Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

EasyMANET for Android

Proyecto Final de Carrera

Ingeniero Técnico en Informática de Sistemas

Autor: Salvador Francisco Morera Soler

Director: Carlos Miguel Tavares Calafate

20 de septiembre de 2013

Resumen

Tradicionalmente, a pesar de sus innegables ventajas frente a otras alternativas, las redes móviles *ad hoc* (MANET) han tenido un uso muy limitado. Sin embargo, con la proliferación actual de dispositivos inalámbricos con la potencialidad de utilizarlas, se hace particularmente interesante intentar llevarlas al usuario común. La plataforma EasyMANET tiene éste como su objetivo, ofreciendo al usuario una plataforma a partir de la cual es posible comunicarse con otros usuarios de forma sencilla y transparente aun cuando no exista una infraestructura a nivel de red y de servicios de red que le den soporte. El presente proyecto representa un intento por acercar EasyMANET al sistema operativo más extendido a día de hoy en dispositivos móviles: Android. Con este objetivo se ha centrado el desarrollo en el componente principal de EasyMANET, una interfaz de DNS visual a partir de la cual es posible establecer el contacto entre usuarios y disponer de todo tipo de servicios. En lo que respecta a servicios, se ha desarrollado un servicio de chat, siendo sencillo extender la aplicación a muchos otros servicios en red en un futuro.

Palabras clave: EasyMANET, VisualDNS, Android, ad hoc, red inalámbrica, red móvil ad hoc, MANET.

Tabla de contenidos

1. Introducción 8
1.1. Motivación 8
1.2. Objetivos 8
1.3. Estructura del documento 9
2. La plataforma EasyMANET 10
2.1. Redes móviles *ad hoc* 10
2.1.1. El modelo OSI 10
2.1.2. Protocolos de encaminamiento 11
2.1.3. El estándar 802.11 12
2.1.4. Limitaciones 13
2.2. Autoconfiguración 14
2.3. Visual DNS 15
2.4. Servicios 16
3. Implementación en Android 18
3.1. Introducción a Android 18
3.1.1. Entorno de desarrollo 18
3.1.2. Estructura de un proyecto 19
3.1.3. Android *manifest* 20
3.1.4. Principios de diseño de Android 21
3.1.5. Activities 24
3.1.6. Intents 25
3.1.7. Services 26
3.1.8. Multithreading 28
3.2. Backend 29
3.2.1. Definición de clases 29
3.2.2. Soporte a Visual DNS 31
3.2.3. Servicios 36
3.3. Frontend 37
3.3.1. MainActivity 37

3.3.2. ChatActivity 38
3.3.3. SettingsActivity 39
4. Validación 42
4.1. Entorno de pruebas 42
4.2. Workflow 42
4.3. Problemas 47
5. Conclusiones 50

1. Introducción

1.1. Motivación

Hoy en día, todo el mundo desea estar conectado permanentemente y en todo lugar. El mercado ofrece multitud de opciones para satisfacer esta necesidad tanto a nivel de infraestructuras (tales como redes UMTS y Wi-Fi, NFC o Bluetooth), como de dispositivos (*smartphones*, *tablets*, ordenadores portátiles) y aplicaciones (WhatsApp, Skype, Line y muchísimas más).

Pero todas estas soluciones, y combinaciones de ellas, tienen inconvenientes desde el punto de vista del usuario. Todas las aplicaciones mencionadas requieren de conexión a internet que, aunque es accesible desde casi cualquier lugar gracias a las redes UMTS, no está disponible en todas las situaciones. Además, el acceso a internet lleva asociado un coste económico. Las tarifas planas ofertadas por los operadores tienen límites en cuanto a los datos consumidos mensualmente. Por su parte, las comunicaciones vía NFC y Bluetooth sólo permiten el enlace de dispositivos a muy cortas distancias y no permiten la comunicación entre dispositivos que no sean directamente visibles entre sí.

Por otro lado existen las *mobile ad hoc networks* (MANET) cuyo propósito es el establecimiento de redes de comunicación inalámbrica entre dispositivos móviles sin más infraestructura que los propios dispositivos. El inconveniente principal a la hora de utilizar esta tecnología es que resulta complicada de configurar para el usuario medio.

La plataforma EasyMANET [1] propone una solución a estos problemas al integrar la gestión transparente de una MANET con una configuración lo más sencilla posible para el usuario.

El sistema operativo (SO) Android, con más de 1000 millones de dispositivos activados y una tasa de 1.5 millones de activaciones diarias [2], se ha convertido en el sistema dominante en computación móvil.

Es por todo esto que la implementación de una aplicación lista para usar la plataforma EasyMANET en dispositivos con el SO Android resulta interesante.

1.2. Objetivos

El principal objetivo del presente proyecto es la implementación de una aplicación compatible con Android lista para ser usada con la plataforma EasyMANET. Para ello deberá implementar dos elementos principales. Primero, un servicio de anuncio a través del cual el dispositivo da a conocer su presencia en la red. En segundo lugar, un servicio de descubrimiento de nodos para mantener una lista de los dispositivos conectados y accesibles en un momento dado y obtener la información de éstos (dirección IP, los servicios que ofrece y una imagen identificativa). Estos dos elementos sirven como base del servicio Visual DNS, cuyo cometido es presentar al usuario de una manera gráfica los nodos que componen la red y los servicios que ofrece cada uno de ellos.

Como objetivo secundario, y para demostrar el correcto funcionamiento de los servicios anteriormente mencionados, de todas las posibles funcionalidades que la aplicación podría

ofrecer al usuario (*chat*, videoconferencia, geolocalización, intercambio de archivos…) se implementará un *chat* basado en texto. Es objetivo, también, del presente proyecto diseñar la aplicación de tal modo que sea relativamente simple extenderla para ofrecer más funcionalidades.

Además, se tendrá en cuenta la especial idiosincrasia de un SO como Android, pensado principalmente para su funcionamiento en teléfonos móviles con lo que ello comporta: restricciones severas en cuanto a uso de memoria y la posibilidad de que en cualquier momento, y por causas externas a nuestra aplicación (recepción de una llamada, giro de pantalla, etc.), ésta sea detenida o incluso destruida. La aplicación deberá responder a estos eventos de la manera en que un usuario de una aplicación comercial esperaría.

Un tercer objetivo secundario es el de que la aplicación tenga un aspecto, si no profesional, al menos estándar y homologable al de otras aplicaciones comerciales presentes en la *Play Store* <*https://play.google.com/store/apps*>. Para ello, en el diseño de la interfaz de usuario se seguirá en la medida de lo posible las guías ofrecidas por Google en el *site* dedicado a ello. [3]

1.3. Estructura del documento

En el siguiente capítulo, "La plataforma EasyMANET", se explica qué es EasyMANET y qué ventajas ofrece sobre las redes móviles *ad hoc* tradicionales. Para ello se hace una breve introducción a las arquitecturas de red y las dificultades que presentan las MANET frente a las redes tradicionales. A continuación, se describen las soluciones que aporta EasyMANET a estas dificultades.

En el capítulo 3, "Implementación en Android", se describe cómo se ha conseguido llevar las funcionalidades de EasyMANET a Android. En el punto 3.1 se describen las particularidades del sistema Android como plataforma para los desarrolladores. En los siguientes puntos del capítulo se detalla el modo en que se ha implementado la aplicación "EasyMANET for Android", que da nombre al presente proyecto.

En el capítulo 4, se detallarán las pruebas de validación realizadas tras el desarrollo de la aplicación, para comprobar su correcto funcionamiento.

En el capítulo final, "Conclusiones", se analizará el grado de cumplimiento de los objetivos indicados en el punto 1.2. Se explicarán también en este capítulo, algunas mejoras y ampliaciones de funcionalidad que sería interesante incorporar a la aplicación en un futuro.

2. La plataforma EasyMANET

En este capítulo se presenta la plataforma EasyMANET. En el primer punto se introducirán las redes móviles *ad hoc*, su arquitectura y limitaciones. Los siguientes puntos del capítulo están dedicados a las soluciones que aporta EasyMANET a dichas limitaciones.

2.1. Redes móviles *ad hoc*

Ad hoc es una locución latina que se traduce literalmente como “para esto”. En el contexto que nos ocupa, la locución se refiere al hecho de en una red móvil *ad hoc* (MANET por sus siglas en inglés) no es necesaria una infraestructura preexistente para ser establecida. Al contrario de las redes centralizadas, en que todos los nodos deben ser capaces de comunicarse en todo momento directamente con un punto de acceso, en las redes móviles *ad hoc* sólo es necesario estar en contacto con, al menos, un nodo cualquiera de la red.

En una MANET son los propios nodos los que se encargan de las tareas de encaminamiento y reenvío de datos necesarias para la comunicación entre nodos que no son directamente accesibles entre sí.

Además de descentralizada, este tipo de redes están compuestas por dispositivos móviles, lo que les confiere una gran flexibilidad. Esta ventaja sobre las redes tradicionales con estaciones fijas entraña dificultades añadidas, entre las que destaca una topología de red sujeta a constantes e impredecibles cambios. El carácter distribuido de las MANET también supone ciertas limitaciones de las que se hablará más adelante.

Estas dos características de las MANET, descentralización y movilidad, las habilita para ser usadas en entornos y situaciones en las que el uso de una red centralizada o cableada no sería posible o adecuado, o resultaría más costoso desde el punto de vista económico.

Entre los usos posibles que se le puede dar a una MANET se encuentra el establecimiento de redes en situaciones en que las que el uso o despliegue de infraestructura es imposible, como en entornos de guerra o tras desastres naturales. Otros ejemplos de uso incluyen: el intercambio de grandes archivos entre varios dispositivos en cualquier lugar, sin tener que hacer uso de las redes UMTS, y su relativa lentitud, así como los costes económicos asociados; el establecimiento de una red de comunicaciones y geolocalización para las tareas de extinción de incendios forestales en una zona sin cobertura UMTS, o la comunicación mediante *chat* basado en texto sin depender de que todas las conversaciones pasen por servidores de terceros, con el riesgo de pérdida de privacidad que ello comporta.

2.1.1. El modelo OSI

El modelo de Sistemas Abiertos de Interconexión (OSI por sus siglas en inglés) definido por ISO/ IEC provee de un marco común para el desarrollo de estándares y arquitecturas con el propósito de interconexión de sistemas [4].

Este modelo divide la arquitectura de una red en siete capas, definiendo las responsabilidades de cada una de ellas en el conjunto y como deben interactuar entre sí. Cada capa debe ser independiente de las demás en la realización de sus funciones. En conjunto puede verse como una pila, donde cada capa se apoya en la anterior para proveer de servicios a la superior. Es la llamada pila OSI, cuyas capas y sus respectivas funcionalidades ilustra la figura 1.

Figura 1. La pila OSI.
Fuente: Wikimedia Commons, autor Marco Bertolini, adaptación: <http://commons.wikimedia.org/wiki/User:LordT>

2.1.2. Protocolos de encaminamiento

Cuando dos nodos de una red no son directamente accesibles entre sí y se debe enviar un paquete de datos entre ellos, se requiere que éste efectúe varios saltos. Para ello debe encontrarse un camino o ruta que le permita llegar a su destino a través de la red, reenviándose de nodo a nodo. Para determinar qué camino seguirá el paquete se utilizan los protocolos de encaminamiento o enrutamiento.

Los protocolos de encaminamiento se enmarcarían en la tercera capa del modelo OSI, el nivel de red, entre cuyos cometidos se encuentra la determinación de rutas lógicas para los paquetes de datos.

Como se ha comentado, en una MANET los dispositivos son móviles y no se dispone de una infraestructura previa. Estas dos características suponen que los algoritmos clásicos de encaminamiento utilizados en redes cableadas y con infraestructura no sean adecuados para su uso en las MANET. Dado el actual interés que suscita el estudio de las redes móviles *ad hoc*, se han desarrollado múltiples propuestas de protocolos de enrutamiento adaptados a sus características. Previamente esbozaremos una clasificación de los algoritmos de

encaminamiento para después mencionar algunos que cuentan con las características necesarias para ser usados en redes móviles *ad hoc*.

Los protocolos de encaminamiento pueden clasificarse atendiendo a diversos aspectos:

- Centralizados o distribuidos: cuando es centralizado, la ruta la determina un único nodo al que los demás consultarán antes de enviar un paquete de datos. En los distribuidos todos los nodos colaboran para determinar la ruta que seguirá el paquete.

- Reactivos o proactivos: los protocolos proactivos se basan en el mantenimiento de tablas de enrutamiento, actualizadas periódicamente, que son consultadas para determinar un camino. Los reactivos determinan las rutas bajo demanda en el momento en que son necesarias. Existen también soluciones híbridas que combinan técnicas de ambos tipos.

- Adaptativos o estáticos: los algoritmos adaptativos son capaces de reaccionar a cambios en el estado de la red de topología, congestión u otros, mientras que los estáticos no.

Según estas clasificaciones diremos que un protocolo adaptado a su uso en MANETs no puede ser centralizado, dado que no todos los nodos tienen porqué ser capaces de acceder al nodo central. Tampoco puede ser estático pues la propia naturaleza de una red móvil es dinámica. Así pues, los protocolos utilizados en estas redes deben ser adaptativos y distribuidos.

Como ejemplos de algoritmos de enrutamiento que cumplen estos dos requisitos mencionaremos el *Ad hoc On-Demand Distance Vector* (AODV) [5], el *Dynamic MANET On-demand* (DYMO) [6] y el *Optimized Link State Routing Protocol* (OLSR) [7].

De ellos, los protocolos AODV y DYMO son de tipo reactivo para evitar el efecto de sobrecarga de la red que puede producirse al intentar mantener las tablas de ruta continuamente actualizadas en redes tan potencialmente cambiantes, mientras que el OLSR es de tipo proactivo lo que minimiza el tiempo necesario para establecer un enlace entre dos nodos.

2.1.3. El estándar 802.11

El propósito del estándar 802.11 *“Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications”* es proporcionar conectividad inalámbrica para estaciones fijas, portátiles y móviles dentro de un área local, definiendo para ello especificaciones de un control de acceso al medio (MAC) y de varias capas físicas. [8]

El estándar 802.11 define pues especificaciones para dispositivos en la primera capa de la arquitectura OSI -nivel físico-, y la segunda –nivel de enlace de datos.

Para la capa física el estándar 802.11 ha definido múltiples especificaciones a lo largo de los años entre las que comentaremos las más comúnmente adoptadas a día de hoy por los dispositivos con SO Android:

- 802.11a: operando en la banda de radiofrecuencia relativamente menos usada de los 5 GHz ofrece menores interferencias. El inconveniente de operar a estas frecuencias

es que esta banda tiene menor capacidad de penetración, por lo que ofrece menores rangos de distancia operativa media. Es capaz de alcanzar tasas de transmisión de datos entre 6 Mb/s y los 54 Mb/s.

- 802.11b: opera en la banda de los 2.4 GHz, muy utilizada en dispositivos de otros tipos, como los hornos microondas o los dispositivos Bluetooth, por lo que es propensa a las interferencias. Es capaz de alcanzar velocidades de 11 Mb/s.
- 802.11g: es el resultado de la fusión de características de los dos anteriores, siendo compatible con 802.11b. Opera en la banda de los 2.4 GHz y alcanza tasas de transmisión de hasta 54 Mb/s.
- 802.11n: supone una mejora sobre las anteriores ya que es capaz de operar en ambas bandas de frecuencia y a velocidades de hasta 600 Mb/s. Además, es compatible con 802.11 a/b/g.

2.1.4. Limitaciones

En una red basada en el *Internet Protocol* (IP) establecida con infraestructura lo primero que necesita hacer cada cliente conectado, si no se ha establecido manualmente, es obtener sus parámetros de configuración, para lo que realiza una petición a un servidor. Habitualmente estos servidores implementan el protocolo *Dynamic Host Configuration Protocol* (DHCP). Un servidor DHCP, ante una petición correcta, proporciona al cliente una dirección IP única, la máscara de subred, una ruta por defecto que se usará en ausencia de una ruta conocida hasta otro nodo de la red, y una o varias IPs donde localizar a los servidores que proporcionan el servicio de DNS (*Domain Name System*).

