Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Trabajo Fin de Máster

Máster en Ingeniería de Sistemas e Informática

Cusro 2013/2014

Estudio y Adaptación de la Plataforma de Agentes Móviles SPRINGS para Entornos Inalámbricos

Néstor Fabio Muñoz García

Director: Sergio Ilarri Artigas

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Noviembre 2014

Estudio y Adaptación de la Plataforma de Agentes Móviles SPRINGS para Entornos Inalámbricos

RESUMEN

Las plataformas de agentes móviles son sistemas que permiten tener ciertos "programas" peculiares. Estos programas se diferencian de los típicos por ser capaces de moverse de una máquina a otra de forma autónoma eligiendo en qué maquina llevarán a cabo su ejecución.

Esto abre un abanico de posibilidades en computación distribuida y entornos móviles, ya que los procesos pueden distribuir su carga de trabajo en función de una o varias máquinas y decidir qué tarea realizar dependiendo de su posición mientras su entorno va variando.

A pesar de que existan varias plataformas de agentes móviles, el trabajo se centra en una única plataforma de agentes móviles, desarrollada por el grupo de trabajo de la universidad de Zaragoza SID (Sistemas de Información Distribuida), llamada SPRINGS (Scalable PlatfoRm for movINg Software).

Aunque en un principio esta plataforma sólo se ejecutaba en máquinas virtuales Java, recientemente el grupo de investigación clonó la plataforma y la modificó para que se ejecutara en dispositivos Android, manteniendo toda la esencia de la plataforma pero sin tener en cuenta toda la potencia que este sistema les proporcionaba con las comunicaciones inalámbricas.

El objetivo de este trabajo es estudiar el posible desarrollo de nuevas funcionalidades que permitan utilizar de forma más adecuada los distintos entornos inalámbricos que pueden estar disponibles (Wifi, 3G, Bluetooth). Para conseguir este objetivo, es necesario:

- Estudiar la plataforma en su estado actual.
- Modificar la plataforma para utilizar todos los recursos disponibles de comunicación inalámbrica (WiFi, redes móviles, Bluetooth).
- Implementar nuevas funcionalidades como la búsqueda de servicios o el desplazamiento de agentes a un área geográfica.

El trabajo realizado representa un primer paso preliminar para el potencial desarrollo futuro de una plataforma de agentes móviles para Android que esté perfectamente adaptada al entorno móvil. El grupo SID pretende seguir progresando su investigación en este tema.

Índice general

Abstract **3**

1. Introducción **7**
1.1. Motivación 8
1.2. Estructura de la memoria 8

2. La plataforma de agentes móviles SPRINGS **9**
2.1. Análisis de la estructura básica de SPRINGS 9
2.2. Conociendo la plataforma 10
2.2.1. Ejecución de un servidor RNS 10
2.2.2. Ejecución de un contexto y conexión con el servidor RNS anterior 11
2.2.3. Ejecución de la plataforma con agentes 12
2.3. La plataforma SPRINGS en Android 12
2.3.1. Estado de la plataforma 13
2.3.2. Ejecución y problemas con la plataforma 13

3. Abstracción de las Comunicaciones: Sockets Abstractos **15**
3.1. La librería Sockets abstractos 15
3.1.1. Diseño de la librería 16
3.1.2. Integración de la librería en SPRINGS 18
3.2. Creando una red Mesh 20
3.2.1. Los nodos de la red: Node 20
3.2.2. Los puertos de la red: Port 22
3.3. Diseño de la nueva red Mesh 23
3.3.1. Conexiones virtuales 23
3.3.2. El Protocolo de comunicación 25
3.3.3. La conexión puente 28
3.4. Implementación de la nueva red Mesh 29
3.4.1. El gestor del nodo: NodeManager 29
3.4.2. Implementación del puerto: Port 29
3.4.3. Nodo genérico: Node 30
3.4.4. El comprobador de puertos y nodos: PortsChecker 31

4. Ampliación de Funcionalidades **33**
4.1. Comunicación Android-Java 33
4.2. Añadiendo agentes geográficos 35
4.2.1. Diseñando los agentes geográficos 35

4.2.2. Añadiendo la funcionalidad 36

5. Conclusiones **39**
5.1. Trabajos futuros 40
5.2. Evaluación del proyecto 41
5.3. Tiempo dedicado 41

Capítulo 1

Introducción

Dada la potencia de los ordenadores, con los que se pueden realizar muchas tareas, es extraño que los programas estén limitados a ejecutarse con el procesador con el que fueron compilados. Aunque esta barrera ha sido más o menos superada por los sistemas operativos a bajo nivel y por ciertas librerías a alto nivel (.Net, Java), la idea de que un programa se pueda mover de un ordenador a otro para que continúe su ejecución sea por el motivo que sea (necesita ciertos datos o el ordenador en el que se encontraba se va a apagar) es muy buena.

Los programas que se "mueven" de un ordenador a otro se conocen como agentes móviles [1] [2]. Los agentes móviles permiten:

- Reducir la carga de la red: en lugar de que un cliente esté constantemente pidiéndole datos a un servidor para realizar su trabajo, éste podría enviarle un agente al servidor, que realice sus tareas allí, pidiéndole todos los datos que necesite, y regrese al cliente con los resultados.

- Realizar ejecuciones asíncronas y autónomas: en entornos con dispositivos móviles es muy común que se establezcan conexiones, pero estas conexiones son muy frágiles dependiendo de cada situación. Con los agentes móviles se le podría encargar a un agente que saliera a la red a realizar una determinada tarea y, a partir de ahí, el agente es independiente del dispositivo móvil pudiendo quedar desconectado o incluso apagado. Cuando el agente termine su tarea entonces volverá al dispositivo móvil que lo creó con los resultados obtenidos.

- Evitar los fallos que se producen por determinadas condiciones del ordenador: si por ejemplo el ordenador se va a apagar y el agente lo detecta, puede decidir irse a otro ordenador a continuar su ejecución, mientras que una aplicación típica se vería afectada por el apagón y lo único que podría hacer es guardar su estado antes de cerrar forzosamente.

- Tener un asistente personal: como los agentes tienen la capacidad de poder desplazarse a otros ordenadores, también pueden realizar ciertas tareas en nombre de otros. Por ejemplo la búsqueda y la manipulación de cierta información en una red personal o privada.

- Monitorizar en tiempo real: si estamos esperando a que cierto sensor alcance cierto nivel para realizar alguna tarea, no es necesario crear un cliente

que esté captando todo el rato el valor actual de ese sensor. Basta con enviar un agente al servidor donde se encuentra el sensor y, en cuanto alcance el nivel que esperábamos, el agente nos avise o actúe él directamente.

- Generar un procesamiento en paralelo: En lugar de tener un programa que genere muchas subtareas, consumiendo notablemente los recursos del ordenador, podemos crear un agente que se encargue de generar múltiples clones para realizar la tarea en paralelo y que estos clones se distribuyan por toda la red. Una red de ordenadores estará menos saturada que un único ordenador si realizan la misma tarea en paralelo.

Este trabajo es un paso para el futuro desarrollo de una plataforma de agentes móviles para Android que esté perfectamente adaptada al entorno móvil. El grupo de investigación SID pretende seguir progresando su investigación en este tema.

1.1. Motivación

A pesar de que las plataformas de agentes móviles [3] [4] [5] (por ejemplo Concordia [6], MAP [7] o JAMES [8]) llevan existiendo desde hace mucho tiempo, una vez visto el gran potencial que pueden aportar [9] [10], podrían suponer una gran herramienta para el futuro de la tecnología [11] [12], en especial con el Internet de las cosas. Por ejemplo, en lugar de tener los electrodomésticos conectados a la red, tendríamos a un agente que realizaría las tareas que le ordenásemos, en el momento deseado y a nuestro gusto.

Con este trabajo se pretende estudiar y trabajar con una plataforma de agentes móviles (SPRINGS [13], que se ejecuta en Java [14]) para aprender cómo funciona [15] y así poder manejar agentes móviles. con mis propias manos una de estas plataformas tan interesantes.

1.2. Estructura de la memoria

A partir de esta introducción, el capítulo 2 trata de explicar qué es SPRINGS, la plataforma de agentes móviles con la que se ha trabajado en este trabajo y la plataforma SPRINGS en Android, una modificación del SPRINGS original para que funcionara en Android.

En el capítulo 3 describe la creación de toda la librería de este proyecto que debe usar la plataforma para cumplir con los objetivos (que la plataforma utilice todos los medios inalámbricos que dispone Android, la comunicación multisalto entre dispositivos alejados, etc.).

El capítulo 4 trata de cómo ha tenido que ser dividida la librería para que pudiera ser utilizada junto a SPRINGS para Android en una máquina virtual Java que no fuese Android. También describe los pasos que se han dado para que los agentes puedan ser geográficos.

Finalmente el capítulo 5 describe las conclusiones a las que se ha llegado con el trabajo, se indican nuevas vías de desarrollo para la plataforma que han quedado abiertas, se evalúa el estado del proyecto y se describe el tiempo dedicado al proyecto.

Capítulo 2

La plataforma de agentes móviles SPRINGS

SPRINGS [13] es una plataforma de agentes móviles creada por el grupo de investigación SID de la universidad de Zaragoza, centrándose en los problemas de escalabilidad y en el mantenimiento de la eficiencia de la plataforma durante la localización de agentes en escenarios dinámicos donde los agentes se encuentran moviéndose continuamente.

La plataforma está compuesta principalmente por:

Agentes: Son los programas que se ejecutan en la plataforma y los que deciden moverse o desplazarse de un contexto a otro.

Contextos: son las "zonas" donde se encuentran los agentes. Cada contexto tiene su nombre propio y tienen toda la información de los agentes que contienen.

Servidores de Nombres de Región o RNS: (Region Name Server) Son servidores que registran los contextos conectados en una misma zona. Todos los contextos de la misma zona se encuentran registrados en un único RNS. Mantienen también la información actualizada de los agentes de estos contextos que hayan hecho alguna operación de movimiento o de comunicación recientemente.

Esta plataforma utiliza RMI (java Remote Method Invocation), es un sistema que permite llamar a los métodos de las clases que se encuentran remotamente en otro ordenador, como base para la transmisión de agentes.

2.1. Análisis de la estructura básica de SPRINGS

El contenido del proyecto SPRINGS se encuentra distribuido en varios módulos o paquetes. Los que más destacan son:

agent: En este paquete se encuentran programadas las funciones que pueden realizar los agentes. Por ejemplo, moverse de un contexto a otro, programar tareas e incluso llamar a otros agentes. En este paquete se encuentra:

SpringAgent: Es la interfaz que incluye todos los métodos que tendrá un agente.

SpringAgent_RMIImpl: Es una implementación de la clase anterior utilizando el sistema RMI de Java.

context: En este paquete se encuentran descritos los Contextos. Destacan:

ContextAddress: Es una clase que representa la localización de un Contexto. Contiene la dirección, el puerto, el protocolo y el nombre del contexto.

Context: Es una interfaz que contiene todos los métodos de un contexto.

Context_RMIImpl: Es una implementación de la interfaz anterior utilizando el sistema RMI de Java.

ContextInterfaceForAgents: Es una interfaz diseñada para que los proxies de los agentes hagan llamadas a un determinado contexto.

ContextLauncher: Es una clase que ejecuta un contexto.

rns: En este paquete se encuentra todo lo relacionado con los Servidores de Nombres de Región (Region Name Server, RNS). Contiene:

RegionNameServer: Es una clase abstracta que representa a un servidor RNS. Contiene los contextos que se han registrado en él y alguna funcionalidad interna.

RegionNameServerInterface: Es una interfaz para describir los métodos que se podrán utilizar en el RNS.

RegionNameServer_RMIImpl: Es una clase que implementa la clase y la interfaz anteriores utilizando RMI como sistema de conexión.

RegionNameServerLauncher: Es una clase que ejecuta un RNS.

test: Aquí se encuentran ciertas clases con pruebas realizadas en la plataforma, agentes simples y pequeñas pruebas entre agentes.

util: En este paquete se encuentran una serie de funciones para simplificar la programación de la plataforma y herramientas para comprobar o analizar el funcionamiento de la misma.

