4 Software

A lo largo de esta sección se pretende explicar el código que se ha desarrollado para la aplicación, comenzando por la estructura del proyecto, y comentando posteriormente las clases y las actividades (*activities*). Los códigos fuente completos de las mismas pueden encontrarse en el Anexo B de esta misma memoria, siendo el objetivo de esta sección comentar las responsabilidades de cada clase/actividad a un nivel de abstracción mayor, sin pretender entrar con demasiada profundidad en el código de las mismas, en la medida de lo posible.

4.1 Aplicación smartphone

Como se ha comentado anteriormente, el sistema operativo elegido para el desarrollo de la aplicación será **Android**, el sistema operativo móvil desarrollado por Google, basado en el kernel de linux, y principalmente diseñado para dispositivos táctiles como teléfonos inteligentes o *tablets*. Se trata del sistema operativo más instalado del mundo [9], siendo el sistema operativo más vendido en *tablets* desde 2013 [10], y predominante en el mundo de los *smartphones* sin ninguna duda, extendiéndose de los dispositivos de gama baja hasta dispositivos de lujo, el cual es uno de los principales motivos para su elección como sistema operativo para el que desarrollar la aplicación.

Para el desarrollo de la aplicación se ha elegido utilizar **Android Studio**, el IDE (entorno de desarrollo integrado) oficial de Google. Si bien existen otras posibilidades en el mercado (siendo otra de las opciones más populares Eclipse), Android Studio ofrece ciertas ventajas sobre los demás al tratarse del único al que Android soporta de forma oficial: por ejemplo, con respecto al sistema de *build*, el IDE de Google nos ofrece **Gradle**, un sistema basado en Apache Maven que introduce su propio DSL (*Domain Specific Language*, Lenguaje de dominio específico) con el fin de obtener una mayor versatilidad en los scripts (al contrario que Eclipse, por ejemplo, que utiliza Apache And, basado en xml y por tanto más rígido y menos versátil), así como mejores herramientas de autocompletado de código y refactorizado, desarrolladas específicamente para el código de Android, que se añaden a las ya presentes en IntelliJ IDEA, el IDE de JetBrains en el que se basa Android Studio. Ofrece también herramientas para previsualizar (e incluso editar de manera visual) las distintas actividades.

Android Studio provee además la estructura base de una aplicación Android con cada nuevo proyecto que se inicialice, reduciendo la laboriosa tarea de tener que crearla a mano por el desarrollador y tener que escribir toda la configuración para Gradle. Es por ello que, a lo largo de esta sección, se ignorará esta configuración (así como otros muchos ficheros) generados de forma automática por el IDE -llamados normalmente por su denominación inglesa *boilerplate*, código necesario para el funcionamiento de la aplicación pero usualmente generado automáticamente y que no forma parte de la aplicación como tal-, pues no son realmente relevantes para el objetivo de esta memoria. Nos centraremos, por tanto, en el código desarrollado específicamente para este proyecto, dejando de lado tanto el proceso de *build* como otros aspectos básicos de la estructura de un proyecto de aplicación Android.

4.1.1 Estructura del proyecto

La estructura del proyecto se representa en el siguiente árbol de directorios:

```
/
├── .gradle/
├── .idea/
├── app/
│   ├── build/
│   ├── libs/
│   ├── src/
│   │   └── main/
│   │       ├── java/
│   │       │   └── com.pfc.app.iHelp/
│   │       │       ├── activities/
│   │       │       │   ├── HelpMessageActivity.java
│   │       │       │   ├── MainActivity.java
│   │       │       │   ├── MainMenuActivity.java
│   │       │       │   └── RescueMessageActivity.java
│   │       │       └── classes/
│   │       │           ├── GPSHandler.java
│   │       │           ├── RestClientTask.java
│   │       │           ├── ToastHandler.java
│   │       │           └── WifiHandler.java
│   │       ├── res/
│   │       │   ├── drawable/
│   │       │   ├── layout/
│   │       │   │   ├── activity_help_message.xml
│   │       │   │   ├── activity_main.xml
│   │       │   │   ├── activity_main_menu.xml
│   │       │   │   └── activity_rescue_message.xml
│   │       │   ├── mipmap-hdpi/
│   │       │   ├── mipmap-mdpi/
│   │       │   ├── mipmap-xdpi/
│   │       │   ├── mipmap-xxdpi/
│   │       │   ├── mipmap-xxxdpi/
│   │       │   ├── values/
│   │       │   │   ├── colors.xml
│   │       │   │   ├── dimens.xml
│   │       │   │   ├── strings.xml
│   │       │   │   └── styles.xml
│   │       │   ├── values-v21/
│   │       │   └── values-w820dp/
│   │       └── AndroidManifest.xml
│   ├── app.iml
│   ├── build.gradle
│   └── proguard-rules.pro
├── build/
│   ├── generated/
│   └── intermediates/
├── gradle/
├── build.gradle
├── gradle.properties
├── gradlew
├── gradlew.bat
├── local.properties
├── MyApplication.iml
└── settings.gradle
```