Un red móvil *ad hoc*, no proporciona estos servicios *per se*. Tradicionalmente, al carecer de infraestructura, en las redes *ad hoc* se debe configurar cada estación participante asignándoles de forma manual una dirección IP única y la máscara de subred. Además, carecen de servicio de DNS, por lo que un usuario que quiera establecer un enlace con otro, deberá conocer su dirección IP, lo que resulta farragoso. Otro problema, derivado de los otros dos, es que al usuario le resulta imposible conocer a priori qué tipo de servicios ofrece cada participante de la red.

Todas estas limitaciones hacen que las MANET resulten difíciles e incómodas de utilizar incluso para usuarios expertos, reduciendo el interés del usuario común en sus, por otra parte, amplias posibilidades.

La plataforma EasyMANET pretende dar respuesta a estas dificultades aportando para ello métodos para la configuración automática de los parámetros de red y un sistema de identificación visual de nodos, llamado Visual DNS. Este servicio ofrece al usuario, de una manera gráfica, información sobre los miembros que componen la MANET y qué servicios ofrecen.

2.2. Autoconfiguración

En una MANET basada en el estándar 802.11 todos los nodos deben compartir ciertos parámetros: el nombre de la red (*Service Set Identifier*, SSID), el canal usado, el protocolo de seguridad usado, la clave de seguridad, la versión IP, la máscara de subred, el protocolo de encaminamiento y la salida a Internet, si la hubiera. Además, cada nodo debe poseer una dirección IP única en la red, dentro del rango permitido.

EasyMANET propone dos métodos para la autoconfiguración, con intervención mínima por parte del usuario, de todos los parámetros necesarios para configuración de la red y de una dirección IP única para cada uno de ellos.

Por una parte, se propone un método centralizado completamente automatizado basado en Bluetooth, tecnología también presente en una gran cantidad de dispositivos con capacidades Wi-Fi, entre ellos la práctica totalidad de los que instalan Android como SO. De esta manera, el establecimiento de la MANET se realiza en dos fases. En la primera cada nodo participante deberá obtener sus parámetros comunicándose con un servidor a través de Bluetooth. Una vez obtenidos, los nodos ya son capaces de comunicarse entre sí a través del interfaz Wi-Fi configurado en modo *ad hoc*.

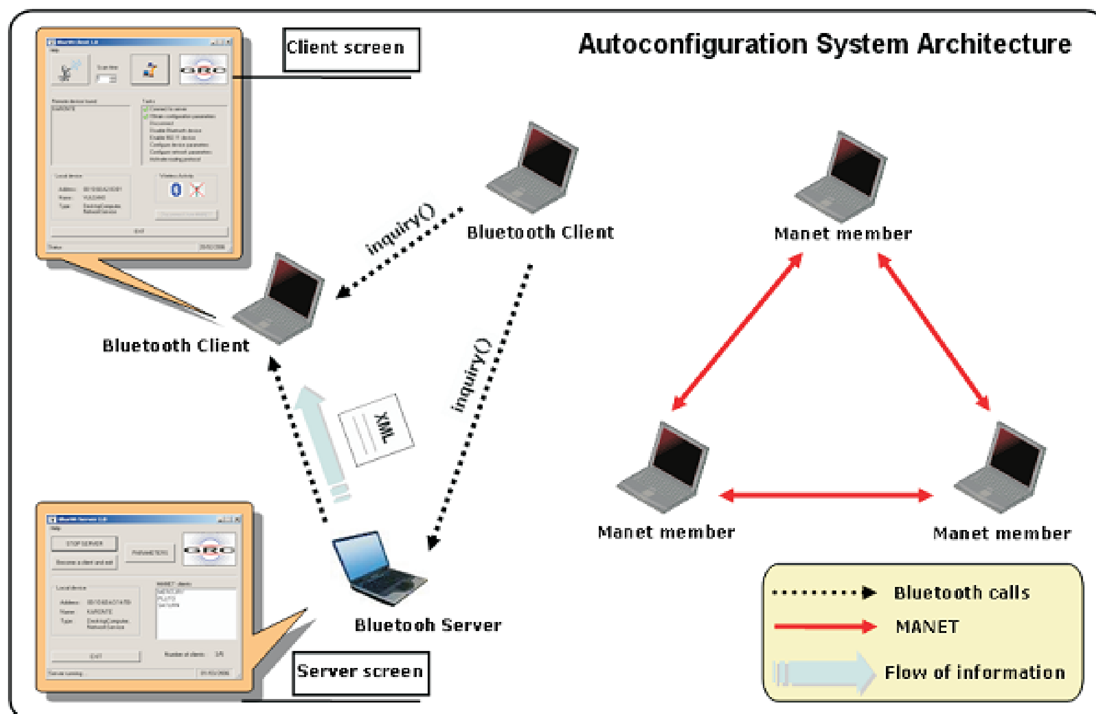

Figura 2. Autoconfiguración por Bluetooth.
Fuente: Integrated Architecture for Configuration and Service Management in MANET Environments. [1]

El otro método propuesto es distribuido y hace uso de la misma tecnología basada en IEEE 802.11 que se utilizará posteriormente una vez establecida la MANET. Esto supone utilizar un enlace para configurar la red de un tipo tal que es el mismo que se pretende configurar. A esta dificultad se añade el hecho de que las comunicaciones inalámbricas deben ser encriptadas para proveer de privacidad a los participantes. Para solucionar este problema aparentemente irresoluble, EasyMANET propone la utilización de la única información no codificada modificable por el usuario en una red basada en 802.11: el SSID. Como el SSID se incorpora en

los *beacons* que todos los miembros de una MANET retransmiten periódicamente, este método no añade tráfico a la red. La solución argumenta que en dicho campo, los bytes reservados para el nombre de la red casi nunca son usados en su totalidad, por lo que propone restringir a sólo los primeros bytes el SSID, y utilizar los restantes para retransmitir el resto de parámetros necesarios (versión de IP en uso, tipo de encriptación utilizada, protocolo de enrutamiento, prefijo de subred y máscara de red si se usa IPv4, y semilla aleatoria para su uso en la obtención de una clave de sesión).

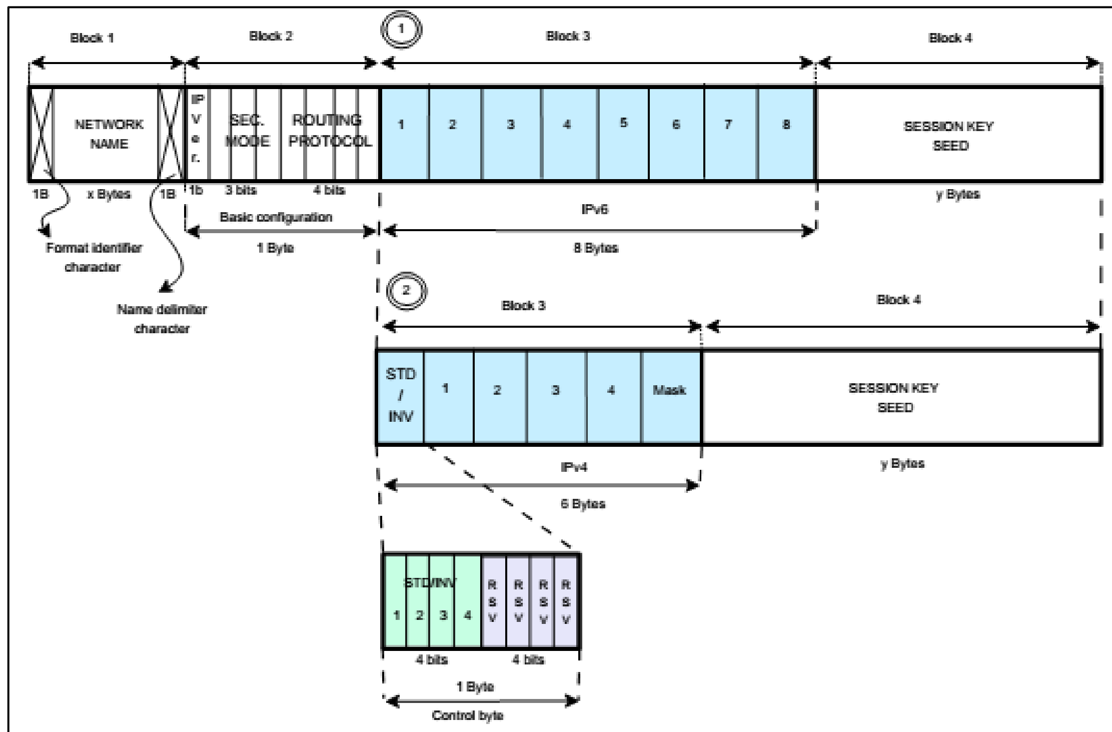

Figura 3. Particionado del SSID.
Fuente: Integrated Architecture for Configuration and Service Management in MANET Environments. [1]

2.3. Visual DNS

Una vez obtenida automáticamente una configuración válida en las capas 2 y 3 de OSI, el siguiente paso en el propósito de acercar al usuario medio el uso de las MANET es ofrecer un sistema de traducción de direcciones a nombre y de descubrimiento de servicios.

La solución que propone EasyMANET se denomina Visual DNS. Este servicio, que se enmarca en el nivel de aplicación de la pila OSI, proporciona una representación visual mediante un nombre y una imagen de cada nodo, yendo más allá del DNS tradicional que sólo ofrece traducción de nombres. Además, ofrece al usuario una interfaz para el acceso a los servicios que cada nodo ofrece.

De este modo Visual DNS soluciona ambas limitaciones de las MANET. Para ello implementa un servicio de descubrimiento de vecinos, de manera que, eventualmente, cada nodo de la red dispondrá de la información (imagen y nombre de usuario) de los demás y los servicios que ofrecen, para presentarle al usuario esta información de una manera visual.

Para que el servicio sea operativo el usuario debe proporcionar un nombre y una imagen. Tras ello, puede poner en marcha el servicio, con lo que Visual DNS comenzará a recabar la información de lo demás nodos y a diseminar su propia información.

El protocolo utilizado en el servicio de descubrimiento de vecinos funciona como sigue. Periódicamente, cada nodo transmite a sus vecinos inmediatos un paquete UDP con una lista de todas las IP que conoce junto con un número de secuencia. Cuando un nodo recibe un paquete UDP se compara la lista recibida con la propia para comprobar si contiene algún par de dirección-nº de secuencia desconocido. Si es el caso, el nodo enviará una petición a través de TCP hacia el nodo que envió la lista, para obtener la información (nombre, foto y servicios ofertados) de los nodos desconocidos. Una vez obtenida la información sobre los nodos, se añaden a la propia lista de nodos. Cuando un nodo modifica su lista de servicios, incrementa su número de secuencia para indicar a los demás que existe información nueva. Cuando Visual DNS modifica su lista de usuarios conocidos, actualiza la interfaz gráfica para informar al usuario.

2.4. Servicios

El objetivo final de toda la arquitectura explicada es el de proporcionar servicios de una manera sencilla a los usuarios de una MANET. Como se ha descrito, Visual DNS se encarga del descubrimiento de los servicios puestos a disposición por cada nodo de la red. Tras ello, utiliza esta información para proporcionar al usuario una interfaz que le permita iniciarlos. Lo que no se ha hecho es especificar cuáles serán estos servicios.

EasyMANET se define como una plataforma extensible para proporcionar servicios en entornos MANET. Efectivamente, sobre EasyMANET pueden construirse multitud de servicios en los que intervengan pares o grupos de usuarios sin necesidad de hacer cambios en la arquitectura subyacente. El servicio Visual DNS, combinado con las soluciones aportadas para las capas dos y tres de la pila OSI, se encarga de abstraer para el programador las complicaciones inherentes al hecho de estar trabajando sobre una red móvil *ad hoc*. Gracias a ello, dispone de una lista de nodos, con sus direcciones IP, por lo que se puede diseñar los servicios como lo haría para el entorno de una red con infraestructura, utilizando para ello las herramientas propias del lenguaje elegido para la implementación.

Efectivamente, para añadir servicios a la plataforma sólo es necesario implementar un servidor que escuche en el puerto asignado a cada uno (o añadir la lógica necesaria a un único servidor multiprotocolo), la lógica asociada y una interfaz gráfica de usuario.

Ejemplos de estos posibles servicios son: cualquier tipo de comunicación entre usuarios (texto, voz, vídeo y combinaciones entre ellas), intercambio de archivos, sistemas visuales de geolocalización de los miembros de la MANET, o juegos en red.

3. Implementación en Android

En el presente capítulo se hace una introducción al SO Android desde el punto de vista del desarrollador de aplicaciones. Después se explica el modo en que se ha implementado la aplicación EasyMANET for Android, su *frontend* – la parte con la que el usuario interactúa – y su *backend* – la parte que da soporte operativo a la aplicación.

Primeramente, en el punto 3.1, "Introducción a Android", se hace una breve introducción al desarrollo de aplicaciones Android en general y a los principales componentes de una aplicación. Se presta especial atención a los elementos de los que se ha hecho uso en el desarrollo de EasyMANET for Android. Concretamente, se detallan: el entorno de desarrollo utilizado, la estructura de un proyecto, la función del archivo AndroidManifest.xml y el uso que de éste se hace, los principios de diseño generales de las aplicaciones Android, los principales componentes de la API de Android que se utilizan en EasyMANET, y las distintas posibilidades que se le ofrecen al desarrollador en materia de programación concurrente.

En la siguiente sección, en el punto 3.2, se explica la implementación del *backend* de la aplicación. En la primera parte, se explican las clases básicas que sirven como modelo, y los mecanismos empleados a efectos de comunicación entre nodos. A continuación, se detalla la implementación del servicio VisualDNS realizada a través de las principales clases definidas para ello. Finalmente, se ve el soporte que se da para la implementación de servicios, con el de *chat* como ejemplo de ello.

Para finalizar el capítulo, en el punto 3.3, se expone la implementación de la interfaz gráfica de usuario, o *frontend*, a través de las *activities* que se han diseñado e implementado.

3.1. Introducción a Android

3.1.1. Entorno de desarrollo

A continuación se enumeran las herramientas empleadas para la implementación de EasyMANET for Android:

- Java Development Kit 6 (JDK 6)
- Eclipse Juno SR2
- *Plugin* Java Development Tools (JDT) para Eclipse
- *Plugin* Android Development Toolkit (ADT) v. 22 para Eclipse
- *Plugin* Subclipse 1.8 para Eclipse
- Android SDK Platform 4.2.2 (API 18)

La combinación de estas herramientas conforma un entorno de desarrollo integrado (IDE, por sus siglas en inglés), que proporciona todo lo necesario para la edición del código, compilado, depuración y diseño de la interfaz gráfica de usuario (GUI) del proyecto EasyMANET en Android. El *plugin* Subversion (SVN) se ha utilizado para el control de revisiones. Las sucesivas revisiones se han almacenado en un repositorio *online* accesible públicamente en <https://www.assembla.com/spaces/easymanet-for-android>.

3.1.2. Estructura de un proyecto

Cuando se crea un nuevo proyecto de Android en el entorno descrito, Eclipse lanza un asistente que permite la introducción de unos parámetros básicos: nombres de la aplicación, del proyecto y del *package* Java, y número de versión objetivo y mínima requerida del SDK. Además, el asistente también permite elegir un aspecto visual general de la GUI, así como la creación del icono de la *app* y la configuración de una *activity* (ver 3.1.5.) inicial básica. Al finalizar el asistente, el *plugin* ADT genera automáticamente la estructura de directorios, los archivos básicos necesarios, y enlaza las librerías de Android.

Entre los archivos generados se encuentran varios documentos XML entre los que destaca el AndroidManifest.xml (ver 3.1.3.) que ya contendrá un esqueleto básico con su estructura según los parámetros introducidos. También se genera el archivo XML que define el *layout* de la *activity* principal, así como otros archivos de recursos para definir dimensiones, estilos y *strings*.

Por otra parte se generan varios archivos de tipo .java. El archivo correspondiente al código fuente para controlar el comportamiento de la *activity* y el archivo R.java. El archivo R.java resulta particularmente interesante.

