo mostramos por pantalla
              append(in.readUTF(),null);
              //Cerramos la conexion
              sc.close();
           }
           catch(Exception e){
              mostrarAlarma(e,this, 0);
              return;
           }
        }
     }
}
```

IV.5.4 Clase SPPClienteMIDlet

```
package SPPBluetooth;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
```

```
public class SPPClienteMIDlet extends MIDlet implements CommandListener {

    //Creamos las variables necesarias
    public static SPPClienteMIDlet SPPc= null;
    public static Display display;

    private SPPCliente c =null;
    private Mensaje msg = null;

    //Objetos Bluetooth necesarios
    public LocalDevice dispositivoLocal;
    public DiscoveryAgent da;

    //Lista de dispositivos y servicios encontrados
    public static Vector dispositivos_encontrados = new Vector();
    public static Vector servicios_encontrados = new Vector();

    public static int dispositivo_seleccionado = -1;

    //Constructor
    public SPPClienteMIDlet(){
        SPPc = this;
    }

    //Implementamos el ciclo de vida del MIDlet
    public void startApp() {
        display = Display.getDisplay(this);

        c = new SPPCliente();
        msg = new Mensaje();

        //Mostramos la lista de dispositivos(vacia al principio)
        c.mostrarDispositivos();
        display.setCurrent(c);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    //Este metodo se encarga de las tareas necesarias para salir del MIDlet
    public static void salir(){
        SPPc.destroyApp(true);
        SPPc.notifyDestroyed();
        SPPc = null;
    }

    //Este metodo se encarga de dar un aviso de alarma cuando se produce una excepcion
    public void mostrarAlarma(Exception e, Screen s, int tipo){
        Alert alerta=null;
        if(tipo == 0){
            alerta = new Alert("Excepcion","Se ha producido la excepcion "+e.getClass().getName(), null,
AlertType.ERROR);
        }
        else if(tipo==1){
        alerta = new Alert("Error","No ha seleccionado un dispositivo ", null, AlertType.ERROR);
        }
        else if(tipo==2){
        alerta = new Alert("Informacion","El mensaje ha sido enviado ", null, AlertType.INFO);
        }
```

```
        alerta.setTimeout(Alert.FOREVER);
        display.setCurrent(alerta,s);
    }

    //Manejamos la accion del usuario
    public void commandAction(Command co, Displayable d){
        if(d==c && co.getLabel().equals("Busqueda")){
            //Limpiamos la lista
            dispositivos_encontrados.removeAllElements();
            servicios_encontrados.removeAllElements();
            try{
                dispositivoLocal = LocalDevice.getLocalDevice();
                dispositivoLocal.setDiscoverable(DiscoveryAgent.GIAC);
                da = dispositivoLocal.getDiscoveryAgent();
                da.startInquiry(DiscoveryAgent.GIAC,new Listener());

                c.escribirMensaje("Por favor espere...");
            }
            catch(BluetoothStateException be){
                mostrarAlarma(be,c, 0);
            }
        }
        else if(d==c && co.getLabel().equals("Enviar")){
            dispositivo_seleccionado = c.getSelectedIndex();
            //Nos aseguramos de que el usuario selecciono un dispositivo
            if(dispositivo_seleccionado == -1 || dispositivo_seleccionado >= dispositivos_encontrados.size()){
                mostrarAlarma(null, c,1);
                return;
            }
            display.setCurrent(msg);
        }
        else if(d==c && co.getLabel().equals("Salir")){
            salir();
        }
        else if(d==msg && co.getLabel().equals("OK")){
            servicios_encontrados.removeAllElements();
            //Buscamos el servicio de puerto serie en el dispositivo seleccionado
            RemoteDevice dispositivo_remoto =(RemoteDevice)dispositivos_encontrados.elementAt
                                                    (dispositivo_seleccionado);
            try{
                //Buscamos en el puerto serie 0x1101
                da.searchServices(null,new UUID[]{new UUID(0x1101)},dispositivo_remoto,new Listener());
            }
            catch(BluetoothStateException be){
                mostrarAlarma(be, c, 0);
            }
        }
    }

