COMUNICACIÓN ENTRE DISPOSITIVOS BLUETOOTH

Juan Pablo Sevilla Martín y Pablo García Sánchez

Director de proyecto: Juan Julián Merelo Guervós

E.T.S. De Ingenierías Informática y Telecomunicación
Universidad de Granada
Curso 2005/2006

ÍNDICE

Agradecimientos:

Es tradición incluir esta sección en cada proyecto, como culminación de una carrera. Y nosotros no vamos a ser menos, así que lo haremos sucintamente.

Queremos agradecer, por supuesto, a todos los profesores de la E.T.S. de Ingenierías en Informática y Telecomunicación, por ayudarnos a conseguir el "ingenio" necesario para ser "ingeniero".

Especialmente queremos dar las gracias a Juan Julián Merelo Guervós por estar siempre disponible, tanto virtual como físicamente y esperamos seguir en contacto con él para todo lo que se tercie.

Finalmente, y también siguiendo la tradición, gracias a nuestras familias y amigos, por estar siempre al otro lado de la pantalla cuando se les necesita.

De verdad. Gracias.

1. INTRODUCCIÓN

Es un hecho innegable la aceptación de la tecnología móvil por parte de la sociedad. Actualmente los usuarios de telefonía móvil no sólo buscan la posibilidad de efectuar y recibir llamadas, sino que las empresas han descubierto que el añadir nuevas tecnologías a los terminales, como cámaras de fotos, reproductores de mp3 y tecnología Bluetooth interesa cada vez a más gente. Así mismo, el creciente aumento de ventas de ordenadores portátiles, PDAs y el auge de la tecnología inalámbrica nos llevan a la conclusión de que la sociedad quiere romper con el inmovilismo tecnológico y estático y busca maneras de comunicación distintas.

Ejemplos claros de esta tendencia van desde el Campus Virtual Inalámbrico de la Universidad de Granada hasta la tecnología PTT (Push To Talk) aún no implantada, pasando por el auge de redes Wi-Fi en cada vez más hogares o el nacimiento de FON, una iniciativa para compartir estas redes y crear una red inalámbrica universal. La tecnología del futuro, si se nos permite usar esta definición, podría considerarse como la comunicación total entre todo el mundo, en cualquier lugar y en cualquier momento.

Una de las tecnologías que más crecimiento está teniendo es la tecnología Bluetooth, que permite la comunicación sin cables entre dispositivos móviles, aunque en distancias no superiores a 100 metros. Cada vez más esta tecnología se está implantando en los nuevos terminales, y la distancia de comunicación está aumentando, pero sin embargo su uso no está siendo aprovechado del todo. Prácticamente su utilización por parte de los usuarios consiste en el intercambio de archivos con otros dispositivos, sin llegar a

aprovechar una comunicación "interactiva", como tal. La finalidad de este proyecto es crear un sistema que permita la comunicación entre dispositivos Bluetooth, creando un protocolo fácil de usar que permita el envío de mensajes entre varios dispositivos permitiendo una comunicación fluida sin coste alguno.

Las ventajas de esto podrían ser, por ejemplo:

- Uso en docencia: Por ejemplo, para que varios estudiantes participen en una asignatura, envíen respuestas de los ejercicios o votaciones sobre algún tema o ejercicio. Además el profesor podría controlar la asistencia de los alumnos, publicar en bitácoras, hacer búsquedas…
- Comunicación en una empresa: Debido a que el uso de esta tecnología es gratuito, los empleados podrían comunicarse entre sí desde cualquier parte de la empresa sin costes económicos.
- Ocio: Para charlar con compañeros de clase o para hacer nuevas amistades en un bar, así como videojuegos multi-jugador

Estos son sólo unos ejemplos de lo que podría realizarse con este protocolo. Hay que decir que la tecnología Bluetooth está creciendo cada vez más y que prácticamente todos los usuarios de telefonía móvil contarán con ella en sus dispositivos de aquí a muy pocos años.

El nombre de nuestro proyecto, Ulfsark, significa "Piel De Lobo", nombre que recibían cierto tipo de guerreros nórdicos, en un homenaje a la palabra Bluetooh, nombre del rey noruego y danés que unificó las tribus danesas, suecas y noruegas.

Ulfsark pretende ser la base que permita una comunicación fácil entre dispositivos Bluetooth (ordenadores, teléfonos móviles, PDAs…)

que soporten Java. Hemos decidido utilizar este lenguaje debido a la enorme facilidad que conlleva su utilización y que permitirá la creación de un sin fin de aplicaciones basadas en nuestro protocolo. El problema radica en que actualmente muchos dispositivos móviles que soportan Java no incluyen el API JSR-82, necesario para la creación y utilización de aplicaciones Java que requieran el uso de Bluetooth, debido a su reciente creación. Sin embargo, el número de dispositivos que la utilizan crece día a día.

La finalidad de este proyecto es la creación de un protocolo basado en Java, que permita la creación de aplicaciones de comunicación entre dispositivos Bluetooth, desde un chat, hasta un videojuego, pasando por envío de noticias u otros temas interesantes. Como ejemplo crearemos un chat que permita la comunicación entre un servidor y varios clientes que se conecten a él.

2. OBJETIVOS

Nuestra intención es crear una aplicación que utilice Bluetooth como método de comunicación entre dispositivos móviles.

Antes de empezar hemos de aclarar algunos conceptos:

- ¿Qué tipo de aplicación necesitamos?
- ¿Cómo debe funcionar?
- ¿Qué materiales/hardware nos hace falta?

Pensemos en un chat entre dispositivos Bluetooth. Los dispositivos más comunes que usan esta tecnología son los teléfonos móviles, por lo que desarrollaremos esta aplicación adaptada a estos aparatos. Sin embargo, tendríamos que generalizar un chat que también sirva para otros dispositivos, como un PC, por ejemplo.

Aplicación:

Un chat entre dispositivos Bluetooth. Un dispositivo hará de servidor y permitirá que varios clientes se conecten a él. Una vez realizada la conexión, la comunicación se realizará intercambiando mensajes de texto.

Dispositivos:

En primer lugar tenemos que escoger los dispositivos Bluetooth que utilizaremos. Existen infinidad de dispositivos que soportan esta tecnología, desde teléfonos móviles hasta videoconsolas, pasando por teclados y ratones. Sin embargo no todos los dispositivos Bluetooth

son programables, y entre los que lo son, no se programan de la misma manera.

Hemos pensado en programar teléfonos móviles. Son aparatos que la gente siempre lleva encima, y como hemos dicho en la introducción, van a tener un gran auge en el futuro. Muchos de ellos además, pueden programarse de manera fácil. Además permiten crear un chat en cualquier parte. Pero ya que disponemos de ordenadores, también es interesante que se comuniquen con estos.

Lenguaje:

Existen varios lenguajes para programar estos dispositivos y que soporten las opciones Bluetooth.

1) Lenguajes propietarios de cada marca: Es una opción inviable, ya que implicaría programar cada dispositivo con un código distinto, amén de que no todos ellos se ofrecen al público.

2) C++: Es un lenguaje muy extendido y que soportan gran cantidad de móviles. Sin embargo casi todos son de la marca Nokia y pertenecientes a la serie de teléfonos inteligentes (SmartPhones) que tienen Symbian como sistema operativo. Además, al investigar en foros y sitios web llegamos a la conclusión de que no es tarea fácil programar dispositivos Bluetooth con este lenguaje.

3) Java: Es soportado por más dispositivos que C++ (incluyendo además los propios dispositivos Symbian), así como otros dispositivos que no son teléfonos móviles (ordenadores, PDAs…) y su programación resulta más sencilla. Y una de las ventajas de Java es que no necesita ser compilado para cada plataforma.

A la vista queda que la opción más sensata es la de escoger Java como lenguaje de programación. Con ello podemos crear una base que funcione en multitud de dispositivos sin tener que hacer apenas cambios, y utilizando el paradigma de la Programación Orientada a Objetos, la cual simplifica el método de la construcción de un proyecto de tal envergadura. Además, este es un tipo de proyecto típico de programación por capas, con lo que aprovecharemos más si cabe este paradigma.

También permite la emulación del dispositivo en un ordenador, siendo mucho más fácil realizar las pruebas.

3. TECNOLOGÍAS UTILIZADAS

3.1 BLUETOOTH

Bluetooth es la norma que define un estándar global de comunicación inalámbrica que posibilita la transmisión de voz y datos entre diferentes equipos mediante un enlace por radiofrecuencia. Los principales objetivos que se pretende conseguir con esta norma son:

- Facilitar las comunicaciones entre equipos móviles y fijos.
- Eliminar cables y conectores entre éstos.
- Ofrecer la posibilidad de crear pequeñas redes inalámbricas y facilitar la sincronización de datos entre nuestros equipos personales.

Tecnología

La especificación de Bluetooth define un canal de comunicación de máximo 720Kb/s con rango óptimo de 10 metros (opcionalmente 100m).

La frecuencia de radio con la que trabaja está en el rango de 2.4 a 2.48Ghz con amplio espectro y saltos de frecuencia con posibilidad de transmitir en Full Duplex con un máximo de 1600 saltos/seg. Los saltos de frecuencia se dan entre un total de 79 frecuencias con intervalos de 1Mhz; esto permite dar seguridad y robustez.

El protocolo de banda base (canales simples por línea) combina switching de circuitos y paquetes. Para asegurar que los paquetes no lleguen fuera de orden, los slots pueden ser reservados por paquetes síncronos, un salto diferente de señal es usado para cada paquete.

Por otro lado, el switching de circuitos puede ser asíncrono o síncrono. Tres canales de datos síncronos (voz), o un canal de datos síncrono y uno asíncrono, pueden ser soportados en un solo canal. Cada canal de voz puede soportar una tasa de transferencia de 64 Kb/s en cada sentido, la cual es suficientemente adecuada para la transmisión de voz. Un canal asíncrono puede transmitir como mucho 721 Kb/s en una dirección y 56 Kb/s en la dirección opuesta, sin embargo, para una conexión asíncrona es posible soportar 432,6 Kb/s en ambas direcciones si el enlace es simétrico.

La CPU del dispositivo se encarga de atender las instrucciones relacionadas con Bluetooth del dispositivo anfitrión, para así simplificar su operación. Para ello, sobre la CPU se ejecuta un software denominado Link Manager que tiene la función de comunicarse con otros dispositivos por medio del protocolo LMP.

Entre las tareas realizadas por el LC y el Link Manager, destacan las siguientes:

- Envío y Recepción de Datos.
- Empaginamiento y Peticiones.
- Determinación de Conexiones.
- Autenticación.
- Negociación y determinación de tipos de enlace.
- Determinación del tipo de cuerpo de cada paquete1. - Ubicación del dispositivo en modo sniff o hold.

ARQUITECTURA DE BLUETOOTH

Introducción a la pila de protocolos de Bluetooth:

La pila de protocolos de Bluetooth puede ser dividida en dos componentes: el Host Bluetooth y el Controlador Bluetooth (o módulo de radio Bluetooth). La Interfaz Controlador Host (Host Controller Interface: HCI) provee una interfaz estandarizada para estos dos elementos. Veamos el diagrama de bloques de la pila de protocolos Bluetooth. La pila de protocolos se compone de protocolos que son específicos a la tecnología inalámbrica Bluetooth, como L2CAP y SDP y otros protocolos adoptados como OBEX. La pila puede dividirse en cuantro capas de acuardo a su propósito como vemos en la siguiente tabla:

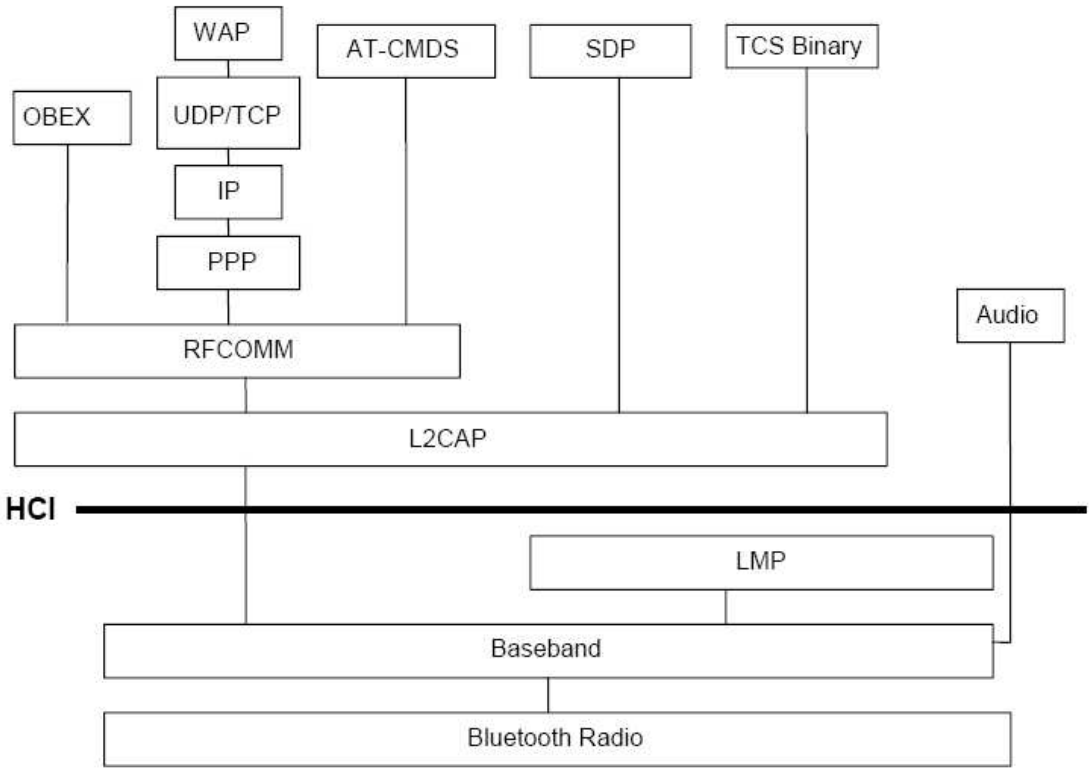
Grupo de Protocolos	**Protocolos en la pila**
Protocolos del núcleo Bluetooth	Banda Base, Protocolo de Control de Enlace, L2CAP y SDP
Protocolo de Reemplazo de Cable	RFCOMM
Protocolo de Control de Telefonía	TCS Binario
Protocolos Adoptados	PPP, UDP/TCP/IP, OBEX, WAP

La capa de banda base crea el enlace por radiofrecuencia físico entre las unidades Bluetooth haciendo una conexión. El Protocolo de Control de Enlace (Link Manager Protocol: LMP) es responsable de la inicialización del enlace entre dispositivos Bluetooth y administra aspectos de seguridad como la encriptación y la autentificación. L2CAP adapta protocolos de la capa superior a la banda base. Multiplexa entre varias conexiones lógica y conexiones creadas por las capas superiores. Los datos de Audio se enlutan directamente desde y para la bande basa y no atraviesan el L2CAP. El SDP se usa para consultar información del dispositivo, servicios o características de servicios. RFCOMM emula las señales de control y datos del RS-232 sobre la banda base de Bluetooth, ofreciendo capacidades de transporte para servicios superiores que usan un mecanismo de transporte mediante interfaz serie. TCS Binario define la señalización de llamada de control para el establecimiento de llamadas de voz y datos entre dispositivos Bluetooth.

Además de los protocolos, el SIG Bluetooth ha definido los perfiles Bluetooth. Un perfil Bluetooth define maneras estándar de

usar protocolos y sus características para un uso en particular. Un dispositivo Bluetooth puede soportar uno o más perfiles.

Los cuatro perfiles "genéricos" son los Perfiles de Acceso Genérico (Generis Access Profile, GAP), el Perfil Puerto Serie (Serial Port Profile, SPP), Perfil de Aplicación de descubrimento de Servicios (Service Discovery Application profile, SDAP), y el Perfil de Intercambio de Objetos Genéricos (Generic Object Exchange Profile, GOEP).



Pila de protocolos Bluetooth

CAPAS DE LA PILA DE PROTOCOLOS

- Host Controller Interface (HCI)

Es una capa de software que intercambia todos los datos entre un host (por ejemplo, un PC) y un controlador (un dispositivo Bluetooth USB). Datos y voz pasan a través del HCI.

- Logical Link Control An Adaptation Protocol (L2CAP)

Es la capa núcleo de la pila por la que todos los datos deben pasar. Incluye segmentación de paquetes y reordenación de datos, así como multiplexación de protocolos. Permite por lo tanto aceptar datos de SDP y RFCOMM, al estar en una capa superior. Sólo transporta datos, no voz.

- Service Discovery Protocol (SDP)

Se utiliza para descubrir servicios. Lo comentaremos más adelante.

- RFCOMM

Puerto de serie inalámbrico o protocolo de reemplazo de cable. Se comentará más adelante.

- Telephony Control Protocol Specification (TCS, TCS Binary, TCS-BIN)

Se usa para enviar señales de control a dispositivos que quieren utilizar las capacidades de audio dentro de Bluetooth.