Cada vez que se compila el proyecto, el Android Asset Packaging Tool (AAPT), genera automáticamente la clase R a partir de los recursos presentes en el proyecto. En esta clase se le otorga a cada recurso un identificador único de tipo público y estático. Esto proporciona una manera simple y eficaz de acceder a los recursos del proyecto desde las clases Java implementadas por el programador. Además sirve para realizar, en tiempo de compilación, la comprobación de que las referencias siguen siendo válidas. [9]

Como se observa en la figura 4, existen, dentro de la carpeta “res”, varias carpetas con nombres similares que se diferencian en un sufijo. Estos sufijos sirven para proporcionar valores diferentes para las posibles configuraciones de los dispositivos Android en tiempo de ejecución. Así, por ejemplo, existen varias carpetas de nombre “drawable-xxxx” para las distintas resoluciones de pantalla que se pueden encontrar en los dispositivos Android. El SO busca los recursos adecuados para una configuración determinada en tiempo de ejecución. Si se encuentran recursos específicos a la situación, se utilizarán éstos. Si no se encuentran, se utilizarán los de la carpeta definida por defecto. El programador sólo tiene que hacer referencia al identificador único de cada recurso, definido en la clase R, que es el mismo independientemente de en qué carpeta se encuentre. Se evita, de este modo, la complicación que resultaría de tener que tener en cuenta todas las posibles combinaciones de configuraciones durante la programación.

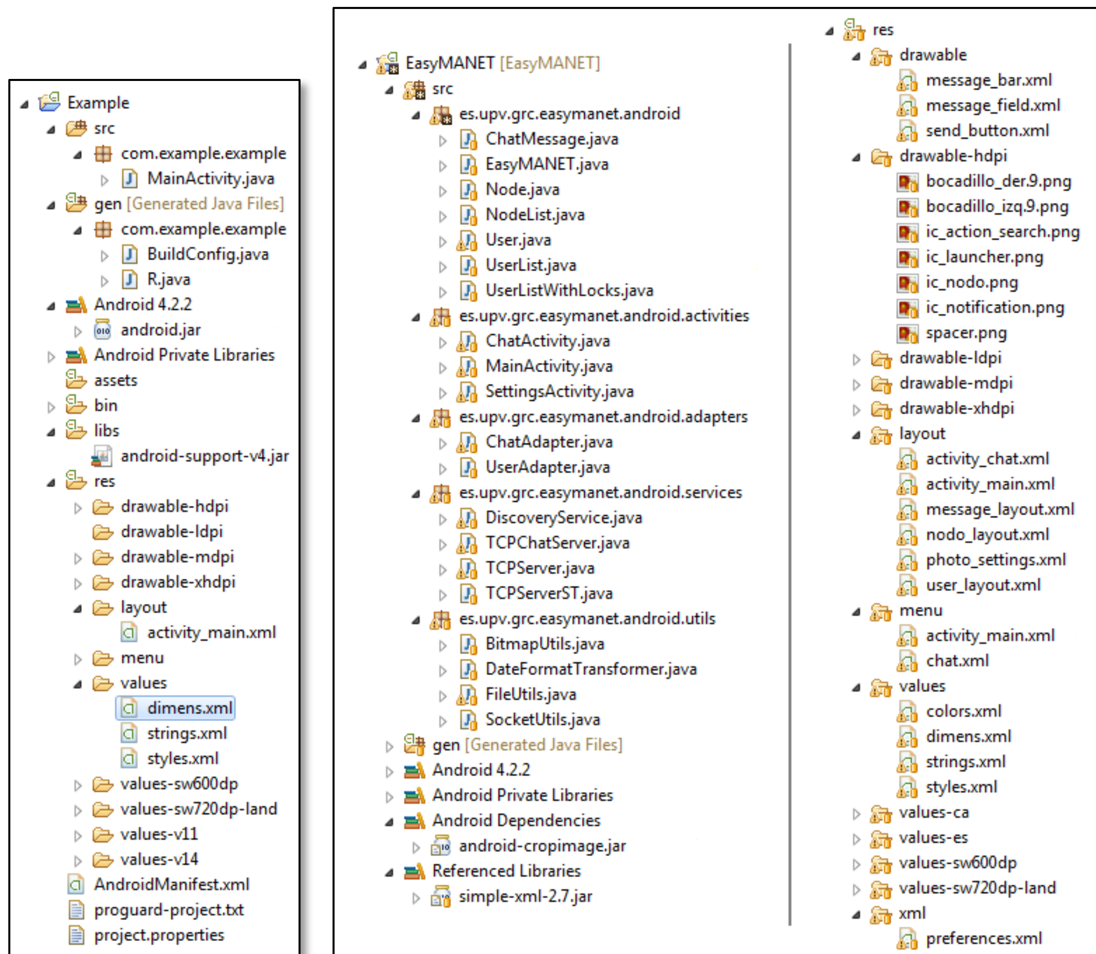

Figura 4. Izq: estructura de un proyecto Android en Eclipse.
Der: estructura del proyecto EasyMANET for Android.

Para el proyecto EfA se han creado distintos recursos en función de las resoluciones de pantalla y de idioma. La aplicación cambia de idioma de manera transparente al usuario, en función del idioma en que tenga configurado su SO. Si el usuario usa su dispositivo en español o valenciano/catalán, la aplicación se mostrará en estos idiomas, en cualquier otro caso, lo hará en inglés. Para conseguirlo, se han definido las carpetas adicionales valuesca y values-es, que contienen, cada una, un archivo denominado strings.xml, donde se encuentran las traducciones. En la figura 4 derecha, se presenta la estructura del proyecto EasyMANET, donde pueden observarse los recursos utilizados y la jerarquía de clases Java.

3.1.3. Android *manifest*

Todo proyecto Android debe tener en su directorio raíz un archivo llamado AndroidManifest.xml. Este archivo contiene información básica sobre la aplicación que el sistema debe conocer antes de poder ejecutarla.

El AndoidManifest.xml de la aplicación desarrollada contiene, entre otros, los siguientes elementos:

- El nombre del *package* Java que servirá como identificador único de la aplicación, en nuestro caso: “`es.upv.grc.easymanet.android`”.

- Una declaración de la versión mínima del API de Android que la aplicación requiere. Esto determina la versión del SO que un dispositivo debe tener instalada para poder ejecutar la aplicación. En nuestro caso se indica una API mínima de 8, lo que se corresponde con la versión 2.2 de Android.

- Una declaración de los componentes de la aplicación y de las clases que las implementan, así como de las capacidades los componentes. Nuestra aplicación tiene varias *activities* y un *service*. En el caso de las aplicaciones con interfaz de usuario, se declara una de las *activities* como la inicial, MainActivity en "*EasyMANET for Android*" (EfA, en adelante), convirtiéndose de esta manera en el punto de entrada a la aplicación.

- Una enumeración de las características que la aplicación requiere que tenga el dispositivo en que se ejecute para poder funcionar, y si son opcionales u obligatorias. Por ejemplo, en nuestra aplicación es preciso que el dispositivo disponga de conectividad Wi-Fi, y es conveniente aunque opcional disponer de una cámara para obtener la imagen del usuario.

- Una enumeración de los permisos que requiere la aplicación. Como medida de seguridad, antes de instalar una nueva aplicación, el SO informa al usuario de las funcionalidades sensibles de que las que hará uso. Por otra parte, una aplicación que intenta utilizar una parte de la API para la que no ha pedido permiso se detiene. De esta manera se asegura que el usuario esté informado de lo que hacen las aplicaciones que instala y que las aplicaciones no hagan más de lo que declaran. Para el funcionamiento de EfA se declaran permisos para acceder al estado de la Wi-Fi y cambiarlo, escribir en la memoria externa y mantener el dispositivo despierto aunque el usuario no interactúe con él.

3.1.4. Principios de diseño de Android

Uno de los problemas de los que adolece, aunque cada vez en menor medida, el SO Android es la falta de uniformidad en cuanto a diseño de las interfaces de usuario de las miles de *apps* presentes en su tienda. Para poner remedio a esta situación, poco después del lanzamiento de la versión 4.0 de Android, Google puso en línea el *site* "Design | Android Developers" [3].

En este *site* Google proporciona recursos, guías, patrones y normas, para que los desarrolladores de *apps* diseñen interfaces que no desentonen con el aspecto del resto del SO. Muchas de estas consideraciones son aplicables sólo a *apps* que vayan a ejecutarse en dispositivos con Android 4.0, mientras que, como se vio, el *target* de la aplicación desarrollada es Android 2.2. No obstante, durante el diseño de EasyMANET for Android se han seguido, en la medida de lo posible, las indicaciones de Google a este respecto. A continuación, enumeraremos algunas de las premisas utilizadas.

En la sección "Devices and Displays [10] se recuerda al diseñador que Android funciona en millones de dispositivos con diversos tamaños y resoluciones de pantalla. Para que una aplicación se vea bien en todos ellos, se requiere la creación de los recursos de imagen a diferentes resoluciones, para ser usados según corresponda. En EfA, se ha tenido en cuenta esto en la creación de todas las imágenes usadas por la aplicación. En la figura 5, se muestra el ejemplo de la guía junto con los iconos creados para la imagen por defecto del usuario.

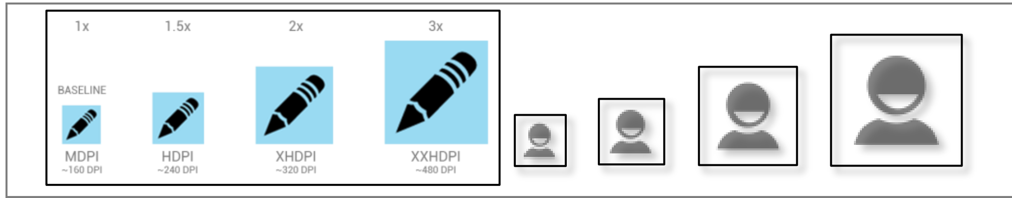

Figura 5. Izq: ejemplo de icono a distintas resoluciones.
Der: imagen de usuario por defecto en EasyMANET for Android.

En la sección "Metrics and Grids" [11] se proponen medidas para el tamaño de los elementos interactivos de la interfaz, y los márgenes que se deben dejar entre cada elemento (ver figura 6).

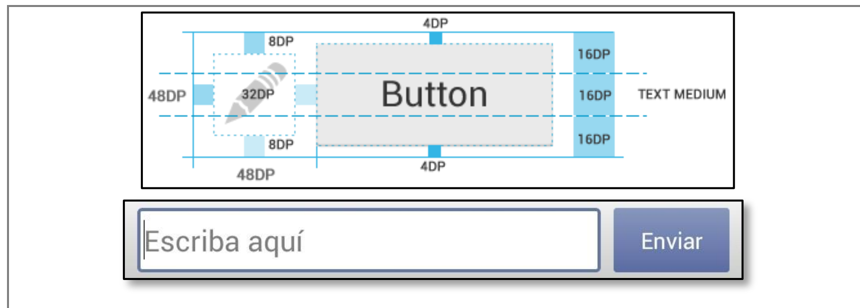

Figura 6: Arriba: parrilla de disposición de elementos interactivos.
Abajo: panel de introducción de texto y botón de envío en la *activity* para el *chat*.

En la sección "Iconography" [12] se ofrecen guías adicionales para el diseño de iconos según su función, que se han tenido en cuenta en el diseño del icono de la notificación que indica al usuario que los servicios de EasyMANET se están ejecutando. La figura 7 contiene los ejemplos aportados en la guía, junto con el icono que usa EfA y su aspecto en la propia notificación.

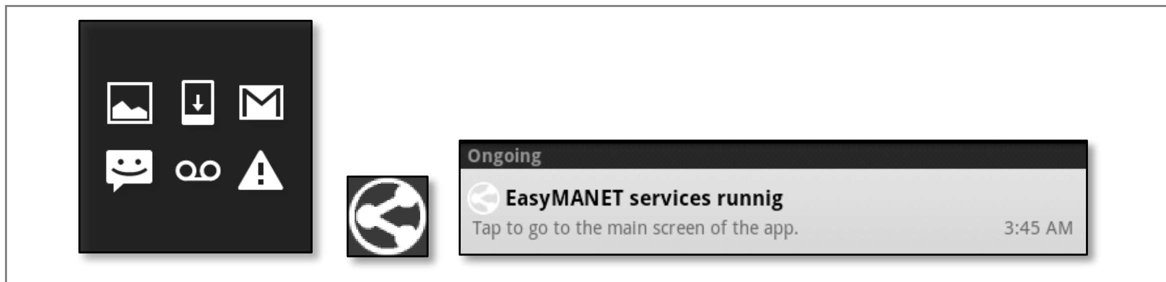

Figura 7. Izq: ejemplos de diseño de iconos de notificación. Centro: icono de notificación de EfA.
Der: aspecto de la notificación de EfA desplegada

En otra sección de la guía,"Writing Style" [13], también se presenta brevemente una especie de libro de estilo para dirigirse al usuario mediante texto. Los textos deben ser concisos, sencillos, y amistosos con el usuario. Es decir, cuanto más cortos y menos técnicos mejor. Además, para resultar amistosos pueden dirigirse al usuario directamente, cuando, por ejemplo, se requiera que actúe. Todas estas sugerencias se han tenido en cuenta en la elaboración de los textos de EfA.

En algunas situaciones, cuando el usuario invoca una acción resulta conveniente pedirle confirmación o informarle de lo que va a ocurrir. En "Confirming & Acknowledging" [14] se ofrece una serie de criterios para decidir en qué ocasiones es conveniente hacer una cosa, la otra o ninguna de las dos. Siguiendo estos criterios, nuestra *app* pide confirmación cuando otro

usuario intenta iniciar una conversación de *chat*, porque ello puede resultar en la pérdida de su conversación con otro usuario, pero no lo hace cuando es el propio usuario quien la inicia. Por otra parte, se informa al usuario de las consecuencias de cambiar una preferencia mientras se ejecutan los servicios de EasyMANET, pero no se le pide ninguna confirmación posterior, porque esto no supone para él mayor perjuicio que volverlos a iniciar. La figura 8 ilustra estos dos usos en la aplicación.

Figura 8. Izq: un Toast para informar al usuario.
Der: un diálogo de confirmación.

Existe también una guía para las preferencias de las *apps*. En ella, entre otras cosas, se exhorta al desarrollador a “no caer en la tentación de convertir todo en una preferencia” [15]. Entre los consejos que se dan, se incluye el de no convertir en configurable en las preferencias algo que la mayoría de los usuarios no cambiarán jamás.

Dado que el uno de los objetivos de la plataforma EasyMANET es acercar al usuario medio al uso de las MANET, se estimó conveniente que, al menos, se proporcionaran valores por defecto para los puertos de escucha de los servicios. Como el usuario medio desconoce lo que es un puerto, es de suponer que la mayoría jamás cambiaría estos valores. Por estos motivos, y de acuerdo con los consejos de Google, la implementación de EasyMANET for Android, a diferencia de otras existentes, no contempla que el usuario pueda modificar los puertos.

Para el diseño de los ítems que contiene la GridView de usuarios conectados se ha seguido el modelo sugerido en “Grid Lists” [16]. Se usa de fondo la imagen del usuario, con su nombre superpuesto con un fondo semitransparente en la parte inferior.

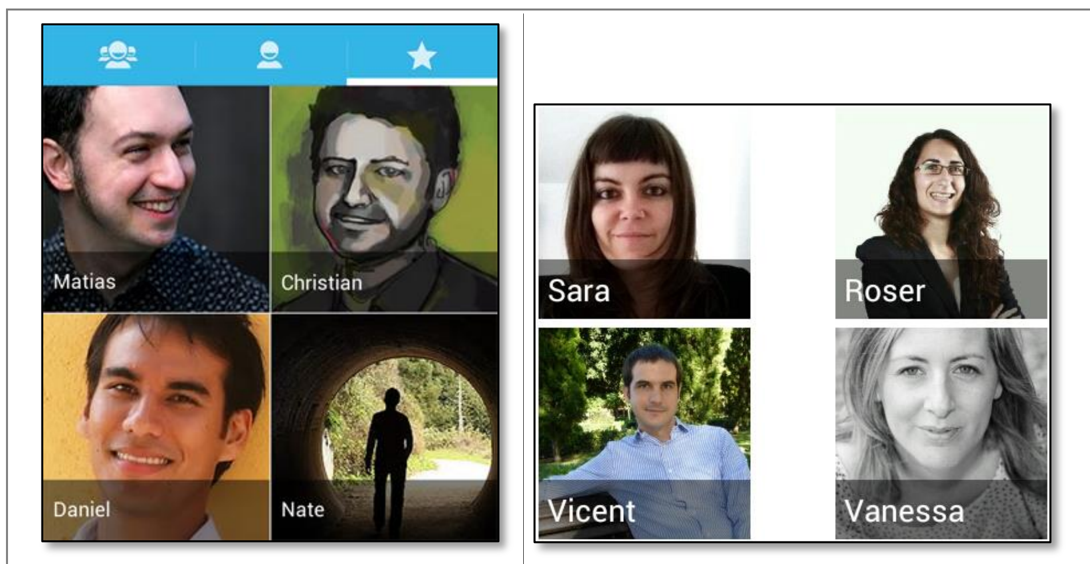

Figura 9. Izq: ejemplo de GridView.
Der: GridView de usuarios conectados en EfA.