    //Este metodo se va a encargar de enviar un mensaje al primer ServiceRecord usando el
    //Serial Port Profile
    public void enviarMensaje(String msg){
        ServiceRecord sr = (ServiceRecord)servicios_encontrados.elementAt(0);
        //Obtenemos la URL asociada a este servicio en el dispositivo remoto
        String URL = sr.getConnectionURL(ServiceRecord.NOAUTHENTICATE_NOENCRYPT,false);
        try{
            //Obtenemos la conexio y el stream de este servicio
            StreamConnection con = (StreamConnection)Connector.open(URL);
            DataOutputStream out = con.openDataOutputStream();
            //Escribimos datos en el stream
            out.writeUTF(msg);
            out.flush();
```

```
            //Cerramos la conexion
            out.close();
            con.close();
            mostrarAlarma(null, c, 2);
        }
        catch(Exception e){
            mostrarAlarma(e, c, 0);
        }
    }

     //Implementamos el DiscoveryListener
    public class Listener implements DiscoveryListener{

    //Implementamos los metodos del interfaz DiscoveryListener

        public void deviceDiscovered(RemoteDevice dispositivoRemoto, DeviceClass clase){
            System.out.println("Se ha encontrado un dspositivo Bluetooth");

            dispositivos_encontrados.addElement(dispositivoRemoto);
        }

        public void inquiryCompleted(int completado){
            System.out.println("Se ha completado la busqueda de dispositivos");

            if(dispositivos_encontrados.size()==0){
                Alert alerta = new Alert("Problema","No se ha encontrado dispositivos",null, AlertType.INFO);
                alerta.setTimeout(3000);
                c.escribirMensaje("Presione descubrir dispositivos");
                display.setCurrent(alerta,c);
            }
            else{
                c.mostrarDispositivos();
                display.setCurrent(c);
            }
        }

        public void servicesDiscovered(int transID, ServiceRecord[] servRecord){
            System.out.println("Se ha encontrado un servicio remoto");
            for(int i=0;i<servRecord.length;i++){
                ServiceRecord record = servRecord[i];
                servicios_encontrados.addElement(servRecord);
            }
        }

        public void serviceSearchCompleted(int transID, int respCode){
            System.out.println("Terminada la busqueda de servicios");

            //Si encontramos un servicio, lo usamos para mandar el mensaje(todos los
            //servicios que hemos buscado son de puerto serie)
            if(servicios_encontrados.size()>0){
                enviarMensaje(msg.getString());
            }
            else{
                //Si no encontramos ningun servicio de puerto serie
                c.mostrarDispositivos();
                display.setCurrent(c);
            }
        }
    }
}
```

IV.5.5 Clase SPPCliente

```
package SPPBluetooth;

import javax.microedition.lcdui.*;
import javax.bluetooth.*;

public class SPPCliente extends List{

   public SPPCliente(){
      super("Cliente SPP",List.EXCLUSIVE);
      addCommand(new Command("Busqueda",Command.SCREEN, 1));
      addCommand(new Command("Enviar",Command.SCREEN, 1));
      addCommand(new Command("Salir",Command.EXIT, 1));

      this.setCommandListener(SPPClienteMIDlet.SPPc);
   }

   //Este metodo se encarga de limpiar la pantalla y de mostrar un mensaje
   public void escribirMensaje(String str){
      for(int i=0;i<this.size();i++) delete(i);
      append(str,null);
   }

   //Este metodo muestra los "friendly names" de los dispositivos remotos
   public void mostrarDispositivos(){
      for(int i=0;i<this.size();i++) delete(i);

      if(SPPClienteMIDlet.dispositivos_encontrados.size()>0){
         for(int i=0;i<SPPClienteMIDlet.dispositivos_encontrados.size();i++){
         try{
                RemoteDevice dispositivoRemoto = (RemoteDevice)
                SPPClienteMIDlet.dispositivos_encontrados.elementAt(i);
                append(dispositivoRemoto.getFriendlyName(false),null);
         }catch(Exception e){
            System.out.println("Se ha producido una excepcion");
         }
         }
      }
      else append("Pulse Busqueda",null);
   }
}
```

IV.5.6 Clase Mensaje