- Wireless Access Protocol (WAP)

Es un protocolo adoptado en Bluetooth para cubrir sus necesidades. Requiere que PPP, IP y UDP estén presentes en la pila.

- Object Exchange (OBEX)

Es un protocolo de comunicación inicialmente definido por la Asociación de Datos Infrarrojos (IdDA) Al igual que WAP, OBEX

fue definido por otro grupo, pero fue adoptado por el SIG Bluetooth. Es útil para transferir objetos como archivos entre dispositivos Bluetooth. OBEX no requiere que TCP y IP estén presentes en la pula, pero el fabricante es libre de implementar OBEX sobre TCP/IP.

- Bluetooth Networ Encapsulation Protocol (BNEP)

Es una capa en la pila Bluetooth que permite a otros protocolos de red ser transmitidos sobre Bluetooth, por ejemplo Ethernet. Encapsula paquetes TCP/IP en paquetes L2CAP antes de transmitirlos por la capa L2CAP.

- Human Interface Device Protocol (HID)

Es otro protocolo adoptado y proveniente de la especificación SUB. Lista las reglas y guías para transmitir información desde y hacia interfaces humanas como letrados, razones, o mandos de consolas.

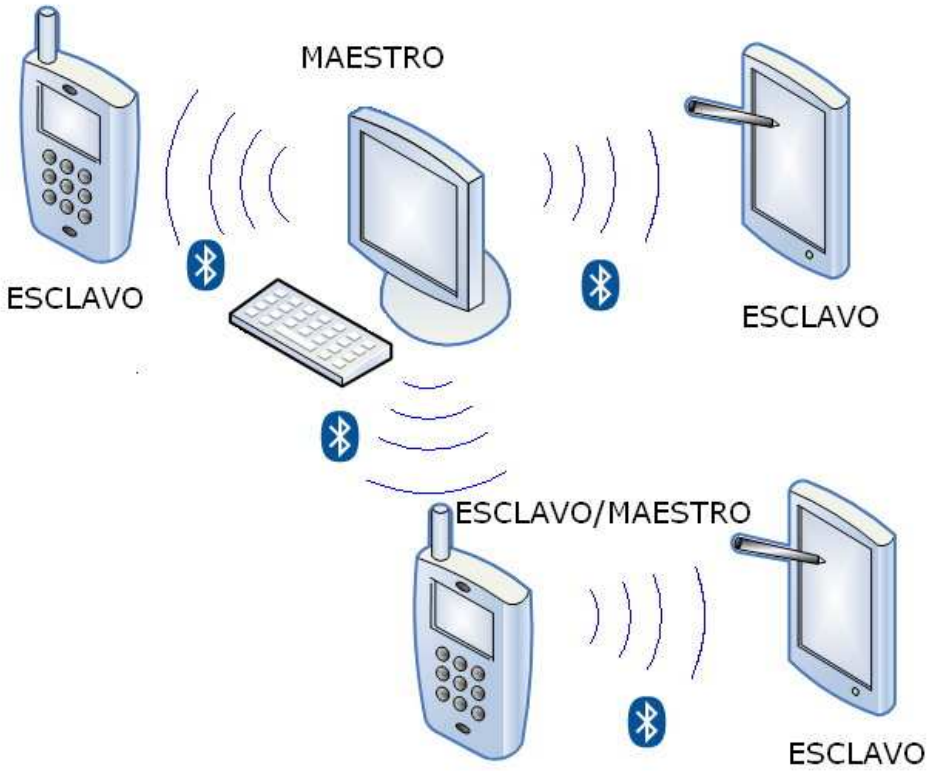
AREAS DE TRABAJO PERSONALES (PERSONAL AREA NETWORK, PAN)

Cuando dos o más dispositivos Bluetooth dentro de un rango establecen una conexión, se forma una area de trabajo personal. Estas pueden ser *piconets* o *scatternets*.

Una piconet tiene un solo maestro y hasta siete esclavos. El maestro es el dispositivo que inicia la conexión. El dispositivo que se acepta la conexión automáticamente se convierte en esclavo. Los roles maestro/esclavo no están predefinidos.

Red Piconet

¿Qué ocurre si un octavo dispositivo quiere unirse a la piconet? El maestro no invita a nuevos miembros a unirse hasta que al menos uno de los miembros antiguos la abandona o entra en estado inactivo. Sin embargo, si uno de los esclavos soporta multipunto entonces el nuevo dispositivo puede conectarse a ese esclavo creando una *scatternet.*



Red Scatternet

3.2 Java

En la actualidad no es realista ver Java como un simple lenguaje de programación, si no como un conjunto de tecnologías que abarca a todos los ámbitos de la computación con dos elementos en común:

• El código fuente en lenguaje Java es compilado a código intermedio interpretado por una Java Virtual Machine (JVM), por lo que el código ya compilado es independiente de la plataforma.

• Todas las tecnologías comparten un conjunto más o menos amplio de APIs básicas del lenguaje, agrupadas principalmente en los paquetes java.lang y java.io.

Java 2 Platform, Enterprise Edition (J2EE)

Esta versión está orientada al entorno empresarial. El software empresarial tiene unas características propias marcadas: está pensado no para ser ejecutado en un equipo, sino para ejecutarse sobre una red de ordenadores de manera distribuida y remota mediante EJBs (Enterprise Java Beans).

Java 2 Platform, Standard Edition (J2SE)

Esta edición de Java es la que en cierta forma recoge la iniciativa original del lenguaje Java. Tiene las siguientes características:

- Inspirado inicialmente en C++, pero con componentes de alto nivel, como soporte nativo de strings y recolector de basura.
- Código independiente de la plataforma, precompilado a bytecodes intermedio y ejecutado en el cliente por una JVM (Java Virtual Machine).

- Modelo de seguridad tipo *sandbox* proporcionado por la JVM.
- Abstracción del sistema operativo subyacente mediante un juego completo de APIs de programación.

Esta versión de Java contiene el conjunto básico de herramientas usadas para desarrollar Java Applets, así cómo las APIs orientadas a la programación de aplicaciones de usuario final: Interfaz gráfica de usuario, multimedia, redes de comunicación, etc.

Java ME

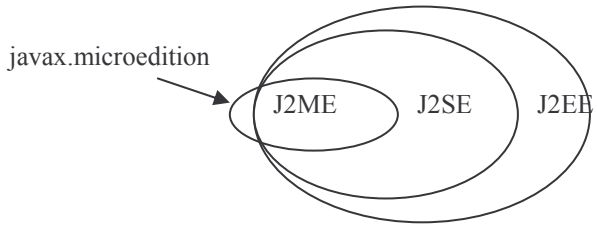
Java Platform, Micro Edition o Java ME (anteriormente conocido como Java 2 Platform, Micro Edition o J2ME), es una colección de APIs para el desarrollo de software para dispositivos con pocos recursos, como PDAs, teléfonos móviles y similares. Java ME es formalmente una especificación, aunque el término se usa a menudo para referirse a las implementaciones de esta especificación en tiempo de ejecución. Fue desarrollado bajo el Java Community Process como JSR 68.

Esta edición tiene unos componentes básicos que la diferencian de las otras versiones, como el uso de una máquina virtual denominada KVM (Kilo Virtual Machine, debido a que requiere sólo unos pocos Kilobytes de memoria para funcionar) en vez del uso de la JVM clásica, inclusión de un pequeño y rápido recolector de basura y otras diferencias.

J2ME contiene una mínima parte de las APIs de Java. Esto es debido a que la edición estándar de APIs de Java ocupa 20 Mb, y los dispositivos pequeños disponen de una cantidad de memoria mucho

más reducida. En concreto, J2ME usa 37 clases de la plataforma J2SE provenientes de los paquetes java.lang, java.io, java.util. Esta parte de la API que se mantiene fija forma parte de lo que se denomina "configuración".

Como vemos, J2ME representa una versión simplificada de J2SE. Sun separó estas dos versiones ya que J2ME estaba pensada para dispositivos con limitaciones de proceso y capacidad gráfica. También separó J2SE de J2EE porque este último exigía unas características muy pesadas o especializadas de E/S, trabajo en red, etc. Por tanto, separó ambos productos por razones de eficiencia. Hoy, J2EE es un superconjunto de J2SE pues contiene toda la funcionalidad de éste y más características, así como J2ME es un subconjunto de J2SE (excepto por el paquete javax.microedition) ya que, como se ha mencionado, contiene varias limitaciones con respecto a J2SE.



Existe mucha documentación relativa a J2ME en Internet, así que simplemente haremos mención a las partes más importantes que utilizaremos en nuestro proyecto, saltándonos partes que hemos considerado ajenas a nuestro trabajo.

Para empezar tenemos que hablar de los perfiles. Un perfil es un conjunto de APIs orientado a un ámbito de aplicación determinado. Los perfiles identifican un grupo de dispositivos por la funcionalidad que proporcionan (electrodomésticos, teléfonos

móviles, etc.) y el tipo de aplicaciones que se ejecutarán en ellos. Las librerías de la interfaz gráfica son un componente muy importante en la definición de un perfil. Aquí nos podemos encontrar grandes diferencias entre interfaces, desde el menú textual de los teléfonos móviles hasta los táctiles de los PDAs.

Configuraciones

Las configuraciones se componen de una máquina virtual y un conjunto mínimo de bibliotecas de función. Proporcionan la funcionalidad básica para un conjunto de dispositivos que comparten características similares, tales como gestión de memoria o conectividad a la red.

En la actualidad existen dos configuraciones J2ME:

Connected Limited Device Configuration (CLDC)

Connected Device Configuration (CDC)

Configuración CLDC

Esta configuración está diseñada para dispositivos con conexiones de red intermitentes, procesadores lentos y memoria limitada como son teléfonos móviles, asistentes personales (PDAs), etc. Está orientado a dispositivos que cumplan las siguientes características:

Procesador:16 bit/16 MHz o más

Memoria: 160-512 KB de memoria total disponible para la plataforma Java

Alimentación: Alimentación limitada, a menudo basada en batería

Trabajo en red: Conectividad a algún tipo de red, con ancho de banda limitado habitualmente

La especificación CLDC se ha desarrollado dentro del Java Community Process[sm] (JCP[sm]) junto con 500 partners que

representan a las industrias de fabricantes de dispositivos wireless, proveedores de servicios y terminales de venta.

Sun proporciona la implementación de referencia de CLDC (CLDC Reference implementation, CLDC RI) que incluye la máquina virtual K (K Virtual Machine, KVM). Sun también proporciona la implementación del CLDC HotSpotTM, disponible para usos comerciales bajo licencia.

Esta máquina virtual está orientada a la nueva generación de dispositivos con una cantidad de memoria disponible mayor. La CLDC RI es adecuada para dispositivos que cumplan las siguientes características:

Procesador:32 bits

Memoria: 512 KB/1 MB de memoria total disponible para la plataforma Java

Alimentación: Alimentación limitada, a menudo basada en batería

Trabajo en red: Conectividad a algún tipo de red, con ancho de banda limitado habitualmente

La máquina virtual K toma la K de Kilobyte, haciendo referencia al poco tamaño que ocupa la plataforma, un mínimo de 70 KB

Existen tres versiones de CLDC:

CLDC 1.1 (JSR 139): CLDC 1.1 es una revisión de la especificación CLDC 1.0 e incluye nuevas características como son punto flotante o soporte a referencias débil, junto con otras mejoras. CLDC 1.1 es compatible con versiones anteriores y sigue soportando dispositivos pequeños o con recursos limitados.

Existen implementaciones de referencia.

CLDC 1.0 (JSR 30)

CLDC HotSpot ImplementationTM: Es una máquina virtual muy optimizada que presenta una diferencia de rendimiento muy alta frente a la KVM. Incluye características que soportan una ejecución

más rápida de aplicaciones y una gestión de recursos más eficientes, manteniendo los requisitos en cuanto a plataforma de ejecución

Configuración CDC

Esta configuración se ha desarrollado para dispositivos con 2 MB o más de memoria disponible para la plataforma, incluyendo RAM y memoria flash o ROM. Estos dispositivos requieren todas las características y funcionalidades de la JVM e incluyen teléfonos móviles de última generación, asistentes personales (PDAs), terminales de punto de venta (TPVs), etc y permiten tanto redes con conexión intermitente o de alta velocidad, sin cablear (wireless) o cableadas. La CDC incorpora el perfil Foundation (Foundation Profile), aunque están disponibles los perfiles Personal y Personal Basis para aplicación que requieren una interfaz gráfica de usuario compleja o Web. La plataforma CDC para J2ME incorpora:

- Una máquina virtual Java (Java Virtual Machine, JVM) completa compatible con J2SE 1.3.1
- Las bibliotecas de clase y APIs mínimas para que el sistema funcione
- Soporte completo para carga de clases, gestión de subprocesos (threads) y mecanismos de seguridad

PERFILES

Los perfiles proporcionan al diseñador de los productos flexibilidad para soportar distintos tipos de dispositivos móviles personales con entornos de aplicaciones Java compatibles. Dado que las necesidades de las familias de dispositivos son muy diferentes, dentro de J2ME se han definido distintos perfiles para proporcionar opciones en función de las necesidades existentes:

Foundation Profile: Este perfil define una serie de APIs sobre la CDC orientadas a dispositivos que carecen de interfaz gráfica como, por ejemplo, decodificadores de televisión digital.

Personal Profile: El *Personal Profile* es un subconjunto de la plataforma J2SE v1.3, y proporciona un entorno con un completo soporte gráfico AWT.

El objetivo es el de dotar a la configuración CDC de una interfaz gráfica completa, con capacidades web y soporte de *applets* Java.

RMI Profile: Este perfil requiere una implementación del *Foundation*

Profile se construye encima de él. El perfil RMI soporta un subconjunto de las APIs J2SE v1.3 RMI. Algunas características de estas APIs se han eliminado del perfil RMI debido a las limitaciones de cómputo y memoria de los dispositivos.

PDA Profile: El PDA Profile está construido sobre CLDC. Pretende abarcar PDAs de gama baja, tipo Palm, con una pantalla y algún tipo de puntero (ratón o lápiz) y una resolución de al menos 20000 pixels (al menos 200x100 pixels) con un factor 2:1. No es posible dar mucha más información porque en este momento este perfil se encuentra en fase de definición.

Mobile Information Device Profile (MIDP): Este perfil está construido sobre la configuración CLDC. Al igual que CLDC fue la primera configuración definida para J2ME, MIDP fue el primer perfil definido para esta plataforma. Este perfil está orientado para dispositivos con las siguientes características:

- Reducida capacidad computacional y de memoria.
- Conectividad limitada (en torno a 9600 bps).
- Capacidad gráfica muy reducida

- Entrada de datos alfanumérica reducida.
- 128 Kb de memoria no volátil para componentes MIDP.
- 8 Kb de memoria no volátil para datos persistentes de aplicaciones.
- 32 Kb de memoria volátil en tiempo de ejecución para la pila Java.

Los tipos de dispositivos que se adaptan a estas características son: teléfonos móviles, buscapersonas (pagers) o PDAs de gama baja con conectividad.

El perfil MIDP establece las capacidades del dispositivo, por lo tanto, especifica las APIs relacionadas con:

- La aplicación (semántica y control de la aplicación MIDP).
- Interfaz de usuario.
- Almacenamiento persistente.
- Trabajo en red.
- Temporizadores.

Las aplicaciones que realizamos utilizando MIDP reciben el nombre de *MIDlets* (por simpatía con *APPlets*). Decimos así que un MIDlet es una aplicación Java realizada con el perfil MIDP sobre la configuración CLDC.

3.3 JSR-82

INTRODUCCIÓN

Debido a que la especificación de Bluetooth cubre muchas capas y perfiles, la API JSR-82 se centra sobre todo en las siguientes áreas:

1) Sólo transimisión de datos, no de voz.
2) Los siguientes protocolos:
   a. L2CAP (sólo orientado a conexión)

b. RFCOMM

c. SDP

d. OBEX (Protocolo de Intercambio de Objetos)

3) Los siguientes perfiles:

a. Perfil de Acceso Genérico (Generic Access Profile: GAP)

b. Perfil De Puerto Serie (Serial Port Profile: SPP)

c. Perfil Genérico de Intercambio de Objetos (Generic Object Exchange Profile: GOEP)

La API intenta servir las siguientes características:

1. Registrar servicios
2. Descubrir dispositivos y servicios
3. Establecer conexiones RFCOMM, L2CAP y OBEX.
4. Conducir esas actividades en una forma segura.

3.3.1 PARTE A: DESCUBRIMIENTO

Debido a que los dispositivos Bluetooth son móviles, necesitan una manera de encontrar otros dispositivos y la manera de aprender que pueden hacer esos dispositivos. La API que vamos a utilizar incluye la forma de descubrir dispositivos, encontrar servicios y anunciar servicios a otros dispositivos.

A) DESCUBRIMIENTO DE DISPOSITIVOS.