Para diferenciar entre los mensajes propios y los del usuario remoto en el chat de texto, se ha optado por utilizar globos de texto al estilo de los cómics, tal y como se hace también en multitud de *apps* de mensajería. Dado que los mensajes pueden variar en longitud y número de líneas, se hace imposible el uso de una imagen estática de fondo. Para lograr el escalado de las imágenes que representan los globos de texto, de manera que se deformen del modo deseado, se ha utilizado la técnica nine-patch [17]. Para crear las imágenes utilizadas, reflejadas en la figura 10, se ha utilizado la herramienta "Draw 9-patch" [18], disponible en el SDK de Android.

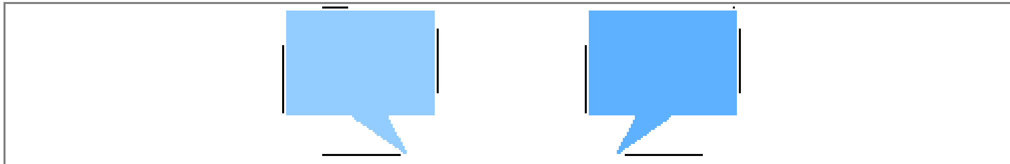

Figura 10. Izq: imagen 9-patch utilizada como fondo de los mensajes del usuario local. Der: imagen 9-patch utilizada como fondo de los mensajes del usuario remoto.

3.1.5. Activities

Las *activities* en Android pueden ser vistas como una abstracción de una unidad lógica de acción, englobando lo que ve el usuario, y el comportamiento de lo que ve tras sus acciones. Es decir: algo que el usuario puede hacer. Una *activity* es pues, para el usuario, algo similar a una ventana en entornos de escritorio, o una Form si pensamos desde el punto de vista del programador.

Android separa muy claramente las dos vertientes de la interfaz: la parte visual y la de interacción. El aspecto visual de la GUI se declara en archivos XML. Estos archivos son cargados en una o varias clases Java, que extienden de la clase Activity, y modelan el comportamiento.

Una *activity* ocupa, típicamente, la práctica totalidad de la pantalla del dispositivo. Por esta razón, y por razones de eficiencia, en un momento dado sólo puede haber una activity en ejecución: la que está en primer plano. Se definen varios estados posibles para una activity que tienen que ver con la gestión de la memoria que hace Android.

Si la *activity* está en primer plano se dice que está en ejecución o activa. Si ya no tiene el foco, pero es parcialmente visible, se dice que está pausada. En estado de pausa normalmente se mantiene en memoria su información de estado y sus miembros (variables), aunque puede ser destruida por el sistema en situaciones de extrema necesidad de memoria. Si la activity está totalmente oculta está en estado de parada. En este estado todavía se mantiene en memoria, pero es muy probable que el sistema la destruya y libere su memoria para usarla en otro sitio. Por último, si ha sido destruida por el sistema y el usuario vuelve a ella debe ser completamente reiniciada y restaurada a su estado anterior. [19]

Todos estos cambios de estado pueden producirse en cualquier momento, bien sea por acción directa del usuario o por cualquier otro evento, como, por ejemplo, la recepción de una llamada. Por este motivo, resulta crucial que la aplicación sea capaz de reaccionar convenientemente ante tales eventos.

Para gestionar todos los posibles cambios de estado de una *activity* se definen una serie de funciones que son llamadas justo antes de que ocurra uno. En el diagrama de la figura 11, se

ilustran los caminos que puede recorrer una *activity* durante lo que se conoce como su ciclo de vida. Los óvalos de colores representan los posibles estados y los rectángulos las funciones llamadas con cada cambio de estado.

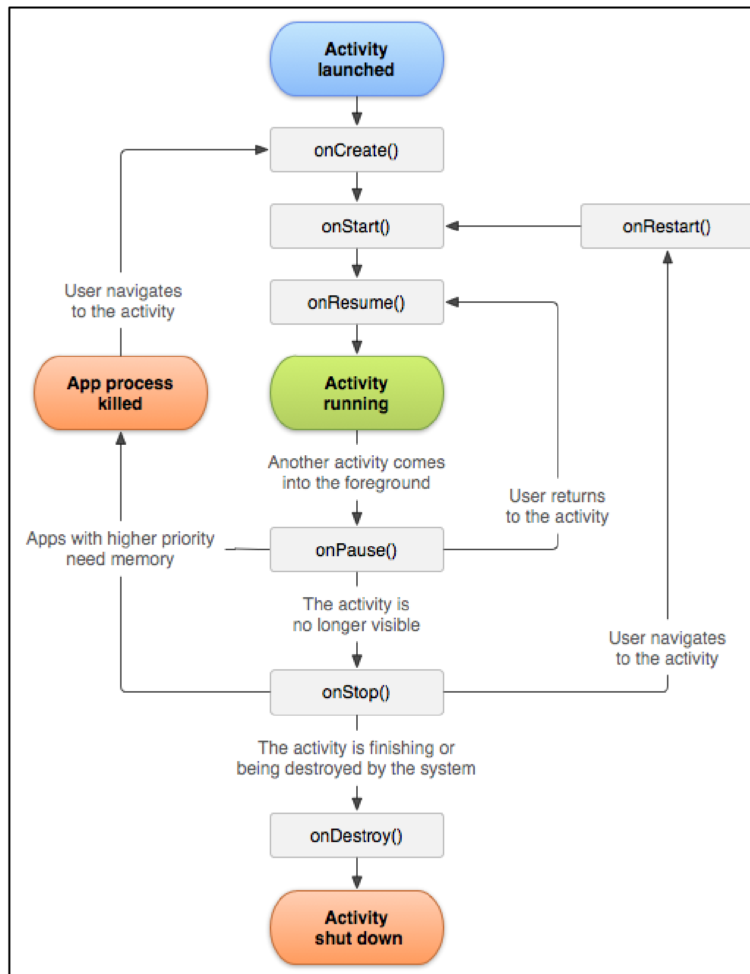

Figura 11. Ciclo de vida de una *Activity*.
Fuente: <http://developer.android.com/guide/components/activities.html>
Licencia: Creative Commons Attribution 2.5

3.1.6. Intents

Un objeto de la clase Intent encapsula una petición al SO para que realice alguna acción. Como respuesta a esta petición, Android puede iniciar una *activity*, un *service* o enviarlo a modo de *broadcast* a todo aquel componente que haya declarado su interés en ese tipo de *intent*. Este mecanismo puede verse como un sistema de comunicación entre componentes donde un *intent* es un mensaje. Los *intents* son asíncronos, lo que significa que el componente que lo envía no detiene su ejecución a la espera de que se complete. [20]

Un *intent* puede ser explícito o implícito. En un intent explícito, el remitente especifica claramente qué componente debe recibirlo y tratarlo. En uno implícito sólo se declara el tipo de receptor. En EasyMANET for Aandroid, de utilizan ambos tipos. Un ejemplo de explícito lo tenemos cuando la *activity* principal inicia o detiene el *service* de descubrimiento de vecinos. Cuando el usuario toca en su imagen, se lanza un *intent* implícito pidiendo tomar una fotografía

con la cámara del dispositivo. En este caso, Android le pregunta al usuario cuál de las aplicaciones capaces de hacerlo desea utilizar.

3.1.7. Services

Como se ha visto, el SO Android sólo garantiza que una *activity* se mantendrá en memoria mientras sea enteramente visible para el usuario. Para la ejecución de tareas que deben continuar incluso aunque el usuario abandone la *activity* existen los *services*.

Un *service* se define como un componente de aplicación que se ejecuta en segundo plano y no tiene GUI. Existen dos tipos de *services*: los locales y los remotos. Los locales proporcionan su servicio sólo a la aplicación que los define, mientras que los remotos pueden proporcionarlo a cualquier aplicación presente en el dispositivo que lo solicite.

Aunque las posibilidades de que la memoria de la que hace uso un *service* sea reclamada por el SO son menores que en el caso de las *activities*, también es posible que esto ocurra. Dependiendo del tipo de tarea encomendada al *service* esto puede resultar más o menos inconveniente. No es lo mismo, por ejemplo, una tarea de actualización de *feeds* de un lector de noticias que la reproducción de música en segundo plano. En el primer caso, no es demasiado grave que la tarea se detenga y vuelva a reintentarse más adelante cuando las condiciones lo permitan. En el segundo, la música que está escuchando el usuario se interrumpiría sin motivo aparente. Evidentemente, la segunda situación debe evitarse en la medida de lo posible.

Para el caso de tareas que puedan suponer un perjuicio para el usuario si son interrumpidas, existen los llamados *foreground services* (paradójicamente, servicios en primer plano). Declarando un servicio como de tipo *foreground*, se consigue que el SO sólo reclame su memoria en casos de extrema necesidad. Para recalcar el hecho de que se trata de un servicio en primer plano, la API exige que tales *services* informen al usuario de que están siendo ejecutados, mediante una notificación en la barra de notificaciones.

El servicio utilizado en EasyMANET for Android para realizar las tareas necesarias para el funcionamiento de Visual DNS y los servicios ofertados es de tipo local y en primer plano, por razones que se verán en el punto dedicado a la implementación de Visual DNS.

Los *services*, como las *activities*, tienen su propio ciclo de vida (figura 12), aunque es más sencillo que el de estas últimas. Un *service* puede ser creado de dos maneras: mediante la llamada, desde una *activity*, a los métodos `startService(Intent i)` o `bindService(Intent service, ServiceConnection conn, int flags)`. Estas dos maneras de iniciarse suelen relacionarse con los tipos de servicio local y remoto, respectivamente. El modo en que se inicia el *service* determina también el flujo que seguirá en su ciclo de vida y bajo qué circunstancias será detenido.

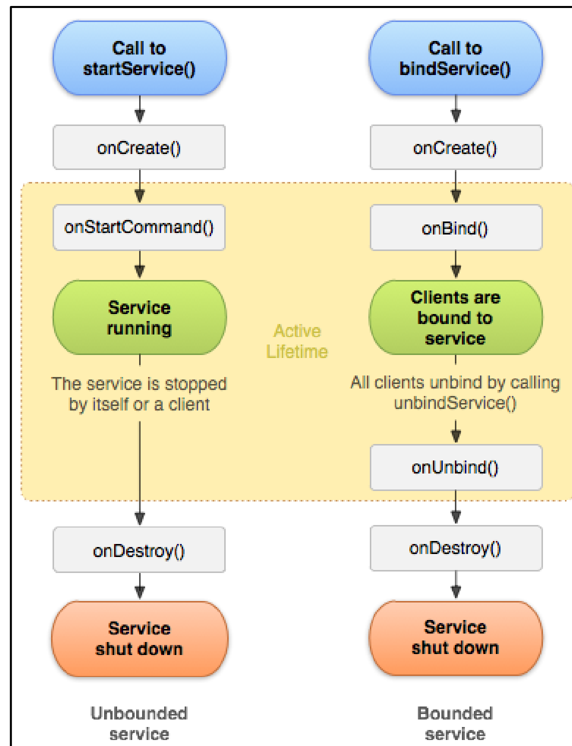

Figura 12. Ciclo de vida de un *Service*.
Fuente: <http://developer.android.com/guide/components/services.html>
Licencia: Creative Commons Attribution 2.5

En el caso de utilizar `startService(Intent i)` para iniciar un servicio, éste permanecerá en ejecución hasta que se detenga a sí mismo o se le indique explícitamente, con una llamada a `stopService(Intent i)`, que debe hacerlo.

En el caso de iniciar un *service* mediante una llamada a `bindService(Intent service, ServiceConnection conn, int flags)`, se iniciará si no lo estaba ya y se incrementará un contador de *bindings*. Cuando un componente se desconecta del *service*, mediante una llamada a `unbindService(ServiceConnection con)`, se decrementa el contador. Cuando el contador llega a cero, el service se detiene.

Para nuestra aplicación, se desea que nuestro *service* se inicie y se detenga mediante la pulsación de un botón por parte del usuario. También se requiere que siga ejecutándose independientemente de la *activitiy* que se encuentre en primer plano, para que los servicios y Visual DNS no dejen de funcionar. Además, como ya se ha comentado, se trata de un *service* de tipo local. Por estos motivos, para que sólo se detenga tras una petición explícita, parece evidente que la aproximación adecuada para iniciarlo sea el uso de la llamada a la función `startService(Intent i)`. Y de hecho lo es, y es la que se usa en EfA. Pero, tras iniciar el *service,* se hace también una llamada a `bindService(Intent service, ServiceConnection conn, int flags)` para poder obtener una instancia del objeto de tipo *Service* permitiendo a la *activity* llamar a sus métodos públicos.

3.1.8. Multithreading

Hay un aspecto crucial a tener en cuenta cuando se diseña una *app* para Android: el *thread* principal se encarga de la GUI. Así, todo lo que ocurre en una *activity* se ejecuta en este *thread*, denominado UI *thread* (*user interface thread*). El UI *thread* es, también, el único que puede modificar la interfaz de usuario. Por tanto, si se realiza una operación larga en el código de una *activity*, la interfaz queda bloqueada mientras dure. Android es muy expeditivo con estas situaciones y pone límites de tiempo muy cortos antes de considerar que una aplicación no responde y cerrarla con un mensaje de error.

Otra característica a tener en cuenta es que, aunque los *services* están pensados para ejecutarse en segundo plano, esto no significa que sean un *thread* en sí mismos. Es decir, su código se ejecuta en el UI *thread*. Así, si se van a realizar operaciones potencialmente largas o de duración indeterminada, el programador debe asegurarse de que no lo hagan en el *thread* principal, para no bloquear la GUI.

Para facilitar el desarrollo de aplicaciones concurrentes y la comunicación asíncrona entre hilos, la API de Android pone a disposición del programador varios mecanismos, aparte de los clásicos de las bibliotecas Java SE. Puesto que EasyMANET for Android es una aplicación que hace multitud de operaciones de red, que son de duración indeterminada, se ha hecho uso de varias de estas herramientas. A continuación, se resumen las utilizadas y el lugar en que se utilizan.

Al arrancar el *service* implementado, se lanzan varios Threads clásicos de la biblioteca de Java, cuyas funciones se explican en el punto 3.2, “Backend”. De ellos, el *thread* encargado de la recepción de peticiones vía TCP utiliza un ThreadPoolExecutor [21]. Esta clase mantiene y gestiona un *pool* de *threads*, de tamaño fijo o variable, que asigna a las tareas que se le envíen para su ejecución.

Para facilitar la ejecución de tareas en segundo plano y publicar los resultados en el UI *thread*, la API de Android poroporciona las AsyncTasks [22]. Esta clase abstracta se implementa mediante la definición de cuatro métodos: *onPreExecute, onProgressUpdate, onPostExecute y doInBackground*. De ellos, los tres primeros se ejecutan en el UI *thread*, pudiendo modificar la interfaz. El primero que se ejecutará, *onPreExecute*, sirve para preparar la tarea y modificar la interfaz si fuera necesario para, por ejemplo, poner una barra de progreso. A continuación se ejecuta *doInBackGround*, que es el encargado de realizar la tarea, pudiendo publicar resultados parciales. Cada vez que se publica un resultado parcial, el método *onProgressUpdate* es ejecutado. Cuando la tarea finaliza, se ejecuta *onPostExecute*, que podrá actualizar la interfaz para informar al usuario del resultado de la operación.

Aunque quizás no sea el método más conveniente, se han utilizado varias AsyncTasks en la Activity del *chat*. Se ha hecho así, porque, aunque implícito, era objetivo de este proyecto el uso la mayor cantidad posible de elementos de la API Android. Además, como se ha comentado, las AsyncTasks son una herramienta sencilla de utilizar.