```
package SPPBluetooth;

import javax.microedition.lcdui.*;

public class Mensaje extends TextBox{

   public Mensaje(){
      super("Introducir mensaje","Hola Mundo",50,TextField.ANY);

      addCommand(new Command("OK",Command.SCREEN, 1));
      this.setCommandListener(SPPClienteMIDlet.SPPc);
   }
}
```

V Anexos

V.1 Desarrollo de aplicaciones mediante Sun ONE Studio 5 ME

V.1.1 Introducción

En este anexo, vamos a relatar los pasos básicos necesarios para desarrollar aplicaciones J2ME mediante *Sun ONE Studio 5 ME*. Para una mayor profundidad en el tema, se pueden consultar los tutoriales que se encuentran en access1.sun.com/s1s5me_survey/documentation.

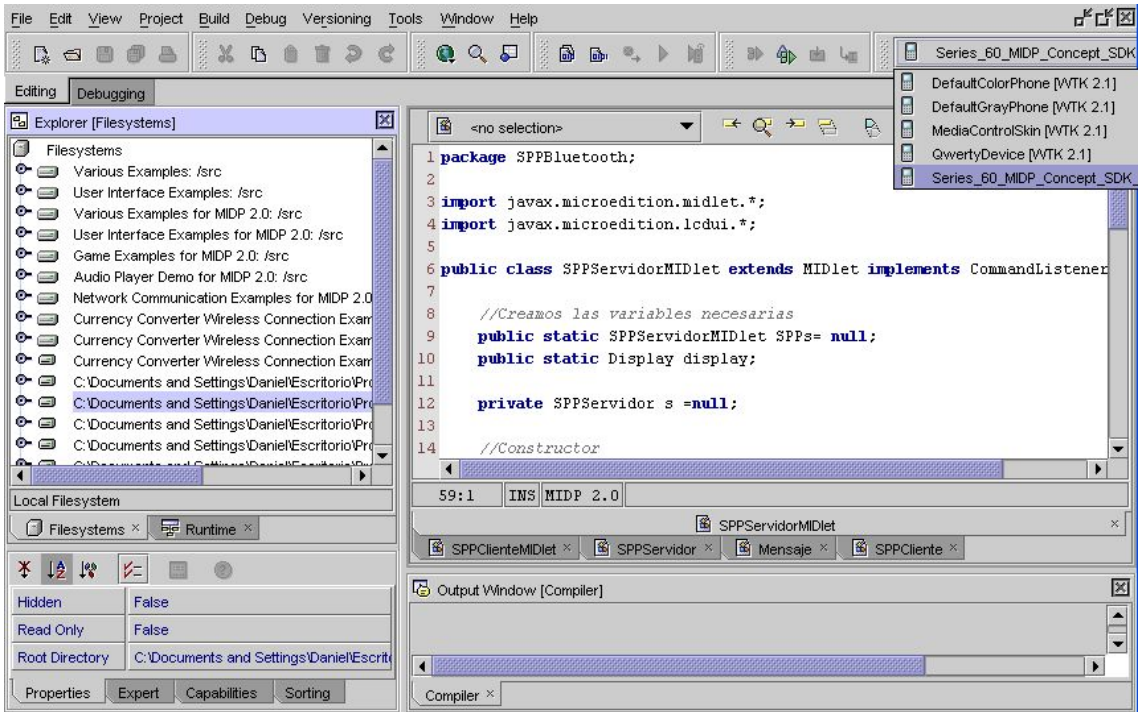
El *Sun ONE Studio 5 ME* es un entorno de desarrollo integrado (IDE), basado en la plataforma de desarrollo NetBeans, que permite el uso de la tecnología J2ME.

Para poder desarrollar aplicaciones J2ME mediante *Sun ONE Studio 5 ME* será necesario tener previamente instalado el J2SE SDK versión 1.4.1_02 o posterior. El *Sun ONE Studio 5 ME* incluye:

- J2ME Wireless Toolkit 2.1
- J2ME Wireless Toolkit 1.0.4_01 (opcional)
- J2ME Wireless Connection Wizard (opcional)
- Dos aplicaciones completas.

El *Sun ONE Studio 5 ME* permite además, integrar otros emuladores y entornos de desarrollo, como por ejemplo el *Nokia Development Kit 2.0* .