Con el objetivo de mantener una separación entre actividades y clases, se han creado sendos directorios dentro del paquete iHelp. Aunque realmente no todo el código está contenido aquí (pues, como se verá más adelante, el layout de las actividades está definido dentro de la carpeta *res*), se utilizará esta separación conceptual a lo largo del capítulo para detallar la implementación de la aplicación.

4.1.2 *Activities*

Una ***activity***, o actividad, es una abstracción de Android que se define sencillamente como una sola acción que el usuario puede llevar a cabo. Casi todas las actividades interactúan con el usuario, de modo que la clase `Activity` se encarga de crear una ventana en la cual se coloca la UI. Normalmente las actividades se presentan al usuario como ventanas a pantalla completa, aunque pueden anidarse múltiples actividades dentro de otras. Es importante destacar que todas las actividades, para poder comenzar, han de declarar la etiqueta `<activity>` dentro del fichero `AndroidManifest.xml`.

Las actividades se manejan mediante una pila. Cuando una nueva actividad empieza, se pone "encima" de la pila y se convierte en la aplicación que está corriendo, quedando la que estuviese corriendo anteriormente en segundo plano, y no volverá al primer plano mientras exista otra actividad por encima de ella en la pila. El ciclo de vida de una actividad, o *lifecycle*, es el conjunto de estados en los que una actividad puede encontrarse desde el momento de su creación hasta el momento de su destrucción.

El siguiente diagrama muestra los diferentes caminos que puede seguir el *lifecycle* de una actividad en Android:

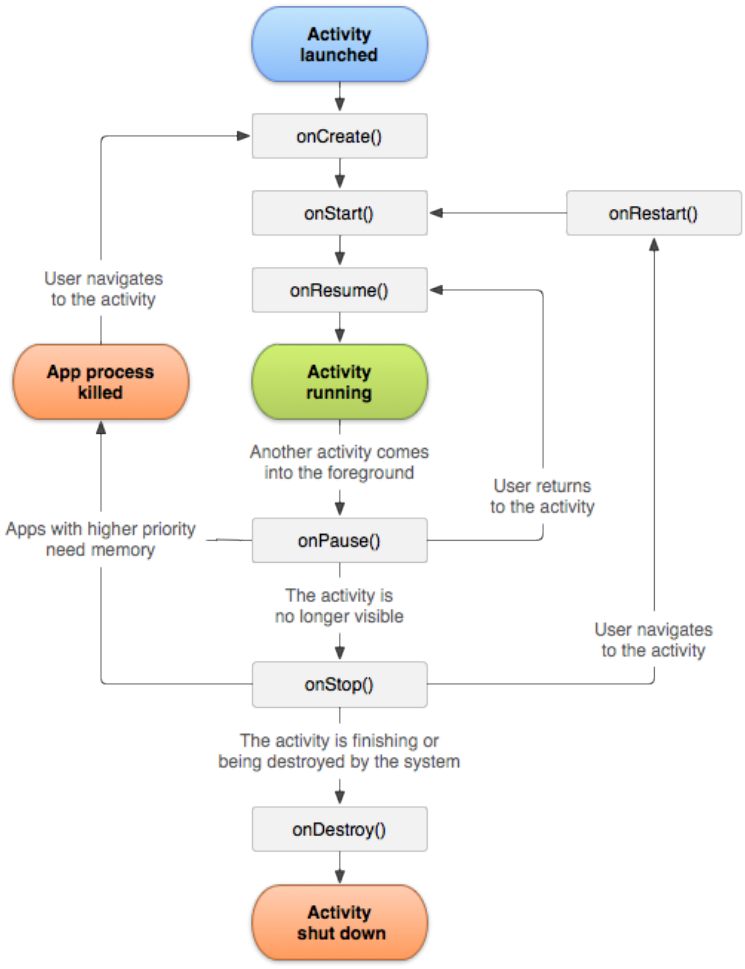

Figura 4.1 Ciclo de vida de una actividad en Android.