2.2. Conociendo la plataforma

Para intentar ejecutar SPRINGS, primero hay que ejecutar un servidor RNS y luego ejecutar contextos que se conecten a ese servidor RNS.

Al leer el código fuente, se observa que hay muchos ejecutables para lanzar los servidores. Los más relevantes son RegionNameServerLauncher y ContextLauncher que se encuentran en los módulos rns y context respectivamente.

2.2.1. Ejecución de un servidor RNS

Para ejecutar el servidor RNS, lo más recomendable es ejecutar el script RNSLauncher. En el script hay varios valores que hay que modificar para que se ejecute con tu propio entorno personal:

classpath: En esta variable hay que especificar dónde está el archivo springs.jar.

path: Indica en qué lugar se encuentran los ejecutables de java. El valor introducido en este caso es: /usr/lib/jvm/jdk1.7.0/bin

instrucción de ejecución: Esta instrucción ejecuta el servidor finalmente.

Pero para poder ejecutarse, es necesario especificar un archivo con instrucciones o políticas de seguridad que se encuentra dentro del propio jar de springs.jar. Una vez extraído el archivo y guardado en un directorio conocido y del que se disponga permisos, ya se puede ejecutar SPRINGS para tener el servidor RNS.

La instrucción en este caso es:

"java -Djava.security.policy=./security.policy springs.rns.RegionNameServerLauncher $* "

Como se puede apreciar, en esta instrucción se indica el main que se va a ejecutar (springs.rns.RegionNameServerLauncher) y recoge todos los parámetros que nos pasen al llamar al script ($*).

Al ejecutar el script sin parámetros, obtenemos el siguiente mensaje:

"-pNN port"

que indica que hace falta especificar un puerto para ejecutar el servidor RNS.

Para terminar y tener finalmente el servidor ejecutando, se añade como parámetro el puerto deseado (en este caso 54321), por lo que la ejecución del servidor RNS a través del script RNSLauncher en este caso es:

"./RNSLauncher -p54321"

2.2.2. Ejecución de un contexto y conexión con el servidor RNS anterior

Para ejecutar este contexto es necesario utilizar el script ContextLauncher, que a su vez llamará al main de la clase springs.context.ContextLauncher.

Al igual que antes, se debe cambiar la variable classpath y la variable path por los valores anteriores.

El comando que se ejecuta es similar al del RNS salvo que esta vez se llama al main de ContextLauncher.

Tras ejecutar el script sin parámetros, éste devuelve como salida los parámetros que faltan. A saber:

-pNN: El puerto en el que va estar escuchando el contexto.

-n: El nombre que va a tener el contexto.

-r: La dirección en la que se encuentra el servidor RNS.

-l: (Opcional) El archivo de log para guardar los sucesos que ocurran durante la ejecución del contexto.

-cNN: (Opcional) El puerto del servidor de clases. Si se especifica, el contexto puede ejecutar Agentes aunque no tenga su código.

-s: (Opcional) Si el servidor de clases debe ser ejecutado.

Si se ejecuta el contexto sin algún parámetro, el propio programa avisa. Por ejemplo si se ejecuta el contexto introduciéndole sólo el puerto:

"./ContextLauncher -p23456"

SPRINGS nos dice: "Error: you must specify a name for the context!"

Finalmente para ejecutar el contexto C1 en el puerto 50001 y conectándolo al RNS anterior, se escribe:

"./ContextLauncher -p50001 -n C1 -r rmi://localhost:54321"

2.2.3. Ejecución de la plataforma con agentes

Una vez que ya tenemos en funcionamiento un RNS y un contexto, es la hora de lanzar un agente.

Para crear un agente es necesario crear una nueva clase que herede de la clase SpringsAgent_RMIImpl (sólo se encuentran implementados los agentes con RMI).

Cuando un agente es creado, se llama al método main() del propio agente para empezar su ejecución. Es en ese método donde se tiene que empezar a programar la tarea que realizará el agente.

Para permitir que un agente se mueva a otro contexto, se utiliza:

moveTo: Para moverse a un determinado contexto, que puede ser especificado con un ContextAddress o con el propio nombre del contexto pasado como String. Además se puede añadir la función a llamar una vez que el agente se haya movido y los parámetros para esa función en el caso de que los necesite.

moveToURL: Para moverse a un determinado contexto dada una dirección URL (que contendrá la dirección del contexto y el protocolo que va a utilizar o el puerto para conectarse). Al igual que antes, también se puede añadir la función que se desea llamar una vez el agente se haya movido y los parámetros para esa función (en el caso de que los necesite).

Para crear este agente, simplemente se hace que nada más ser creado muestre algo por pantalla en el método main(), y luego se mueva al contexto C1 con el método moveTo("C1", "end") ejecutando finalmente un método end() que muestre por pantalla que ha llegado al contexto C1 y que va a terminar su ejecución.

Una vez creada la clase con el código que ejecutará este agente, es necesario tener un contexto que ejecute el agente. Se crea una clase Test que en su ejecución main(String [] args) cree un contexto (el nombre es indiferente) que se conecte al RNS que he ejecutado antes y que cree el agente que se ha programado antes. Una vez el agente se haya creado, mostrará por la pantalla el mensaje de que ha sido creado y en el terminal del contexto C1 mostrará que va a terminar su ejecución.

Así, se comprueba que el agente se ha creado y se ha movido correctamente.

2.3. La plataforma SPRINGS en Android

Una vez probada la plataforma SPRINGS, es momento de estudiar el proyecto modificado de SPRINGS utilizando Android, también desarrollado por el grupo de investigación SID.

Este proyecto mantiene la esencia de SPRINGS, la misma estructura de clases desarrollada por el grupo SID, pero no utiliza RMI debido a que, en Android, esas librerías no se encuentran de forma nativa y son muy pesadas para incluirlas junto al proyecto de SPRINGS. Además, la máquina virtual de Android no es igual a la de Java puro de Oracle, por lo que podría haber problemas si se utiliza el código RMI original.

Así, en este proyecto, en sustitución de RMI, se encuentra la librería lipeRMI, que viene a ser una librería con la misma funcionalidad que RMI pero más ligera.

2.3.1. Estado de la plataforma

La parte interna de SRPINGS para Android no cambia prácticamente nada, exceptuando el cambio de llamadas de RMI a lipeRMI. El cambio más llamativo es el volcado de código que ha sido llevado de RegionNameServer a RegionNameServer_RMIImpl debido a problemas con lipeRMI que no podía encontrar los métodos de la clase RegionNameServer en la clase RegionNameServer_RMIImpl si no se encontraban en la clase de RegionNameServer_RMIImpl.

En lipeRMI las clases que más destacan son:

Server: Es un simple servidor que espera en un determinado puerto a que un cliente de lipeRMI se conecte.

Client: Es el cliente de lipeRMI que establece la conexión con el servidor de lipeRMI y genera un conjunto de conexiones locales para permitir llamar a distintos métodos de forma remota.

CallHandler: Es una clase que permite realizar llamadas remotas. Conoce la clase a la cual está conectada y también sus métodos.

2.3.2. Ejecución y problemas con la plataforma

Se intenta poner en marcha SPRINGS para Android creando un servidor RNS, un par de contextos y un agente que se mueva de un contexto a otro.

Problema: Permission.INTERNET Ha habido un problema que se produce justo cuando un contexto se intenta conectar al servidor RNS para registrarse. El problema viene con una excepción IOException que indica que no se ha podido establecer la conexión.

Solución Tras investigar el establecimiento de la conexión en Android, se encuentra en la documentación de Android que hay que añadir un permiso llamado INTERNET para poder abrir Sockets.

Una vez añadida la línea:

<uses-permission android:name="android.permission.INTERNET" />

en el archivo AndroidManifest.xml (es un archivo que describe cómo se encuentra definido un proyecto Android, en este caso SPRINGS para Android) se consigue que la plataforma pueda utilizar los Sockets.

Una vez añadido este permiso, la prueba ha sido un éxito.

Problema: StreamCorruptedException Una vez conseguido registrar un contexto en un RNS se detecta un nuevo problema: la comunicación entre estas dos clases.

La excepción viene desde una parte interna del código, la clase ConnectionHandler (que es la que se encarga de llamar a los métodos remotos, recibir llamadas de métodos y devolver resultados). La excepción en concreto es: StreamCorruptedException (que se produce al intentar leer un Objeto cuya información de descripción del objeto no se puede encontrar o está corrupta).

Solución Durante las pruebas de ejecución y depuración, se detectaron dos causas.

Primera: Intentar mezclar RMI de SPRINGS con lipeRMI de SPRINGS para Android. Tras indagar en ambos códigos confirmamos que definitivamente RMI no era compatible con lipeRMI y, por lo tanto, los agentes, contextos y servidores de nombres de regiones de RMI no iban a ser compatibles con sus respectivos de lipeRMI.

Segunda: Utilizar un emulador de Android tanto para crear contextos como RNS. Los emuladores de Android, aunque ejecutan el sistema operativo propio de Android, no establecen las conexiones con los módulos de conexión que utilizaría un dispositivo físico, sino que la establecen a través de un servicio NAT (Network Address Translation). Esto permite al emulador salir al exterior y conectarse con normalidad, pero no le permite actuar fácilmente como un servidor y, debido a esto, recibir conexiones nuevas impidiendo el normal funcionamiento de SPRINGS.

Para solucionar este último problema simplemente hay que utilizar dispositivos físicos (como los terminales o tablets) con Android.

Tras utilizar este tipo de dispositivos, el problema desapareció.

Capítulo 3

Abstracción de las Comunicaciones: Sockets Abstractos

Tras las dos pruebas de SPRINGS, se ha comprobado cómo funciona internamente SPRINGS y se procede a diseñar la nueva librería para la plataforma.

Se pretende conseguir que la plataforma trabaje con los distintos tipos de conexiones a la vez (WiFi, Bluetooth y 3G) y establezca conexiones entre los dispositivos móviles distantes de la red para que se puedan comunicar.

3.1. La librería Sockets abstractos

SPRINGS para Android permite ejecutar la plataforma en un dispositivo móvil (un terminal), así que ya es un entorno móvil (que es lo que se pretendía en un principio), pero el entorno en el que se ejecuta es solamente WiFi [16] [17] y los idspositivos deben estar en la misma red para que la plataforma se ejecute.

Para conseguir que el entorno se ejecute en diferentes tipos de red (Bluetooth, WiFi, redes móviles, etc) el sistema debe recibir, por cada conexión que realiza, diferentes tipos de conectores ("Sockets") en función del tipo de conexión que se realice.

Los tipos de conectores son únicos y esto obliga a los programadores a mantener el código con todos los tipos posibles de conectores. Esto es poco escalable y obliga a la plataforma a que compruebe constantemente los tipos de conectores durante la ejecución. Además, si apareciese un nuevo tipo de conexión, se tendría que modificar toda la plataforma para agregar este nuevo tipo.

En lugar de añadir y combinar todas las conexiones, la mejor opción es utilizar el patrón Adapter [18] para que, independientemente del tipo de conexión, la plataforma utilice un socket como hacía con el socket de java. Entonces todas las conexiones quedarán encapsuladas siguiendo un mismo patrón (que tenga la misma interfaz).

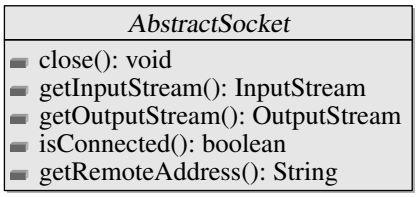

Figura 3.1: Interfaz AbstractSocket

3.1.1. Diseño de la librería

La clase base es una interfaz con las cabeceras que utiliza SPRINGS de java.net.socket. La interfaz se llamará AbstractSocket (ver fig: 3.1).

Una vez creada esta interfaz, se crearán los tipos de conexiones que utilizará la plataforma implementando la interfaz anterior (ver imagen: 3.2), en concreto:

EthernetSocket: que utiliza un host o IP y un puerto para la conexión TCP/IP.