Una aplicación puede obtener una lista de dispositivos usando la función startInquiry() (no bloqueante) o retrieveDevices() (bloqueante). La primera requiere que la aplicación especifique un listener; este listener es avisado cuando se encuentran nuevos dispositivos de una investigación. Si una aplicación no tiene porqué esperar a que la investigación empiece, la API incluye el método retrieveDevices(), que devuelve la lista de dispositivos que han sido encontrados en una investigación anterior o dispositivos que se han clasificado como pre-conocidos. Los dispositivos pre-conocidos son aquellos dispositivos que están definidos en el Centro de Control Bluetooth como dispositivos con los que el dispositivo local contacta frecuentemente. Así este método no realiza una investigación, pero proporciona una manera rápida de obtener una lista rápida de dispositivos que podrían estar en el área. Una vez que el dispositivo se descubre se suele iniciar una búsqueda de servicios, como veremos más adelante.

CLASES DE DESCUBRIMIENTO DE DISPOSITIVOS

Interfaz javax.bluetooth.DiscoveryListener

Esta interfaz permite a una aplicación especificar un listener de eventos que responderá a eventos relacionados con la investigación. Esta interfaz también se usa para la búsqueda de servicios. El método deviceDiscovered() se llama cada vez que un dispositivo se encuentra durante una investigación. Cuando la investigación se termina o cancela, se llama al método inquiryCompleted(). Este método recibe como argumento INQUIRY_COMPLETED, INQUIRY_ERROR o INQUIRY_TERMINATED.

Clase javax.bluetooth.DiscoveryAgent

Esta clase proporciona método para descubrimiento de servicios y dispositivos. Para los segundos está el método startInquiry() para iniciar el dispositivo local en modo investigación y el método retrieveDevices() devuelve información sobre los dispositivos encontrados que fueron encontrados a partir de la investigación previa realizada por el dispositivo local. También proporciona una manera de cancelar una investigación a través del método cancelInquiry().

B) DESCUBRIMIENTO DE SERVICIOS

CLASES DE DESCUBRIMIENTO DE SERVICIOS

Las siguientes secciones proporcionan un breve resumen de las clases involucradas en el descubrimiento de servicios.

Clase javax.bluetooth.UUID

Esta clase encapsula enteros sin signo de 16, 32 o 128 bits de largo. Esta clase se utiliza para representar un identificador único y universal utilizado como valor para un atributo de un servicio. Sólo los atributos representados por UUIDs pueden ser buscados en el SDP Bluetooth.

Clase javax.bluetooth.DataElement

Esta clase contienen varios tipos de dato que un atributo de servicio puede tomar.

Pueden ser:

- Enteros con o sin signo que tienen uno, dos , cuatro, ocho o dieciséis bytes de longitud.
- String
- bolean
- UUID
- secuencias de cualquiera de alguno de esos tipos escalares.

La clase también presenta una interfaz para construir y recuperar el valor de un atributo de servicio.

Interfaz javax.bluetooth.ServiceRecord

Esta interfaz define un par ID,valor. *ID* es un entero sin signo de 16 bits y *valor* es un DataElement. Un DataElement, es un un valor auto-descriptivo de uno de los tipos comentados anteriormente. Además, para proporcionar el dispositivo servidor Bluetooth del cual un ServiceRecord ha sido obtenido, esta interfaz tiene el método populateRecord() para recuperar atributos de servicio deseados.

Clase javax.bluetooth.DiscoveryAgent

Proporciona metodos para el descubrimiento de servicios y dispositivios. Soporta descubrimiento de servicios en modo no-bloqueante y proporciona una manera de cancelar una transacción de búsqueda de servicios en progreso.

Interfaz javax.bluetooth.DiscoveryListener

Esta interfaz permite a una aplicación especificar un EVENT LISTENER que responda a eventos de descubrimiento de servicios y dispositivos. El método servicesDiscovered() se llama WHENEVER los

servicios se descubren. El método serviceSearchCompleted() es llamado cuando la búsqueda de servicios ha terminado o se ha cancelado.

C) REGISTRO DE SERVICIOS

Hemos visto que una aplicación Bluetooth tiene las siguientes responsabilidades:

1) Crear un registro de servicio que describe el servicio ofrecido por la aplicación.
2) Añadir un registro de servicio que describe el servicio ofrecido por la aplicación.
3) Registrar las medidas de seguridad de Bluetooth asociadas con un servicio que debería ser reforzada para las conexiones con los clientes.
4) Aceptar conexiones desde clientes que soliciten el servicio ofrecido por la aplicación.
5) Actualizar el registro de servicio en la SDDB del servidor si las características del servidor cambian.
6) Eliminar o deshabilitar el registro de servicio en la SDDB del servidor cuando el servicio ya no está disponible.

Al conjunto de los elementos 1,2,5 y 6 que comprende las responsabilidades que el servidor tiene que cumplir para anunciar un servicio a los clientes se denomina registro de servicio.

RESPONSABILIDADES DEL REGISTRO DE SERVICIOS

Lo anterior hace referencia al registro de servicios desde la perspectiva general de Bluetooth. En lo relativo a la API de Java que nosotros usaremos el registro de servicios es un esfuerzo colaborativo entre la aplicación servidora, la implementación de la API y la pila Bluetooth.

La figura siguiente muestra como colaboran estos componentes.

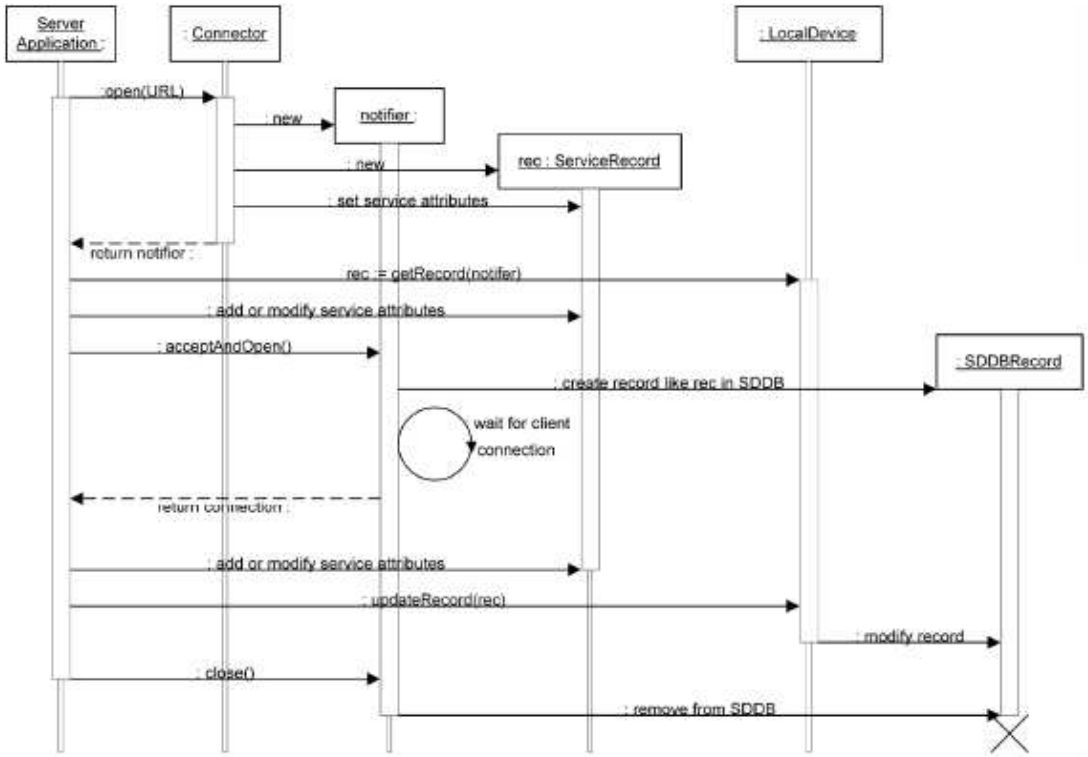


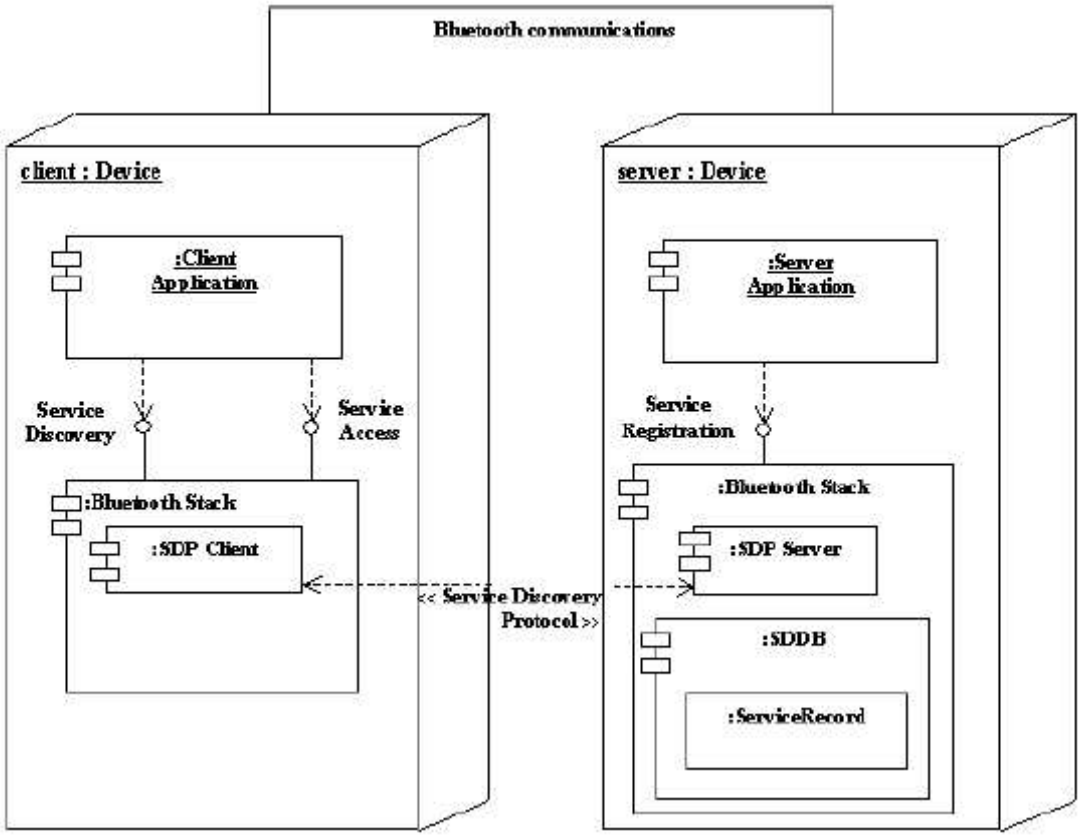
Diagrama de colaboración para el registro de servicios

Podemos ver en esta imagen que cuando la aplicación del servidor llama a *Conector.open()* con un string de conexión URL como argumento para un servidor, la implementación cre un nuevo *ServiceRecord*. Un registro de servicio se añade a la SDDB por la implementación cuando la aplicación del servidor llama *acceptAndOpen()*. La aplicación servidora puede acceder a su *ServiceRecord* llamando a *getRecord(),* y puede hacer modificaciones a ese *ServiceRecord*. Estas modificaciones también se realizan a los correspondientes registros de servicio en la SDDB cuando la aplicación servidora llama a *updateRecord().* Finalmente, el registro de servicio de la aplicación se borra de la SDDB cuando la aplicación servidora envía un mensaje *close* a el *notifier* para el servicio.

CLASES DEL REGISTRO DE SERVICIOS

Interfaz javax.bluetooth.ServiceRecord

Un registro de servicio describe un servicio Bluetooth a los clientes. Los registros de servicio están compuestos por un conjuntos de atributos de servicio, cada uno de los cuales es un par consistente un ID de atributo y un valor de atributo.



Un servidor proporciona un Registro de Servicio que permite a los clientes conectarse

Un servidor SDP de la pila Bluetooth mantiene una "base de datos" de registros de servicio que describe los servicios del disposivo servidor. Un servicio ejecutar-antes-de-conectar añade su ServiceRecord a la SDDB llamando a acceptAndOpen(). Los clientes

utilizan SDP para consultar al servidor SDP para cualquier servcio de su interés. Un ServiceRecord proporciona suficiente información para permitir a un cliente SDP conectar al servicio Bluetooth en el dispositivo servidor.

La aplicación servidora también puede usar el método setDeviceServiceClasses() de la clase ServiceRecord para activar algunos de los bits del dispositivo para reflejar el nuevo servicio ofrecido.

Clase javax.bluetooth.LocalDevice

Esta clase proporciona el método getRecord() que una aplicación servidora puedo usar para obtner su ServiceRecord. El servidor puede modificar el objeto ServiceREcord añadiendo o modificando atributos. El servicio actualizado puede ser colocado en la SDDB realizando notifier.acceptAndOpen() o usando el método updateRecord() de LocalDevice.

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

Una ServiceRegistrationException se lanza cuando un intento de añadir o modificar un registro de servicio en la SDDB falla.

Los fallos de registro de servicio pueden ocurrir:

- Durante la ejecución de Connector.open(), cuando la implementación crea un nuevo registro de servicio para el servicio especificado por Connector.open().
- Cuando un servicio ejecutar-antes-de-conectar invoca el método acceptAndOpen() y la implementación intenta añadir el registro de servicio asociado con el notifier a la SDDB.
- Después de la creación inicial del registro de servicio, cuando la aplicación servidora intenta modificar el registro de servicio en la SDDB usando el método updateRecord().

3.3.2 PARTE B: ADMINISTRACIÓN DE DISPOSITIVOS

Esta parte describe las posibilidades de cambiar la manera en la que el dispositivo local responde al dispositivo remoto. Veremos las clases que representan los objetos Bluetooth esenciales (como LocalDevice y RemoteDevice), los métodos para acceder a las propiedades de esos objetos, como sus nombres y sus direcciones Bluetooth y los métodos para administrar los estados de LocalDevice, así como hacer el dispositivo visible.

Los dispositivos Wireless son pontencialmente más vulnerables a escuchas y al spoofing (falsificación del origen de los mensajes) que los dispositivos cableados. La tecnología Bluetooth incluye varias técnicas para evitar estas vulnerabilidades. Algunas de estas técnicas, como el salto de frecuencias, se aplican universalmente a todas las comunicaciones Bluetooth. Otras características, como encriptación y autentificación pueden activarse o desactivarse en función de las necesidades de la aplicación.

A) GENERIC ACCESS PROFILE

Las clases LocalDevice y RemoteDevice representan los objetos esenciales de Bluetooth. Estas clases proporcionan características para administración de dispositivos que son parte del Perfil de Acceso Genérico (Generis Access Profile, GAP). Los métodos de control estándar para el dispositivo local están en la clase LocalDevice. Las clases DeviceClass y BluetoothStateException proporcionan soporte para la clase LocalDevice. DeviceClass tienen métodos para recuperar los valores para las clases de servicio superiores y las clases inferiores y superiores que describen las propiedades de un dispositivo. Para terminar, la clase RemoteDevice representa un dispositivo remoto y proporciona métodos para recuperar información sobre ese dispositivo remoto.

CLASES GAP

Clase javax.bluetooth.LocalDevice

Esta clase proporciona acceso y control sobre el dispositivo Bluetooth local. Está diseñada para cumplir completamente los requisitos del GAP como está definido en la especificación Bluetooth.

Clase javax.bluetooth.RemoteDevice

Esta clase representa un dispositivo Bluetooth remoto y proporciona información básica sobre un dispositivo remoto, incluyendo la dirección Bluetooth del dispositivo y su nombre amigable (el nombre Bluetooth del dispositivo).

Clase javax.bluetoot.BluetoothStateException extends java.io.IOException

Esta excepción se lanza cuando un dispositivo no puede servir una solicitud que normalmente soporta debido al estado de la radio. Por ejemplo, algunos dispositivos no permiten INQUIRY cuando el dispositivo está conectado a otro dispositivo.

Clase javax.bluetooth.DeviceClass

Esta clase define valores para el tipo de dispositivo y los tipos de servicio en un dispositivo.

B) SEGURIDAD

Este capítulo describe los métodos disponibles a las aplicaciones para solicirtar comunicaciones Bluetooth seguras. Las aplicaciones cliente y servidor opcionalmente pueden añadir parámetros a la cadena de conexión que se pasa como argumento a a Connector.open() para especificar la seguridad requerida para

conexiones. Además es posible para diferentes conexiones que traten entre diferentes servicios para tener diferentes niveles de seguridad. Los parámetros en la cadena de conexión pueden ser usados para establecer medidas de seguridad al mismo tiempo que se establece la conexión. Los métodos de la clase RemoteDevice puede ser usados en cualquier momento por las aplicaciones cliente y servidor para solicitar un cambio en la seguridad para una conexión particular.

SOLICITUDES DE SEGURIDAD EN LA CADENA DE CONEXIÓN

Las aplicaciones servidoras pueden usar uno de los métodos open de la clase javax.microedition.io.Connector del CLDC para crear un objeto notificador que pueda ser usado para esperar a un cliente que conecte. Para un servidor, los componentes de la cadena de conexión del argumento del método open proporcionan suficiente información para crear un objeto de la clase apropiada de notificador, y para crear el registro de servicio apropiado. Sin embargo, parámetros opcionales pueden añadirse al string (cadena) de conexión para especificar los requisitos del servidor para conexiones con clientes. Estos parámetros son para autentificación, encriptación y intercambio maestro/esclavo.