La API de Android proporciona, además, multitud de herramientas para facilitar la comunicación asíncrona entre procesos y *threads*. De entre ellos, se han utilizado las clases Handler [23] y ServiceConnection [24], los ya comentados *intents* y el contenedor de mensajes Parcel [25].

En EasyMANET for Android, un Handler es empleado para enviar mensajes desde el *service* hacia la *activity* principal. Para la comunicación en el otro sentido, se usa la clase ServiceConnection.

Como ya se ha visto, entre otros muchos usos, los *intents* pueden ser utilizados para iniciar una *activity*. Dentro de los prámetros que se pueden incluir en un *intent*, se incluyen los objetos que implementan la interfaz *Parcelable*. En el *intent* utilizado para iniciar la *activity* que implementa el *chat* de nuestra aplicación, se incluye el usuario con el que se conversará, en forma de *parcel*.

3.2. Backend

3.2.1. Definición de clases

Se han definido dos clases para representar las entidades básicas de la aplicación: usuario y nodo. También se han implementado dos clases que son listas de objetos del tipo usuario y nodo, con el propósito encapsular sus funcionalidades, facilitar su serialización para transmitirlas a través de la red, y asegurar que se evitan condiciones de carrera en su uso concurrente. A continuación se detalla cada una de ellas y se explica el mecanismo de serialización utilizado.

Clase Node

La clase *Node* contiene la información más básica relativa a cada dispositivo miembro de la MANET. Ésta información incluye su dirección IP en formato String, un *timeStamp* que contiene el momento en que se conectó a la red y el atributo *lastSeen* que es actualizado por cada cliente en el propio nodo antes de retransmitir la lista de nodos conocidos.

El atributo *timestamp* de la clase Node se utiliza con un propósito similar al del código de secuencia que especifica EasyMANET en el protocolo de descubrimiento de vecinos, con algunas salvedades. En la implementación realizada en EfA, no se ha contemplado la posibilidad de que un usuario cambie su configuración básica mientras está conectado a la MANET. Si un usuario modifica sus parámetros (nombre, foto o servicios) el sistema actualiza su *timeStamp* y los demás usuarios lo consideran a todos los efectos como un usuario nuevo. Esto es así para prevenir la posibilidad de que a un nodo recién conectado a la red se le asigne la IP de un nodo que se hubiera desconectado recientemente y el resto de nodos trataran al nuevo como si el antiguo hubiera, simplemente, cambiado su configuración. Atendiendo a esto, un nodo se considera equivalente a otro si tienen la misma IP y el mismo *timeStamp*. Se ha implementado la función con signatura `Node.equals(Node other)` teniendo en cuenta esto para simplificar el diseño de los algoritmos y poder utilizar los métodos de modificación y búsqueda implementados en las colecciones Java.

El atributo *lastSeen* de la clase *Node* es utilizado para que los nodos inactivos durante un cierto periodo de tiempo sean eliminados de las listas de usuarios. Este atributo no se tiene en cuenta en evaluación de la función `Node.equals(Node other)`.

Clase User

La clase *User* contiene un objeto del tipo *Node* y los atributos necesarios para que Visual DNS pueda realizar sus funciones: el nombre del usuario, su imagen y los servicios que ofrece.

Se establece una correlación directa entre la clase *Node* y la clase *User*, pues, como se ha comentado, si se cambia algún parámetro de la configuración, se cambia el campo *lastSeen* del objeto de tipo *Node*, considerándose a todos los efectos un usuario nuevo. La función `User.equals(User other)` refleja esta relación uno a uno, considerando que un usuario equivale a otro si y sólo si sus nodos resultan iguales (*i.e.* el método `user.equals(other)` devuelve *true* si y sólo si `user.getNode().equals(other.getNode())` devuelve *true*).

La clase *User* implementa la interfaz *Parcelable*. La clase *Parcel* de Android es utilizada como un mecanismo de comunicación entre procesos, que resulta más eficiente que la serialización y posterior desserialización del objeto que se desea comunicar.

Clase NodeList

La clase *NodeList* representa una lista de nodos y se utiliza principalmente para facilitar la conversión de las listas a formato XML.

Internamente implementa la lista como una *ArrayList*. Además de los métodos de acceso y modificación habituales de las colecciones de Java, implementa el método `NodeList.addAllAbsent(List<Node> nodeList)`, que añade a la *NodeList* los elementos de la lista recibida como parámetro que no se encuentran en la del objeto sobre el que se hace la llamada.

Clase UserList

Finalmente, la clase *UserList* modela las listas de usuarios. Internamente, esta clase representa la lista como un objeto de tipo *CopyOnWriteArrayList* [26]. La clase *CopyOnWriteArrayList* es una variante de *ArrayList* en la que todas las operaciones que modifican la lista se realizan sobre una copia nueva de ésta, que sólo será visible a los *threads* lectores cuando el *thread* modificador la deje en un estado consistente. De este modo se solucionan los posibles problemas derivados del acceso concurrente a la lista.

Además, la clase *UserList* encapsula algunas de las funcionalidades requeridas por los algoritmos de descubrimiento de vecinos. Implementa, entre otros:

- `int removeIdleUsers(int limitInSecs)`, que elimina de la lista los usuarios que no han sido vistos durante un periodo de tiempo determinado por el parámetro *limitInSecs*, devolviendo el número de usuarios eliminados.

- `ArrayList<User> mergeLists(NodeList receivedNL)`, utilizado a la recepción de cada paquete UDP para comprobar si la lista de nodos recibida contiene nodos desconocidos y actualizar el campo *lastSeen* de los conocidos, según sea necesario. Devuelve una lista de usuarios construida con los nodos desconocidos para su posterior utilización en la petición de información de cada nodo.

- `List<Node> getNodes()`, que devuelve una lista de los nodos de los usuarios contenidos en la *UserList*.

Serialización

Si se desea que se pueda establecer una EasyMANET en la que puedan participar distintos tipos de dispositivos (smartphones y PCs) con diversos SO (Android, iOS, BlackBerry OS, Windows, Linux y MacOS), no podemos confiar en los mecanismos propios del lenguaje utilizado en cada plataforma para la serialización de objetos. Por esta razón, pensando en una

posible estandarización de la comunicación entre nodos para hacerla independiente de la implementación, se optó por la utilización de mensajes en el formato XML. No son, por supuesto, el objeto y el alcance de este proyecto suficientes para proponer tal estándar. Aun así parece interesante adoptar un formato que pueda ser fácilmente adaptable a ese posible futuro estándar. Atendiendo a este objetivo se diseñó la comunicación entre nodos como se explica a continuación.

Las clases *NodeList* y *UserList* son utilizadas a efectos de comunicación entre nodos de la red. La clase *NodeList* se utiliza en la retransmisión mediante *broadcast* de la lista de nodos conocidos por cada uno. La clase *UserList* se utiliza tanto en el envío de las peticiones de la información adicional relativa a un usuario como en el de las respuestas.

Para la construcción de las peticiones y la interpretación de las respuestas se ha utilizado la librería Simple [27]. Esta librería permite la conversión de clases Java en documentos de tipo XML, y viceversa, de una manera relativamente sencilla. Para ello se utiliza un sistema de anotaciones Java, con el que se indica qué atributos, de las clases que nos interesa convertir, se incluirán en el XML generado.

Se tuvo un problema debido a las distintas codificaciones de fechas que hacen las diversas versiones de Android y la interpretación que de ellas que hace Simple. Para solventarlo se quitó del formato de fechas la zona horaria, que era la parte que ocasionaba los errores, permitiendo a Simple interpretarlas correctamente. Esto no debería ser un problema, ya que en una red *ad hoc* podemos suponer que todos los dispositivos estarán en la misma zona horaria.

3.2.2. Soporte a Visual DNS

La funcionalidad asociada al servicio Visual DNS de la aplicación se ha implementado como un *Service* de Android en la clase *DiscoveryService*. Es un servicio del tipo *Foreground Service* por dos motivos. Primeramente porque se pretende que el SO lo mantenga en memoria todo el tiempo posible. Si se detiene y se vuelve a iniciar el servicio, se destruiría la lista de usuarios conocidos, con lo que sería necesario volver a obtenerla, y, además, el usuario propio contendría un nuevo *timeStamp* con lo que dejarían de funcionar todos los servicios iniciados con algún usuario porque los demás lo considerarían un usuario nuevo. El segundo motivo es que los servicios en primer plano informan al usuario, mediante una notificación, de que se están ejecutando. Esto resulta conveniente, pues el continuo uso de la Wi-Fi que hace la aplicación resulta en un consumo de la batería del dispositivo muy notable. Se pretende evitar así que el usuario incurra involuntariamente en ese consumo en el caso de que olvidara desactivar el servicio, al pasar la aplicación a segundo plano.

A continuación se introducirá el ciclo de vida del *DiscoveryService*. En los siguientes puntos se detallarán las clases de las que hace uso para realizar sus funciones: *AnnounceThread*, *DiscoveryThread* y *TCPServer*.

Ciclo de vida de DiscoveryService

Cuando se crea el servicio, en la llamada a la función *onStartCommand* de su ciclo de vida, se realizan varias acciones. Primeramente se obtiene un WifiManager.WifiLock [28], para mantener la interfaz Wi-Fi en funcionamiento, incluso aunque el dispositivo entre en estado de bajo consumo. A continuación se intentan obtener los parámetros básicos de red, IP y máscara de subred. Si no se logran obtener, el servicio se detiene. Si se logran obtener, se genera el

objeto de tipo *Node* que representará al nodo propio. Después, se crea el objeto de tipo *User* a partir del nodo y las preferencias del usuario. Se crea la lista de usuarios conocidos de tipo *UserList* y se le añade el usuario propio. Tras ello, se crean e inician los *threads* encargados de realizar las tareas encomendadas a Visual DNS: *AnnounceThread*, *DiscoveryThread* y *TCPServer*. Finalmente se inician los servicios que el usuario eligió poner a disposición de los demás usuarios de la MANET.

El método *onBind* simplemente devuelve un objeto del tipo Binder. Este objeto es después pasado a la función onServiceConnected, que se ejecuta en la *activity*.

El método *onUnbind* limpia todos los posibles mensajes pendientes en la cola del *handler* de la activity. Tras ello, devuelve una constante que indica al sistema que está permitido volver a enlazar al *service* después de que *onUnbind* haya sido llamado.

Por último, *onDestroy* es llamado a la detención del *service*. Este método se encargará de detener ordenadamente todos los *threads* lanzados en el método *onStartCommand* y liberar todos los recursos en uso.

```
int timesRunned = 0;
NodeList nodeList = new NodeList();
byte[] buf;
DatagramPacket packet;
while (!Thread.currentThread().isInterrupted()) {
  // Eliminación de usuarios inactivos.
  if (++timesRunned % REMOVE_IDLE_USERS_FREQUENCY == 0 &&
    getUsersList().removeIdleUsers(USERS_TIMEOUT) > 0 &&
    getMainActivityHandler() != null) {
      getMainActivityHandler()
        .obtainMessage(MainActivity.NEW_USERS).sendToTarget();
  }
  // Construcción de la lista de nodos
  nodeList.clear();
  mSelfNode.setLastSeen(new Date());
  nodeList.add(mSelfNode);
  nodeList.addAllAbsent(getUsersList().getNodes());
  // Serialización con Simple
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  serializer.write(nodeList, out);
  buf = out.toByteArray();
  // Broadcast del packet
  packet = new DatagramPacket(buf, buf.length, myBcastIA, PORT);
  mSocket.send(packet);
  // Espera entre envíos
  AnnounceThread.sleep(TIME_BETWEEN_ANNOUNCES);
}
```

Figura 13. Código del método `AnnounceThread.run()`.

Clase AnnounceThread

La clase *AnnounceThread* se encarga de construir una lista de nodos a partir de la lista de usuarios conocidos, eliminando en el proceso los usuarios inactivos, convertirla a formato XML y de retransmitirla mediante *broadcast* de un paquete UDP a los vecinos situados a sólo un salto de distancia. Después espera un tiempo determinado, antes de volver a iniciar todo el ciclo. Si la lista de usuarios es modificada como resultado de este proceso, se notifica a la interfaz gráfica haciendo uso del *handler* de la *MainActivity*.

En la figura 13 se muestra el método `AnnounceThread.run()` que implementa su funcionalidad, omitiendo parte de la lógica para el tratamiento de interrupciones y excepciones.

Clase DiscoveryThread

La clase *DiscoveryThread* implementa el descubrimiento de los usuarios presentes en la red. De manera resumida, se encarga de la recepción de las listas de nodos enviadas por los otros dispositivos, compararlas con la propia utilizando el método `ArrayList<User> mergeLists(NodeList receivedNL)`, emitir las peticiones necesarias vía conexión TCP a los remitentes para obtener la información y las imágenes de los usuarios, y, cuando modifica la lista de usuarios conocidos, comunicárselo a la interfaz de usuario a través del *handler*.

El código del método *run* de la clase *DiscoveryThread* es muy sencillo por lo que no se comentará. Merece la pena, no obstante, detenernos en los métodos *obtainUsersInfo* y *obtainUsersImage*, de los que *run* hace uso para obtener la información y las imágenes de los usuarios, respectivamente.

```java
private UserList obtainUsersInfo(InetAddress from, List<User> newUsers) {
  UserList userListResponse = null;
  Socket socket = null;
  PrintWriter out = null;

  socket = new Socket(from, DiscoveryService.PORT);
  socket.setSoTimeout(10000);

  out = new PrintWriter(socket.getOutputStream(), true);
  String petitionType = new String("GET_USERS_INFO"+"\r\n");
  out.print(message);
  UserList usersUL = new UserList();
  usersUL.getUsers().addAll(newUsers);
  // Envía lista de usuarios solicitados
  serializer.write(usersUL, out);
  // Recepción de respuesta con info básica de usuarios
  InputStreamReader isr = new InputStreamReader(socket.getInputStream());
  BufferedReader br = new BufferedReader(isr);
  userListResponse = serializer.read(UserList.class, br, false);
  return userListResponse;
}
```

Figura 14. Código del método `DiscoveryThread.obtainUsersInfo()`.

La figura 14 ilustra el código del método *obtainUsersInfo*, omitiendo el tratamiento de excepciones y también el cierre de *streams* y *sockets*. A continuación se desglosará su funcionamiento.

El método *obtainUsersInfo* recibe como parámetros una *InetAddress* y la lista de los usuarios de los que se desea obtener la información, en formato *List* por cuestiones de eficiencia. Primero crea una conexión de tipo TCP y establece un *timeout*. A continuación envía la primera parte de la petición, en la que se indica el tipo de petición: información de usuarios. Genera a partir de la *List<User>* recibida un objeto de tipo *UserList*, necesario para la correcta serialización y útil para la posterior desserialización en objetos del tipo *User*, que se pueden buscar directamente en la lista de usuarios conocidos del otro extremo de la conexión. La línea `serializer.write(usersUL, out)` efectúa la serialización de la lista y la envía. Finalmente la línea `userListResponse = serializer.read(UserList.class, br, false)` recibe la respuesta en forma de UserList y la asigna a la variable que será el retorno del método.

La figura 15 ilustra el código del método *obtainUsersImage*, una vez más, omitiendo el tratamiento de excepciones, y el cierre de *streams* y *sockets*. A continuación se desglosará su funcionamiento.

El método *obtainUsersImage*, funciona de modo similar a *obtainUsersInfo* con la salvedad de que efectúa una iteración sobre la lista de usuarios recibida para realizar la petición de cada imagen de usuario de manera individual.

```
private void obtainUsersImage(InetAddress from, UserList newUsers) {
  Iterator<User> it = newUsers.iterator();
  Socket socket = null;
  Scanner in = null;
  PrintWriter out = null;
  while (it.hasNext()) {
    // Petición
    User current = it.next();
    socket = new Socket(from, DiscoveryService.PORT);
    socket.setSoTimeout(10000);
    in = new Scanner(socket.getInputStream());
    out = new PrintWriter(socket.getOutputStream(), true);
    // Se envía tipo de petición
    String message = new String("GET_USER_IMG"+"\r\n");
    out.print(message);
    // Se envía usuario del que se solicita la imagen
    serializer.write(current, out);
    if (in.nextLine().equals("OK")) {
      // Recibimos la imagen
      Bitmap userImg = BitmapFactory.decodeStream(socket.getInputStream());
      current.setImgBmp(userImg);
    } else {
      newUsers.remove(current);
    }
  } // end while
}
```

Figura 15. Código del método `DiscoveryThread.obtainUsersImage()`.