Si la instalación ha sido correcta, al arrancar el programa, nos debería aparecer una ventana similar a esta:



Entorno de desarollo de *Sun ONE Studio 5 ME*

V.1.2 Creación de MIDlets y MIDlet Suites

Una aplicación escrita para MIDP es llamada *MIDlet*. Estos son subclases de la clase java.micoedition.midlet.MIDlet. Los *MIDlets* son empaquetados en *MIDlet Suites* , que consisten en dos archivos:

- Java Application Descriptor (.jad)
- Java Archive (.jar)

Antes de poder crear un MIDlet, necesitamos un directorio en el sistema, y montarlo en el IDE. Un sistema de ficheros es comparable a un directorio en el sistema operativo. Cuando se monta un sistema de ficheros, este es incluido en el *Java classpath* , que es necesario para poder compilar, ejecutar y depurar el código.

Los pasos necesarios para montar un sistema de ficheros son:

1. Seleccionamos **File->Mount Filesystem**.
2. Seleccionamos **Local Diectory**. Hacemos click en **Next**.
3. Usamos el *wizard* para navegar hasta el directorio elegido. Seleccionamos este directorio y hacemos click en **Finish** para completar el proceso de montaje.
4. En el directorio montado (que aparecerá en la parte derecha de la ventana), con el botón derecho del ratón elegimos **New->Java Package**.
5. Elegimos un nombre para el paquete, y hacemos click en **Finish**.