SOLICITUD DE SERVICIO PARA AUTENTICACIÓN

Autenticación Bluetooth significa la verificación de la identidad del dispositivo remoto. La autentificación consiste en un esquema de reto y respuesta entre dispositivos que requiere una llave de 128 bits generada a partir del código PIN compartido por ambos dispositivos. Si los códigos PIN entre ambos dispositivos no concuerdan, el proceso de autenticación falla.

El parámetro *authenticate* tiene la siguiente interpretación cuando se usa en la cadena de conexión de una aplicación servidora.

- Si *authenticate=true*, la implementación intenta verificar la identidad de cada dispositivo cliente
- Si *autenthicate=false*, la implementación no intenta verificar la identidad de los dispositivos clientes que intentan conectar al servicio.

Si el parámetro *authenticate* no está presente en la cadena de conexión, entonces la implementación no intente verificar la identidad de los clientes a menos que otros parámetros presentes en la cadena requieran este chequeo de identidad.

No todos los sistemas Bluetooth soportan autenticación. Incluso si la autenticación se soporta es posible para *authenticate=true* entrar en conflicto con las opciones de seguridad que el usuario ha establecido usando el BCC. Una *BluetoothConnectionException* se lanza en el método *Connector.open()* si *authenticate=true* y la autenticación no se soporta, o si esta autenticación entra en conflicto con las opciones de seguridad del dispositivo. Si hay un conflicto entre las necesidades de seguridad de una aplicación y las opciones de seguridad del dispositivo, algunas implementaciones del BCC podrían intentar eliminar el conflicto solicitando al usuario cambiar las opciones del dispositivo.

SOLICITUDES DE SERVIDOR PARA ENCRIPTACIÓN

La encriptación puede ser aplicada a las comunicaciones sobre un enlace de datos entre dos dispositivos Bluetooth. Cuando está activada, la encriptación se aplica a toda la transferencia de datos en ambas direcciones sobre este enlace. El parámetro *encrypt* tiene la siguiente interpretación cunado se usa en la cadena de conexión de una aplicación servidora:

- Si *encrypt=true* la implementación encripta toda comunicación desde y para este servicio

- Si *encrypt=false*, no se requiere encriptación por la aplicación servidora, pero puede usarse si el dispositivo cliente u otras conexiones existentes en el enlace de datos entre estos dos dispositivos requieren encriptación.
- Si el parámetro *encrypt* no está presente en la cadena de conexión, es equivalente a *encrypt=false*.

Debido a que la encriptación Bluetooth requiere una llave de enlace compartida la encriptación requiere autenticación. Esto quiere decir que la combinación *authenticate=false* y *encrypt=true* no es válida y generará una *BluetoothConnecionException*.

Al igual que con la autenticación, no todos los sistemas Bluetooth soportan encriptación. Incluso si se soporta, el caso *encrypt=true* puede entrar en conflicto con las opciones de seguridad establecidas a través del BCC. Una *BluetoothConnectionException* se lanza en el método *Connector.open()* si *encrypt=true* y la encriptación no se soporta o entra en conflicto con las opciones de seguridad del BCC.

SOLICITUDES DE SERVIDOR PARA AUTORIZACIÓN

La autorización es un procedimiento por el cual el usuario de un dispositivo servidor concede acceso a un servicio específico de un dispositivo cliente específico. La implementación de la autorización puede involucrar que se pregunte al usuario del dispositivo servidor si el dispositivo cliente debería permitirse acceder al servicio. También puede involucrar la consulta de una lista de dispositivos que son "confiables" y por tanto se les permite el acceso a todos los servicios.

El parámetro *authorize* tiene la siguiente interpretación cuando se usa en la cadena de conexión de una aplicación servidora:

- Si *authorize=true*, la implementación consulta con el BCC para determinar si el dispositivo cliente que solicita una conexión debería permitirse el acceso a este servicio.
- Si *authorize=false*, se le permite a todos los clientes acceder a este servicio.
- Si no existe el parámetro *authorize* se asume que es *authorize=false*.

Al igual la encriptación, la autorización implica que la identidad del cliente pueda ser verificada a través de la autenticación, por lo que la combinación *authenticate=false* y *authorize=true* no es válida y generará una *BluetoothConnecionException*.

Y al igual que la autenticación y la encriptación, no todos los sistemas Bluetooth soportan la autorización, comportándose igual que estas características como hemos comentado anterioremente y lanzando las mismas excepciones.

Solicitudes de servidor para papel maestro (Server Requests for Master Role)

Los dispositivos Bluetooth forman redes localizadas. Cada red Bluetooth tiene un dispositivo maestro cuyo reloj y frecuencia se usan para sincronizar hasta 7 dispositivos esclavos. Un dispositivo puede ser tanto maestro como esclavo. El dispositivo que inicia la formación de un enlace de datos a otro dispositivo generalmente se convierte en maestro de la red entre esos dos dispositivos. Sin embargo, la tecnología Bluetooth proporciona un procedimiento para un dispositivo esclavo solicitar un intercambio maestro/esclavo usando el parámetro *master* en la cadena de conexión.

Mediante *master=true* hacemos que la implementación solicite que el cliente y el servidor intercambien sus papeles para que el

servidor se convierta en el maestro de la red. Con *master=false* no se fuerza nada.

No todos los dispositivos soportan esta funcionalidad, así que se lanza una excepción *BluetoothConnectionException* en el método *connector.open().*

Solicitudes en la cadena de conexión para el cliente

Las aplicaciones cliente también pueden usar los parámetros *authenticate, encrypt* y *master* en la cadena de conexión de argumento para *Connector.open().* Cuando lo usan los clientes estos parámetros tienen las siguientes interpretaciones:

- Cuando *authenticate=true* la implementación intente verificar la identidad del dispositivo servidor.
- Cuando *encrypt=true* la implementación encripta todas las comunicaciones desde y hacia este servicio. Al igual que con el servidor, *encrypt=true* implica *authenticate=true*.
- Cuando *master=true* el cliente debe asumir el papel de maestro en la comunicación con el servidor, así que la implementación debe rechazar intentes del servidor de iniciar un intercambio de papeles.

Con esta API, el único dispositivo que necesita conceder permiso para usar un servicio es el dispositivo que ofrece ese servicio. Con lo cual el parámetro *authorize* no se permite en las conexiones del cliente y se lanza la excepción anteriormente comentada si aparece el argumento *authorize* en la cadena de conexión. Cuando un cliente intenta conectarse al servicio ofrecido por un servidor, ambos dispositivos tienen sus propias opciones para los parámetros de la cadena de conexión. Estas opciones indican los requisitos que cada servicio tiene para esa conexión. Casi todas las posibles combinaciones de estos parámetros pueden dar lugar a una conexión exitosa. El único problema radica que ambos tengan *master=true*, con lo que el intento de conexión falla. En el lado del cliente se lanza

la excepción *BluetoothConnectionException* en el método *connection.open(),* pero el servidor simplemente rechaza la conexión. La aplicación servidora continúa esperando en una llamada bloqueante en *acceptAndOpen*() hasta que haya una conexión segura.

La seguridad de Bluetooth puede ser solicitada usando el CLDC *javax.microedion.io.Connector* y *javax.bluetooth.RemoteDevice*.

3.3.3 PARTE C: COMUNICACIÓN

Para usar un servicio en un dispositivo Bluetooth remoto, el dispositivo local debe comunicarse usando el mismo protocolo que el servidor remoto. Como las aplicaciones pueden acceder a una amplia gama de servicios Bluetooth, se proporcionan APIs para permitir conexiones que tienen los protoclos RFCOMM, L2CAP y OBEX.

PROTOCOLO DE PUERTO SERIE

El protocolo RFCOMM proporciona emulación de múltiples puertos serie RS-232 entre dos dispositivos Bluetooth. Las direcciones Bluetooth de los dos puntos finales identifican una sesión RFCOMM. Sólo una sesión RFCOMM puede existir entre cualquier par de dispositivos a la vez, pero una sesión puede tener más de una conexión. El número de conexiones que pueden establecerse a la vez en un dispositivo Bluetooth depende de éste. Un dispositivo puede tener más de una sesión RFCOMM, tantas como cada sesión esté enlazada aun dispositivo diferente. Esta característica se soporta en la API jsr-82, pero según la especificación es opcional, por lo que algunas pilas Bluetooth no la soportan.

Una aplicación que ofrece un servicio basado en el Perfil de Puerto Serie (Serial Port Profile, SPP) es un servidor SPP. Una aplicación que inicia una solicitud de conexión a un servidor SPP es un cliente SPP. Un registro de servidor SPP es un servicio en la SDDB. Como parte del proceso de registro de servicio, un identificador de canal de servicio se añade al registro de servicio (service record). Un cliente localiza el servicio usando la API de descubrimiento de servicio. Puede conectar al servicio especificando la dirección del servidor y el identificador del canal de servicio. Después de establecer la conexión los datos pueden ser transmitidos en ambas direcciones

entre el cliente y servidor. La negociación de los parámetros de conexión y control se manejan automáticamente por la implementación SPP de la conexión.

URLs para servidor y cliente

Las URLs de conexión para el servidor y el cliente son de la siguiente forma:

Para el **servidor**:
btspp://localhost:UUID[;PARAMETROS DEL SERVIDOR]

UUID es una secuencia de 1 a 32 números en hexadecimal

PARAMETROS DEL SERVIDOR pueden ser:
master
name
encrypt
autenthicate
authorize

Todos ellos siendo =true o =false, como ya hemos visto, salvo name, que es de la forma name=Texto

Para el **cliente** la cadena de conexión es de la forma

btspp://direccion:canal[;PARAMETROS DEL CLIENTE]

dirección es una cadena de 12 dígitos en hexadecimal y canal un número del 1 al 30.

En cuanto a *PARAMETROS DEL CLIENTE* tenemos master, encrypt y autenthicate, usándose de la misma forma que con el servidor.

REGISTRO DE SERVICIO DE PUERTO SERIE

Un servidor SPP crea un objeto de tipo StreamConnectionNotifier de la siguiente forma:

- Usando la cadena apropiada para un servidor SPP como argumento de Connector.open(); y
- Realizando un casting del resultado obtenido de Connector.open() a la interfaz StreamConnection Notifier.

Ejemplo:

StreamConnectionNotifier interface.StreamConnectionNotifier service =

(StreamConnectionNotifier) Connector.open(

"btspp://localhost:10203040506070809OA1B1C1D1D1E100;name=SPPEx");

StreamConnection con =

(StreamConnection) service.acceptAndOpen();

El servidor usa el método *acceptAndOpen*() para indicar que está listo para aceptar una conexión de un cliente. Este método bloquea hasta que un cliente conecta.

El servicio SPP puede aceptar múltiples conexiones de diferentes clientes llamando a *acceptAndOpen*() varias veces. Un nuevo objeto *StreamConnection* se crea para cada conexión aceptada. Cada cliente accede al mismo registro de servicio y conecta al servicio usando el

mismo canal RFCOMM del servidor. Si el dispositivo no acepta múltiples conexiones, entonces se lanza una *BluetoothStateException* desde *acceptAndOpen*().

ESTABLECIMIENTO DE CONEXIÓN EN EL CLIENTE

Antes de que un cliente SPP pueda establecer una conexión con un servicio SPP, debe descubrir el servicio. Una conexión cliente incluye la dirección Bluetooh del dispositivo del servidor y el identificador del canal del servidor para ese servicio. El método getConnectionURL() en la interfaz ServiceRecor se usa para obtener esta URL.

Invocando al método Connector.open() con una conexión SPP cliente se obtiene un objeto StreamConnectio, que representa la conexión SPP desde el lado del cliente.

Veamos como un cliente establece una conexión con un servicio identificado por el identificador del canal del servidor=5 en un dispositivo con dirección '0050C000321B'

```
StreamConnection con =
(StreamConnection)
Connector.open("btspp://0050C000321B:5");
```

4. REALIZACIÓN DEL PROYECTO

Ulfsark constituye un framework para el desarrollo de aplicaciones que utilicen tecnología bluetooth en las comunicaciones. El principal objetivo es abstraer detalles del API especificado en el documento jsr-82, y ofrecer al programador las siguientes características:

1. Interfaz sencilla: el desarrollo de un programa simple debe ser casi instantáneo.
2. Envío/recepción de datos asíncronos
3. Flujo de datos delimitado en paquetes
4. Programación orientada a eventos
5. Modelo cliente-servidor

4.1 Motivación

4.1.1 Simplificación del API

Para aquellos familiarizados con la tecnología Java™, hablar de la simplificación de un API puede resultar absurdo, ya que se trata de un lenguaje bastante amigable, no sólo por la sintaxis del propio lenguaje sino también por los hábitos de los programadores.

No obstante, aunque no pretendemos reducir el coste de desarrollo a grandes empresas del mundo de las telecomunicaciones, sí puede resultar útil para gente no experimentada. Así pues, la utilidad de esta simplificación debe ser considerada desde un punto de vista docente, no profesional.

En cualquier caso, esto implicará una cierta reducción de la funcionalidad: un mayor grado de abstracción implica un menor grado de control. Se pretende que la librería Ulfsark haga innecesario el uso de clases del paquete javax.bluetooth.

Si el tamaño y la sencillez del código de una aplicación que utilice tecnología Bluetooth se decrementa drásticamente sin perder excesiva funcionalidad, el objetivo se considerará conseguido.

4.1.2 Envío/recepción de datos asíncrono

En una aplicación que utilice el API de bluetooth (jsr-82), el intercambio de datos se realiza de forma continua[1] mediante operaciones *read* y *write*. Una vez establecida la conexión, se obtiene un objeto de tipo L2CAPConnection o StreamConnection.

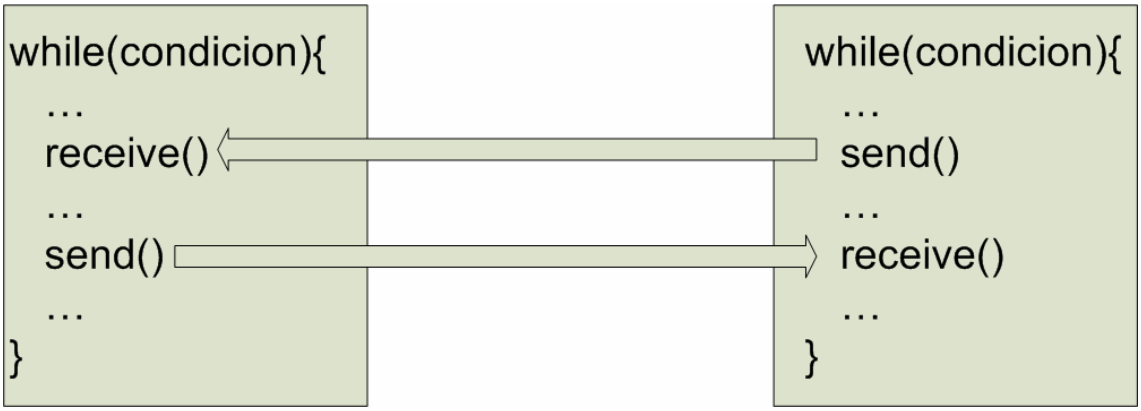
En el primer caso, el objeto ofrece dos funciones para transferencia de datos:

- Send(byte[] b) envia una serie de bytes.
- Receive(byte[] b) lee una serie de bytes.

En el segundo caso, lo que ofrece son métodos para abrir flujos de lectura y escritura:

- openInputStream() y openOutputStream() devuelven, respectivamente, flujos de lectura y escritura de bytes para cualquier propósito.
- openDataInputStream() y openDataOutputStream() devuelven, respectivamente, flujos de lectura y escritura de bytes, pero orientados al envío/recepción de texto.

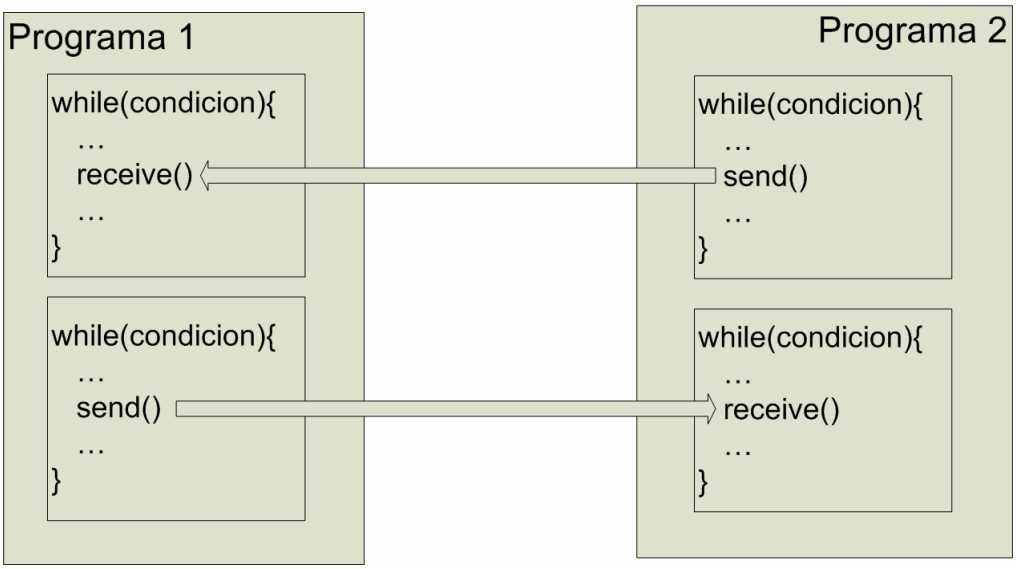
Por tanto, una vez establecida la conexión, los programas deben permanecer en un bucle de lectura/escritura continuo. El objeto StreamConnection permite realizar las operaciones de lectura y escritura a través de métodos de distintos objetos. No obstante, en ambos casos, la operaciones de lectura implican un bloqueo para el programa, que no sabe en qué momento recibirá datos a través de la conexión, y desconoce la cantidad a recibir. Además, según este esquema, los programas deben comunicarse de modo síncrono, reduciendo la flexibilidad.