BluetoothSocket: que utiliza el hardwareAddress y el Identificador único universal, UUID (Universal Unique Identifier) para establecer una conexión mediante Bluetooth.

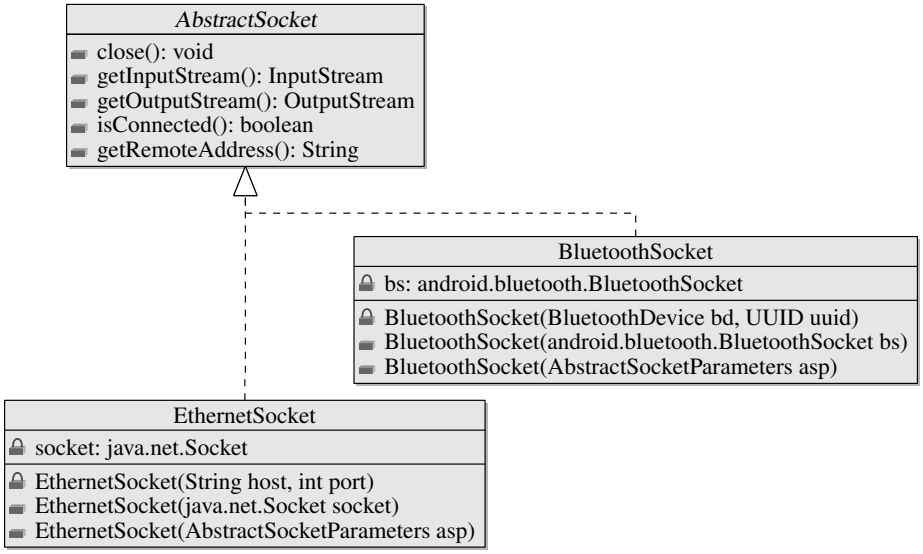

Figura 3.2: Clases EthernetSocket y BluetoothSocket que implementan la interfaz AbstractSocket.

Dentro de sus respectivas clases están los verdaderos sockets y las funciones de las clases llaman a las funciones del socketque contiene, utilizando para ello el patrón wrapper [19].

También se generará una interfaz para los servidores de Sockets con la misma interfaz necesaria de java.net.ServerSocket. La interfaz se denomina AbstractServerSocket (ver figura: 3.3).

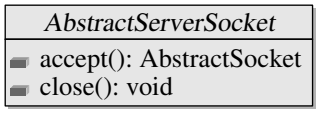

Figura 3.3: Interfaz AbstractServerSocket

Y, al igual que con los Socket, también se necesitan los respectivos servidores (ver fig. 3.4):

EthernetServerSocket: Representará un servidor que escuchará a través de un puerto.

BluetoothServerSocket: Representará a un servidor que escuchará en un determinado UUID.

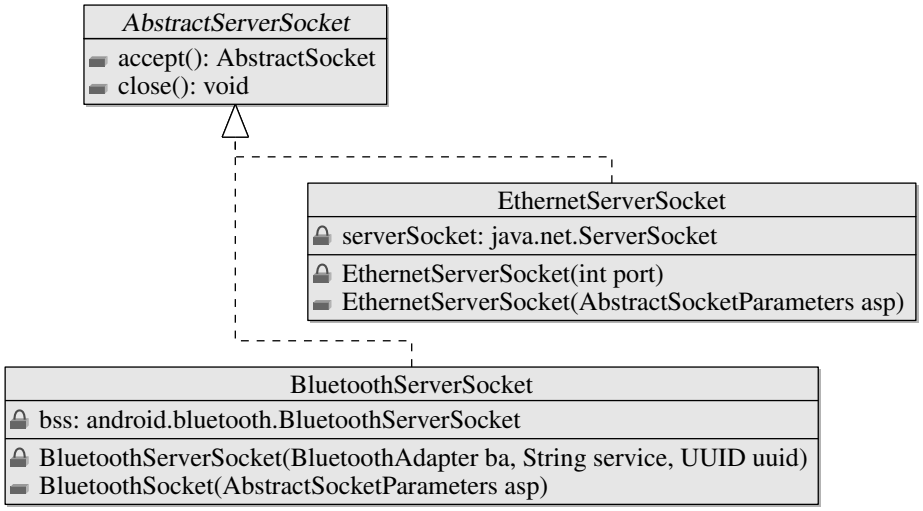

Figura 3.4: Clases EthernetServerSocket y BluetoothServerSocket que implementan la funcionalidad de AbstractServerSocket

Aunque aún hay problemas derivados de la plataforma:

Los parámetros necesarios tienen que llegar a su correspondiente Socket o Servidor y que se inicie de forma adecuada. Por ejemplo no podemos iniciar un servidor de Bluetooth que escuche en un puerto IP.

Por esto es necesario que, para crear tanto los sockets como los Servidores, reciban un parámetro común que luego ellos comprobarán si es correcto o no.

Para recibir un parámetro abstracto, es necesario crear una interfaz que represente eso mismo, parámetros abstractos (ver fig. 3.5):

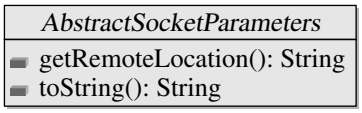

Figura 3.5: Interfaz AbstractSocketParameters

Ahora, cada tipo de conexión debe moldear una clase que extienda la funcionalidad de AbstractSocketParameters para obtener los parámetros deseados:

- En el caso de Ethernet, la clase se denomina EthernetSocketParameters que tiene un método estático para comprobar si los parámetros abstractos son del tipo de Ethernet y algunos métodos para obtener la IP o el puerto.
- En el caso de Bluetooth, la clase se denomina BluetoothParameters que, al igual que el anterior, tiene un método para comprobar si los parámetros abstractos son del tipo de Bluetooth y otros métodos para obtener, por ejemplo, la dirección hardware del Bluetooth y el UUID.

(Ver fig. 3.6)

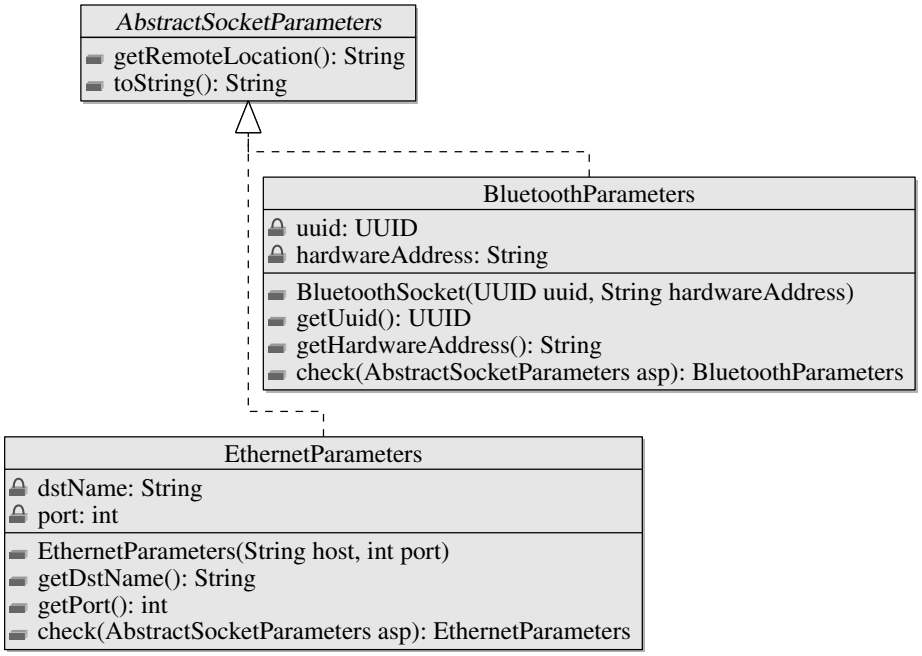

Figura 3.6: Clases EthernetParameters y BluetoothParameters que implementan la interfaz AbstractSocketParameters

3.1.2. Integración de la librería en SPRINGS

Para conseguir que SPRINGS para Android siguiera funcionando correctamente (sin cambiar su estructura) con los Sockets de Bluetooth y cualquier otro tipo de Socket (tanto los que ya existen, como los que pueden existir en un

futuro si se crea un nuevo tipo de conexión), es necesario modificar un poco la plataforma.

En realidad sólo hay que modificar todas las partes donde se usa un Socket en SPRINGS para Android (la gran mayoría están en LipeRMI) por un AbstractSocket y, a su vez, hay que cambiar los ServerSocket por AbstractServerSocket y los parámetros (en algunos puntos se usa un IP y un puerto, en otras solamente un puerto) por AbstractSocketParameters.

Además ya que se está modificando SPRINGS para Android, voy a aprovechar para poder:

Cerrar un Contexto y un Servidor de Nombres de Región (RNS) Para cerrar un contexto o un servidor lo único que hace falta es:

- Obtener el servidor de lipeRMI que se crea, en la función start() y guardarlo.
- crear un método close() o stop() en las clases donde se utilice este servidor de lipeRMI para que, cuando se vayan a cerrar los servidores, también llamen a la función close() del servidor lipeRMI.

Y también, en el contexto, es recomendable llamar a la función padre stop() para que termine la ejecución del objeto "_accessService", a demás de llamar al servidor RNS al que se encontraba conectado para indicarle que este contexto no va a estar más tiempo conectado.

Mantener la conexión Para mantener la conexión es necesario que el servidor RNS compruebe cada cierto tiempo si los contextos, a los que está conectado, siguen operativos. Esto puede resultar demasiada carga para un único servidor (el que tenga que recibir peticiones de todos los contextos y además comprobar si se encuentran conectados). Así que lo voy a hacer al contrario. Los contextos serán los que envíen una débil señal al servidor, cada cierto tiempo, para indicar que siguen operativos.

Cuando un servidor se cierre, intentará indicarle a su servidor RNS que se va a cerrar y es muy recomendable que lo haga sin ningún agente. Si contiene algún agente, el contexto debe avisar a los agentes que contiene indicando que el contexto en el que se encuentran se va a cerrar para que realicen la tarea que tengan programada, o terminen su ejecución.

Para esto he añadido en la interfaz RegionNameServerInterface una función para que un contexto pueda enviarle una señal al servidor RNS. Una vez reciba esa señal, lo que hará el servidor RNS es anotar el momento en que lo recibió.

Cuando el servidor RNS vaya a establecer una conexión con el contexto y no lo consiga, comprobará el último momento en que recibió la última señal del contexto. Si pasa un determinado tiempo desde la última señal recibida hasta el momento actual sin haber recibido ninguna señal, entonces el Servidor RNS decidirá que el contexto está caído y cerrará la conexión con tal contexto.

También se ha añadido un evento (onCloseContext()) a los agentes que se disparará cuando el contexto en el que se encuentren vaya a ser cerrado.

En ese evento es donde el agente debe decidir si debe moverse a otro contexto o terminar su ejecución.

3.2. Creando una red Mesh

Ya están diseñados los distintos tipos de conexión, pero sólo se pueden utilizar si el medio está al alcance, es decir, es como realizar una conexión a una máquina a la que te puedes comunicar.

En otras palabras, ahora el tipo de comunicación, que realizaría cada uno de los dispositivos de comunicación, sería directa, del terminal de origen al terminal de destino (P2P [20] [21] [22] [23]).

El problema viene cuando hay dispositivos a los que no te puedes conectar directamente debido a que no están al alcance (por ejemplo por Bluetooth, si los dos dispositivos están a más de 10 metros, cuando el alcance máximo del Bluetooth es de 10 metros) o que no tienen los mismos dispositivos para comunicarse (uno utiliza solamente WiFi y otro utiliza únicamente Bluetooth).

En estos casos es necesario utilizar un sistema que implemente una red Mesh [24] [25] y permita la comunicación entre varios dispositivos que no están conectados directamente.

Para comunicarse todos los dispositivos de la misma red Mesh, lo que debe hacer cada dispositivo es ir buscando periódicamente a quién se puede conectar. Esto lo realizará:

- Primero directamente a los dispositivos que tenga cerca.
- Luego les preguntará a esos dispositivos, cuales son los dispositivos a los que ellos se pueden conectar.