Existen por tanto cuatro estados:

- Si una actividad se muestra en la pantalla, está encima de la pila, y es por tanto la actividad activa.
- Si una actividad pierde el foco pero sigue estando visible (es decir, que otra actividad que no está a pantalla completa está por encima), entonces la actividad está pausada. Esto quiere decir que está viva (mantiene su estado y la información que estuviese en proceso sigue ahí), pero puede ser destruida por el sistema en caso de recursos memoria extremadamente baja.
- Si una actividad está complemente oculta por otra actividad, está parada. Mantiene el estado y la información, pero no es visible y será destruida por el sistema operativo en cuando la memoria haga falta para cualquier otro recurso.
- Si una actividad está pausada o parada, el sistema puede quitarla de la memoria bien "pidiéndole" que termine o directamente matando el proceso. Cuando se muestre de nuevo, deberá ser completamente reseteada.

MainActivity

Esta sencilla actividad es el punto de entrada de la aplicación desarrollada en este proyecto. Visualmente se ofrece al usuario un botón con el texto *Start scanning*, que será quien, mediante el método `onClick`, llame a la función `startScanning` de nuestra actividad. Este método tan sólo lanza una nueva actividad, llevándonos al menú principal, representado por `MainMenuActivity`.

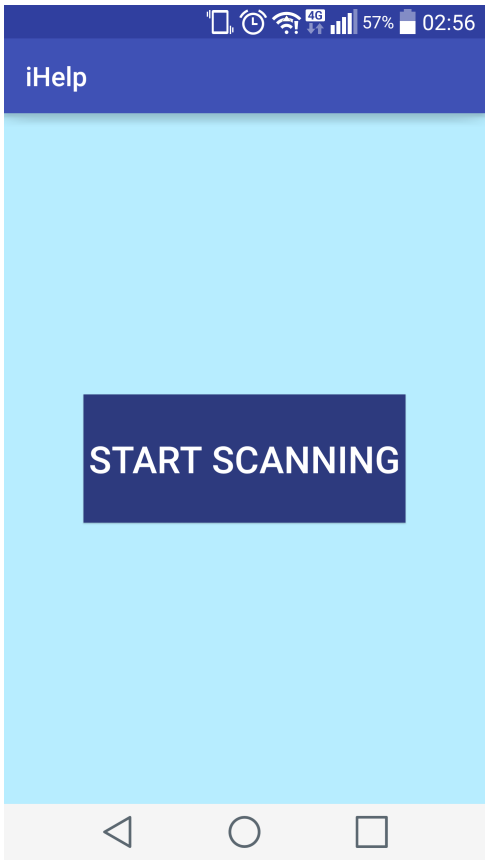

Figura 4.2 Punto de entrada de la aplicación.

MainMenuActivity

Esta actividad es, visualmente, el menú principal de la aplicación, desde el que se da la posibilidad al usuario de solicitar la localización del punto de ayuda más cercano o bien de pedir un rescate.

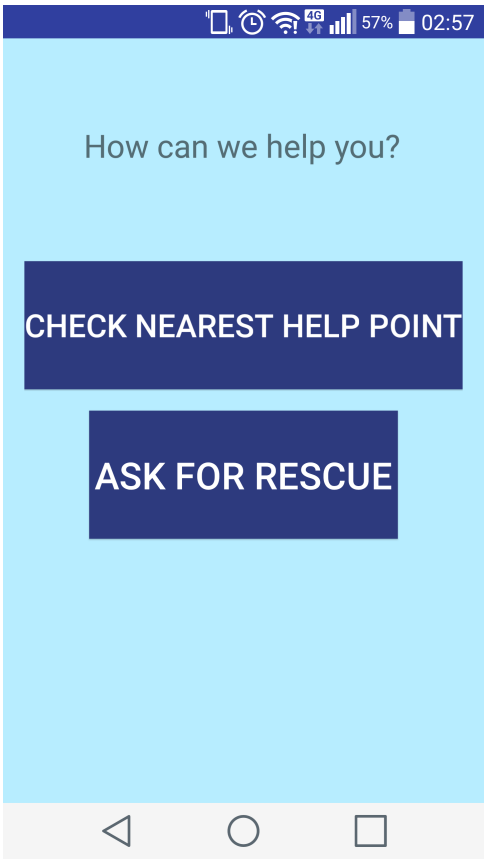

Figura 4.3 Menú principal de la aplicación.