Visto esto, las facilidades que pretende ofrecer Ulfsark son las siguientes:

- Envío/recepción de datos asíncrono
- Flujo de datos delimitado en paquetes
- Programación orientada a eventos

Una primera mejora al caso explicado con anterioridad, es la introducción de hebras para paralelizar las operaciones de lectura y escritura. De este modo, cada vez que se disponga de un dato listo para ser enviado, no será necesario esperar a tener el turno de envío.



La idea subyacente en Ulfsark es automatizar este proceso, de modo que la lectura se inicie, en una hebra paralela, al crear la conexión.

4.1.3 Flujo de datos delimitado en paquetes

Si la comunicación entre los dos dispositivos está basada en bytes (cada byte es un paquete de datos que contiene información suficiente), sería una tarea simple implementar las acciones a realizar al recibir cada byte. Lo común es que no sea así: los programas se comunican mediante paquetes de tamaño mayor que 1 y normalmente variable. Por tanto, habría que implementar el envío y recepción de datos en paquetes siguiendo un patrón determinado que sirva de "estandar" en el contexto de un software determinado.

Otra de las misiones asignadas a Ulfsark es la creación de un patrón genérico para los paquetes de bytes, que sirva como base para la comunicación en distintos contextos.

4.1.4 Programación orientada a eventos

El objetivo es eliminar la necesidad de programar bucles de lectura/escritura. Tal y como se describió anteriormente, una vez lanzada la aplicación, la lectura debe realizarse de forma automática en una hebra paralela. Si a esto añadimos el método de delimitación de paquetes, el programa podría simplificarse del siguiente modo:

1. Se implementa un método asociado a la recepción de un paquete: *paqueteRecibido()*
2. Al crear la conexión, se inicia automáticamente una hebra que se encarga de la lectura del flujo de entrada. La hebra principal puede encargarse de otras tareas mientras tanto.
3. Cada vez que reciba un paquete completo, la hebra de lectura ejecuta el método *paqueteRecibido.*

El paradigma de programación pasaría a ser orientado a eventos o, como también suele denominarse a estos casos, un híbrido entre programación declarativa e imperativa, ya que en lugar de programarse el camino o ruta principal del programa, se programan los métodos asociados a diferentes eventos que puedan surgir, siendo el código interno de estos métodos totalmente imperativo (secuencia de instrucciones).

4.1.5 Modelo cliente-servidor

Al utilizar Ulfsark como base para la construcción de una aplicación tenemos que

Para la creación de una aplicación siguiente el modelo cliente-servidor, se dispondrán de las tres entidades siguientes:

- Servidor: esta entidad puede crear un servicio Bluetooth™ con identificador y atributos arbitrarios. Una vez creado el servicio, la aplicación recibe nuevos clientes que deseen conectarse, y permite el intercambio de información.

- Cliente: esta entidad puede conectarse a un servidor Ulfsark e intercambiar información

- Buscador de servicios: busca servicios Bluetooth con identificador arbitrario. Es un complemento para el cliente.

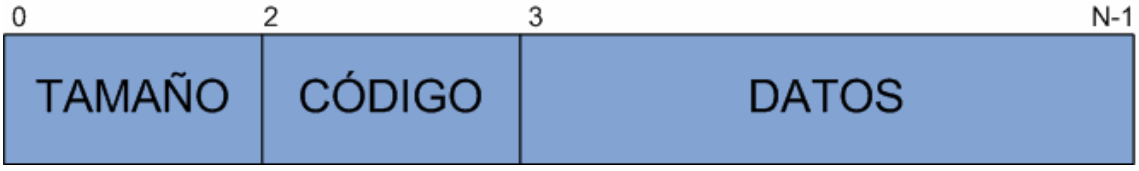
4.2 DISEÑO

4.2.1 FORMATO DE LOS PAQUETES

A la hora de definir un formato para los paquetes utilizados en la transmisión de datos, tenemos en cuenta varios factores:

- El tamaño debe ser variable.
- El tamaño máximo debe ser considerablemente grande, para posibilitar el envío de mensajes de tamaño muy variado en un solo paquete.
- La cabecera debe contener alguna información adicional para facilitar la creación de protocolos basados en este tipo de paquete.

Siguiendo estos principios, definimos el protocolo **UMP** (Ulfsark Messaging Protocol) como un protocolo básico de comunicaciones en el que sólo se establece la forma de segmentar el flujo de datos en paquetes siguiendo este formato:



- Los dos primeros bytes indican el tamaño de la sección de datos.
- El tercer byte es un código que representa información acerca del tipo de paquete.
- El resto de bytes constituyen los datos.

Hay que destacar las siguientes características:

1. **Rango de tamaño**: 3-65538 bytes (con cabecera incluida)

2. **Delimitación basada en tamaño de paquete**

Los dos primeros bytes indican al receptor el número de bytes que debe extraer del flujo de entrada para obtener un paquete completo.

Este método de delimitación tiene una ventaja obvia: es simple e intuitivo. Sólo requiere la inserción de 2 bytes en cada paquete. La desventaja podría ser suficientemente perniciosa como para desecharlo: un error en la comunicación (se pierden datos por el camino, o llegan datos incorrectos) tendría consecuencias catastróficas. Los paquetes recibidas serían incoherentes desde ese momento, pues los bytes interpretados como cabecera no serían los enviados con tal propósito, y por consiguiente se extraerían tramas de tamaño aleatorio. Lo más grave sería que, una vez perdida la secuencia de paquetes, sólo podría recuperarse por una casualidad. Además, si no se usa un sistema de control de errores, el programa podría no saber que está recibiendo paquetes erróneos.

Sabiendo esto ¿Qué razón hay para utilizar este método de delimitación de tramas? El protocolo L2CAP ofrece un canal de transmisión libre de errores. Por tanto, el único error posible en la transmisión será la pérdida de la conexión, y será fácilmente detectable ya que las operaciones read() de Java devuelven -1 en caso de error.

3. Clasificación de los paquetes

El segundo componente de la cabecera es un byte denominado *código*. Mediante este código, se pueden distinguir distintos tipos de paquetes, estableciendo cada programa su propio protocolo basado en unos tipos concretos de paquetes. Por ejemplo, en el caso de un chat, podrían usarse estos:

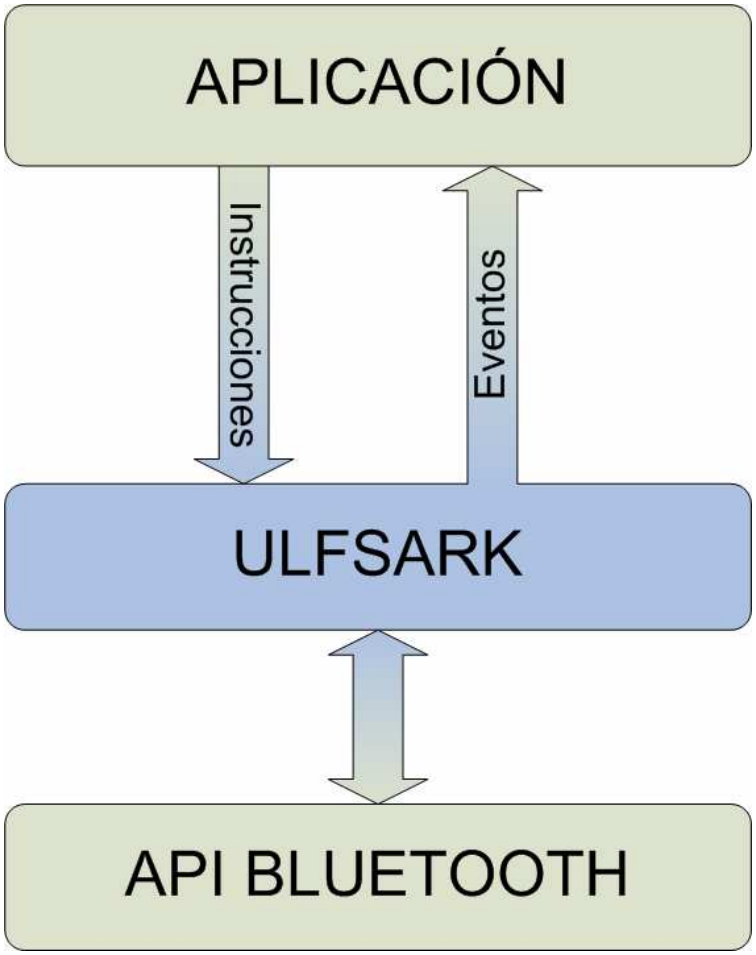
- Mensaje
- Mensaje privado
- Cambio de nick
- etc.

La lectura de datos podría realizarse de la siguiente forma:

1. Leer paquete del flujo de entrada
2. Determinar el tipo de paquete (código)
3. Extraer datos del paquete. Éste podría tener una estructura interna determinada. Por ejemplo, volviendo al caso del chat, en un mensaje privado debe indicarse el receptor del mensaje y el contenido.
4. Realizar acciones necesarias, en función del tipo de paquete

4.2.2 ARQUITECTURA

Tal y como se indicó en los objetivos, Ulfsark pretende constituir una capa intermedia entre el API de Bluetooth y una aplicación cualquiera. También se indicó que uno de los objetivos era automatizar la lectura en paralelo de los paquetes recibidos, y la generación de eventos. El siguiente diagrama esquematiza el papel de Ulfsark en una aplicación que utilice Bluetooth.



En este apartado, definiremos los servicios que ofrecerán las distintas entidades que componen la librería (servidor, cliente y buscador) así como los eventos que generan.

Hay que tener en cuenta un aspecto muy importante: la librería está orientada a la plataforma J2ME. Esto no significa que no pueda ser utilizada en otras (J2SE, principalmente), sino que en el desarrollo de ésta debemos considerar ciertas características singulares de la programación para dispositivos móviles que reducirán nuestro campo de movimiento.

Una de las características de la tecnología Java es el entorno seguro en el que se ejecutan los programas, permitiendo la restricción/concesión de privilegios a las aplicaciones que se ejecutan en la máquina virtual. De este modo, un programa no puede acceder a la red o a un sistema de archivos sin permiso expreso del usuario, aunque esto se puede automatizar con ficheros de permisos.

Como es de esperar, este principio también se respeta en la plataforma J2ME. Sin embargo no debemos contar con la posibilidad de automatizar la concesión/restricción de los servicios, ya que desconocemos el sistema operativo sobre el que se ejecuta la la máquina virtual. La solución que se adoptó es la siguiente: dotar al usuario (u obligar, visto de otro modo) de la capacidad de permitir el acceso a los servicios por parte de una aplicación, en el momento en que ésta los solicita. Una vez concedido el permiso, no se volverá a consultar durante la misma ejecución.

Supongamos un MIDlet, que debe ser ejecutado en un dispositivo móvil. Al iniciar la aplicación, se crea una hebra que se encarga de los eventos relacionados con la interfaz. Cuando se introduce un comando desde el teclado del teléfono (por ejemplo, seleccionando una entrada de un menú con una tecla OK), la hebra citada ejecutará

un procedimiento asociado a ese evento. Si dentro de este procedimiento se solicita el acceso a un recurso externo (sistema de archivos, red, etc.), se mostrará una pantalla consultando al usuario si desea permitir a la aplicación el acceso a dicho recurso.

Aquí surge un problema:

- La hebra se bloquea esperando la contestación del usuario
- La respuesta recibida no se procesará, pues la hebra de la interfaz está bloqueda

Esto produce interbloqueo, con lo que la interfaz de la aplicación quedará inservible. Sólo hay una solución: los métodos que requieren permiso del usuario, deben ser ejecutados en otra hebra. Esto no es condición suficiente: la hebra de la interfaz debe continuar su ejecución independientemente, para poder atender los nuevos eventos. Por tanto, esto introduce cierto asincronismo en la ejecución de estas funciones, al no poder iniciarlas y esperar a que terminen. Es importante recalcar que este problema sólo incumbe al código ejecutado en la hebra de la interfaz gráfica. No obstante, muchos métodos fundamentales se iniciarán por orden del usuario desde la interfaz.

Así pues, hemos de preparar la librería de modo que la ejecución de los procedimientos no pueda producir interbloqueo si se desea utiliza en la hebra de la interfaz. Para conseguir esto, los métodos pertinentes se ejecutarán de la siguiente forma:

- Supongamos el procedimiento P, que contiene una llamada a una función crítica (que solicita permiso al usuario)

- Supongamos que en la hebra H_1 se ejecuta el procedimiento P

- La llamada a P creará una hebra H_2, dentro de la cual se ejecutará el procedimiento crítico P_c

- La llamada a P termina independientemente de que P_c no lo haya hecho(en su hebra H_2), y H_1 continua su curso sin saber nada acerca del éxito o fracaso de P_c

- En H_2, cuando P_c termina, se genera un evento Pc_completado. Éste evento modificará el estado de la clase y, si es conveniente, de la interfaz gráfica. También puede ser util el evento Pc_error, en caso de que suceda algo inesperado en P_c.

Esta filosofía es la que seguiremos en los métodos de comunicación (Bluetooth) iniciales, es decir, aquellos previos al resto de las operaciones de comunicación. En el caso del servidor, la operación fundamental es la creación del servicio Bluetooth. En el cliente, la conexión a un servicio. Una vez ejecutados, si el usuario concede el permiso necesario, éste no volverá a ser requerido. La búsqueda de servicios también se realiza en una nueva hebra, y genera eventos asociados a ésta, aunque en principio no requiere permiso del usuario.

En las listas posteriores, se marcaran con un asterisco los métodos citados, así como los eventos asociados.

Servidor

1. Servicios:
   - Iniciar servicio*: se dispone a crear el servicio Bluetooth. La finalización de esta operación será asíncrona, y vendrá anunciada por un evento.
   - Finalizar servicio: desconecta a los clientes y elimina el servicio del SDDB.
   - Enviar mensaje a cliente
   - Desconectar cliente
   - Consulta del estado del servidor: permite saber si el servicio está activo.