Con estos pasos, hemos creado un nuevo paquete en el sistema de ficheros que hemos montado.

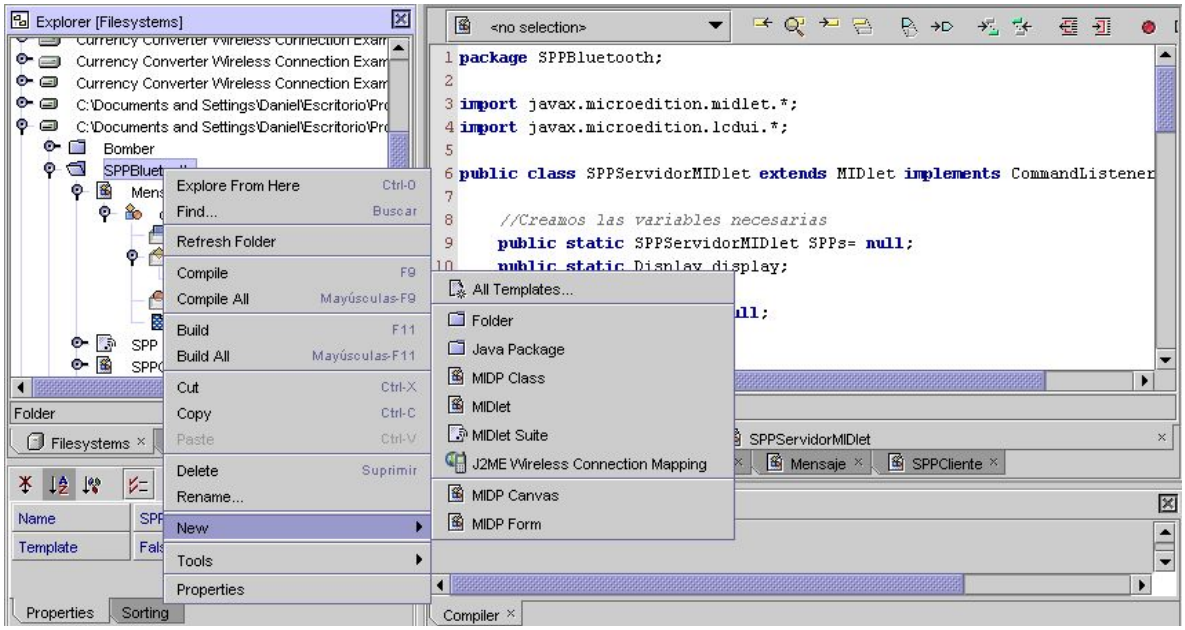
Aunque podríamos trabajar con MIDlets individuales, es mejor para el diseño y testeo, crear MIDlets dentro de una MIDlet Suite. Esta ayuda a empaquetar aplicaciones MIDlet y prepararlas para su descarga.

Los pasos necesarios para crear un MIDlet Suite son:

1. Con el botón derecho del ratón en el paquete creado anteriormente, elegimos **New->MIDLet** Suite del menu.
2. En el *wizard* del MIDlet Suite, escribimos el nombre del MIDlet Suite y hacemos click en **Next**.
3. En la página **Add MIDlet**, hacemos lo siguiente para crear un MIDlet dentro del MIDlet Suite:

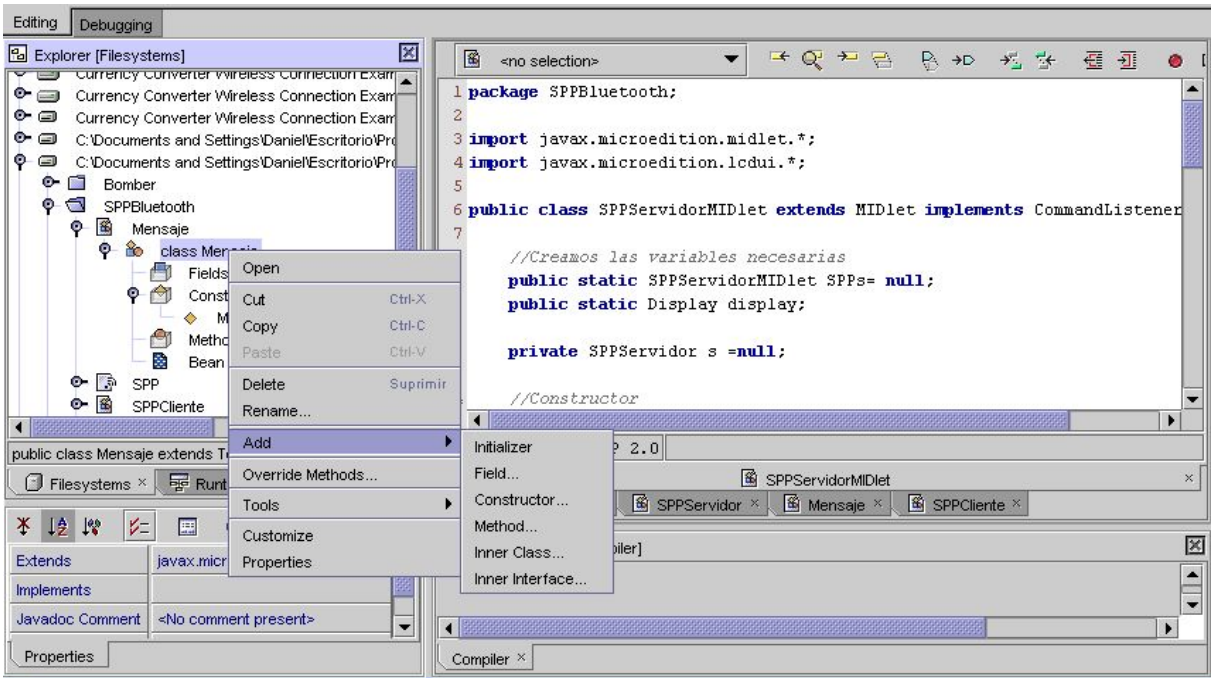
    1. Seleccionamos la opción **Create New MIDlet**.
    2. Introducimos el nombre del paquete y el de la clase.
    3. Si queremos seleccionar una plantilla, la seleccionamos de la lista y luego elegimos MIDlet.
    4. Hacemos click en **Next**.

4. En la página **MIDlet Properties**, elegimos el nombre que nos va a aparecer cambiando el nombre del campo **MIDlet Displayable Name**.
5. En la página **Suite Configuration**, hacemos click en **Finish** y elegimos el MIDP y CLDC que deseamos.



Creación de MIDlets y MIDlet Suites en *Sun ONE Studio 5 ME*