Sin embargo, a nivel lógico, se trata de la actividad más importante de la aplicación. En primer lugar, en el método `onCreate` (el cual es invocado al instanciar la actividad), se instancian los distintos objetos del apartado clases, esto es, `toastHandler`, `wifiHandler` y `gpsHandler`. Acto seguido, es este menú principal quien se encarga de llamar al método `scanNetworksAndConnectToBest` del objeto wifiHandler, y también de asegurarse de que la aplicación tiene los permisos necesarios para acceder al GPS del teléfono antes de, utilizando un *ThreadPoolExecutor*, lanzar un hilo con la tarea del GpsHandler.

En el caso en el que el servicio de localización no esté activado, la aplicación mostrará un mensaje al usuario indicándole que deberá activarlo para poder utilizar la aplicación, proveyendo un acceso directo a la sección de ajustes del teléfono que permite hacerlo (la función `showAlert` se encargará de esto). En caso de que no se pueda obtener una localización GPS, la aplicación mostrará un mensaje indicándolo e impedirá el uso de los botones para realizar las llamadas a la API. Esta lógica se implementa en el método `disableButtonsAndShowMessage`.

Esta actividad tiene también, por supuesto, los *callbacks* asociados a los eventos *click* de la interfaz, llamados respectivamente `help` y `rescue`, quienes, haciendo uso de la tarea `RestClient`, se encargarán de realizar la llamada correspondiente a la API, de manejar el error, y de lanzar las actividades correspondientes para informar al usuario con la respuesta del servidor. En caso de un error de red causado por la pérdida de conexión, la aplicación se encargará de relanzarse (mediante el método `handleNetworkError`).

HelpMessageActivity

El objetivo de esta actividad es simplemente mostrar al usuario un mensaje con la respuesta del servidor, de forma que pueda acudir, dentro del escenario afectado por el desastre ocurrido, a un punto de ayuda. Para ello, además de mostrar la información que provee la Raspberry (un nombre identificable de la localización, las coordenadas GPS y la distancia a la que se encuentra), esta actividad cuenta con el método privado `getDirection`, del cual hace uso para poder indicar al usuario en qué dirección se encuentra su destino.

```
private String getDirection (Double deviceLat, Double deviceLon, Double lat, Double lon) {
    if (lat > deviceLat) {
        if(lon >= deviceLon + 0.1) {
            return "North-East";
        } else if ( deviceLon > lon - 0.1 && deviceLon < lon + 0.1 ) {
            return "North";
        } else {
            return "North-West";
        }
    } else {
        if(lon >= deviceLon + 0.1) {
            return "South-East";
        } else if (deviceLon > lon - 0.1 && deviceLon < lon + 0.1) {
            return "South";
        } else {
            return "South-West";
        }
    }
}
```

Código 4.1 getDirection.

El objetivo de esta actividad es tratar de dar la máxima información posible al usuario de forma que pueda localizar el punto de ayuda más cercano a su posición de la forma más sencilla posible, dada la imposibilidad de utilizar servicios más completos como Google Maps al no disponer de conexión a internet en una situación de emergencia, debido al daño recibido por la infraestructura de comunicaciones o bien por la saturación de la misma debido al uso masivo de un elevado número de dispositivos al mismo tiempo. La red de los drones no provee de servicios de conexión a internet, si no que provee comunicación inalámbrica a un sistema específico de ayuda para las víctimas.

RescueMessageActivity

De nuevo, el fin de esta actividad es simplemente mostrar al usuario la respuesta del servidor tras llamar a la ruta `/rescue`. En este caso es tan sencillo como simplemente asignar el valor del texto al elemento correspondiente en la vista, tomándolo directamente de los parámetros con los que se ha invocado a la actividad, sin requerir lógica adicional como en el caso anterior. Esto se puede apreciar en el siguiente fragmento de código extraído diretamente del código de la actividad:

```
        (...)

        textView = (TextView) findViewById(R.id.textView2);

        String text = getIntent().getStringExtra("message");

        changeText(text);
    }

    public void changeText(String text) {
        textView.setText(text);
    }
```

Código 4.2 Fragmento de RescueMessageActivity.