2. Los eventos generados por el servidor serán los siguientes:
   - Servicio iniciado*: la creación del servicio se ha completado con éxito. A partir de este momento, un cliente puede conectarse si lo desea.
   - Cliente conectado
   - Cliente desconectado
   - Mensaje recibido de un cliente

Cliente

1. Servicios
   - Iniciar conexión a servidor*
   - Desconectar
   - Enviar mensaje

2. Eventos
   - Conexión con servidor completada*
   - Error en la conexión (no se llegó a conectar)*
   - Mensaje recibido

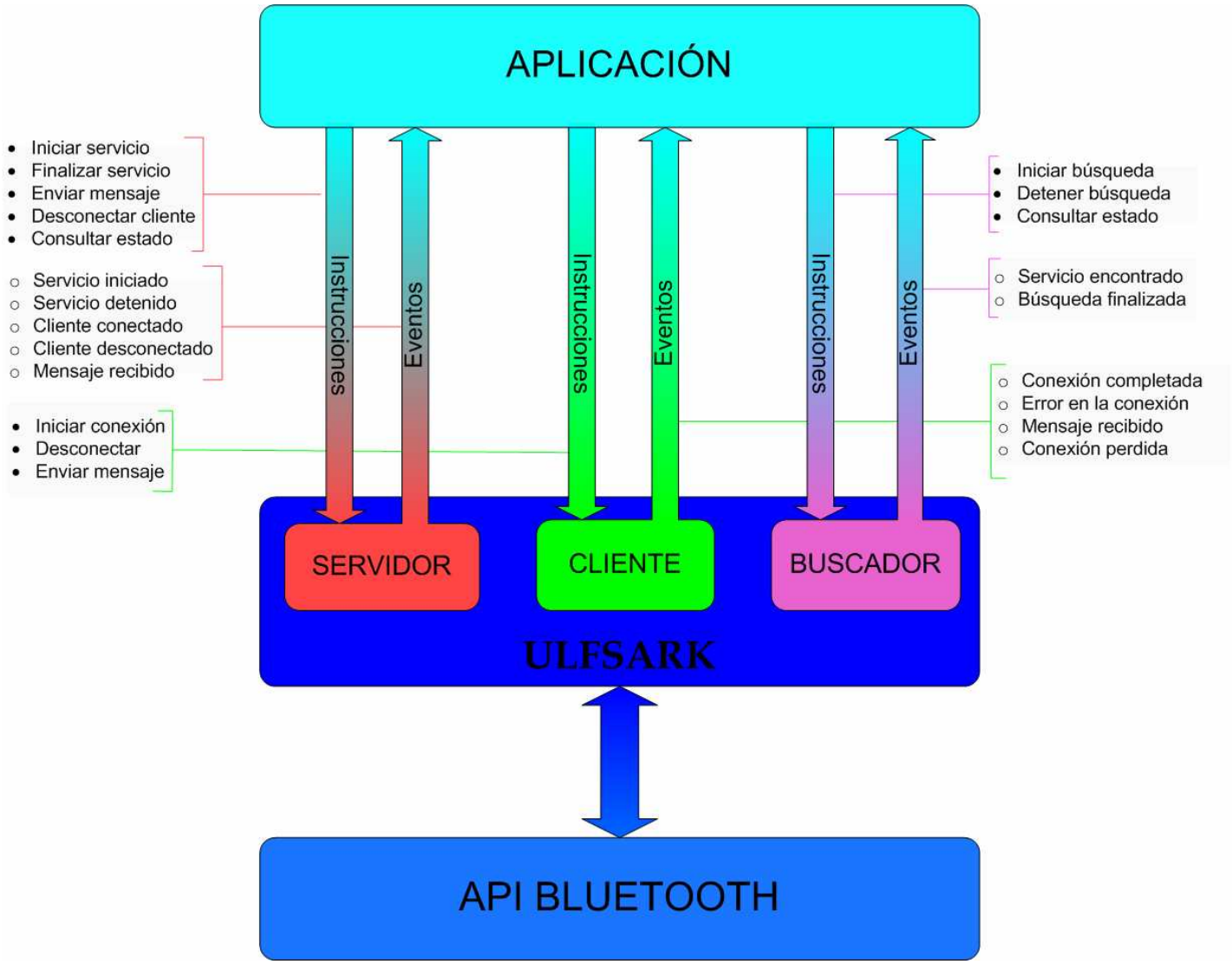
- Conexión perdida

Buscador

1. Servicios
   - Iniciar búsqueda
   - Detener búsqueda
   - Consultar estado del buscador

2. Eventos
   - Servicio encontrado
   - Búsqueda finalizada

De esta forma, queda definida la arquitectura externa de Ulfsark.


APLICACIÓN
Iniciar servicio
Finalizar servicio
Enviar mensaje
Desconectar cliente
Consultar estado
Servicio iniciado
Servicio detenido
Cliente conectado
Cliente desconectado
Mensaje recibido
Iniciar conexión
Desconectar
Enviar mensaje
Instrucciones
Eventos
Instrucciones
Eventos
Instrucciones
Eventos
Iniciar búsqueda
Detener búsqueda
Consultar estado
Servicio encontrado
Búsqueda finalizada
Conexión completada
Error en la conexión
Mensaje recibido
Conexión perdida
SERVIDOR
CLIENTE
BUSCADOR
ULFSARK
API BLUETOOTH

4.2.3 DISEÑO EN CAPAS

Introducción al Diseño en Capas

La programación por capas es un estilo de programación en la que el objetivo primordial es la separación de la lógica de negocios de la lógica de diseño, un ejemplo básico de esto es separar la capa de datos de la capa de presentación al usuario.

La ventaja principal de este estilo, es que el desarrollo se puede llevar a cabo en varios niveles y en caso de algún cambio sólo se ataca al nivel requerido sin tener que revisar entre código mezclado.

Además permite distribuir el trabajo de creación de una aplicación por niveles, de este modo, cada grupo de trabajo está totalmente abstraído del resto de niveles, simplemente es necesario conocer la API que existe entre niveles.

En el diseño de sistemas informáticos actual se suele usar las arquitecturas multinivel o Programación por capas. En dichas arquitecturas a cada nivel se le confía una misión simple, lo que permite el diseño de arquitecturas escalables (que pueden ampliarse con facilidad en caso de que las necesidades aumenten).

El diseño más en boga actualmente es el diseño en tres niveles (o en tres capas):

1.- **Capa de presentación:** es la que ve el usuario, presenta el sistema al usuario, le comunica la información y captura la información del usuario dando un mínimo de proceso (realiza un filtrado previo para comprobar que no hay errores de formato). Esta capa se comunica únicamente con la capa de negocio.

2.- **Capa de negocio:** es donde residen los programas que se ejecutan, recibiendo las peticiones del usuario y enviando las respuestas tras el proceso. Se denomina capa de negocio (e incluso de lógica del negocio) pues es aquí donde se establecen todas las reglas que deben cumplirse. Esta capa se comunica con la capa de presentación, para recibir las solicitudes y presentar los resultados, y con la capa de datos, para solicitar al gestor de base de datos para almacenar o recuperar datos de él.

3.- **Capa de datos:** es donde residen los datos. Está formada por uno o mas gestor de bases de datos que realiza todo el almacenamiento de datos, reciben solicitudes de almacenamiento o recuperación de información desde la capa de negocio.

Hemos comprobado en varias asignaturas que el modelo de capas se adapta perfectamente a un protocolo de comunicación (como TCP/IP).

Empecemos por la capa más baja. Tenemos que pasar de la capa física (ya implementada) a una capa que permita la comunicación entre dos dispositivos. Hace falta pues definir los paquetes de datos y lo más importante, un servidor y un cliente. El objetivo de esta capa es ser la base para cualquier aplicación (un chat, un juego…), por lo que el servidor, el cliente, y el paquete deberían ser lo más abstracto posible.

La segunda capa es la capa de aplicación. El cliente y servidor de esta capa utilizarán los métodos de las capas inferiores sin preocuparse de cómo está implementada. Pero hay que diferenciar el tipo de dispositivo en el que se ejecutará la aplicación, por lo que hace falta una última capa.

La última capa es la que distingue entre interfaces de la aplicación. De esta manera podemos utilizar la aplicación en dispostivos distintos, como PDAs, móviles o ordenadores personales.

Hemos decidido en los objetivos realizar un chat, por lo que la segunda capa (de aplicación) estará desarrollada para cumplir los servicios de un chat (entrar en una sala, cambiar el nick, enviar mensajes privados…). Esta capa se basa en la primera, por lo que el servidor y cliente de chat utilizarán las funciones de la capa inferior relativas al servidor y cliente de ésta última. Finalmente la capa superior dependerá del dispositivo que queramos usar, ya que la interfaz de éstos ha de desarrollarse de distinta forma.

Veamos un esquema de la arquitectura de capas que desarrollaremos:

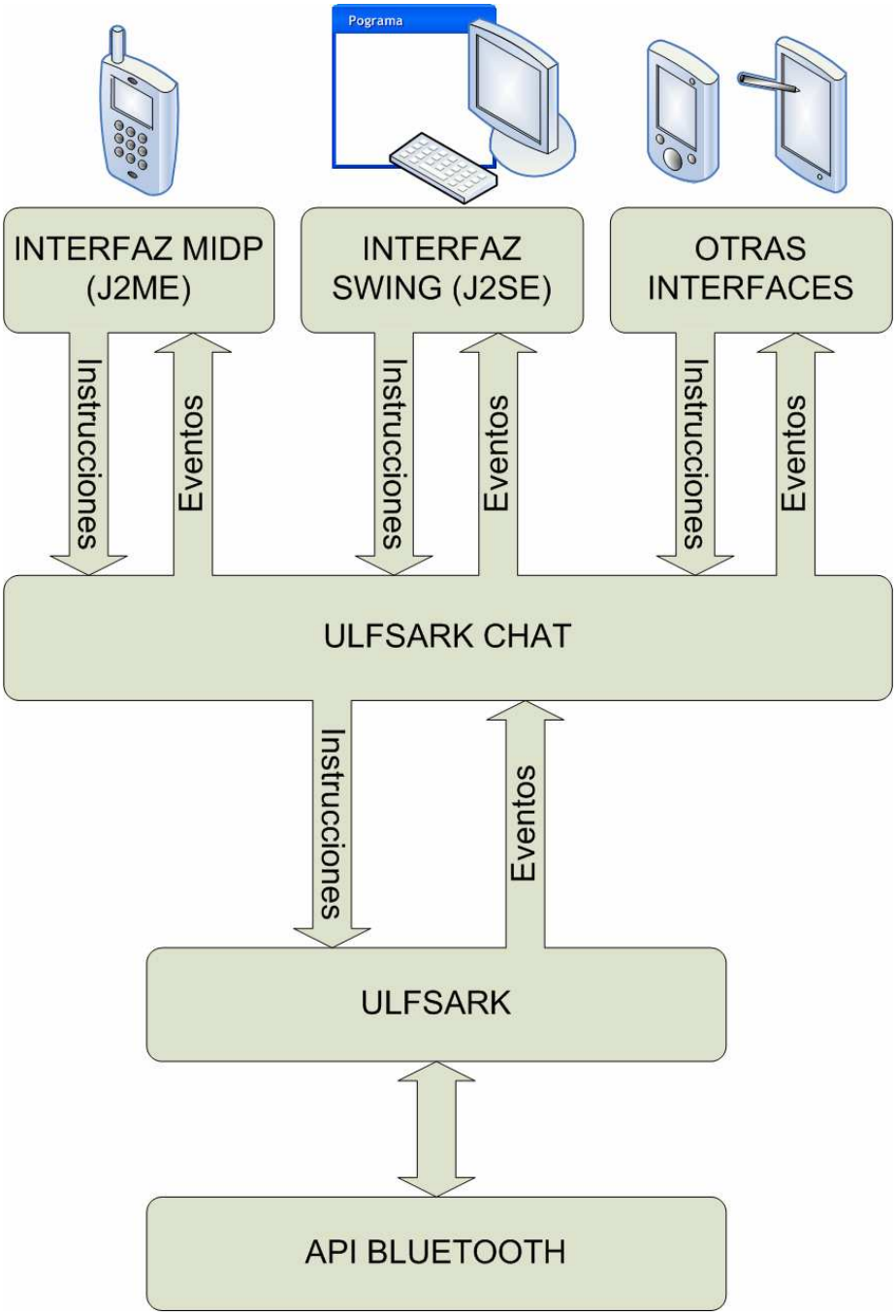


Diagrama de Capas

Diseño de cada capa

En esta parte vamos a comentar las clases que hemos usado para definir la arquitectura anterior, y el uso de distintos mecanismos, como las hebras, para la comunicación y sincronización.

4.2.3.A) CAPA ULFSARK

Como ya hemos dicho, en esta capa vamos a definir el cliente y servidor que usaremos en las capas superiores.

BUSCADOR SERVICIOS:

Esta clase facilita el uso de las clases DiscoveryAgent/DiscoveryListener, comentadas en la sección 3.3.1.

El constructor recibe un único UUID para identificar al servicio en sí y un DiscoveryListener para escuchar los eventos producidos. Estos eventos pueden ser:

Servicio encontrado: Proporciona directamente la url de conexión, el resto de parámetros son informativos, pero no son necesarios para conectar.

Búsqueda finalizada (de dispositivos y de servicios)

-Notificación: para enviar mensajes a la capa superior e informar de errores.

La búsqueda funciona de la siguiente forma:

1) Se inicia la búsqueda de dispositivos (con el método *iniciarBusqueda*)

2) Cada vez que se encuentra un dispositivo (evento *DeviceDiscovered*) se introduce en un vector de dispositivos. Al finalizar la búsqueda se produce el evento *InquiryCompleted*.

3) Para cada dispositivo del vector de dispositivos:

a) Se extrae del vector de dispositivos.

b) Se inicia la búsqueda de servicios de ese dispositivo. Al recibir un evento *servicesDiscovered* se lanza un evento *evtServicioEncontrado* por cada servicio.

c) Al recibir *ServiceSearchCompleted* volvemos al paso 3.

4) Si el vector de dispositivos está vacío se lanza el evento *evtBusquedaFinalizada*.

SERVIDOR:

Un servidor tiene tantas hebras lectoras y registros cliente como clientes haya.

El constructor de la clase recibe como parámetros el UUID y los atributos en un hash de la forma (clave:Integer, Atributo:String).

El método iniciarServicio() creará una hebra que iniciará el servicio y después entrará en un bucle continuo de la siguiente forma.

```
while(servicioActivo){
   acepta conexion entrante
   crea un registroCliente para el cliente
   inicia una hebra de lectura (Servidor.HebraLectora)
}
```

Es la hebra principal creada la que avisará a las capas superiores de la creación de un servicio mediante el evento *evtServicioIniciado*.

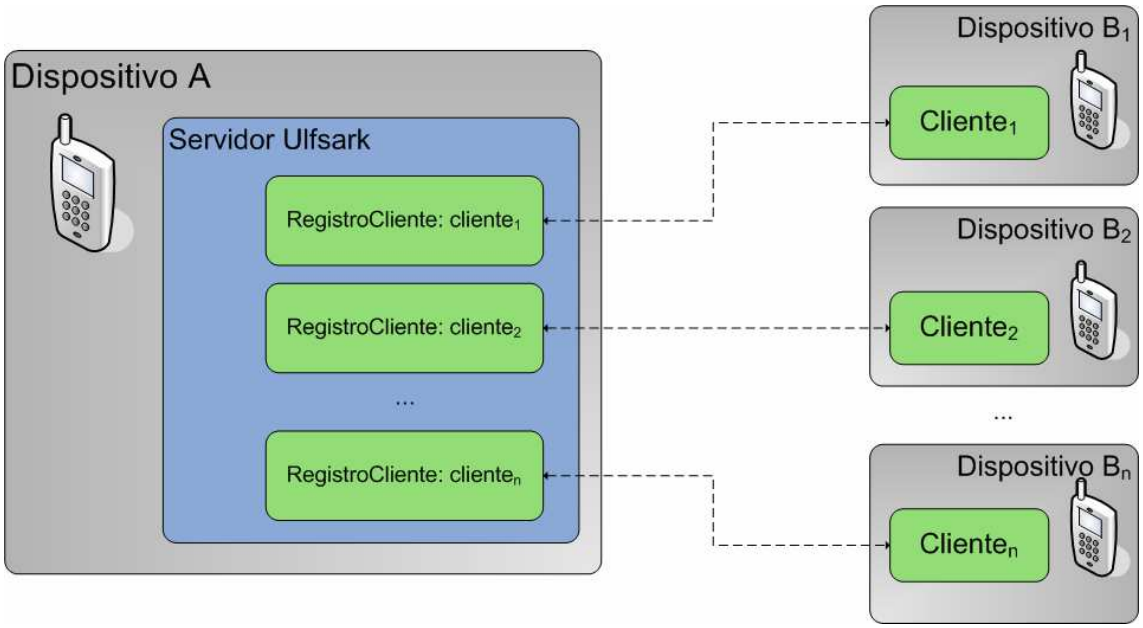


Diagrama del servidor

Los **métodos** del servidor son los siguientes:

- *void ModificarAtributo(int id, String valor)*

    - Modifica los atributos del registro de servicio (ServiceRecord)
- *void DetenerServicio()*
    - Cierra el notificador de conexiones
    - Desconecta a todos los clientes
- *void desconectarCliente(RegistroCliente rc)*
    - Finaliza la hebra lectora asociada a ese cliente
    - Elimina el RegistroCliente rc del vector.
    - Elimina la hebra lectora del cliente.
- *void añadirListener()*
    - Añade un listener que recibirá los eventos del servidor
- *UUID obtenerUUID()*
    - Consulta el identificador del servicio
- *boolean estaActivo()*
    - Devuelve *true* si se aceptan nuevos clientes

Los eventos que genera son:

- *evtServicioIniciado():* Se genera en la hebra principal (la que inicia el servicio y después acepta las conexiones)
- *evtClienteAñadido(RegistroCliente rc):* Se genera también en la hebra principal
- *evtMensajeRecibido(UMPpacket msg, RegistroCliente rc):* Se genera en la hebra lectora correspondiente a ese cliente.
- *evtClienteDesconectado(RegistroCliente rc):* Se genera en la hebra lectora correspondiente al cliente que desconecta.