A continuación podemos escribir el código del MIDlet de dos maneas distintas; o bien escribiendo el código directamente en el editor, o bien usando las herramientas para añadir métodos, campos, constructores, clases, interfaces, inicializaciones..

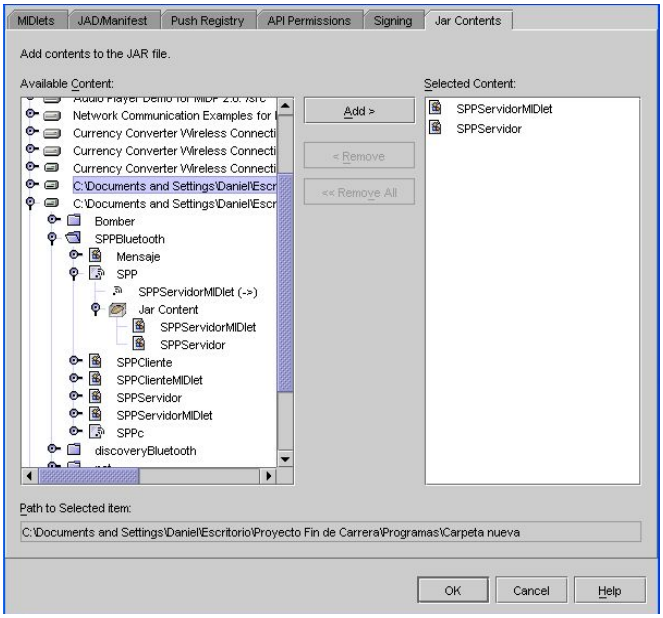


Herramientas para escribir código en *Sun ONE Studio 5 ME*

Finamente, una vez que hayamos escrito el código, nos queda empaquetar nuestra aplicación. Cuando creamos la MIDlet Suite, creamos el paquete esencial de la aplicación. Ahora necesitaremos añadir todos los archivos adicionales de nuestra aplicación.

Para añadir archivos al MIDlet Suite, haremos lo siguiente:

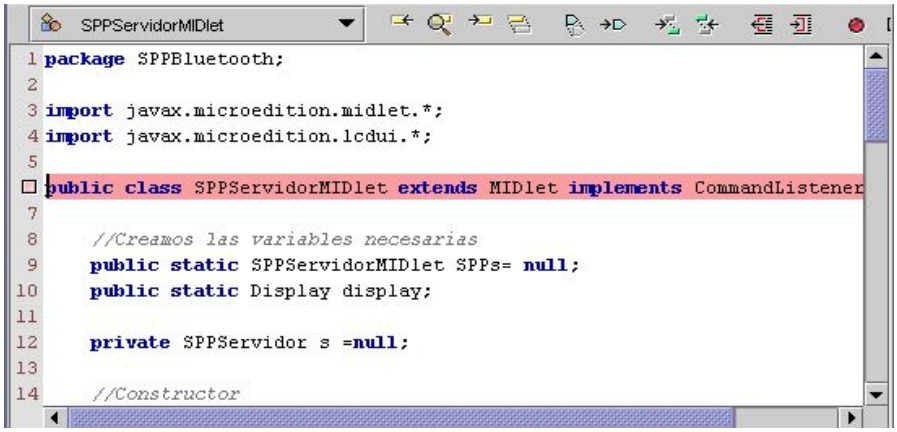
1. Hacemos click con el botón derecho del ratón en el MIDlet Suite y elegimos la opción **Edit Suite.**
2. Seleccionamos la pestaña **Jar Context**.



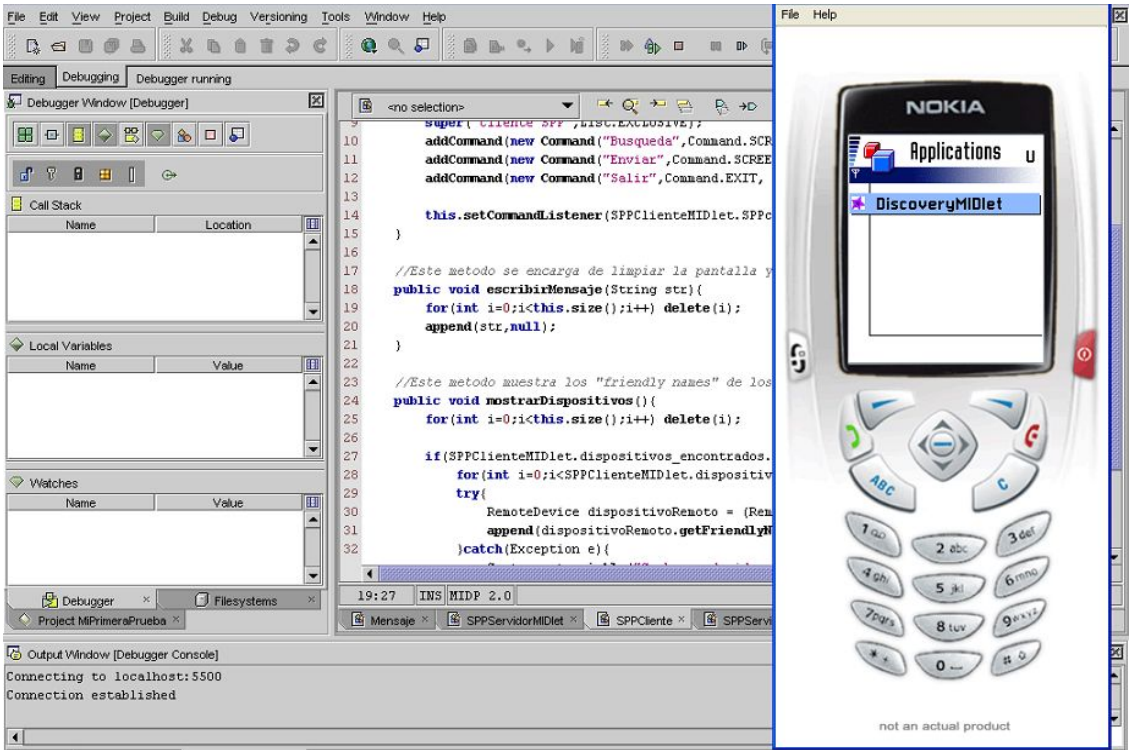
3. Añadimos los ficheros que queremos agregar al MIDlet Suite, seleccionándolos y haciendo click en el botón **Add**.
4. Finamente haremos click en **OK**. Vemos que los ficheros se han añadido al Jar Context.

V.1.3 Depuración de MIDlets y MIDlet Suites

El primer paso para la depuración de un programa es establecer un *breakpoint* en el código donde se quiere examinar la ejecución del programa. Para ello, hacemos click en el número de la línea de codigo en que queremos establecer el *breakpoint*. La línea se volverá de color rosa:



A continuación, lanzaremos la aplicación en modo depuración, o bien haciendo click en **Debug->Start Sesion->Run in Debugger,** o bien seleccionando el icono apropiado del editor de texto (Alt-F5). Hecho esto, se lanzará el entorno de depuración y el emulador del dispositivo seleccionado.



Debugger de *Sun ONE Studio 5 ME* con el emulador *Nokia Series 60 SDK 0.31*

Para depurar el programa, interactuaremos con el emulador, e iremos recogiendo la información necesaria en el entorno de depuración. La información necesaria acerca de las variables y métodos es recogida en las ventanas de la izquierda del depurador.

Cuando se alcance el *breakpoint*, esta línea cambiará del rosa al verde. A partir de entonces el depurador tiene el control de la aplicación, y podremos ejecutar una línea cada vez con **Debug->Step Over**, o *step througth* (saltar) hasta el siguiente *breakpoint* con ***<F8>***.

Si llegados a un *breakpoint* que contiene métodos, queremos consultar el código de dichos métodos, usaremos **Debug->Step Into** o **<F7>** . Para consultar el valor de las variables podemos poner el cursor encima de la variable deseada, y el valor de dicha variable aparecerá al lado del cursor.

Para terminar la sesión de depuración, haremos **Debug->Finish.**

V.2 Método uuidToName