4.1.3 Clases

GPSHandler

`GPSHandler` implementa la interfaz *Callable* con el fin de poder ejecutarse en segundo plano en un *ThreadPoolExecutor*, de forma que pueda realizar operaciones sin bloquear el hilo principal de ninguna actividad. Para ello, implementa el método `call()`. Este método se encargará de utilizar una de las mútliples abstracciones que el sistema operativo Android provee para el manejo de la localización de usuario, `LocationListener`, con el objetivo de obtener la última localización del dispositivo. Este objeto se encarga de ejecutar ciertos

métodos cuando haya cambios en el estado de la localización: *onLocationChanged*, *onStatusChanged*, *onProviderEnabled* y *onProviderDisabled*.

El *LocationListener* correrá en un hilo y se encargará de monitorizar el estado de la localización, mientras que el método `getLocation` proveerá la última localización conocida en el momento en que es llamado.

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

Código 4.3 Declaración de los permisos necesarios para GPSHandler en AndroidManifest.xml.

RestClientTask

Esta clase extiende a la abstracción de Android *AsyncTask*, que provee una forma de realizar operaciones en *background* con el fin de publicar los resultados en el hilo de UI sin necesidad de manejar hilos o concurrencia a más bajo nivel. En realidad, no es más que una clase construida sobre *Thread* y *Handler*, que no constituye un verdadero *framework* de concurrencia: la idea es utilizar `AsyncTask` para operaciones cortas, que no duren más de unos cuantos segudos.

`RestClientTask` sobreescribe el método `doInBackground` de `AsyncTask` donde se encarga de realizar una llamada http al servidor de la Raspberry, para posteriormente procesar la respuesta y devolver un objeto JSON al hilo principal de interfaz de usuario desde el que se invocó la tarea. Es responsabilidad también de la tarea el configurar la conexión antes de realizar cualquier llamada al servidor.

De nuevo es importante señalar que el permiso `INTERNET` será necesario para que nuestra aplicación pueda realizar llamadas http:

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

Código 4.4 Declaración de los permisos necesarios para RestClientTask en AndroidManifest.xml.

ToastHandler

Un *toast* es un mensaje que se muestra en pantalla durante unos segundos al usuario para luego volver a desaparecer automáticamente sin requerir ningún tipo de actuación por su parte, y sin recibir el foco en ningún momento (o dicho de otra forma, sin interferir en las acciones que esté realizando el usuario en ese momento).

El objetivo de esta clase es proveer una pequeña capa de abstracción sobre la clase Toast de Android, pudiendo tener una sola instancia en toda la aplicación desde la que poder mostrar mensajes al usuario sin necesidad de acceder directamente a Toast en cada ocasión en que sea necesario. Provee además una forma de centralizar la configuración de los *toasts*, pues en caso de necesitar cambiar la configuración, tan sólo habremos de acudir a esta clase para editarla, y no tener que cambiar cada *toast* de manera individual.

WifiHandler

La clase WifiHandler pretende albergar toda la lógica relacionada con la conexión inalámbrica (relacionada con la conexión, no con la comunicación, esto es, no se ocupará de la comunicación entre la aplicación Android y el servidor web, si no de la conectividad inalámbrica). El punto de entrada de la lógica de la clase es el método `scanNewtorksAndConnectToBest`, el cual es llamado desde la actividad del menú principal.