CLIENTE:

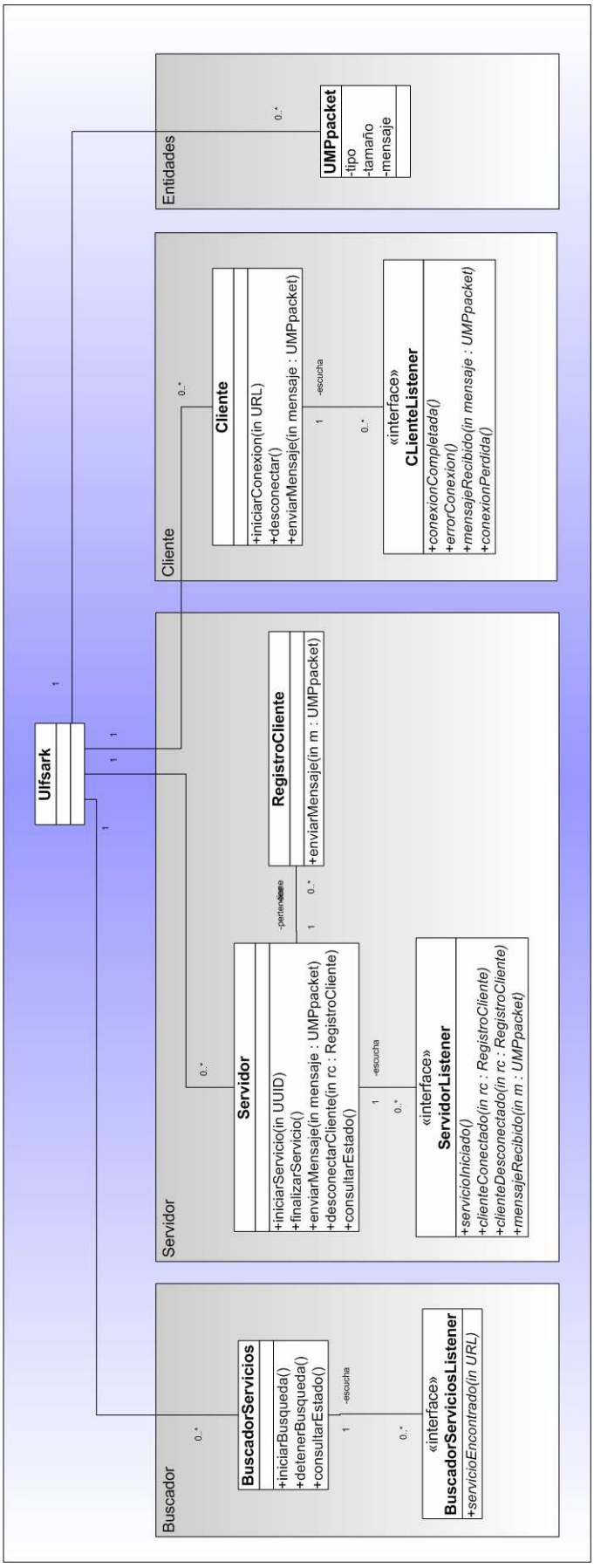
El cliente al crearse puede conectar a cualquier servicio, por lo que el buscardor será quien le otorgue la UUID del servicio al que quiere conectarse.

Métodos del cliente:

- *conectar(String URL)*
  - Crea una hebra *HebraConexión*
  - Esta hebra realiza la conexión al servidor y crea una hebra de lectura *HebraLectora*
  - Genera el evento *evtConexion*() al terminar
- *añadirListener*
- *conectar*
- *desconectar*
- *getURL*

Eventos Generados:

- *evtMensajeRecibido(UMPpacket mensaje):* En la hebra lectora
- *evtDesconexión(String msg)*
- *evtConexion()*
- *evtErrorConexion()*


Ulfsark
Entidades
UMPpacket
-tipo
-tamaño
-mensaje
Cliente
Cliente
+iniciarConexion(in URL)
+desconectar()
+enviarMensaje(in mensaje : UMPpacket)
«interface»
CLienteListener
+conexionCompletada()
+errorConexion()
+mensajeRecibido(in mensaje : UMPpacket)
+conexionPerdida()
-escucha
Servidor
Servidor
+iniciarServicio(in UUID)
+finalizarServicio()
+enviarMensaje(in mensaje : UMPpacket)
+desconectarCliente(in rc : RegistroCliente)
+consultarEstado()
RegistroCliente
+enviarMensaje(in m : UMPpacket)
«interface»
ServidorListener
+servicioIniciado()
+clienteConectado(in rc : RegistroCliente)
+clienteDesconectado(in rc : RegistroCliente)
+mensajeRecibido(in m : UMPpacket)
Buscador
BuscadorServicios
+iniciarBusqueda()
+detenerBusqueda()
+consultarEstado()
«interface»
BuscadorServiciosListener
+servicioEncontrado(in URL)

4.2.3.B) CAPA ULFSARKCHAT

En esta capa se implementará un chat como ejemplo de aplicación sobre la capa Ulfsark. En cada tipo de servicio tendremos que crear un Buscador, un Servidor y un Cliente basándonos en los comentados anteriormente. Con esto conseguimos separar un elemento abstracto (por ejemplo, un servidor) de un elemento específico para una aplicación (un servidor chat). En esta clase definiremos los campos de los paquetes del protocolo de acorde con nuestros intereses.

Vamos a enumerar las clases anteriormente comentadas junto con las diferencias que modifican su comportamiento (más específico) en esta capa. El buscador, el servidor y el cliente además cuentan con listeners asociados a cada uno de ellos.

BuscadorSalasChat

A diferencia del BuscadorServicios sólo busca un tipo de servicio específico, en este caso, el chat, por lo que tiene menos argumentos y sólo buscará el tipo de servicio y atributos de ulfsarkchat.

Cuando busca servicios, al recibir el *evtServicioEncontrado* enviará el evento *evtSalaEncontrada*(SalaChat, sala) a la capa superior (capa de la interfaz).

El resto se comporta igual que BuscadorServicios.

SalaChat

Esta clase incluye información sobre cada sala como por ejemplo la URL, el nombre, el creador y la dirección.

Servidor

El servidor se comporta como un servidor de chat. Esto quiere decir que a diferencia del que se usa como base en la capa inferior el único servicio que crea es el de chat. Por eso el UUID y los atributos están fijos:

```
public static UUID UUID_Ulfsark_Chat = new
UUID("10203040506070809OA0B0C0D0F0F010", false);

public static final  Integer ID_nombre_sala = new Integer(59169);
public static final  Integer ID_creador = new Integer(59170);
public static final  Integer ID_nclientes = new Integer(59171);
```

Además incorpora los códigos de los distintos paquetes que se usarán en el chat:

```
public static final byte MSG_GLOBAL = 0x0;
public static final byte MSG_PRIVADO = 0x1;
public static final byte MSG_LISTA_USR = 0x2;
public static final byte MSG_NUEVO_USR = 0x4;
public static final byte MSG_ERROR = 0x5;
public static final byte MSG_AVISO = 0x6;
public static final byte MSG_NICK = 0x7;
public static final byte MSG_LOGOUT = 0x8;
public static final byte MSG_WHOIS = 0x9;
public static final byte MSG_LOGIN_OK = 0xA;
public static final byte MSG_USR_DESC = 0xB; //Usuario
desconocido
public static final byte MSG_NICK_USO = 0xC; //Nick ya en uso
```

El servidor tiene un vector de clientes, una tabla asociativa con nicks de los clientes (donde las claves son RegistroCliente) y un vector de listeners. El servidor también puede ser usuario del chat, por lo que también tiene un nick de usuario.

Los **métodos básicos del servidor** son los siguientes:

- setNick(String nick): comprueba que el nuevo nick (nombre de usuario) usar no esté en uso. Si es así cambia de nick e informa a los clientes mediante el mensaje MSG_NICK.

- setNombreSala(String nombre): modifica el atributo nombre de la sala. Al modificar un atributo los nuevos clientes obtendrán estos nuevos valores en el buscador.
- setCreador(String nombre): modifica el atributo nombre del creador de la sala.

Los **métodos relativos al chat** son:

- *iniciar():* Inicia el servicio. No hace nada hasta recibir el *evtServicioIniciado.*
- *reset():* Desconecta a cada uno de los clientes, borra los nicks y detiene el servicio.
- *enviarMensajeGlobal(String nickOrigen, String msg):* Envía un paquete MSG_NORMAL a todos los clientes excepto al cliente cuyo nick es nickOrigen.
- *enviarMensajePrivado(String nickDestino, String msg):* envía un mensaje MSG_PRIVADO al cliente cuyo nick es nickDestino.
- *Vector obtenerListadoUsuarios():* devuelve un Vector con objetos RegistroCliente.

Los eventos recibidos se tratan de la siguiente manera:

ClienteAñadido:

ClienteDesconectado: se envía un paquete MSG_LOGOUT a todos los demás y se envía al listener del servidor el evento evtUsuarioDesconectado (sólo en caso de que estuviera registrado su nick).

ServidorIniciado: informar al listener.

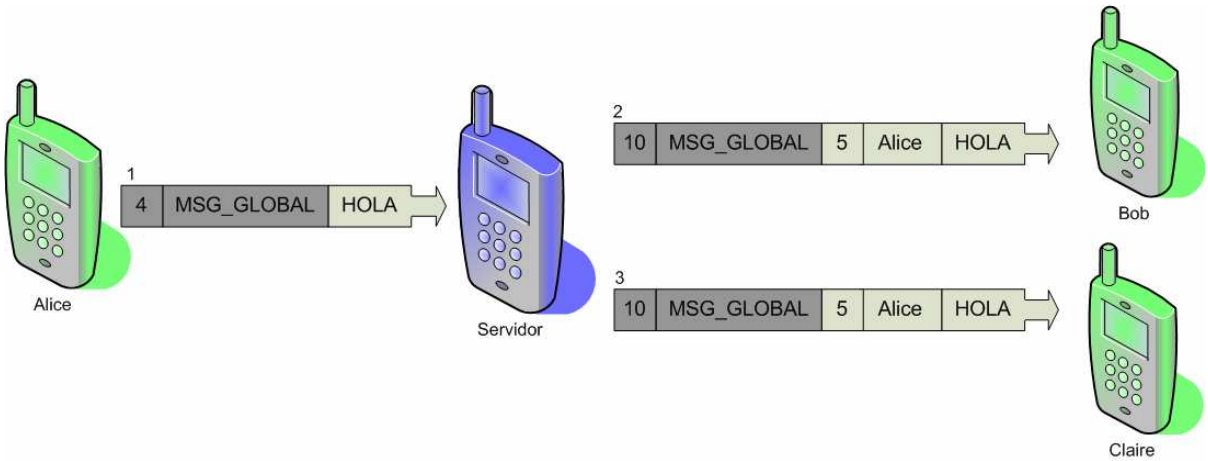
MensajeRecibido: Este evento es el más importante. Lo explicamos a continuación:

Gestión de los mensajes recibidos, en función del código:

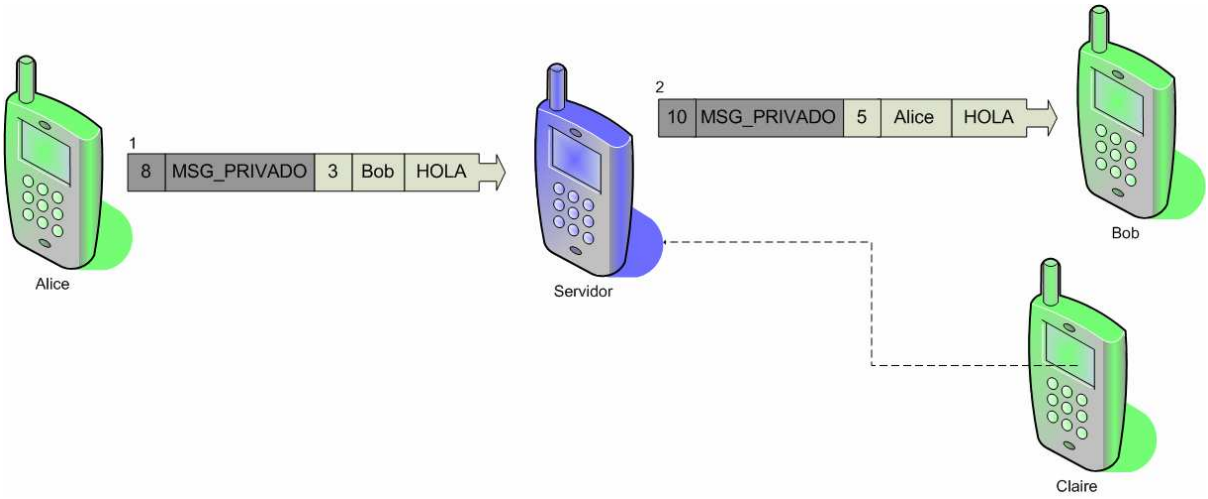
- **MSG_NUEVO_USR**

- Si el nick está en uso, contesta con un paquete MSG_NICK_USO
- Si no, informa al listener (*evtUsuarioConectado*), registra el cliente (guarda el nick), contesta con un paquete MSG_LOGIN_OK y envía al resto de clientes un paquete MSG_NUEVO_USR
- Informa al listener: *evtUsuarioConectado*

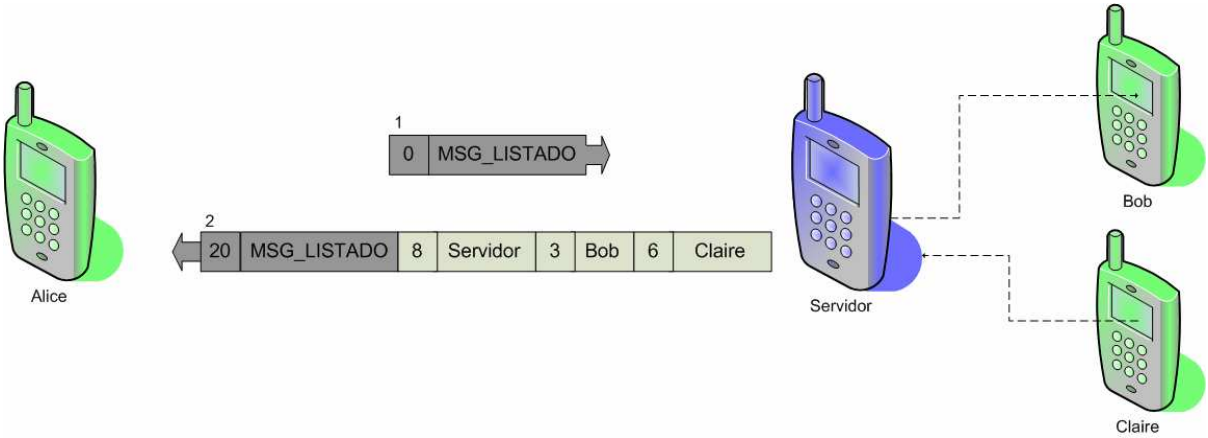
- **MSG_GLOBAL**:
  - envía a todos un paquete MSG_GLOBAL con el mismo texto
  - informa al listener: evtMensajeRecibido



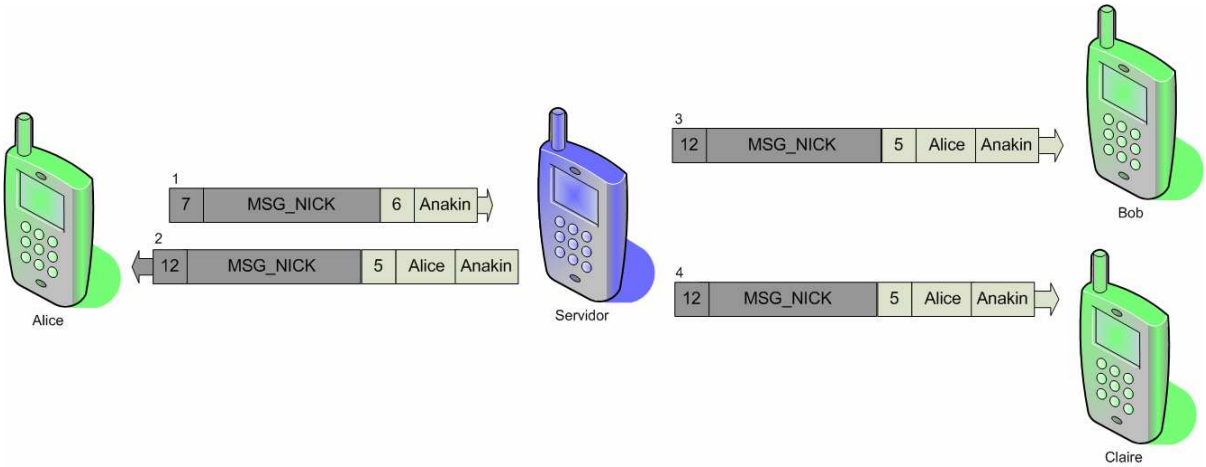
- **MSG_PRIVADO**:
  - Si el nick de destinatario es el del servidor, informa al listener (*evtMensajePrivadoRecibido*)
  - en otro caso, envía un paquete MSG_PRIVADO al cliente que tenga ese nick, con el mismo texto
  - si no hay ningun cliente con ese nick, contesta con un paquete MSG_USR_DESC



- **MSG_LISTA_USR**
  - Crea un paquete MSG_LISTA_USR con los nicks de los usuarios y lo envía al remitente.



- **MSG_NICK**
  - Si el nick está en uso, contesta con un mensaje MSG_NICK_USO.
  - En otro caso, registra el nuevo nick y envia a todos los clientes un paquete MSG_NICK, con el nick anterior y el nuevo.
  - Informa al listener: *evtNuevoNick*.