De esta manera el dispositivo podrá conocer todos los dispositivos de la red y a cuales se puede conectar indirectamente.

Además, mientras van buscando los dispositivos nuevos, también va actualizando la lista de dispositivos a la cual se puede conectar (los dispositivos que tiene cerca). De esta manera, si se aleja de un dispositivo cercano, tal vez pueda alcanzarlo indirectamente mediante otro dispositivo.

La idea principal es utilizar los dispositivos a los que sí se tiene conexión y utilizarlos como medio de comunicación para establecer un camino y, así, llegar al dispositivo destino. (Ver fig.3.7, el camino rojo que comunica dos nodos que no están conectados directamente).

Ninguna de las clases creadas con anterioridad, sirve para este propósito, ya que ambos tipos de conexiones se comunican directamente mediante su medio (WiFi o Bluetooth). Por lo que es necesario crear un sistema propio de comunicación que utilice las clases anteriores y permita conectar los dispositivos entre sí mediante las conexiones directas de los dispositivos cercanos.

3.2.1. Los nodos de la red: Node

Cada terminal Android es considerado como un nodo. Un nodo es cada uno de los puntos de conexión de la red por donde se establecen y se transmiten todas las comunicaciones de la red. De esta forma, cada vez que en la red se establezca una conexión, se hará de un nodo a otro y, si fuese necesario, involucrando a otros nodos para permitir la comunicación entre el nodo emisor y el nodo receptor.

Un nodo (Node) esta compuesto por un conjunto de interfaces de conexión denominadas Port (puerto). A través de cada nodo se permite la conexión (directa o indirectamente) a otro nodo (siempre que haya conexión) y se permite también recibir conexiones de otros nodos.

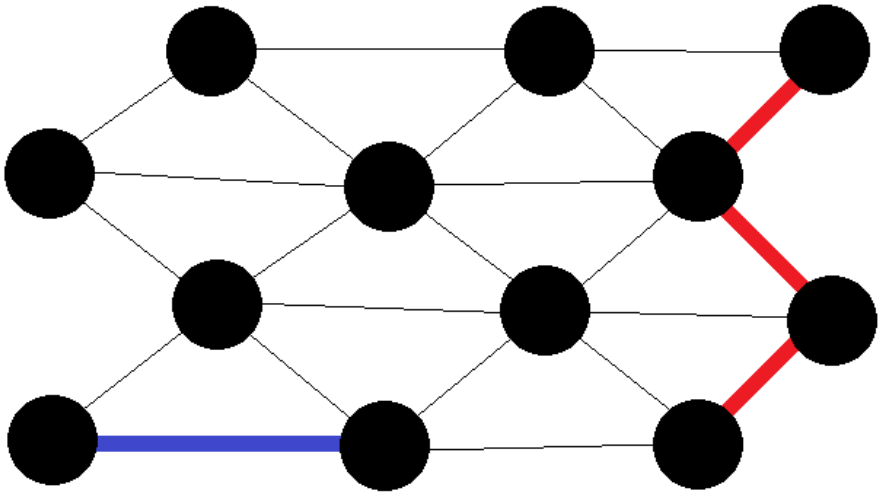

Figura 3.7: Ejemplo de una conexión directa (azul) e indirecta (rojo) entre nodos de una misma red.

Para saber cuál es el mejor camino hacia un nodo determinado, los nodos pueden preguntar a sus nodos adyacentes para saber a qué nodos puede llegar. Si recibe distintos caminos para llegar a un nodo, entonces se tendrá que decidir por el camino más óptimo.

El camino más óptimo se decide en función de algo llamado "coste", y de la cantidad de nodos que tiene que pasar la comunicación para poder conectarse con el nodo final.

Los costes son relativos y dependen de los nodos intermedios. Estos costes no hacen referencia a ningún coste económico, sino a un coste de esfuerzo del dispositivo por conectarse. El coste en cada nodo puede ser diferente y dependerá de su configuración. El concepto de coste se utiliza para poder tener un orden de preferencia de puertos (por ejemplo, si se prefiere las conexiones WiFi a las 3G, entonces el coste del Port WiFi será menor que el 3G, para que tenga preferencia).

Por definición, el coste para conectarse un nodo a sí mismo es 0.

Por ejemplo: Tenemos cuatro nodos A, B, C, D (Ver fig. 3.8).
Los cuatro se comunican de la siguiente forma: A-B-C-D:

- Coste de A:
  - Coste por WiFi: 1
- Coste de B:
  - Coste por WiFi: 1
  - Coste por Bluetooth: 7
- Coste de C:
  - Coste por Bluetooth: 11

- Coste por WiFi: 5

- Coste de D:
  - Coste por WiFi: 37

De A a B la comunicación es mediante WiFi.
De B a C la comunicación es mediante Bluetooth.
De C a D la comunicación es mediante WiFi.
Es decir: A <—(WiFi)—> B <—(Bluetooth)—> C <—(WiFi)—> D
Para llegar de A a D el coste es el siguiente:
Coste(A, WiFi) + Coste(B, Bluetooth) + Coste(C, WiFi) = 1 + 7 + 5 = 13
El coste que tendría A para conectarse a D es 13.

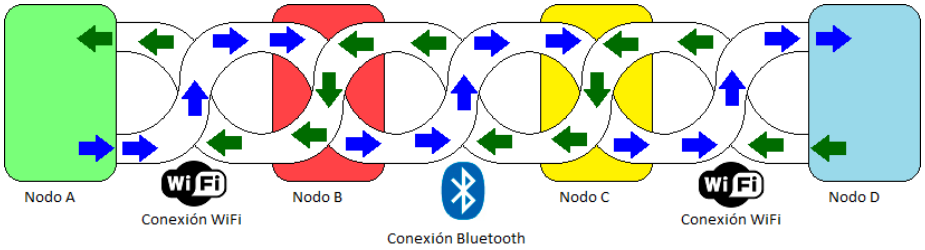

Figura 3.8: Ejemplo visual de la conexión entre A y D pasando por los nodos B y C.

Para mayor seguridad, cada nodo debe habilitar el uso de un determinado sistema de conexión. Por ejemplo, si en un nodo, aunque tenga habilitado el sistema WiFi y Bluetooth, sólo debe utilizar el sistema WiFi, entonces el nodo sólo activará el servicio WiFi. El servicio Bluetooth no lo activará y, por tanto, no será usado por la plataforma.

También se puede deshabilitar las conexiones que ya no se quieran utilizar. En estos casos es necesario reorganizar los nodos que hayan sido conocidos el nodo principal, ya que es posible que el mejor camino para alcanzar un nodo pasase por la conexión que se acaba de cerrar.

3.2.2. Los puertos de la red: Port

Un puerto es una interfaz que establece conexiones hacia otros nodos (siempre que estén a su alcance, por ejemplo si es un puerto Bluetooh y el nodo al que se tiene que conectar está a más de 10 metros, entonces no se establecerá la conexión directa con el nodo). Por ejemplo la tarjeta inalámbrica WiFi o Bluetooth. Aunque parezca que están relacionados con los tipos de conexiones anteriormente descritos, puede haber más de una interfaz con el mismo tipo de conexión. Por ejemplo: una tarjeta de red Ethernet y una tarjeta inalámbrica WiFi utilizan el mismo tipo de conexión (Ethernet).

Estos puertos deben permitir establecer directamente la conexión utilizando su medio y no se deben ver implicados directamente en el establecimiento de conexión entre nodos, sino entre dispositivos. Por ejemplo, un puerto WiFi sólo se debe encargar de conectar con otros dispositivos WiFis en función de los

parámetros que le pasen. Es el nodo el encargado de conectarse con otros nodos, tanto si la conexión es directa mediante el puerto WiFi o indirecta usando dicho puerto.

Para permitir recibir conexiones, un puerto que se encuentre habilitado, siempre tiene que tener un servidor que reciba conexiones a través de dicho puerto. Además, es recomendable que también tenga todos los parámetros posibles para poder establecer la conexión con otros nodos y poder pedir información sobre el estado de la red, establecer una conexión con su nodo o incluso pedirles que establezcan una conexión con un nodo lejano utilizándolo como puente.

Para mejorar la conectividad en redes de tipo Ethernet, donde los nodos se encuentren en diferentes redes conectados mediante Routers con NAT, Los puertos que utilicen conexiones Ethernet están obligados a indicar cuál es la IP que deben preguntar para conectarse a su propio nodo. Por ejemplo: si tenemos un nodo en Internet con una IP 9.10.11.12 y otro nodo en una Intranet con la IP 192.168.20.5, estos dos nodos no se podrán comunicar directamente debido a que hay un Router NAT pasarela que comunica ambas redes. Si el nodo de Internet intenta comunicarse a la IP del nodo de la Intranet, entonces la IP (192.168.20.5) del nodo de la Intranet será buscada en Internet y nunca se podrá establecer la conexión. Pero si el nodo de la Intranet indica que la IP para conectarse a él es la IP pública del Router NAT (la IP de Internet), entonces el nodo de Internet establecerá la conexión con el Router y luego será el Router el que redirecione al nodo de Intranet (utilizando la configuración NAT que tenga).

3.3. Diseño de la nueva red Mesh

En la sección anterior se han descrito los componentes que contiene la red Mesh.

En cambio, en esta sección se describen los componentes lógicos que forman la red para empezar a darle forma.

3.3.1. Conexiones virtuales

Tras describir el sistema, es necesario generar un nuevo tipo de conexión para permitir la conexión con dos nodos que no se pueden conectar directamente.

Este nuevo tipo de conexión se denomina Virtual (Ver fig. 3.9).

Una conexión virtual mantendrá una conexión de un nodo durante todo el tiempo que dure la comunicación, sea del tipo que sea, con otro nodo.

Para crear esta conexión es necesario indicar en los parámetros a qué nodo se quiere conectar y a qué servicio se debe preguntar. En el caso de que los parámetros sean para realizar otro tipo de conexión (Ethernet o Bluetooth), este enlace debe permitir la creación y gestión de esos tipos de enlace, independientemente del puerto, en función de la configuración del sistema (en este caso Android).

Así que los parámetros virtuales se deben poder crear a través de un nombre de un nodo y servicio o con otros parámetros abstractos.

Para los SocketsVirtuales el planteamiento es el mismo, los sockets virtuales tienen un socket abstracto en su interior y lo único que deben hacer es rellamar al socket que contenga (ver fig. 3.10).

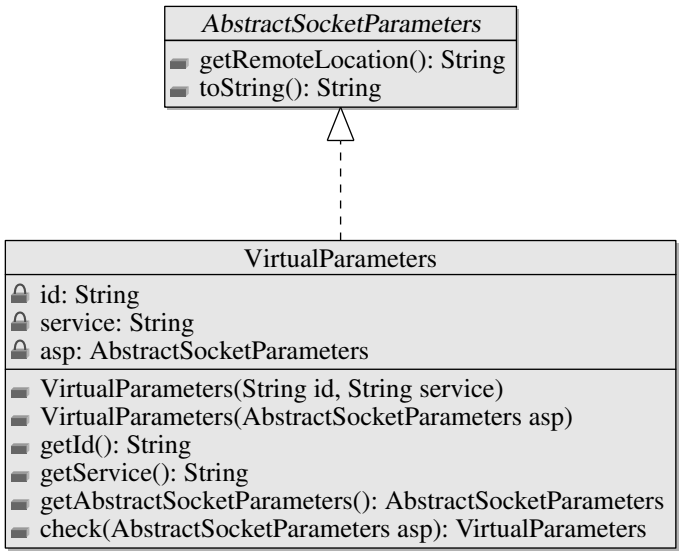

Figura 3.9: Clase VirtualSocketParameters, que implementa la interfaz AbstractSocketParameters

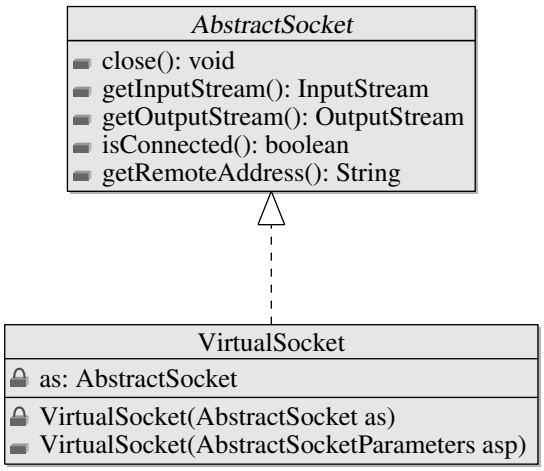

Figura 3.10: Clase VirtualSocket, que implementa la interfaz AbstractSocket

En el caso de los servidores virtuales el planteamiento es un poco distinto, ya que puede que quiera recibir conexiones de diferentes nodos, así que lo mejor es que puedan permitir tener varios servidores abstractos bajo el mismo servidor virtual y cada vez que reciba una nueva conexión, sea del tipo que sea, enviarla a través del método accept() (ver fig. 3.11).