Clase TCPServer

La clase *TCPServer* implementa la recogida de las peticiones relativas a la información de los usuarios y sus imágenes, y su posterior envío. Para ello se han definido dos clases privadas: *ServerThread*, que extiende *Thread* y la clase *TCPServerTask*, que implementa la interfaz *Runnable*.

La clase *ServerThread* implementa un servidor TCP de tipo concurrente, apoyándose para ello en la clase *ThreadPoolExecutor* de la biblioteca Java con una *pool* de *threads* variable según el número de *cores* disponibles. Su único cometido es la creación de un *ServerSocket* para aceptar las conexiones entrantes, y el envío de una *TCPServerTask* al *Executor* para cada una de ellas.

La clase *TCPServerTask* gestiona las peticiones al servidor. Para ello, recibe el *socket* creado por el *serverSocket* tras cada conexión. La primera línea de cada petición indica si se trata de una petición de información de usuarios o de una imagen.

Si lo que se recibe es una una petición de usuarios, se utiliza Simple para la desserialización del documento XML recibido tras la línea de tipo de petición, en un objeto de tipo *UserList*. A continuación se genera una lista de respuesta que contendrá los usuarios de los que se disponga información ampliada. De nuevo, se utiliza Simple para la conversión de la lista en un documento XML y se envía al destinatario.

En el caso de que la petición sea de una imagen de usuario el procedimiento es similar, pero con cada petición se recibe únicamente un usuario. Cada respuesta, pues, contiene únicamente o bien una línea de aceptación, en caso de encontrarse disponible la imagen del usuario solicitado, y a continuación la imagen en cuestión, o bien una línea que indica que no se posee la imagen. En la figura 16 se esboza el código Java del método *sendUserImage* utilizado para realizar esta tarea.

```
private void sendUserImage() {
  out = new PrintWriter(clientSocket.getOutputStream(), true);
  User userRequested;
  // Recibe el usuario del que se solicita la imagen
  userRequested = serializer.read(User.class, br, false);
  // Se busca el usuario en la lista de usuarios conocidos
  User userToSend =
getDiscoveryService().getUsersList().getUser(userRequested);
  if (userToSend != null) { // Si se encuentra
    out.println("OK");
    // Se envía la imagen del usuario solicitado.
    userToSend.getImgBmp().compress(CompressFormat.PNG, 100,
clientSocket.getOutputStream());
  } else {
    out.println("NOT_FOUND");
  }
}
```

Figura 16. Código del método `TCPServerTask.sendUserIamge()`.

Como se ha visto, en el protocolo de peticiones y respuestas diseñado, los clientes obtienen la información de los usuarios desconocidos en varias fases. Primero, deben obtener la información básica de los usuarios y, después, obtener las imágenes, de una en una. El principal

motivo por el que se optó por este diseño tiene que ver con la agilidad de la aplicación a la hora de mostrar nuevos usuarios encontrados. Este protocolo permite que se pueda actualizar la interfaz de usuario en cuanto se dispone de la información mínima para ello (nombre y servicios), para después incluir las imágenes de los usuarios a medida que se vayan recibiendo. En redes pequeñas esta diferencia puede resultar trivial, pero en una red en la que hubiera decenas de nodos conectados permite mejorar mucho la experiencia de usuario. Dicho esto, en la actual implementación de EfA, aunque el protocolo y el servidor están diseñados con todo esto en mente, la parte cliente no se comporta de modo que pueda aprovechar esta ventaja, por motivos en los que abundaremos en el apartado de conclusiones.

3.2.3. Servicios

De entre los servicios que puede ofrecer una MANET se ha elegido implementar a modo de ejemplo un *chat* basado en texto. Dado que el objetivo principal del presente proyecto es la implementación de la plataforma EasyMANET más que la de servicios concretos, se ha optado por un modelo sencillo.

Servicio de chat

El *chat* implementado sólo permite conversaciones entre pares. Además, se pueden aceptar conexiones con más de un cliente, pero la interfaz no permite pasar cómodamente de una sesión a otra. Se evita de este modo las complicaciones de la gestión de la posible destrucción de las *activities* y su posterior reconstrucción, que supondría implementar la persistencia de las conversaciones y las notificaciones cuando se recibiera un mensaje en una sesión que no es la que se encuentra en primer plano, entre otras. Cuando un usuario abandona la *activity* mediante la pulsación del botón atrás, se cierra el *socket* y se pierde la conversación.

Para respaldar este servicio se ha implementado un servidor TCP en la clase *TCPChatServer*. Este servidor, tras recibir una conexión entrante, comprueba si la IP corresponde a un usuario conocido. A continuación genera un *AlertDialog* [29] para obtener una confirmación de que el usuario quiere iniciar una conversación. En caso de que el usuario responda afirmativamente, se inicia una *ChatActivity* pasándole el *socket* obtenido en la conexión. En caso contrario, se cierra el *socket*.

En otros entornos, el paso del *socket* desde el *backend* a la GUI, bien mediante constructor o mediante un *setter*, no supondría mayor complicación. Sin embargo, el mecanismo utilizado por Android para iniciar las *activities* hace que la aproximación clásica no sea viable.

Como ya se ha visto, Android utiliza los *intents* como mecanismo universal para comunicar al sistema la intención de iniciar acciones. Así, para iniciar una *activity* se debe crear un objeto de la clase *Intent*. A este objeto se le pueden añadir parámetros que después pueden ser consultados en alguno de los métodos de inicio del ciclo de vida de la *activity*. El problema es que, por cuestiones de eficiencia, estos parámetros sólo pueden ser de los tipos primitivos de Java, Strings u objetos de clases que implementen las interfaces *Serializable* o *Parcelable*. La clase *Socket* no cumple estos requisitos.

La solución adoptada para este problema pasa por el uso de variables globales. Para ello se define una clase cuyo nombre debe indicarse en la etiqueta `<application>` del AndroidManifest.xml, en nuestro caso, EasyMANET. Esta clase, que extiende Application [30], se instancia cuando se crea el proceso de la aplicación. Cualquier clase que derive de `la`

clase Context [31], como las *activities* o los *services*, o tenga una referencia a ella, puede obtener el objeto global de la aplicación haciendo uso del método `Context.getApplicationContext()`, para acceder a sus métodos y variables.

Otros servicios

Aunque sólo se ha implementado la funcionalidad del *chat*, la clase *User*, el *DiscoveryService* y la GUI se han diseñado de tal manera que la aplicación pueda ser ampliada fácilmente para la provisión de nuevos servicios.

Los servicios disponibles se transmiten entre los usuarios como un mapa clave-valor. Posteriormente este mapa es consultado por la GUI en la generación dinámica de menús tras una pulsación larga en la imagen de un usuario en la MainActivity. El menú generado presenta una entrada por cada servicio prestado por el usuario en cuestión.

Para demostrar esto, en el menú de configuración de la aplicación, se permite al usuario seleccionar también un servicio de intercambio de archivos. No obstante, si el usuario intenta iniciar un intercambio, se le informará de que tal funcionalidad no está disponible.

3.3. Frontend

3.3.1. MainActivity

La MainActivity es la primera que aparece al usuario al arrancar la aplicación. En la parte superior de la *activity*, se muestra la información del usuario, junto con un botón para iniciar y detener los servicios de EasyMANET. La parte inferior contiene una GridView [32] que está compuesta por ítems que se disponen en una matriz bidimensional deslizable. Cada uno de los ítems representa a un usuario componente de la MANET.

El diseño de la MainActivity se define en un archivo llamado activity_main.xml, en el que se especifican los elementos que la componen y su disposición en la pantalla del dispositivo. En este archivo se incluyen todas las *views* [33] que componen la *activity*.

La parte de la información de usuario está formada por varias *views*. Entre ellas se encuentra la que contiene la imagen seleccionada por el usuario para ser usada por el servicio Visual DNS, así como las que informan del nombre de usuario seleccionado, y la dirección IP asignada al dispositivo al conectarse a la red. A la derecha de la información de usuario se encuentra un ToggleButton [34], que sirve para iniciar y detener el DiscoveryService.

En el archivo activity_main.xml, sólo se definen las características generales de la *gridview* de la parte inferior (su posición y la disposición relativa de los ítems que la componen), pero no los ítems en sí mismos. Para la especificación del diseño de un ítem se utiliza otro archivo, user_layout.xml, que se usará en el *adapter*, para la generación de cada *view* dinámicamente.

La funcionalidad asociada a la MainActivity se implementa en una clase que extiende de Activity. Como se ha visto, las *activities* reciben *callbacks* a ciertos métodos en respuesta a sus cambios su estado según su ciclo de vida. Además de estos métodos, que toda clase que extienda de Activity puede implementar, MainActivity también implementa otros en respuesta a las acciones del usuario y su relación con el DiscoveryService. A continuación, se describirá el modo en que operan algunos de estos métodos.

Para configurar su imagen de usuario, éste debe tocar en la propia imagen situada en la zona de información. Como respuesta a ello, la EfA pide al sistema que abra una aplicación para la toma de fotografías, para después recoger el resultado y enviarlo a otra *activity*, CropImageActivity, que permitirá al usuario seleccionar qué zona, cuadrada, de la fotografía tomada desea usar.

Para ello, se construye un *intent* que indica al sistema el deseo de obtener una fotografía de la cámara. Este *intent* es pasado como argumento al método `Activity.startActivityForResult(Intent intent, int requestCode)`, que iniciará la *activity* encargada de tomar la fotografía. Cuando el usuario finalice, MainActivity recibirá un *callback* al método `onActivityResult(int requestCode, int resultCode, Intent data)`. Los parámetros con que es llamado este método nos indican para qué se llamó a la *activity* que está devolviendo su resultado, si éste es válido (para comprobar, por ejemplo, si el usuario canceló la acción) y un *intent* con información sobre el resultado en sí mismo. Si el `resultCode` indica que la acción se realizó con éxito y el `requestCode` corresponde al de la toma de imagen, se utiliza la dirección física de la imagen tomada para pasársela de nuevo a una llamada a `startActivityForResult`, esta vez para iniciar la CropImageActivity, que, a su vez producirá otra llamada a `onActivityResult`. En esta segunda llamada recogeremos, por fin, la imagen que se usará para el usuario, guardándola en memoria persistente e informando a las preferencias de su situación.

MainActivity implementa la interfaz SharedPreferences.OnSharedPreferenceChangeListener. Para ello define el método `onSharedPreferenceChanged (SharedPreferences sharedPreferences, String key)`, que es llamado cuando una preferencia de la aplicación es modificada. En este método se aplican las modificaciones a la interfaz pertinentes. Además, como ya se indicó, un usuario es considerado como un nuevo usuario a todos los efectos si cambia alguno de sus parámetros básicos. Por tanto, este método comprueba, además, si el DiscoveryService está en ejecución y, si es el caso lo detiene, para que, cuando se reinicie el objeto de tipo User, que representa al propio usuario, éste contenga los nuevos parámetros seleccionados.

Como se explicó en el apartado dedicado a los *services* de Android, cuando el usuario inicia el DiscoveryService, no sólo lo arrancamos sino que también enlazamos la Activity con él pasándole un objeto de tipo anónimo que implementa la interfaz ServiceConnection. Esta interfaz define el método onServiceConnected(ComponentName name, IBinder service), que es llamado tras un *binding* correcto. En este método, utilizamos el IBinder recibido para solicitar al DiscoveryService la lista de usuarios conectados para pasársela al *adapter* y notificarle que debe refrescar la GridView. Además, para permitir al DiscoveryService comunicarse con la *activity* cuando encuentra nuevos usuarios, se le pasa un objeto de tipo Handler.

Si el usuario selecciona algún ítem de la GridView que no sea el que representa al propio usuario, se inicia una sesión de *chat* con el usuario seleccionado, lanzándose para ello la ChatActivity.

3.3.2. ChatActivity

La ChatActivity proporciona la interfaz y la funcionalidad para prestar el servicio de *chat*. Está compuesta por una barra de información en la parte superior que contiene la imagen y el nombre del usuario, una parte central deslizable donde se insertan los mensajes enviados y

recibidos, y una barra inferior que contiene un botón para enviar el texto introducido en el campo habilitado a tal efecto. Este diseño queda definido en el archivo activity_chat.xml.

La parte funcional de la *activity* se implementa en la clase ChatActivity. Como se ha visto, existen dos maneras de llegar a ella: tras el usuario iniciar una sesión de *chat* o si responde afirmativamente a la petición de otro usuario de iniciarla. Si la inicia el propio usuario, deberá enviarse una petición y esperar respuesta. En el otro caso, cuando se inicia la *activity*, ya se dispone de un *socket* abierto hacia el otro usuario, por el cual se envía la respuesta afirmativa.

Se han definido tres AsyncTasks para gestionar la petición de conexión, la gestión de peticiones de inicio de sesión, y el posterior intercambio de mensajes a través del *socket*. Tras crearse la *activity*, si la sesión de *chat* ha sido iniciada por el usuario local, se lanza una SendPetitionTask. En caso contrario se obtiene el *socket* desde la clase EasyMANET (como se adelantó en la explicación del TCPChatServer) y se asigna a un miembro de la clase ChatActivity. A continuación, en todos los casos, se lanza una WaitConnectionTask. Si la WaitConnectionTask finaliza con éxito se lanza una SocketListeningTask, y en caso contrario se finaliza la *activity*. Ahora se procede a describir el funcionamiento de estas tres clases, derivadas de AsyncTask.

La clase SendPetitionTask, en su método `doInBackground`, recibe el objeto User que representa al otro usuario. Utilizando la información contenida en el Node del User, crea un *socket* a la dirección IP en al puerto de escucha del TCPChatServer. A continuación envía la petición y espera la respuesta del otro usuario por el *socket*. Si el usuario remoto responde negativamente a la petición, o si ocurre algún problema de conexión, se cierra el *socket* creado y se pone a *null*. Durante todo el proceso se publican resultados parciales, de manera que el método `onProgressUpdate` actualice la interfaz para informar al usuario. Al final, el método devuelve el *socket*. El método onPostExecute recoge el *socket* y, si es *null*, informa al usuario. En caso de ser válido, el *socket* se asigna a la variable de la clase ChatActivity.

La clase WaitConnectionTask, simplemente comprueba periódicamente si la variable miembro de tipo *Socket* de la clase ChatActivity contiene un *socket* válido. Si, pasado un cierto periodo de tiempo, no se obtiene un *socket* válido se cierra la *activity* informando al usuario. Si, por el contrario, se consigue obtener el socket, se lanza la SocketListeningTask.

SocketListeningTask simplemente escuchará en segundo plano los mensajes provenientes del *socket*, publicándolos en la GUI a medida que vayan llegando, hasta que uno de los dos usuarios cierre la activity y, con ello, la conexión.

3.3.3. SettingsActivity

En la SettingsActivity el usuario puede seleccionar su nombre y los servicios que desea ofrecer a los demás usuarios. La primera vez que se ejecuta la aplicación en el dispositivo, el usuario es conducido a ella antes que a la MainActivity. En sucesivas ocasiones, se puede acceder a ella desde la MainActivity, tras pulsar el botón físico de opciones y seleccionar la opción adecuada.

La SettingsActivity no es una extensión directa de Activity, sino de PreferenceActivity [35], que ha sido diseñada para facilitar la tarea de mostrar la jerarquía de preferencias al usuario. Para ello, se declaran, en un archivo XML, los tipos de preferencias que deseamos (una

EditTextPreference [36] y varias CheckBoxPreference [37], en nuestro caso), agrupadas en categorías. Para cada preferencia se consignan una clave única, título, resumen y demás atributos que se deseen definir.

Después, en el método onCreate de la clase SettingsActivity, este XML es utilizado en la línea `addPreferencesFromResource(R.xml.preferences)`. Con esta simple línea de código se consigue que la interfaz se construya dinámicamente y que la jerarquía de preferencias consignada se añada a la jerarquía de la *app*. No sólo se construye la parte visual de la interfaz, sino también el comportamiento: si el usuario modifica alguna de las preferencias, la *activity* se encarga de guardarlas sin que el programador tenga que escribir el código para ello. Cada una de las preferencias puede ser consultada, utilizando su clave única, desde cualquier otro componente con acceso al contexto de la aplicación.