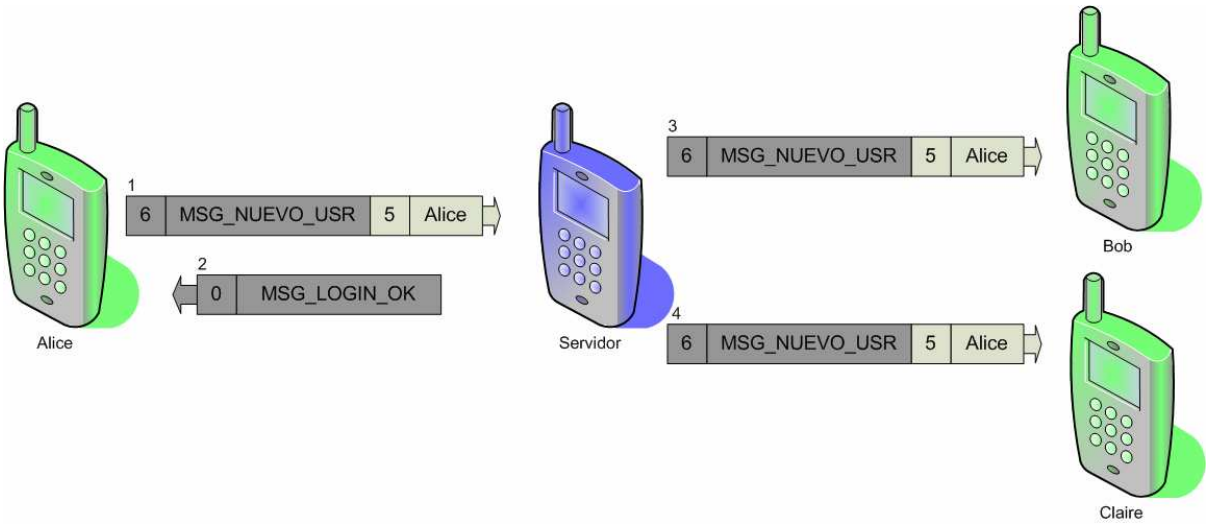


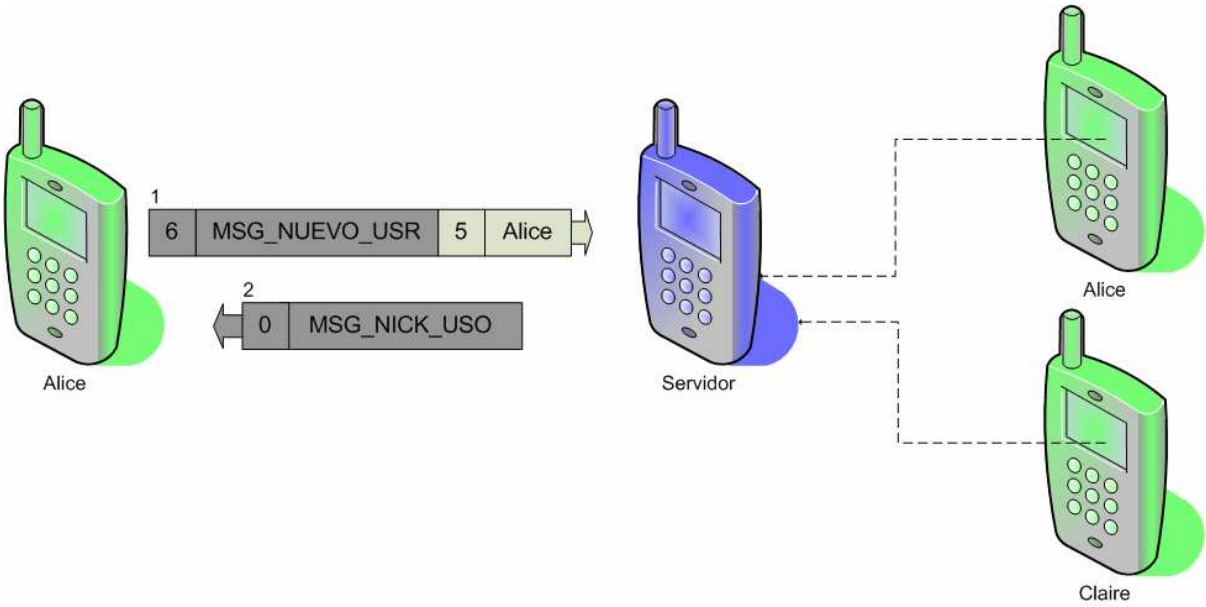
- **MSG_LOGOUT**
  - Informa al listener: *evtUsuarioDesconectado*
  - Desconecta al cliente y borra su registro y su nick

Cliente

Finalmente el cliente, al igual que el servidor utiliza un cliente genérico para especializarse en un solo tipo de servicio.

Para conectarse a un servidor se ejecuta el método conectar(). Al recibir *evtConexion*() la conexión ha finalizado con éxito, por lo que se envía un paquete MSG_NUEVO_USR al servidor que contiene el nick. Cuando el cliente reciba MSG_LOGIN_OK se da por finalizado el registro del cliente.





Métodos del cliente:

- *cambiarNick(String nick)*
- *solicitarCambioNick(String nuevoNick):*envía un paquete MSG_NICK al servidor. El cambio se hace efectivo cuando se recibe un paquete MSG_NICK del servidor, confirmando el nuevo nick.
- *solicitaListadoClientes:* envia un paquete MSG_LISTA_USR al servidor.
- *enviarMensaje(String msg):*envia un mensaje MSG_NORMAL al servidor, con el texto *msg.*
- *enviarMensajePrivado(String nick, String msg)*: envia al servidor un mensaje MSG_PRIVADO, incluyendo el nick del destinatario y el mensaje
- *logout():* envia un mensaje MSG_LOGOUT y desconecta
- *setListener*

Los eventos que pueden ocurrir son los siguientes

- *evtConexion* (comentado arriba)
- *evtErrorConexion*: se envía el evento *evtErrorConexionServidor* al listener
- *evtDesconexion:* se envia al listener el evento *evtConexionPerdida*

- *evtMensajeRecibido*, en función del código genera un evento:
  - MSG_NUEVO_USR -> *evtUsuarioConectado(msg.toString())*
  - MSG_LOGIN_OK -> *evtLoginCompletado()*
  - MSG_NORMAL -> *evtMensajeRecibido(…)*
  - MSG_PRIVADO -> *evtMensajePrivadoRecibido(…)*
  - MSG_LISTA_USR -> *evtListaRecibida(nicks)*
  - MSG_NICK
    - *evtNuevoNick(nickAnterior, nickNuevo);*
    - si nickAnterior es el nick de este cliente, guarda el nick nuevo como nick actual
  - MSG_NICK_USO -> *evtNotificacionServidor("Nick en uso")*
  - MSG_USR_DESC *evtNotificacionServidor(msg.toString() + " no está conectado")*
  - MSG_LOGOUT -> *evtUsuarioDesconectado(…)*

4.2.3.C) CAPA CHATJ2ME Y J2SE

Finalmente la capa de aplicación será una capa que presenta la información de las capas inferiores, y que depende del dispositivo que queramos usar. Con esto podemos comunicarnos por ejemplo entre un PDA y un ordenador simplemente usando las funciones de la capa de servicio (chat). En la capa UlfsarkChat subyacente, se implementa el servidor, cliente y buscador de salas sin especificar ningún aspecto relativo a la presentación de datos. Ésta será la función de la última capa, en la que se diseñarán las interfaces para:

a) Dispositivos móviles con plataforma Java 2 Microedition
b) Dispositivos con plataforma Java 2 Standard Edition

Interfaz J2ME

Es el objetivo principal, pues el chat pretende ser usado principalmente en dispositivos móviles. Lo primero que se mostrará es un menú que contendrá accesos a las distintas funciones del programa.

Menú principal:

1. **Crear sala**: esta opción nos llevará a un formulario simple en el que se solicitará al usuario el nombre de la sala. Desde este formulario, se podrá volver atrás o confirmar la creación de la sala. Una vez confirmado, se mostrará el **formulario de conversación**.

2. **Unirse a sala**: seleccionando esta opción, se mostrará la **pantalla de búsqueda**.

3. **Test de compatibilidad**: mostrará un listado en el que indicarán la disponibilidad de los APIs que necesita el programa para funcionar correctamente. Para esta aplicación, los APIs requeridos son:
   - API bluetooth: necesario para que el chat funcione
   - API fileconnection: necesario para guardar logs de las conversaciones, pero no imprescindible
   - RMS: Memoria persistente del móvil

4. **Opciones**: se presentará en pantalla un formulario con los siguientes componentes:
   - Nick del usuario en el chat
   - Opciones de logging:
     - Directorio donde se guardarán los logs
     - Formato
   - Opciones de blogging:
     - Nombre del blog,
     - Identificador de usuario,

- Contraseña
- URL

Desde este formulario, se podrá volver al menú principal aceptando los nuevos valores o cancelando. Si se acepta, se aprovechará el sistema de gestión de registros del móvil (RMS) para guardarlas de forma persistente sin necesidad de acceder a sistemas de archivos.

5. Información de los autores

6. Salir

Formulario de conversación:

Este formulario contendrá una entrada de texto para enviar mensajes al chat. Además de esto, se imprimirán los mensajes recibidos de otros usuarios del chat (así como los propios mensajes del usuario actual). Tiene un carácter dual, pues se utilizará tanto para el servidor como para el cliente.

Pantalla de búsqueda

Este listado (clase List) mostrará las salas de chat encontradas en el entorno progresivamente. Cada sala se identifica por sus atributos:

- Nombre de la sala
- Nombre del creador de la sala

Cuando se termine la búsqueda, se indicará este evento.

Interfaz J2SE

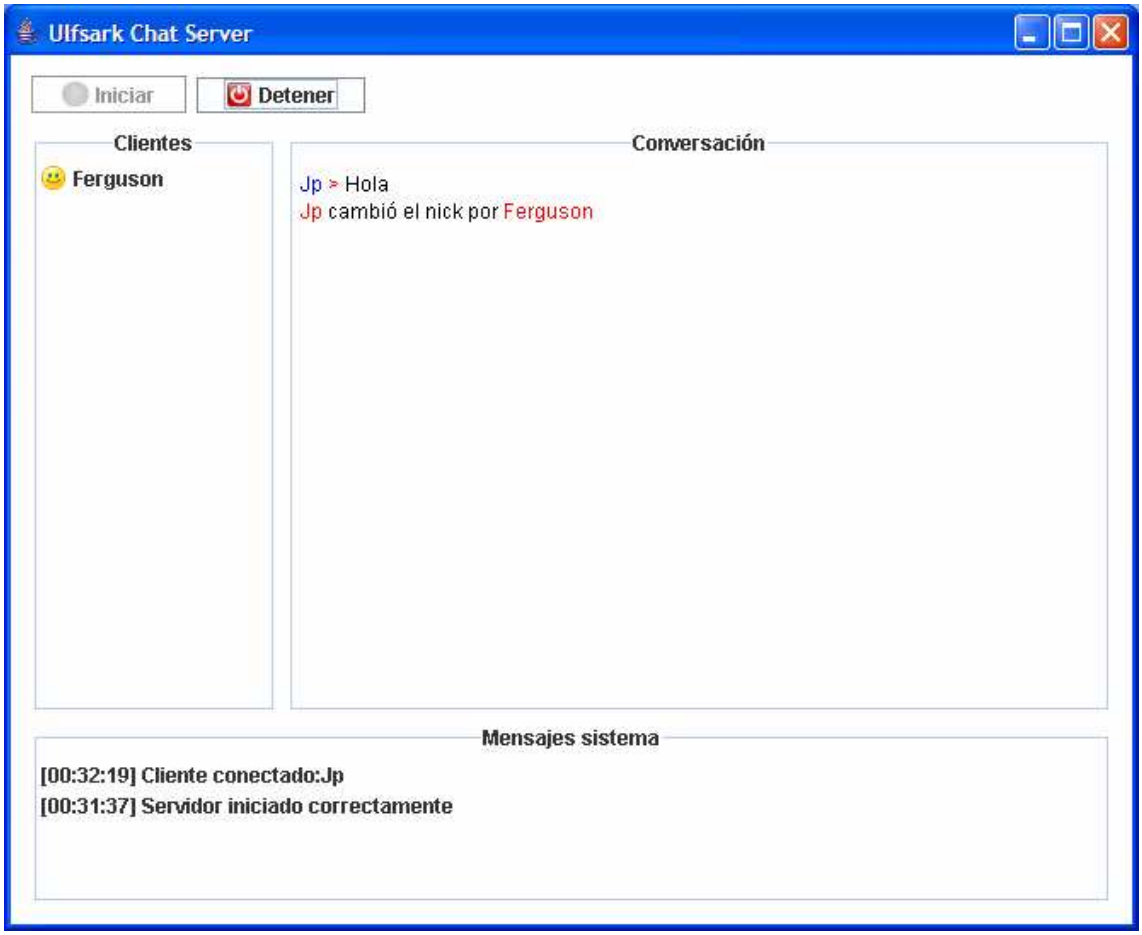
1. Interfaz del servidor de chat

La funcionalidad que se pretende dar al servidor de chat para J2SE es simplementa la creación del servicio y la monitorización de los mensajes transmitidos entre usuarios. Por tanto, la interfaz sólo constará de una ventana con estos componentes:

- Panel con controles para iniciar/detener servicio
- Lista con los clientes conectados
- Panel de conversación

- Panel con los mensajes del sistema.

Cada vez que se reciba un mensaje de un usuario (destinado al propio servidor, a otro usuario concreto o a todos) se mostrará en pantalla.



2. Interfaz del cliente de chat

En este caso, la aplicación debe ser interactiva. Debe permitir al usuario la búsqueda de salas de chat en el entorno, la conexión y la participación en éstas (envío y recepción de mensajes).

Los componentes de esta interfaz serán:

- Panel de controles
- Lista con las salas de chat del entorno

- Panel de conversación (en este caso incluirá entrada de texto para enviar mensajes)

Panel de mensajes del sistema

5. CONCLUSIONES

Ulfsark es una base amigable para el desarrollo de aplicaciones de comunicación entre dispositivos Bluetooth. Hemos pretendido que el desarrollo de este proyecto, el código, la documentación y demás materiales sirva a futuros desarrolladores para realizar aplicaciones más completas.

Una aplicación Ulfsark se compone de tres capas:

- La primera, capa Ulfsark, crea un servidor, cliente y buscador de servicios genérico.
- La segunda, capa del servicio, utiliza los elementos anteriores para crear un servicio específico (en nuestro caso, un chat).
- La última capa es la interfaz para varios dispositivos de diversa índole que permiten la comunicación entre sí.

Con unas pequeñas nociones y conceptos es fácil crear comunicación entre dispositivos Bluetooth. Además el uso de Java hace que el número de dispositivos compatibles sea superior a cualquier otra herramienta, pudiendo crear futuras aplicaciones.

Algunas de esas aplicaciones futuras que se nos ocurren son:

- Sistema de votación

En una pantalla se ofrecen opciones a elegir. Los usuarios votan desde sus móviles una sala vez, y pueden consultarse los resultados en tiempo real, bloquear las votaciones, cambiar de voto…

- Juegos multijugador

Por ejemplo cada paquete envía información de la posición actual del personaje.

- Uso en docencia

Los alumnos pueden comentar sobre las clases, responder a preguntas del profesor, enviar dudas...

- Chat en lugares de ocio

Por ejemplo, en un bar o discoteca, para hacer amigos, ligar…

- Computación distribuida

Creación de *scatternets* dinámicas para computación paralela. En este caso tendríamos que utilizar la clase UMPpacket para crear paquetes de sincronización, de datos o de conexión.

- **Manejo de un PC desde un dispositivo móvil**

Esta es una funcionalidad muy interesante. Existen clases en Java que permiten ejecutar comandos, abrir programas, mover el ratón, cambiar transparencias… El dispositivo móvil puede usarse para controlar la ejecución de películas, transparencias, reproductores de música…

- Impresión de documentos

Existen impresoras Bluetooth que permiten imprimir archivos desde dispositivos como los que mencionamos.

- Otros dispositivos

Existen muchos más dispositivos aparte de PDAs, móviles y ordenadores personales que utilizan la tecnología Bluetooth. Videoconsolas, cámaras de fotos, Mp3, GPS… Ulfsark podría ser utilizado para comunicarse con todos estos dispositivos para un sinfín de aplicaciones.

Estos son unos pocos ejemplos que se nos ocurren, pero sirven para entender el alcance que puede llegar a tener usar a Ulfsark como base. Nuestra intención es, pues, hacer una base fácil de utilizar a futuros usuarios para desarrollar fácilmente cualquier idea creativa que se nos pueda ocurrir.

6. BIBLIOGRAFÍA

- **Bluetooth for Java**. *Bruce Hopkins and Ranjith Anthony*. Ed Apress.
- **JavaTM APIs for BluetoothTM Wireless Technology (JSR-82)** *Specification Version 1.0a JavaTM 2 Platform, Micro Edition*
- **J2ME.** *Sergio Gálvez Rojas Lucas Ortega*. Díaz Dpto. de Lenguajes y Ciencias de la Computación. E.T.S. de Ingeniería Informática Universidad de Málaga
- **JSR-82: Bluetooth desde Java** *Alberto Gimeno Brieba*
- http://java.sun.com
- http://forum.nokia.com
- http://es.wikipedia.org
- http://www.todo-symbian.com
- http://jcp.org/aboutJava/communityprocess/final/jsr075/