```
public static String uuidToName( UUID u ){

  if ( u.equals( new UUID( 0x0001 ) ))  return "SDP";
  else if ( u.equals( new UUID( 0x0003 ) )) return "RFCOMM";
  else if ( u.equals( new UUID( 0x0008 ) )) return "OBEX";
  else if ( u.equals( new UUID( 0x000C ) )) return "HTTP";
  else if ( u.equals( new UUID( 0x0100 ) )) return "L2CAP";
  else if ( u.equals( new UUID( 0x000F ) )) return "BNEP";
  else if ( u.equals( new UUID( 0x1000 ) )) return "ServiceDiscoveryServerServiceClassID";
  else if ( u.equals( new UUID( 0x1001 ) )) return "BrowseGroupDescriptorCerviceClassID";
  else if ( u.equals( new UUID( 0x1002 ) )) return "PublicBrowseGroup";
  else if ( u.equals( new UUID( 0x1101 ) )) return "SerialPort";
  else if ( u.equals( new UUID( 0x1102 ) )) return "LANAccessUsingPPP";
  else if ( u.equals( new UUID( 0x1103 ) )) return "DialupNetworking";
  else if ( u.equals( new UUID( 0x1104 ) )) return "IrMCSync";
  else if ( u.equals( new UUID( 0x1105 ) )) return "OBEX ObjectPushProfile";
  else if ( u.equals( new UUID( 0x1106 ) )) return "OBEX FileTrasnferProfile";
  else if ( u.equals( new UUID( 0x1107 ) )) return "IrMCSyncCommand";
  else if ( u.equals( new UUID( 0x1108 ) )) return "Headset";
  else if ( u.equals( new UUID( 0x1109 ) )) return "CordlessTelephony";
  else if ( u.equals( new UUID( 0x110A ) )) return "AudioSource";
  else if ( u.equals( new UUID( 0x1111 ) )) return "Fax";
  else if ( u.equals( new UUID( 0x1112 ) )) return "HeadsetAudioGateway";
  else if ( u.equals( new UUID( 0x1115 ) )) return "PersonalAreaNetworkingUser";
  else if ( u.equals( new UUID( 0x1116 ) )) return "NetworkAccessPoint";
  else if ( u.equals( new UUID( 0x1117 ) )) return "GroupNetwork";
  else if ( u.equals( new UUID( 0x111E ) )) return "Handsfree";
  else if ( u.equals( new UUID( 0x111F ) )) return "HandsfreeAudioGateway";
  else if ( u.equals( new UUID( 0x1201 ) )) return "GenericNetworking";
  else if ( u.equals( new UUID( 0x1202 ) )) return "GenericFileTransfer";
  else if ( u.equals( new UUID( 0x1203 ) )) return "GenericAudio";
  else if ( u.equals( new UUID( 0x1204 ) )) return "GenericTelephony";
  else  return u.toString();
 }
```

V.2 Dispositivos compatibles con el JSR-82

A medida que pasa el tiempo y nuevos dispositivos van saliendo al mercado, cada vez son más numerosos los dispositivos que implementan el JSR-82. No es de extrañar, por tanto, que en el momento en que se esté leyendo este anexo, hayan aparecido nuevos dispositivos que no figuren aquí. De modo orientativo, expondremos algunos de los dispositivos que hoy en día soportan dicho estándar[12]:

NOKIA 7610

NOKIA 9500

NOKIA 5140

NOKIA 7700

NOKIA 6230

NOKIA 6600

12 Los dispositivos aquí mencionados corresponden a modelos europeos.

Sony Ericsson

P900

Benq P31

Benq P30

SIEMENS
Mobile

SX1

VI Bibliografía

- **Java APIs for Bluetooth Wireless Technology (JSR 82)**
  Specificarion Version 1.0a
  Motorola. Wireless Software, Applications & Services April 5, 2002

- **The Java APIs for Bluetooth Wireless Technology**
  http://wireless.java.sun.com/midp/articles/bluetooth2/ Abril 2003

- **Bluetooth Technology Overview**
  Version 1.0
  http://www.forum.nokia.com Abril 2003

- **www.benhui.net**
  Esta página está dedicada al desarrollo de aplicaciones J2ME.
  Especialmente recomendable su sección dedicada al desarrollo de aplicaciones Bluetooth.

- **Sun ONE Studio 5, Mobile Edition Tutorial**
  access1.sun.com/s1s5me_survey/documentation Octubre 2003