Al final la situación para establecer una conexión es la misma. Siempre hay que identificar el "lugar" al que te quieres conectar y la "puerta" por la que quieres pasar.

Esto se ve claramente con Ethernet, Bluetooth o incluso con el sistema de conexiones virtuales.

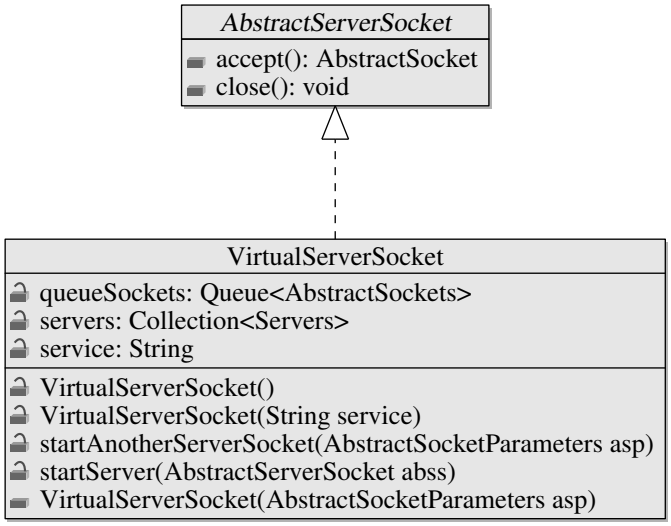

Figura 3.11: Clase VirtualServerSocket, que implementa la interfaz AbstractServerSocket

	Lugar	**Puerta**
Ethernet	Host (IP)	Puerto
Bluetooth	HardwareAddress	UUID
Virtual	Node	Service

3.3.2. El Protocolo de comunicación

Ahora que ya se ha descrito qué es lo que va a haber en la red (nodos) y el sistema de conexión (puertos y sockets abstractos), es el momento de empezar a describir cómo se van a comunicar estos nodos, cómo se van a descubrir y cómo van a establecer la conexión con un nodo lejano.

Para establecer la comunicación simplemente un nodo debe establecer la conexión con otro nodo que se encuentre escuchando, utilizando el mismo sistema (WiFi o Bluetooth, por ejemplo).

Para descubrir y conectarse con un nodo lejano, primero hay que conectarse a un nodo cercano y luego es necesario comunicarse a través de un protocolo.

Una vez establecida la conexión con un nodo cercano en un puerto que sepamos que está escuchando siempre es necesario:

1. El emisor debe que enviar primero, en formato String UTF-8 (Unicode Transformation Format, 8-bit), el tipo de acción que quiere realizar con esta conexión. Si es CONNECT_TO_NODE entonces es que se quiere conectar a un nodo. Si es GET_NODES entonces es que quiere obtener información de todos los nodos a los que puede llegar el receptor en función de su configuración.

2. Opción CONNECT_TO_NODE (Ver fig. 3.12)

   a) Si el emisor se quiere conectar a un nodo, entonces el emisor tendrá que enviar, en formato String UTF-8, el nombre del nodo al que se quiere conectar.

b) El receptor comprobará si es él el nodo que se está buscando, si es él entonces enviará un NODE_FOUND al emisor y utilizará la conexión para comunicarse con el nodo emisor. Si no es él entonces intentará establecer una conexión con el nodo que se está buscando. Si consigue establecer la conexión entonces enviará al emisor NODE_FOUND y este nodo receptor intermedio empezará a actuar como "puente". Si no lo consigue, entonces enviará un NODE_NOT_FOUND y cerrará la conexión.

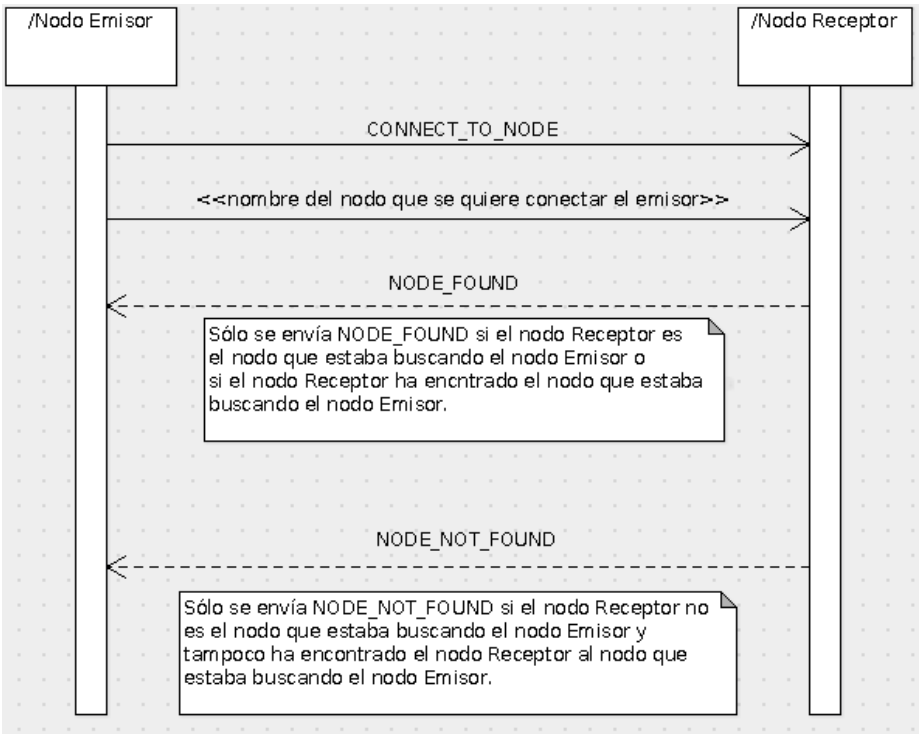

Figura 3.12: Diagrama de secuencia del protocolo que muestra cómo se debe proceder en el caso de que un nodo emisor se intente comunicar con otro nodo.

3. Opción GET_NODES (Ver fig. 3.13)

   a) Si el emisor quiere obtener información de los nodos a los que se puede conectar el receptor, entonces el receptor enviará al emisor toda la información de los nodos a los que se puede conectar más uno, la información del propio receptor. La forma de enviar la información de cada nodo es la siguiente:

      1) Primero se envía el nombre del nodo, en formato UTF-8.
      2) Luego se envía la configuración de los parámetros para poder conectarse a ese nodo. Estos parámetros deben ser del mismo tipo que la conexión que se esté utilizando aunque se almacenen como AbstractSocketParameters. Por ejemplo si la conexión se está realizando por Bluetooh, entonces los parámetros recibidos

serán del tipo BluetoohParameters, pero se almmacenarán en el nodo como AbstractSocketParameters.

3) A continuación se envia el coste para conectarse al nodo.

4) Finalmente la cantidad de saltos o nodos que hay intermedios para llegar al nodo destino.

b) Una vez enviada toda la información de todos los nodos que conoce el receptor, el propio receptor indicará con un valor booleano si quiere la información para conectarse con el emisor.

1) Si el receptor ha enviado un true preguntando por la configuración del emisor entonces el emisor comprobará si puede enviar información. Si puede enviarla le enviará un valor booleano true y empezará a enviarle los parámetros al receptor (de la misma forma que el receptor le había enviado todos los parámetros de todos los nodos que conocía). En el caso contrario le enviará un valor booleano false y cerrará la conexión.

2) Si el receptor ha enviado un false para no preguntar por los parámetros, entonces se cerrará la conexión.

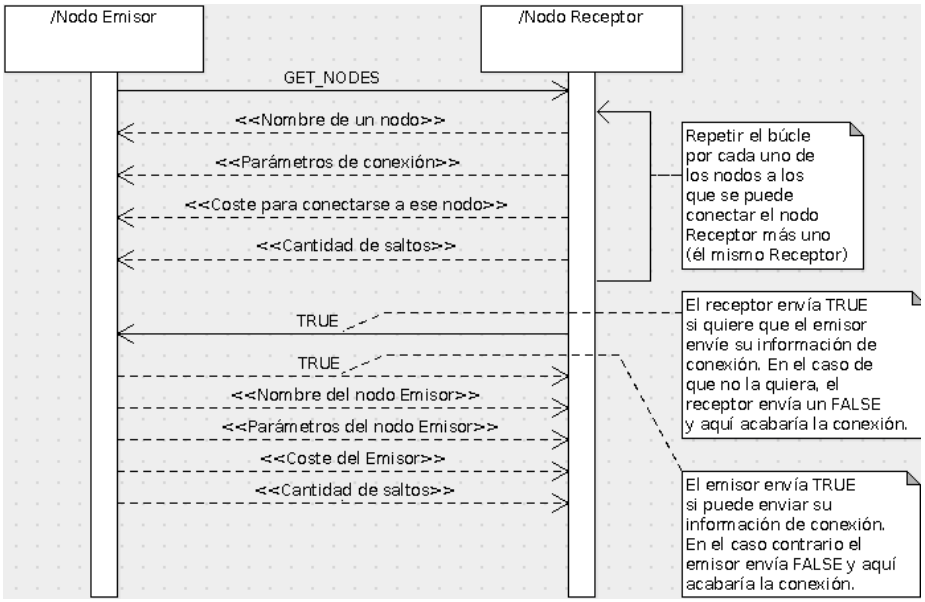

Figura 3.13: Diagrama de secuencia del protocolo en el caso de solicitar la información de todos los nodos a los que se puede conectar.

Todos los envíos de alguna cadena de tipo String se enviarán en formato UTF-8.

También se puede conectar a un determinado dispositivo que tenga un nodo sin querer comunicarse con el nodo, simplemente utilizando otro servicio que no utilice el nodo. Por ejemplo si el nodo escucha con el puerto 9000 del sistema inalámbrico WiFi, cualquier otra aplicación se puede conectar a ese dispositivo con el sistema inalámbrico WiFI utilizando un puerto distinto, y de esta forma no se tiene que acceder al nodo obligatoriamente.

3.3.3. La conexión puente

Una vez establecida la conexión, los nodos intermedios deben enviar todo lo que reciban para poder permitir que dos nodos que no se podían comunicar directamente en un principio, ahora sí que lo pueden hacer.

Para establecer la conexión, los nodos utilizan la parte del protocolo de conexión entre nodos(ver fig. 3.14).