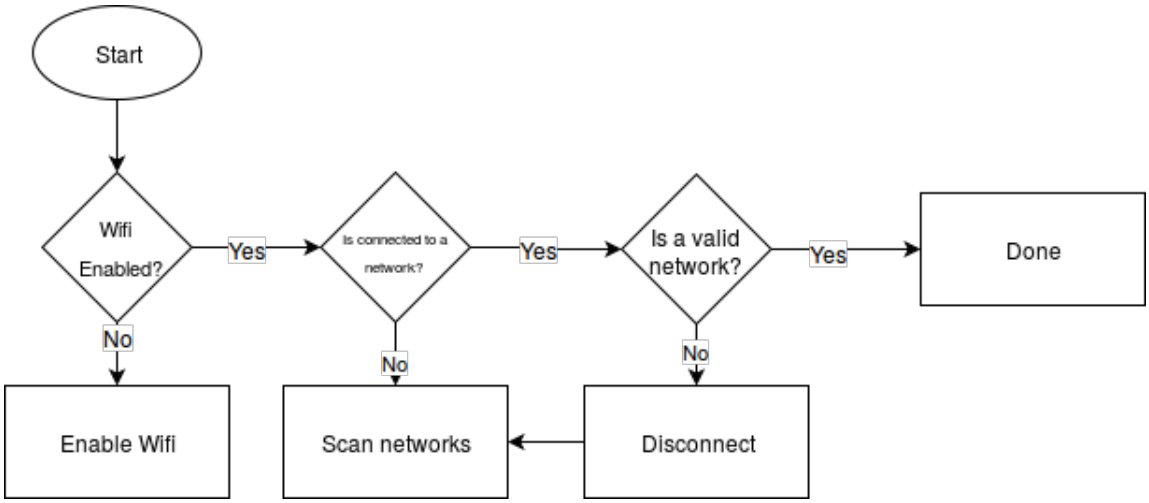

Figura 4.4 Lógica de conexión inalámbrica.

En la figura se explica el funcionamiento del algoritmo para la conexión a una red inalámbrica. El criterio para elegir si una red es válida o no se basa en el SSID, pues se asume que todas las redes inalámbricas estarán anotadas con las siglas "RP" (Raspberry Pi). La referencia de la figura a *ScanNetworks* se refiere al método `scanNetworksAndConnectToBest`. Este método utiliza, al igual que el resto del WifiHandler, la abstracción que provee la API de Android para manejar las conexiones inalámbricas, WiFiManager. Haciendo uso del mismo es capaz de iterar sobre las conexiones disponibles tras el escaneo, para seleccionar aquella cuya intensidad recibida en dB sea mayor. Es importante destacar que, para que la clase WifiHandler pueda llevar a cabo todas estas tareas, hemos de declarar en el manifiesto de nuestra aplicación que hará uso de los permisos `ACCESS_WIFI_STATE` y `CHANGE_WIFI_STATE`:

```
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
```

Código 4.5 Declaración de los permisos necesarios para WifiHandler en AndroidManifest.xml.

Una vez detectada la red inalámbrica con mayor potencia, WifiHandler comprueba si se trata de una red conocida (es decir, configurada pues el dispositivo se ha conectado a ella en algún momento en el pasado), con el fin de crear la configuración correspondiente si esta no está presente.

4.2 Configuración de la Raspberry Pi

Para que la Raspberry Pi sea capaz de emitir WiFi, será necesaria la instalación y configuración de dos servicios: *hostapd* y *udhcpd*, además de la instalación del entorno node.js para poder correr la aplicación que hará de servidor.

4.2.1 udhcpd

udhcpd es un servidor DHCP ligero, pensado para sistemas embebidos. Su nombre proviene de la abreviación de micro (μ, **u**), el protocolo DHCP (**dhcp**) y su propia naturaleza como *daemon* (**d**). Se trata de un programa completamente funcional, que cumple la RFC 2131. Fue originalmente desarrollado en 1999 por Matthew Ramsay y distribuido bajo licencia GNU GPL por Moreton Bay.

Instalación, configuración y puesta en marcha

El método de instalación más común y sencillo es utilizar los paquetes incluidos en los repositorios configurados en el gestor de paquetes de cualquier distribución Linux, pues udhcpd está incluído en la gran mayoría. En Raspbian, por ejemplo, bastaría con introducir en una terminal:

```
sudo apt-get install udhcpd
```

La configuración del servicio es muy sencilla. *udhcpd* utiliza un solo fichero principal de configuración, `/etc/udhcpd.conf`. Con cualquier editor de texto y permisos de superusuario, el fichero debe quedar como sigue:

```
start 192.168.42.2
end 192.168.42.20
interface wlan0
remaining yes
opt router 192.168.42.1
opt lease 864000
```

Código 4.6 udhcpd.conf.

`start` y `end` delimitan el rango de IPs que el dispositivo podrá proporcionar a los clientes. En este caso, se dispone de 18 direcciones IP. `interface` indica la interfaz de red a utilizar por el servicio. Las líneas que comienzan con `opt` son parámetros opcionales, pero necesarios para la configuración deseada. `router` indica la IP de la Raspberry Pi en la interfaz wlan0, y `opt lease` indica el tiempo, en segundos, de "préstamo" que cada cliente tendrá asignada la dirección IP (en este caso, 864000 segundos equivalen a diez días).