En caso de que el usuario acceda a la *activity* de configuración mientras tiene los servicios de EasyMANET activos, se le informa de que se desactivarán si modifica algún parámetro. En la figura 17 se muestra el contenido del archivo preferences.xml utilizado en EasyMANET.

```xml
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory android:title="@string/prefs_profile_options">
  <EditTextPreference
      android:key="userName"
      android:title="@string/prefs_user_name"
      android:summary="@string/prefs_user_name_summary" />
  </PreferenceCategory>
  <PreferenceCategory android:title="@string/prefs_services_offered">
  <CheckBoxPreference
    android:key="chat"
    android:title="@string/prefs_chat_title"
    android:summary="@string/prefs_chat_summary" />
  <CheckBoxPreference
    android:key="fileExchange"
    android:title="@string/prefs_files_title"
    android:summary="@string/prefs_files_summary" />
  </PreferenceCategory>
</PreferenceScreen>
```

Figura 17. Archivo preferences.xml.

4. Validación

En esta sección se procede a validar la aplicación. Para ello, en primer lugar, se describe el entorno en el que se han realizado las pruebas de validación. A continuación se estudia en *workflow* de la aplicación para comprobar su correcto funcionamiento. Por último se detallan algunos problemas detectados, sus causas y las posibles soluciones.

4.1. Entorno de pruebas

A pesar de que la aplicación desarrollada en el presente proyecto está diseñada para trabajar en redes *ad hoc*, durante el desarrollo y las pruebas de validación se ha utilizado un *router* para proveer de direcciones IP y para realizar las tareas de encaminamiento de paquetes. Esto fue así porque, aunque EasyMANET como plataforma ofrece métodos para obtener estos servicios sin la necesidad de infraestructura, estos métodos no estaban implementados en los dispositivos que se han utilizado. Dicho esto, la aplicación funcionaría correctamente en el entorno de una MANET. Como se explicó en el capítulo 2, cada capa del modelo OSI es independiente entre sí. Esto significa que se pueden substituir los algoritmos y métodos empleados en una capa, sin que ello afecte a las capas superiores, mientras se les provea de los mismos servicios. Así pues, en lo que a la aplicación desarrollada respecta, es indiferente el modo en que se opere en el nivel de red.

Los teléfonos móviles utilizados durante el desarrollo de la aplicación han sido un Samsung Galaxy S Plus y un HTC Desire, ambos con la versión 2.3.7 de Android que corresponde con la API 9. La aplicación se ha desarrollado utilizando la API 8 de Android, por lo que la versión mínima del SO que debe llevar instalada un dispositivo para poder ejecutarla es la 2.2. Esto es así porque no se ha utilizado ninguna de las nuevas funcionalidades de la API 9.

Para comprobar la compatibilidad de la aplicación con versiones posteriores de Android y que funciona correctamente con más de dos dispositivos en la red, en las pruebas de validación se ha utilizado también un móvil LG Optimus G con la versión 4.1.2 de Android y una *tablet* Asus Nexus 7 (1ª generación) con la versión 4.3.

4.2. Workflow

A continuación se mostrará el funcionamiento de la aplicación, desde que el usuario la arranca por primera vez tras instalarla, hasta el uso del servicio de *chat*, pasando por todas las posibles situaciones que se pueden dar en el proceso.

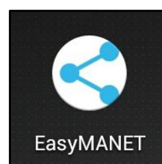

Figura 18. Icono de la aplicación.

Cuando el usuario pulsa en el icono de la aplicación (figura 18), si ésta es la primera ocasión en que lo hace, se le dirige directamente a la *activity* de configuración y se le muestra un mensaje indicándole lo que debe hacer (figura 19-izq). En el resto de ocasiones, se le mostrará directamente la MainActivity y se utilizarán las preferencias guardadas.

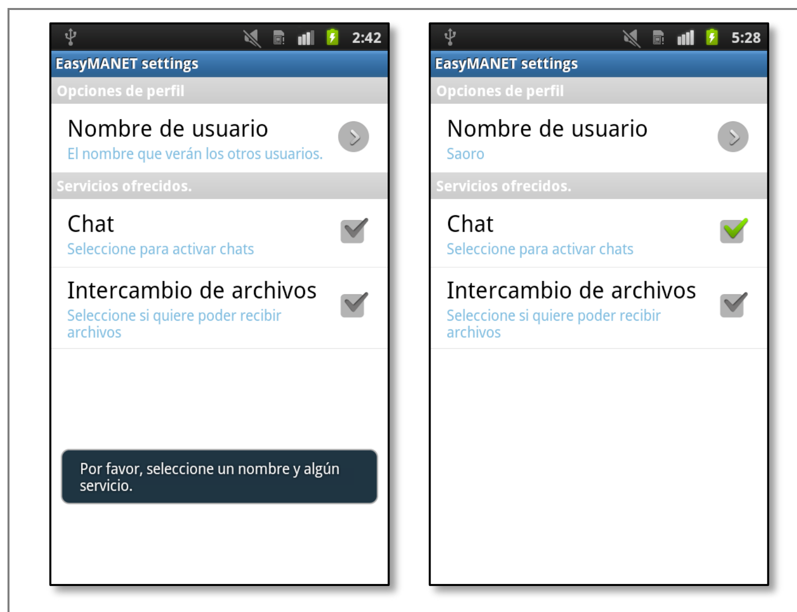

Figura 19. Izq: SettingsActivity tras abrir por primera vez la *app*.
Der: SettingsActivity tras configurar los parámetros.

En SettingsActivity (figura 19-der) el usuario puede elegir el nombre por el que será identificado por los demás usuarios de la red, así como los servicios que quiera poner a disposición de éstos.

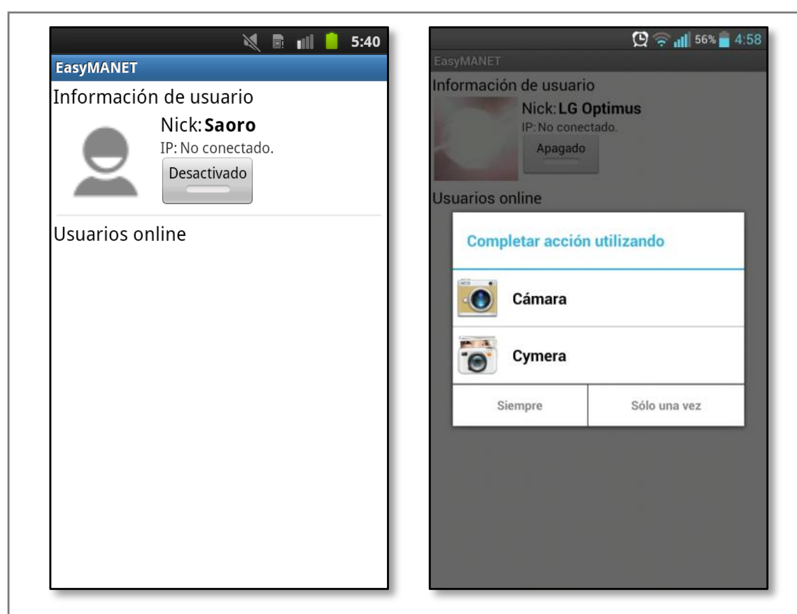

Figura 20. Izq: MainActivity con la imagen por defecto. Der: diálogo de selección de cámara.

Tras pulsar el botón atrás del dispositivo, se le dirige a la *activity* principal (figura 20-izq), en la que se mostrará el nombre seleccionado junto a la imagen de usuario configurada por defecto.

En este punto, el usuario puede configurar su imagen pulsando en la imagen predefinida. Si lo hace, el SO le dará a elegir entre las aplicaciones de cámara instaladas (figura 20-der). Tras seleccionarla, se abrirá la aplicación de su elección para tomar la imagen. Si acepta la imagen tomada, ésta se usa en la *activity* de recortado. En CropImageActvity podrá elegir qué parte de la fotografía realizada quiere utilizar. Cuando quede satisfecho, puede pulsar el botón aceptar, lo que le llevará de nuevo a la MainActivity. Ésta reflejará los cambios efectuados. La figura 21 muestra capturas de pantalla de esta secuencia de acciones. Si el usuario cancela el proceso antes de que éste finalice, la imagen de aquél no será modificada

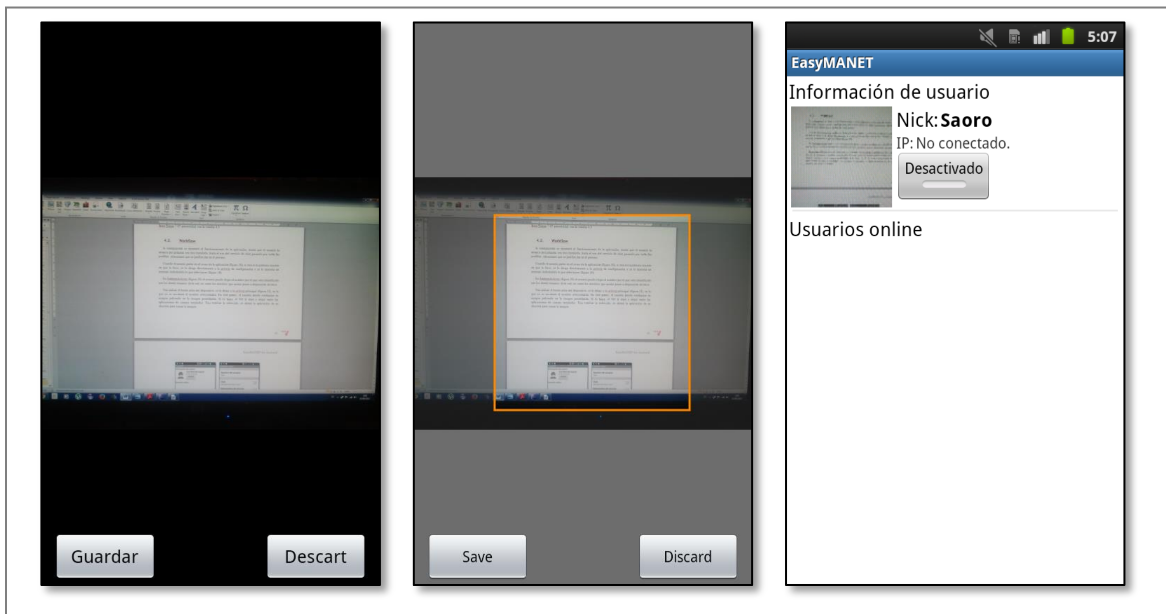

Figura 21. Secuencia de selección de imagen de usuario.

Una vez seleccionados todos los parámetros, el usuario puede pulsar el botón para iniciar EasyMANET. Al hacerlo, se mostrará la IP obtenida en la GUI y la imagen del usuario local en la zona “Usuarios Online”, y se situará una notificación en la barra superior. La aplicación comienza el protocolo de descubrimiento de nodos y, al cabo de un tiempo breve, aparecen en la parte inferior de la MainActivity los usuarios conectados a la MANET en ese instante (figura 22).

Si el usuario hace una pulsación larga sobre la imagen de otro, se le pregunta, a través de un cuadro de diálogo, qué servicio quiere iniciar de los que el usuario remoto tuviera disponibles. Como se ve en la figura 23, si el servicio seleccionado no es el de *chat*, simplemente se le informa mediante un mensaje *toast* de que la funcionalidad no está disponible.

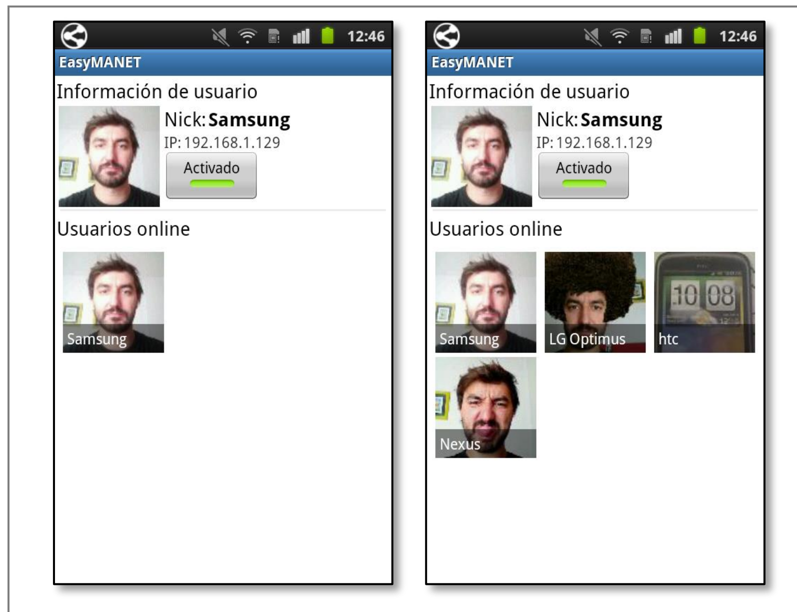

Figura 22. Puesta en marcha de Visual DNS.

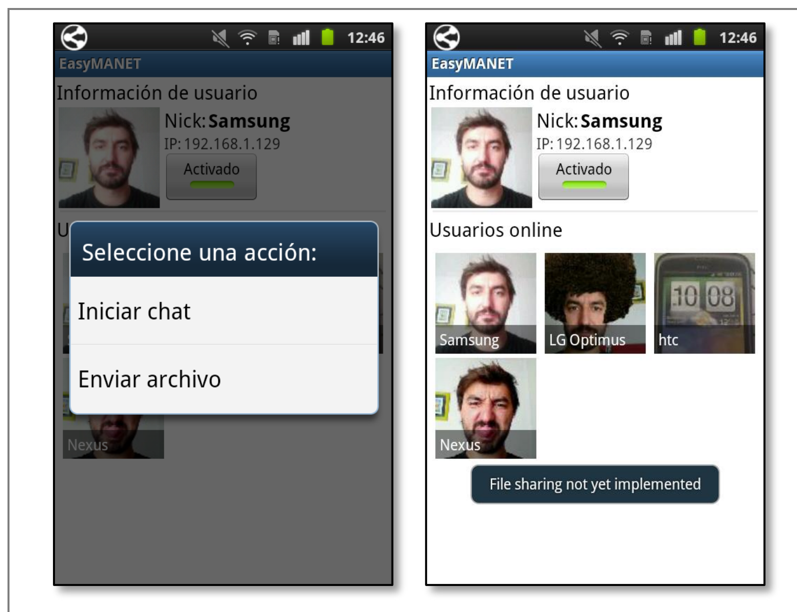

Figura 23. Izq: menú contextual para iniciar servicios.
Der: *toast* avisando de que un servicio no está disponible.

En caso de que el usuario seleccione la opción para iniciar un *chat* o, simplemente, haga una pulsación corta sobre la imagen de otro usuario, se abre la *activity* de chat y se envía al usuario remoto la petición de inicio. Mientras se espera la respuesta del otro usuario no se permite el envío de mensajes, como se puede ver en la figura 24-izq. Al recibir una petición de otro usuario, se muestra un cuadro de diálogo preguntándole al usuario local si desea iniciar una sesión de *chat* con el remoto (figura 24-der).

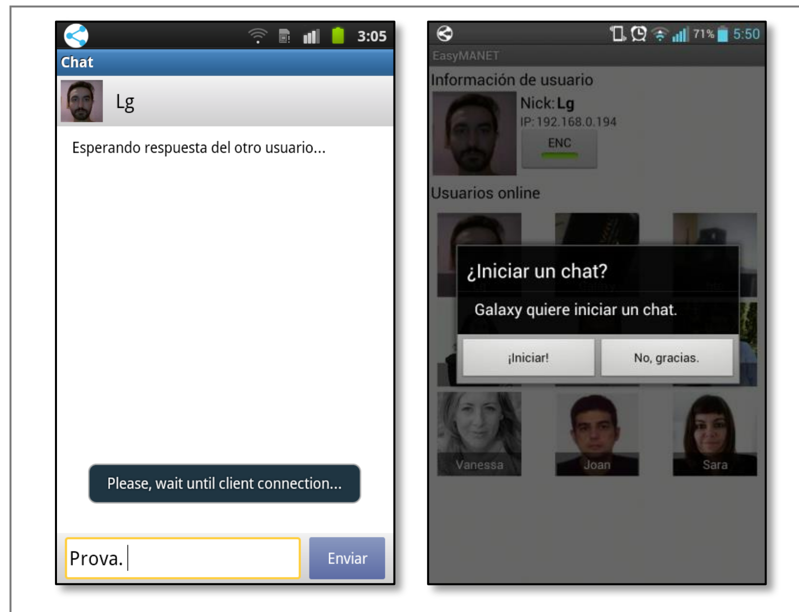

Figura 24. Izq: ChatActivity a la espera de conexión.
Der: Cuadro de diálogo de petición de *chat*.