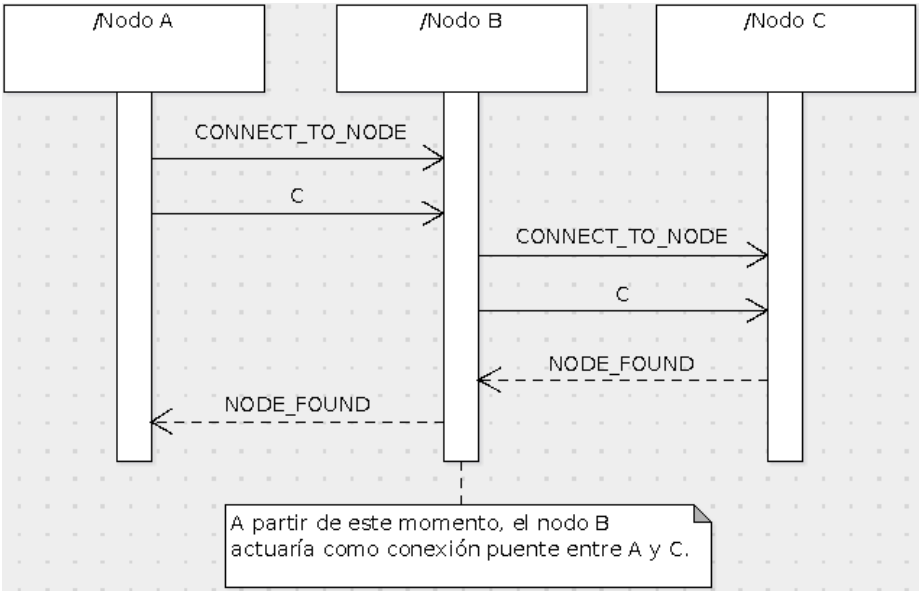

Figura 3.14: Ejemplo del uso del protocolo para establecer una conexión entre A y C usando B como medio. Al final, una vez establecida la conexión, A y C utilizarán a B como conexión puente.

Para establecer la conexión es necesario que, una vez se tenga claro que un nodo va a actuar como puente, entonces debe recoger los dos sockets que mantienen la conexión y enviar la salida de uno a la entrada del otro y viceversa (ver fig. 3.15).

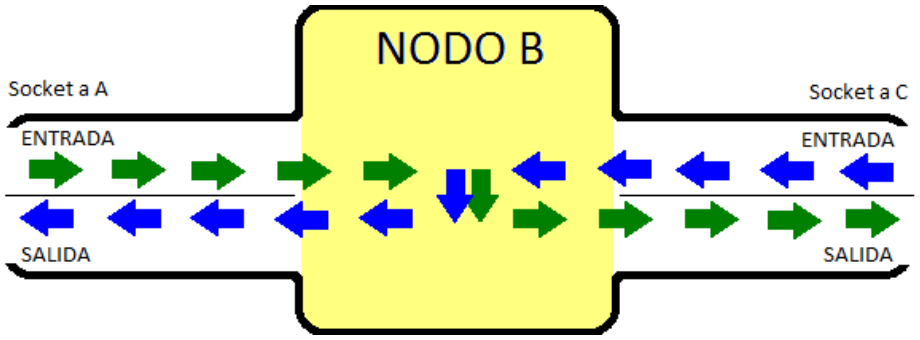

Figura 3.15: Ejemplo visual con los dos sockets donde el nodo B actúa como puente, basado en el ejemplo anterior.

3.4. Implementación de la nueva red Mesh

En la sección anterior se han descrito y diseñado a nivel lógico los componentes de la nueva red.

En esta sección se describe el diseño e implementación de los distintos gestores y clases principales que tiene la red Mesh para la comunicación indirecta entre nodos.

Para que la nueva red funcione, es necesario que sólo se permita un nodo por dispositivo. Para conseguir este objetivo se utiliza el patrón Singleton [26]. La clase que gestione cada nodo de cada dispositivo se denomina NodeManager (Gestor del nodo).

3.4.1. El gestor del nodo: NodeManager

Un NodeManager debe saber en cada momento el camino más rápido para llegar a los nodos más cercanos que tenga (tras realizar al menos una búsqueda), comprobar cada cierto tiempo si sus puertos están operativos y tener la capacidad de conectarse a cualquier nodo, incluso si no los conoce.

Para realizar una búsqueda, el NodeManager debe estar configurado con varios parámetros que le indican qué nodos debe conocer. Uno de ellos es la cantidad de saltos que debe dar para llegar al nodo destino. Si la cantidad de saltos es mayor, entonces el NodeManager no debe conocer el nodo destino ni los caminos para llegar a él, aunque se podrá seguir conectando al nodo destino.

El otro parámetro es el tiempo que debe pasar para cada consulta. Cada cierto tiempo el NodeManager realizará una nueva búsqueda actualizando los nodos a los que puede acceder, los costes que han podido variar y los saltos necesarios para llegar a los nodos.

Los puertos también se comprobarán cada cierto tiempo y, si algún puerto se encuentra deshabilitado en algún momento, entonces se debe cambiar el acceso que se realizaba antes por ese puerto, a otro puerto distinto. Si no hay otro puerto para acceder, entonces ya no se podrá acceder a ese nodo. Si, pasado cierto tiempo, el puerto volviera a estar operativo, entonces volverá a ser habilitado automáticamente a no ser que se haya especificado explícitamente que debe quedarse cerrado.

Para conectarse a cualquier nodo, simplemente el NodeManager debe buscar en los nodos que conoce a ver si se encuentra en esa lista. Si no se encuentra entonces debe utilizar a los nodos que conoce más cercanos y les debe preguntar si pueden establecer una conexión al nodo deseado. Si alguno lo consigue, entonces se utiliza la conexión del nodo al que se preguntó como conexión puente y ya se tendrá la conexión con el nodo deseado. Si ninguno lo consigue entonces se avisa al nodo que realizó la búsqueda que no se ha podido establecer la conexión.

También deben permitir cerrar manualmente un determinado puerto, al igual que abrirlo.

Y finalmente permitir establecer, de forma externa, el orden de preferencia de los puertos.

3.4.2. Implementación del puerto: Port

Los propios puertos (Port) almacenan los resultados de sus últimas búsquedas de nodos para que, en el caso que sea necesario, volver a consultar los

resultados para, por ejemplo, obtener acceso a un nodo que ya no se pueda acceder a él mediante otro puerto. También establecen conexiones con otros puertos de su mismo tipo (WiFi con WiFi, por ejemplo) de otros nodos, pero sin utilizar el protocolo. Para el protocolo y la comunicación con otros nodos ya se encargará el NodeManager del nodo.

Todos los puertos estan identificados para poder acceder a ellos de una forma más rápida y, si fuese necesario, para establecer el orden de los puertos (ver fig. 3.16).

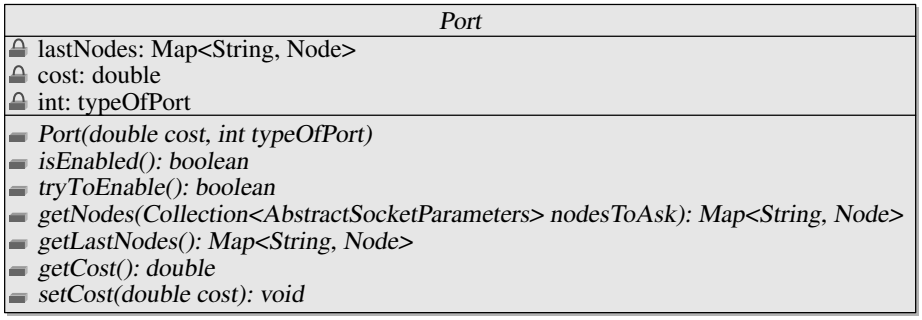

Figura 3.16: Clase Port

3.4.3. Nodo genérico: Node

Un nodo, sin más, en el sistema (clase Node) es la forma de representar cualquier nodo al que podamos llegar desde un NodeManager (es decir, cualquier dispositivo) y está descrito por un Id que identifica el nombre del nodo, tiene una referencia al puerto por el que se accede a él, contiene los parámetros para llegar a ese nodo, y almacena la cantidad de saltos y el coste que cuesta llegar al nodo destino (ver fig. 3.17).

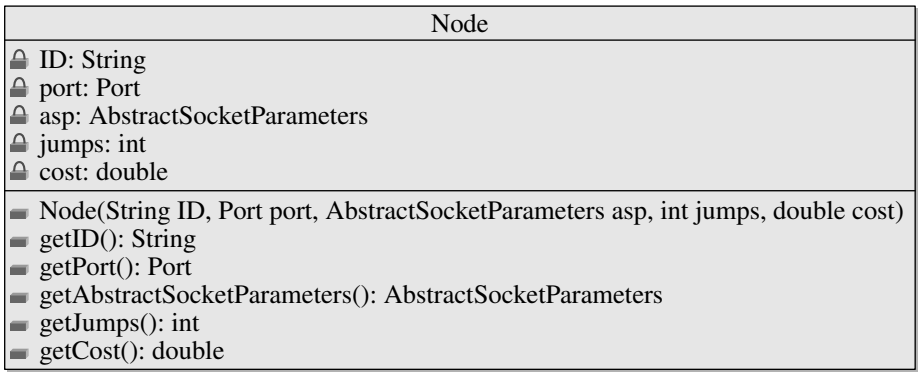

Figura 3.17: Clase Node

Tanto para comprobar los puertos como para realizar búsquedas de nodos es necesario un hilo que realice esta tarea periódicamente. Al hilo que realiza esta tarea se denomina PortsChecker (Comprobador de puertos).

3.4.4. El comprobador de puertos y nodos: PortsChecker

La clase PortsChecker aligera la carga que tiene del NodeManager al comprobar los puertos y buscar nuevos nodos (ver fig. 3.18).

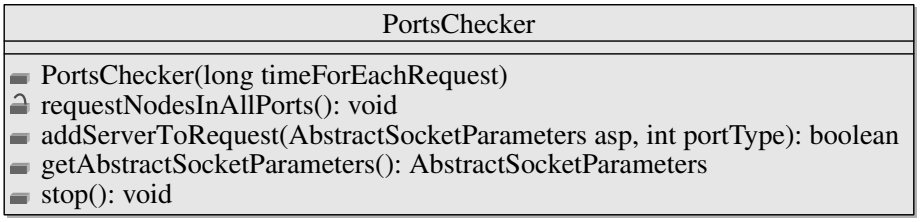

Figura 3.18: Clase PortsChecker

De esta forma el NodeManager es la principal fuente para conectarse al resto de los nodos, establecer el orden de prioridad de los puertos, y para habilitar y deshabilitar puertos (ver fig. 3.19).

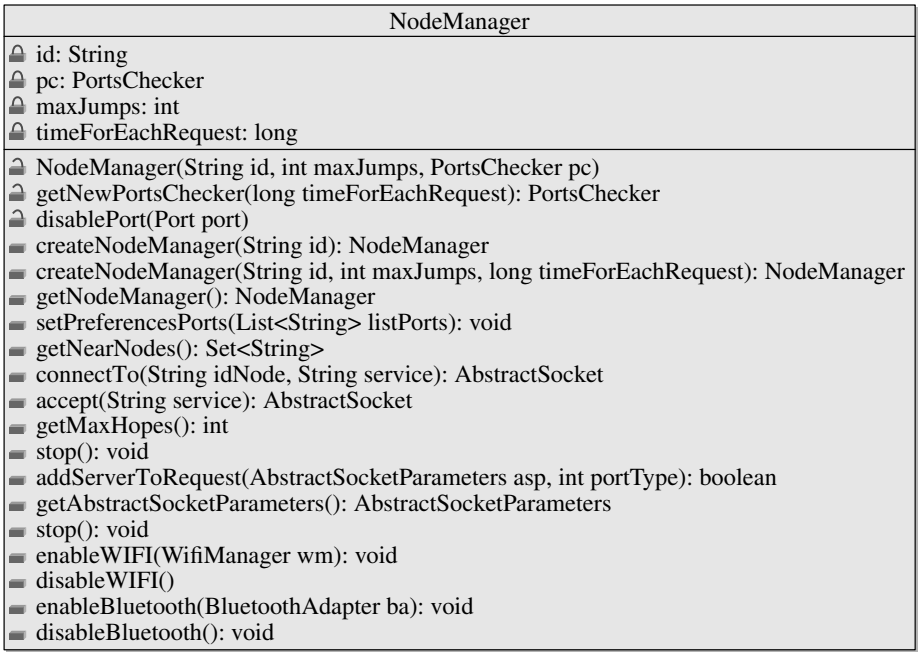

Figura 3.19: Clase NodeManager

Capítulo 4

Ampliación de Funcionalidades

A continuación se presentan todos los cambios que han sido necesarios en las librerías, tanto en SPRINGS para Android como en la recién creada Sockets Abstractos, para que se ejecuten en plataformas Java que no sean Android y, así, permitir tener más tipos de dispositivos en la red y en la plataforma aunque no sean del todo móviles. Además se ha añadido un nuevo tipo de agente.

4.1. Comunicación Android-Java

Una de las metas que se desea alcanzar con la plataforma de agentes móviles es que los terminales Android puedan enviar agentes a equipos no móviles (como por ejemplo ordenadores).

Esto no era posible debido a que SPRINGS para Android no es compatible con el SPRINGS original ya que utiliza LipeRMI y la comunicación entre RMI y LipeRMI no es compatible.

Para permitir que SPRINGS para Android se pueda ejecutar en un equipo con Java pero sin Android simplemente es necesario quitar las líneas de código que hacen referencia a la depuración de Android (funciones: "Log.d()").

En este punto ya se puede ejecutar SPRINGS con LipeRMI en Java, pero es necesario modificar la librería Sockets abstractos ya que los puertos y las conexiones Bluetooth hacen referencia a la librería de Android "android.bluetooth.*" y no a la de Java "javax.bluetooth". Así que la librería de sockets abstractos queda dividida en dos mitades. Una, la más básica, es la que tiene el código java que se pueda utilizar tanto en Android como en equipos virtuales Java y la otra la que contiene el código que sólo se puede ejecutar en Android.

Hay ciertas funciones, especialmente en la clase java.net.NetworkInterface que aparecen en Android 2.3.1 (Gingerbread), pero esa API se encuentra desaconsejada, así que la plataforma de SPRINGS para Android tendrá que ser ejecutada a partir de Android 2.3.3.

El código que se mantiene en la parte Java corresponde a las clases pertenecientes a EthernetSocket y VirtualSocket, además del código generado por los puertos genéricos y los nodos.

Desgraciadamente, el código de Bluetooth hay que dejarlo en Android ya que no pertenece a la librería original de Java, sino que se encuentra en el propio código de Android.

Para poder añadir Bluetooth a la librería de Java es necesario generar una nueva librería que no podrá usar ningún dispositivo Android, pero sí cualquier máquina física. El Bluetooth en Java se encuentra en el paquete "javax.bluetooth.*".

Entonces, la nueva organización de las librerías es la siguiente: (ver fig. 4.1):

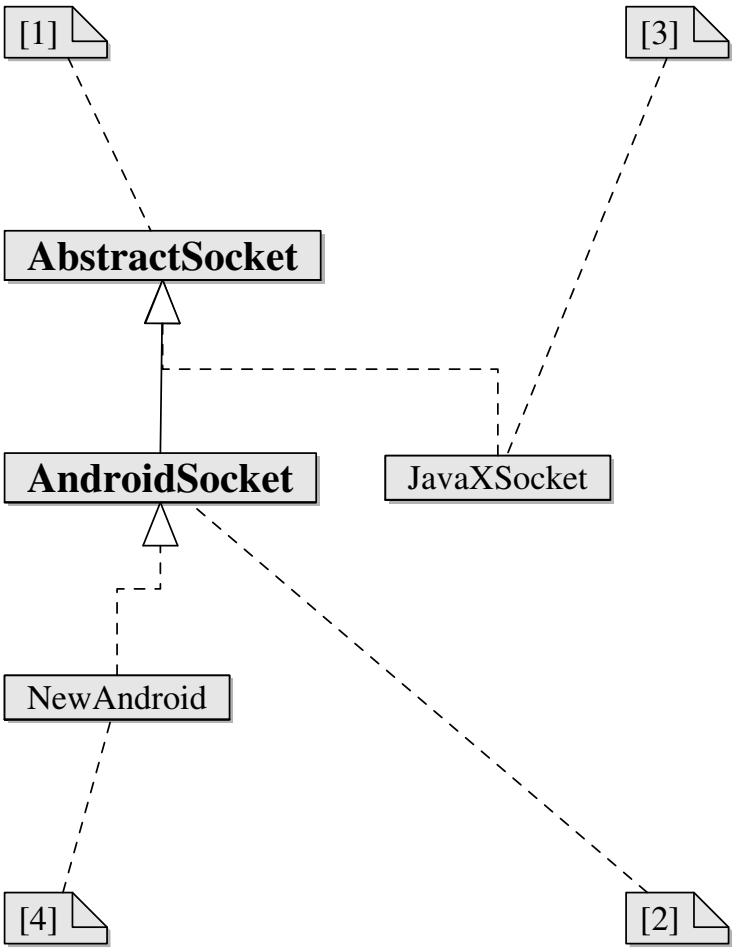

Figura 4.1: Esquema de las librerías

Descripción de la nota:

1: AbstractSocket utiliza únicamente los paquetes java.*.

2: AndroidSocket utiliza la librería de Android 2.3.3 y, a su vez, la librería AbstractSocket.

3: Ejemplo de una supuesta librería que utilizaría javax.* (u otras) para tener nuevas funcionalidades en las máquinas Java, de entre ellas, Bluetooth.

4: Ejemplo de una nueva librería que utilizaría los nuevos tipos de conexiones que en un futuro puedieran añadir en Android.

4.2. Añadiendo agentes geográficos

Los agentes no sólo tienen que ser móviles en un entorno de redes, donde impera la conectividad entre dispositivos, sino que también pueden ser móviles en el sentido geográfico [27] [28] [29], es decir, que se puedan mover a una determinada coordenada geográfica (latitud y longitud) del planeta.

Para permitir que los agentes puedan desplazarse en un contexto geográfico, es necesario indicar en qué posición geográfica se encuentran.

Pero no son los agentes los que realmente se encuentran en un determinado lugar, sino los contextos, que son los que contienen los agentes, los que realmente se pueden desplazar físicamente por el mundo si se están ejecutando en un sistema móvil, como un terminal o un portátil.

Para determinar un lugar físico en la tierra es necesario indicar su latitud y su longitud, es decir, un punto.

4.2.1. Diseñando los agentes geográficos

Para permitir que los agentes se puedan desplazar y buscar un lugar determinado es necesario implementar un sistema que permita a los agentes decir dónde quieren ir o a qué zona quieren viajar.

Esta pequeña API forma parte de SPRINGS para Android y es una manera, más o menos abstracta, de referirse a un lugar, ya que la posición geográfica se puede obtener de diferentes formas. Por ejemplo, en Android para obtener la posición se utiliza las funciones del GPS (Global Positioning System), pero es posible que en otros dispositivos, como un portátil, no tengan el mismo sistema de geolocalización y obtengan la posición de una manera distinta.

Un lugar determinado, queda descrito como un punto, con su latitud y su longitud (ver fig. 4.2).

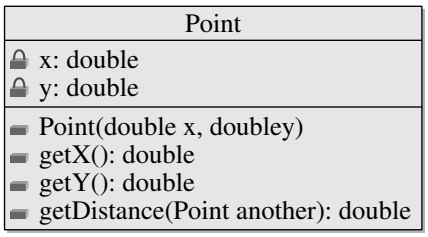

Figura 4.2: Clase Point

Para describir una zona, hay que describir un conjunto de superficies.

Una superficie es un área, y estas áreas se describen en la librería de forma abstracta (ver fig. 4.3). Lo más importante de estas áreas es saber si un punto se encuentra en un área o no:

Para no diseñar todo tipo de áreas, sólo se diseñarán dos tipos (el resto se tendrán que diseñar en un futuro si fuesen necesarias): (ver fig. 4.4):

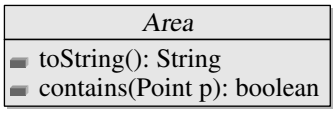

Figura 4.3: Clase abstracta Area

Triangulares : Representa un área triangular determinada o por tres puntos o por tres rectas. Además es necesario un cuarto punto que pertenezca al área para no confundir con el área complementaria.

Circulares : Representa un área circular determinada por un punto y el radio de alcance.

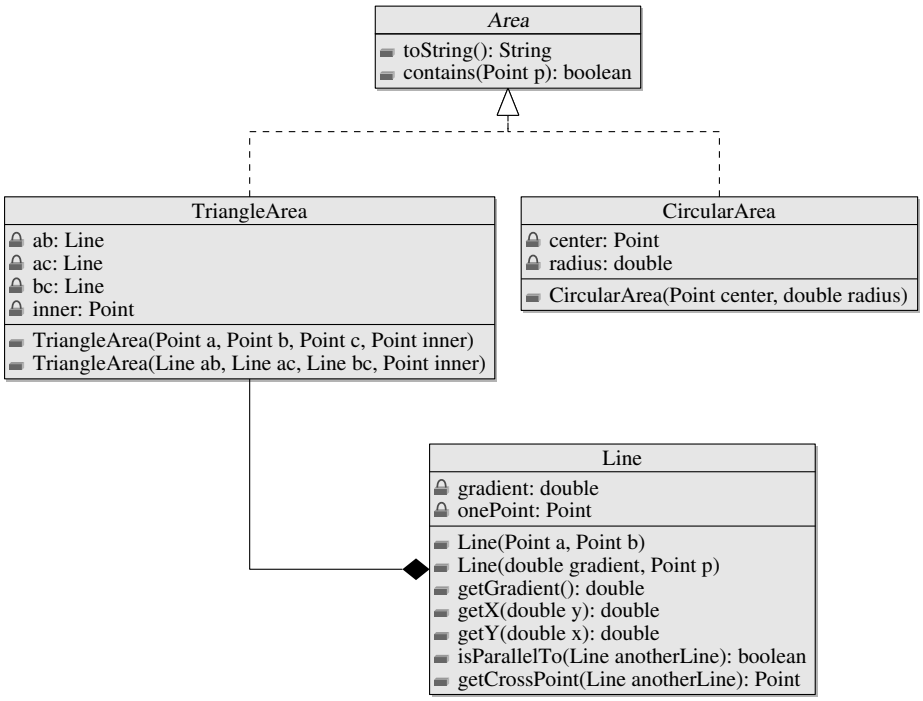

Figura 4.4: Clases TriangleArea y CircularArea que extienden la clase Area. (La clase Line simplemente es utilizada por TriangleArea para facilitar la taera de tratamiento de este tipo de area).

Todas las áreas poligonales pueden ser representadas como un conjunto de áreas triangulares. El resto de las áreas que tengan una curvatura (ovaladas, elípticas, etc.) tendrán que ser implementadas cuando sean necesarias.

4.2.2. Añadiendo la funcionalidad

Tras el diseño y la implementación de las clases anteriores que permiten determinar una zona del planeta, hay que modificar la versión de SPRINGS para Android para que los agentes puedan acceder a esta nueva funcionalidad.

En los agentes de SPRINGS se añaden dos nuevas funciones para que puedan moverse a un contexto que se encuentre en un punto en concreto o en una zona. Una es moveTo(Point) y la otra es moveTo(Zone).

En los contextos se añade una variable para que guarden la última posición en la que se encuentran (puede ser nula si el contexto no sabe dónde se encuentra, bien porque el sistema de localización que utilice no funciona o porque no tiene ningún sistema de geolocalización). También tienen un evento que actualiza la última posición geográfica del contexto.

Por último, el servidor RNS tiene su interfaz, RegionNameServerInterface, modificada para poder actualizar la posición de un contexto y para buscar cuál es el contexto más cercano a un punto o que se encuentre en una determinada área.

Capítulo 5

Conclusiones

Con este trabajo, se ha realizado un primer paso para la extensión de la plataforma de agentes móviles con funcionalidades de interés para entornos inalámbricos, manteniendo la estructura original de la plataforma, lo cual garantiza una mejor compatibilidad.

Los objetivos cumplidos han sido:

- El desarrollo de nuevas funcionalidades que permiten utilizar de forma más adecuada los distintos entornos inalámbricos que puedan estar disponibles (Wifi, 3G, Bluetooth) en la plataforma de SPRINGS para Android. Todos los sistemas tienen como punto en común, a nivel bajo de código, que utilizan sockets. Las conexiones Ethernet utilizan "java.net.socket" y las conexiones Bluetooth utilizan "javax.bluetooth.*" en Java o "android.bluetooth.*" en Android. El esquema de ambas clases socket es muy similar (primero establecen la conexión y luego sacan un InputStream y un OutputStream para el envío de datos). En este proyecto se ha aprovechado esta característica de que el esquema de los sockets sea muy similar para poder crear una clase abstracta Socket (AbstractSocket) y unas envolturas (EthernetSocket y BluetoothSocket) para poder utilizar ambos sockets iniciales de una manera cómoda y fácil.