En la figura 25 se muestra una secuencia de una sesión de *chat*, en la que, una vez establecida la conexión, ambos usuarios intercambian mensajes hasta que uno de los dos abandona la sesión mediante la pulsación del botón atrás, momento en el cual ya no se permitirá el envío de más mensajes en esa sesión.

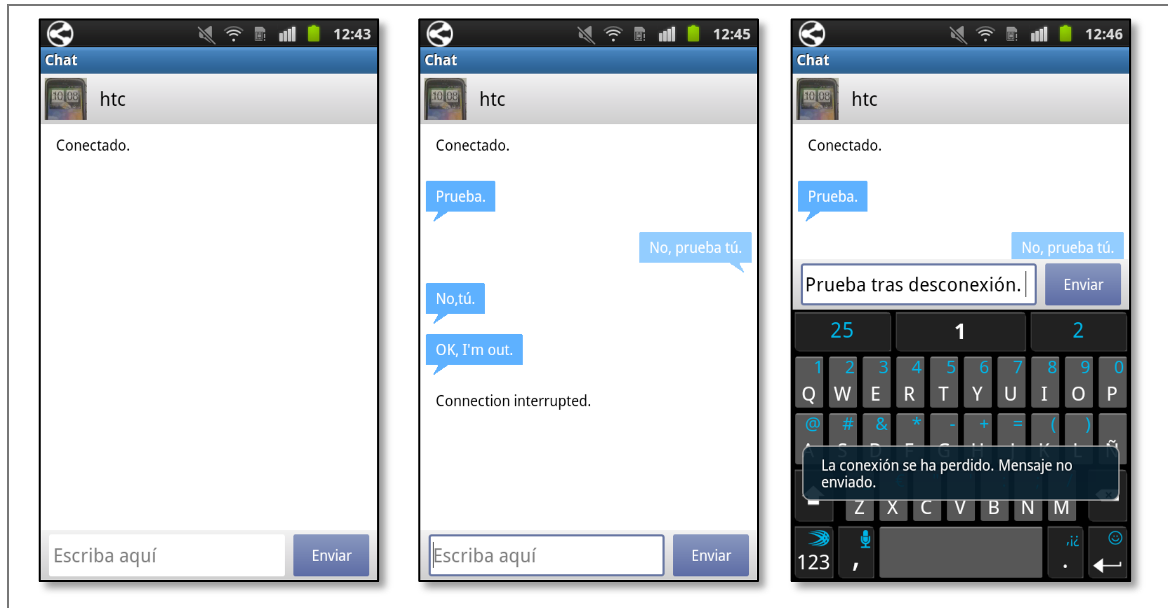

Figura 25. Sesión de *chat*.

4.3. Problemas

Durante las pruebas de validación se han detectado algunos problemas de funcionamiento. Si bien no son graves, estos problemas deberían ser solucionados si se quisiera poner a disposición del público la aplicación. Veámoslos a continuación.

Aunque falle el inicio del DiscoveryService, el botón de la MainActivity se queda conectado: cuando arranca, el DiscoveryService intenta obtener los parámetros de red. Si este proceso falla, el servicio se detiene automáticamente y alerta al usuario mediante un Toast (y el hecho de que la notificación de la barra superior no llega a mostrarse). Sin embargo, el service no tiene manera de notificarle a la MainActivity que se ha producido el error, porque en ese punto todavía no ha obtenido el Handler de la *activity*. Por tanto, el botón para iniciar y detener el *service* se queda conectado. Si se intentara detener el service estando ya detenido, la aplicación se cerraría súbitamente con un mensaje de error. Se evita esta situación porque se ha implementado lógica que comprueba si el servicio está en ejecución antes de intentar detenerlo, pero debería buscarse una alternativa para que el botón muestre el estado real del servicio en este caso.

La toma de imágenes desde la cámara provoca el cuelgue de la aplicación en algunas situaciones: en el móvil LG Optimus G, se produce una parada forzosa de la aplicación al volver de la CropActivity. Esto sucede por motivos que no han podido ser determinados, sólo en ocasiones, y sólo si se usa el software de cámara de LG. Sin embargo, cuando se vuelve a iniciar la aplicación, se observa que la imagen ha sido correctamente tomada y recortada, y está en uso.

A partir de un cierto número de usuarios conectados, la aplicación se cuelga: aunque este error no se ha producido durante las pruebas con cuatro dispositivos, puede llegar a darse. A continuación se detallan los motivos.

El sistema Android asigna una determinada cantidad de memoria a cada máquina virtual encargada de ejecutar una aplicación. Esta cantidad, que varía entre dispositivos, es relativamente baja, debido a que, habitualmente, los dispositivos móviles tienen poca RAM y muchas aplicaciones en ejecución concurrente.

En nuestra aplicación, se hace uso de gran cantidad de imágenes, lo que puede agotar rápidamente la memoria disponible. Para minimizar este problema, la CropViewActivity recorta la imagen de usuario tomada con la cámara, a cualquier resolución, hasta una resolución de 96x96 píxeles. Para guardarla en memoria persistente y transmitirla entre nodos, esta imagen es comprimida en formato PNG, lo que reduce su tamaño a aproximadamente 2 KB. No obstante, cuando recibimos la imagen de un usuario, ésta se guarda en memoria principal, en el objeto User correspondiente, en formato Bitmap. Además, cada objeto User se guarda en dos listas independientes, la del *service* y la del *adapter* de la GridView de la *activity* principal. Un objeto Bitmap para una imagen de 96x96 a 32 bits de color ocupa 36 KB. Por tanto, cada usuario conectado ocupa en memoria un mínimo de 72 KB. Por ejemplo, el teléfono HTC utilizado en las pruebas, tiene un tamaño de *heap* para cada aplicación de 48 MB. Es decir, antes de llegar a los 682 usuarios conectados, la aplicación se quedaría sin memoria. Un límite de 682 usuarios puede parecer razonable en una MANET, pero si se tiene en cuenta que la memoria no se usa sólo para almacenar los objetos User, y que la resolución elegida para las imágenes es pequeña, dadas

las densidades de píxeles de las pantallas de los dispositivos más modernos, se verá que lo que no parece un problema muy acuciante ahora, puede serlo en el futuro.

La solución a este problema pasaría por no guardar los objetos Bitmap directamente en el usuario. Las imágenes recibidas deberían guardarse en memoria secundaria, y ser colocadas en memoria principal sólo cuando fueran a ser visibles en la pantalla del dispositivo. Sin embargo, esta solución acarrea otros problemas. Las memorias utilizadas como almacenamiento secundario son relativamente lentas. Como consecuencia de ello, si simplemente se utilizara la memoria secundaria, cuando el usuario se desplazara por la lista de usuarios conectados, la GUI se comportaría de manera poco fluida, pudiendo llegar incluso a cerrarse, si el bloqueo durase más de 5 segundos.

Para conseguir que el número de usuarios sea virtualmente ilimitado manteniendo la fluidez de la interfaz, se tendría que implementar una solución híbrida. Se debería construir una caché que mantuviera en memoria principal un número reducido de imágenes que se hubieran utilizado recientemente, al tiempo que se cargaran desde memoria secundaria las que probablemente se verían a continuación. Si fuera preciso mostrar un ítem de la GridView del que todavía no se tuviera la imagen en memoria principal, se utilizaría una imagen por defecto. Esta imagen se sustituiría por la real cuando se hubiera finalizado su carga desde memoria secundaria [38].

Como se ve, la implementación de este sistema no hubiera sido sencilla, y tampoco era el objeto de este proyecto, por lo que se optó por tener en memoria principal los Bitmaps. No obstante, la clase User está preparada para hacer el cambio de un sistema al otro sin que haya que rediseñarla.

5. Conclusiones

Como se ha visto en esta memoria, tanto el principal objetivo de este proyecto, el desarrollo de una aplicación para Android con soporte para EasyMANET, como el secundario, la implementación de un servicio de *chat*, se han cumplido satisfactoriamente. Además, como se deseaba, la aplicación se ha diseñado de manera que sea fácilmente ampliable con otros servicios.

La tercera meta que se pretendía alcanzar, que la aplicación tuviera un acabado estándar y profesional, era quizás demasiado ambiciosa. Sin embargo, el aspecto y funcionamiento de EfA son lo suficientemente consistentes como para que, tras unos retoques no muy profundos, se pudiera obtener una aplicación con una apariencia similar a otras *apps* de éxito de la Play Store.

Se partía de un conocimiento completamente superficial de las redes *ad hoc*, y de la plataforma EasyMANET, así como del desarrollo de aplicaciones para dispositivos móviles en general, y para el sistema Android en particular. Estas materias no se cubren en la carrera, por lo que toda noción que se poseía de ellas era sólo fruto de la curiosidad y no de un aprendizaje bien estructurado. A pesar de ello, tras un arduo esfuerzo de documentación y dedicación de tiempo, que han excedido en mucho los previstos inicialmente, se ha conseguido desarrollar una aplicación funcional, consciente del entorno en que se ejecuta, y con un aspecto visual agradable.

Aun así, EasyMANET for Android, como se ha visto, no está exenta de problemas que deberían ser abordados en futuros desarrollos: el tratamiento de las imágenes de usuario debería mejorarse, también habría que pulir algunos detalles de la GUI, o paralelizar la petición de imágenes de usuario a los demás nodos.

A parte de la mejora de detalles de usabilidad, en cuanto a la funcionalidad, sería deseable, por supuesto, el aumento de los servicios que es capaz de ofrecer la aplicación.

Con respecto a la plataforma EasyMANET, sería conveniente que se definiera un método unificado para la transmisión de datos entre los dispositivos. Sería posible entonces el desarrollo de aplicaciones similares a EfA para otros sistemas operativos (Mac OS, iOS, Windows Phone 8, Chrome…), y la adaptación de las existentes, haciéndolas a todas compatibles entre sí. Me permito sugerir aquí el modelo utilizado en la aplicación desarrollada, que hace uso de mensajes en formato XML, en vez de mecanismos propios del SO o del lenguaje de programación utilizado.

Desde un punto de vista personal, el gran esfuerzo realizado se ha visto recompensado con una ampliación de conocimientos prácticos en paradigmas de desarrollo de aplicaciones que sólo se tratan superficialmente durante la carrera, pero que son utilizados extensivamente en el desarrollo para Android.

Bibliografía

[1] José Cano Reyes, "Integrated Architecture for Configuration and Service Management in MANET Environments", Ph.D. thesis, Universitat Politècnica de València, January, 2012.

[2] Eric Schmidt: Google now at 1.5 million Android activations per day, <http://www.engadget.com/2013/04/16/eric-schmidt-google-now-at-1-5-million-android-activations-per/>. Accessed: 10-Sep-2013.

[3] Design | Android Developers, <http://developer.android.com/design/index.html>. Accessed: 10-Sep-2013.

[4] ISO/IEC (1994) *Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model.* ISO/IEC 7498-1:1994.

[5] Charles E. Perkins, Elizabeth M. Belding-Royer, and Samir R. Das, “Ad hoc on-demand distance vector (AODV) routing,” Request for Comments 3561, MANET Working Group, <http://www.ietf.org/rfc/rfc3561.txt>, July 2003, work in progress.

[6] I. Chakeres, C. Perkins, “Dynamic MANET On-demand (DYMO) Routing,” Draft IETF v21, <http://tools.ietf.org/html/draft-ietf-manet-dymo-21>, July 2012, Internet-Draft.

[7] T. Clausen and P. Jacquet, “Optimized link state routing protocol (OLSR),” Request for Comments 3626, MANET Working Group, <http://www.ietf.org/rfc/rfc3626.txt>, October 2003, work in progress.

[8] IEEE Std 802.11™-2 2012. IEEE Standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks— Specific requirements Part 11: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specications, ISO/IEC 8802-11:2012 IEEE Std. 802.11, 2012.

[9] S. Komatineni, *Pro Android 3*. Apress, New York, N.Y., 2011.

[10] Devices and Displays | Android Developers, < http://developer.android.com/design/style/devices-displays.html>. Accessed: 22-Sep-2013.

[11] Metrics and Grids | Android Developers, <http://developer.android.com/design/style/metrics-grids.html>. Accessed: 22-Sep-2013.

[12] Iconography | Android Developers, <http://developer.android.com/design/style/iconography.html>. Accessed: 22-Sep-2013.

[13] Writing Style | Android Developers, <http://developer.android.com/design/style/writing.html>. Accessed: 22-Sep-2013.

[14] Confirming & Acknowledging | Android Developers, <http://developer.android.com/design/patterns/confirming-acknowledging.html>. Accessed: 22-Sep-2013.

[15] Settings | Android Developers, <http://developer.android.com/design/patterns/settings.html>. Accessed: 22-Sep-2013.

[16] Grid Lists | Android Developers, < http://developer.android.com/design/building-blocks/grid-lists.html>. Accessed: 23-Sep-2013.

[17] Canvas and Drawables | Android Developers, <http://developer.android.com/guide/topics/graphics/2d-graphics.html#nine-patch>. Accessed: 23-Sep-2013.

[18] Draw 9-patch | Android Developers, < http://developer.android.com/tools/help/draw9patch.html>. Accessed: 23-Sep-2013.

[19] Activity | Android Developers, <http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle>. Accessed: 19-Sep-2013.

[20] M. Gargenta, *Learning Android*. O'Reilly Media, Sebastopol, Calif, 2011.

[21] ThreadPoolExecutor (Java Platform SE 6), <http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ThreadPoolExecutor.html>. Accessed: 20-Sep-2013.

[22] AsyncTask | Android Developers, <http://developer.android.com/reference/android/os/AsyncTask.html>. Accessed: 20-Sep-2013.

[23] Handler | Android Developers, <http://developer.android.com/reference/android/os/Handler.html>. Accessed: 20-Sep-2013.

[24] ServiceConnection | Android Developers, <http://developer.android.com/reference/android/content/ServiceConnection.html>. Accessed: 20-Sep-2013.

[25] Parcel | Android Developers, < http://developer.android.com/reference/android/os/Parcel.html>. Accessed: 20-Sep-2013.

[26] CopyOnWriteArrayList (Java Platform SE 6), <http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/CopyOnWriteArrayList.html>. Accessed: 12-Jul-2013.

[27] Simple 2.7.0, <http://simple.sourceforge.net/>. Accessed: 8-Aug-2013.

[28] WifiManager.WifiLock | Android Developers, <http://developer.android.com/reference/android/net/wifi/WifiManager.WifiLock.html>. Accessed: 17-Sep-2013.

[29] AlertDialog | Android Developers, <http://developer.android.com/reference/android/app/AlertDialog.html>. Accessed: 17-Sep-2013.

[30] Application | Android Developers, <http://developer.android.com/reference/android/app/Application.html>. Accessed: 17-Sep-2013.

[31] Context | Android Developers, <http://developer.android.com/reference/android/content/Context.html>. Accessed: 17-Sep-2013.

[32] GridView | Android Developers. <http://developer.android.com/reference/android/widget/GridView.html>. Accessed: 15-Jul-2013.

[33] View | Android Developers, < http://developer.android.com/reference/android/view/View.html>. Accessed: 21-Jul-2013.

[34] ToggleButton | Android Developers, <http://developer.android.com/reference/android/widget/ToggleButton.html>. Accessed: 21-Sep-2013.

[35] PreferenceActivity | Android Developers, <http://developer.android.com/reference/android/preference/PreferenceActivity.html>. Accessed: 3-Sep-2013.

[36] EditTextPreference | Android Developers, <http://developer.android.com/reference/android/preference/EditTextPreference.html>. Accessed: 3-Sep-2013.

[37] CheckBoxPreference | Android Developers, <http://developer.android.com/reference/android/preference/CheckBoxPreference.html>. Accessed: 3-Sep-2013.

[38] Displaying Bitmaps Efficiently | Android Developers, <http://developer.android.com/training/displaying-bitmaps/index.html>. Accessed: 20-May-2013.