- Crear y utilizar una red para la comunicación entre los contextos y servidores de nombre de región de la plataforma. Esta red ha es el punto principal de todo el proyecto y permite la conexión, tanto directa como indirecta, de todos los dispositivos que formen la red, la gestión de las comunicaciones, etcétera, independientemente de su sistema de comunicación.

- SPRINGS en Android puede ejecutarse en equipos que no sean Android siempre que tengan una máquina virtual Java. Este objetivo fue añadido durante el desarrollo del proyecto, pero aún así, se llevó a cabo sin muchas dificultades. Este punto es un poco especial porque es la primera vez que, en particular, se diseña una librería que utilice otra librería creada personalmente.

- La plataforma ahora puede utilizar agentes geográficos. Sólo se ha implementado una manera de describir lugares para que los agentes puedan

saltar a esos espacios geográficos, pero es posible que el grupo de trabajo de la universidad de Zaragoza SID siga desarrollando estos agentes y dotándolos de nuevas funcionalidades.

En cuanto a lo personal, principalmente he aprendido mucha ingeniería inversa al investigar y poner en marcha a SPRINGS, la plataforma de agentes móviles, tanto el original, como el port en Android. También he aprendido bastante sobre Android, sus permisos internos y el funcionamiento de este sistema, especialmente en el desarrollo de esta plataforma.

La gestión del proyecto ha sido desproporcionada, sobretodo al principio, ya que he perdido mucho tiempo con los problemas que he tenido poniendo en marcha la plataforma SPRINGS.

Para finalizar, tengo que destacar que el diseño se realiza mejor en grupo, ya que cada persona aporta su punto de vista sobre una idea y añade o mejora otras ideas sobre un tema. Quiero destacar esto porque como he estado yo sólo, he tenido que realizar el diseño de los módulos en solitario. Pero en ciertas ocasiones les he explicado a mis compañeros cómo iba avanzando mi proyecto y me han sugerido nuevas ideas o me han aportado una nueva información que, si hubiésemos estado en un equipo desde el principio, se hubiera tenido en cuenta desde el principio. Por ejemplo las conexiones virtuales, que hacen que el programador se despreocupe de la conexión real que se está estableciendo.

5.1. Trabajos futuros

Una vez finalizada la integración de los entornos inalámbricos de Android en la plataforma SPRINGS, quedan por mencionar unos cuantos problemas que surgen con los entornos inalámbricos. En concreto:

Desconexión del nodo RNS: Si, durante la conexión, el servidor RNS pierde la conexión (por ejemplo está demasiado alejado del punto de conexión) la plataforma de agentes fallará ya que es la pieza clave.

Posible solución: Una posible solución sería distribuir el propio servidor para asegurar que, al menos, la mayoría de las funcionalidades no se pierdan y, cuando todos los RNS estén otra vez conectados, se actualice el servidor por completo con los datos nuevos.

Gestión de las conexiones: Que la plataforma tenga en cuenta las nuevas conexiones, incluso mientras se comunican los nodos.

- Por ejemplo: Si dos nodos establecen una conexión indirecta y, al cabo de un tiempo, están lo suficientemente cerca como para poder establecer la conexión directa, entonces no hace falta que cierren la primera comunicación para abrir la segunda, sino que sea la plataforma que lo haga automáticamente.
- También es útil si la línea por la que se comunican dos nodos cae. Si llegase a ocurrir, entonces se encargaría la plataforma de buscar una ruta alternativa automáticamente, sin que los nodos de la aplicación tengan que realizar ningún esfuerzo.

5.2. Evaluación del proyecto

El proyecto se ha podido terminar cumpliendo todos los requisitos propuestos desde el principio e, incluso, alguno más (que la plataforma SPRINGS para Android se pueda ejecutar en máquinas virtuales Java que no fuesen Android).

Se ha invertido mucho tiempo (tal vez demasiado), en conseguir que la plataforma de SPRINGS para Android funcionara.

El desarrollo de la librería no ha ocupado mucho tiempo (la mitad del tiempo del proyecto más o menos) pero con unos resultados muy satisfactorios como haber conseguido utilizar varios sistemas de comunicación a la vez o el uso de una red multisalto para todos los nodos de la red.

Este proyecto puede resultar muy interesante como punto de partida hacia una plataforma de agentes móviles para Android que esté perfectamente adaptada al entorno. El grupo SID pretende seguir progresando su investigación en este tema.

5.3. Tiempo dedicado

Tras la realización del proyecto, se ha realizado una estimación del tiempo invertido.

Estudio de la plataforma SPRINGS y pruebas de funcionamiento: 78 horas. El mayor tiempo invertido fue en la búsqueda de información de agentes móviles y en plataformas de agentes móviles: cómo es su funcionamiento, cuál es la base general de los agentes móviles, etc.

Estudio, ejecución, depuración y solución de SPRINGS para Android: 243 horas. Esta parte duró mucho debido a la depuración y la búsqueda de las soluciones. Aunque parezca sencillo depurar un programa, no es lo mismo un programa "monohilo" que "multihilo", donde se producen condiciones de carrera que afectan a la depuración que se realiza.

Diseño e implementación de Abstract Socket: 118 horas. Mientras se buscaban los problemas que hubo en SPRINGS para Android, se pudo observar la estructura de la plataforma a fondo y se obtuvieron muchas ideas para el desarrollo de esta librería.

Diseño e implementación de la red Mesh: 161 horas. Esta parte amplía la librería Sockets abstractos para que funcione con dispositivos alejados, creando una red Mesh. Aunque parece un simple concepto, había que desarrollar y diseñar muchas partes para que la red estuviese operativa.

Bifurcación de las librerías: 37 horas. Esta es la parte más corta del proyecto, pero no la mas fácil. La librería Sockets abstractos se tuvo que dividir en dos, una para que funcionara en Java y otra que usase la primera y funcionara en Android. Lo más complicado fue la investigación de todas las instrucciones para saber cuáles eran compatibles en Java y hasta qué versión mínima podía alcanzar con Android.

Adaptación de SPRINGS para tener agentes geográficos: 41 horas. Aunque no era una parte básica del proyecto, los agentes geográficos en entornos móviles son un aspecto muy interesante. Se invirtió bastante tiempo

en estudiar cómo representar un área de forma óptima para la plataforma, porque el planeta se representa como un plano no euclidiano.

Tiempo en el desarrollo del trabajo: 79 horas. Es el tiempo invertido en la redacción, puesta en marcha, pruebas con terminales y máquinas virtuales.

En total son 757 horas para el desarrollo de todo el proyecto.

Bibliografía

[1] D. Chess, C. Harrison, A. Kershenbaum, and T. J. Watson, "Mobile agents: Are they a good idea?," *Communications of the ACM*, 1995.

[2] D. B. Lange and M. Oshima, "Seven good reasons for mobile agents," *Communications of the ACM*, pp. 88–89, 1999.

[3] T. G. Nguyen and T. T. Dang, "Agent platform evaluation and comparison," Tech. Rep. Pellucid 5FP IST-200134519, Institute of Informatics, Slovak Academy of Sciences, June 2002.

[4] Y. Aridor and M. Oshima, "Infrastructure for mobile agents: Requirements and design," *Second International Workshop of Mobile Agents*, pp. 38–49, 1998.

[5] H. Peine and T. Stolpmann, "The architecture of the ara platform for mobile agents," *First International Workshop of Mobile Agents*, pp. 50–61, 1997.

[6] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet, "Concordia: An infrastructure for collaborating mobile agents," *First International Workshop of Mobile Agents*, pp. 86–97, 1997.

[7] A. Puliafito, O. Tomarchio, and L. Vita, "Map: Design and implementation of a mobile agents' platform," *Journal of Systems Architecture*, 2000.

[8] L. M. Silva, P. Simões, G. Soares, P. Martins, V. Batista, C. Renato, L. Almeida, and N. Stohr, "James: A platform of mobile agents for the management of telecommunication networks," *Intelligent Agents for Telecommunication Applications*, pp. 76–95, 1999.

[9] R. Trillo, S. Ilarri, and E. Mena, "Comparison and performance evaluation of mobile agent platforms," *IEEE Computer Society*, 2007.

[10] E. Gómez-Martínez, S. Ilarri, and J. Merseguer, "Performance analysis of mobile agents tracking," *WOSP '07 Proceedings of the 6th international workshop on Software and performance*, pp. 181–188, 2007.

[11] O. Urra, S. Ilarri, R. Trillo, and E. Mena, "Mobile agents and mobile devices: Friendship or difficult relationship?," *Journal of Physical Agents*, 2009.

[12] A. Moreno, A. Valls, and A. Viejo, "Using JADE-LEAP to implement agents in mobile devices," *Research Report 03-008, DEIM, URV*, 2005.

[13] S. Ilarri, R. Trillo, and E. Mena, "Springs: A scalable platform for highly mobile agents in distributed computing environments," *WOWMOM '06 Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, pp. 633–637, 2006.

[14] D. B. Lange and O. Mitsuru, *Programming and Deploying Java Mobile Agents Aglets*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[15] J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, and M. Straßer, "Communication concepts for mobile agent systems," *First International Workshop of Mobile Agents*, pp. 123–135, 1997.

[16] O. Urra, S. Ilarri, and E. Mena, "Agents jumping in the air: Dream or reality?," *Springer Berlin Heidelberg*, pp. 627–634, 2009.

[17] O. Urra, S. Ilarri, and E. Mena, "Testing mobile agent platforms over the air," *IEEE Computer Society*, pp. 152–159, 2008.

[18] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler, "A design framework for highly concurrent systems," *UC Berkeley Tecnical report UCB/CSD-00-1108*, 2000.

[19] C. Kramer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," *Reverse Engineering, Proceedings of the Third Working Conference*, pp. 208–215, 1996.

[20] C. Weckerle and L. Strick, "Mobile agents in a P2P world," *IEEE Computer Society*, pp. 1876–1881, 2004.

[21] D. H. Lee, K. W. Cho, W. S. Jeon, and D. G. Jeong, "Two-stage semi-distributed resource management for device-to-device communication in cellular networks," *IEEE Communications Society*, pp. 1908–1920, 2014.

[22] G. Moro and G. Monti, "W-grid: a cross-layer infrastructure for multi-dimensional indexing, querying and routing in wireless ad-hoc and sensor networks," *IEEE Communications Society*, 2006.

[23] J. Park, H. Youn, and E. Lee, "A mobile agent platform for supporting ad-hoc network environment," *International Journal of Grid and Distributed Computing*, 2008.

[24] A. Raniwala and T. cker Chiueh, "Architecture and algorithms for an ieee 802.11-based multi-channel wireless mesh network," *IEEE Computer Society*, vol. 3, pp. 2223–2234, 2005.

[25] J. Bicket, D. Aguayo, S. Biswas, and R. Morris, "Architecture and evaluation of an unplanned 802.11b mesh network," *MobiCom Mobile Computing and Networking*, pp. 31–42, 2005.

[26] K. Stencel and P. Wegrzynowicz, "Implementation variants of the singleton design pattern," *Springer Berlin Heidelberg*, pp. 396–406, 2008.

[27] M. Durresi, A. Durresi, and L. Barolli, "Emergency broadcast protocol for inter-vehicle communications," *IEEE Computer Society*, pp. 402–406, 2005.

[28] R. Lambiottea, V. D. Blondela, C. de Kerchovea, E. Huensa, C. Prieurc, Z. Smoredac, and P. V. Doorena, "Geographical dispersal of mobile communication networks," *Physica A: Statistical Mechanics and its Applications*, 2008.

[29] I. C. Department of Geography, The University of Iowa, "Modelling adaptive, spatially aware, and mobile agents: Elk migration in yellowstone," *International Journal of Geographical Information Science*, 2005.