Una vez configurado, aún nos queda otro paso, y es activar DHCP. Para ello, editamos el fichero bajo la ruta `/etc/default/udhcpd`, de forma que su contenido quede:

```
# Comment the following line to enable
# DHCPD_ENABLED="no"
```

Código 4.7 /etc/default/udhcpd.

Se puede observar que tan sólo hemos comentado (añadiendo, al principio de la línea, el símbolo #) la configuración que deshabilita DHCPD, habilitándolo de esta forma.

4.2.2 hostapd

hostapd es un *daemon* (demonio) de espacio de usuario para servidores de punto de acceso y autenticación. Implementa puntos de acceso IEEE 802.11, autenticadores IEEE 802.1X/WPA/WPA2/EAP, cliente RADIUS, servidor EAP y servidor de autenticación RADIUS. La última versión soporta Linux (driver Host AP entre otros) y FreeBSD (driver net80211). Está diseñado para ser un programa *daemon* que corra en segundo plano (*background*) y actúe como un componente backend controlando la autenticación, aunque también existen programas cliente frontend que permiten su configuración como *hostapd_cli*, incluido en la distribución de *hostapd*.

Se enumeran a continuación las características WPA/IEEE 802.11i/EAP/IEEE 802.1X soportadas por hostapd:

- WPA-PSK ("WPA-Personal")
- WPA con servidor EAP o servidor RADIUS de autenticación externo ("WPA-Enterprise")
- Gestión de claves CCMP, TKIP, WEP104, WEP40
- WPA y IEEE 802.11i/RSN/WPA2 al completo
- RSN: caché PMKSA, pre-autenticación
- IEEE 802.11r
- IEEE 802.11w
- Wi-Fi Protected Setup (WPS)

Instalación, configuración y puesta en marcha

Al igual que con udhcp, podemos utilizar los repositorios de nuestra distribución para instalar el software. En Raspbian bastaría con introducir en un terminal:

```
sudo apt-get install hostapd
```

Para el caso concreto que nos ocupa, utilizando el *dongle* WiFi Edimax EW7811, el proceso no es tan sencillo. Si bien el dispositivo es *plug and play* en cuanto a conectividad inalámbrica (bastará conectarlo a un puerto USB de la Raspberry para que ésta lo detecte y sea capaz de conectarse a redes inalámbricas,

pues el kernel Raspbian incluye los drivers necesarios), para conseguir que se comporte como un punto de acceso, el proceso es algo más complicado, y será necesario disponer del driver nl80211. El por qué es el siguiente: el EW7811 está basado en el chip Realtek RTL8188CUS. Este chip no soporta el driver estándar nl80211 de *hostapd*, y sus drivers *in-kernel* están limitados en características, por lo que la propia Realtek publicó en su web los drivers para este chip bajo licencia GPL. Será necesario entonces utilizar una versión parcheada de hostapd para incluir estos drivers, o bien hacerlo nosotros mismos. Gracias a la gran comunidad *open source*, no es difícil encontrar dichas versiones ya parcheadas de *hostapd* publicadas por reconocidos ingenieros *senior* expertos en la materia, como Pritam Baral (https://github.com/pritambaral) o Jens Segers (https://github.com/jenssegers).

Si tomamos este último, por ejemplo, el procedimiento de instalación de *hostapd* es sencillo. En primer lugar, y dado que vamos a instalar una versión modificada de *hostapd*, hemos de eliminar cualquier instalación anterior. En una terminal, bastará con ejecutar:

```
sudo apt-get autoremove hostapd
```

El siguiente paso es descargar el código fuente en la Raspberry. Bien sea clonando el repositorio de git, o descargando directamente una versión comprimida del proyecto (pues no nos interesa modificar el código), comenzamos con la instalación. Si nos decantamos por la opción de la descarga directa, los siguientes comandos de terminal descargarán el fichero y lo descomprimirán:

```
wget https://github.com/jenssegers/RTL8188-hostapd/archive/v2.0.tar.gz
tar -zxvf v2.0.tar.gz
```

Una vez descomprimido, tan solo hemos de movernos al directorio que contiene el código y hacer uso del fichero Make para compilar *hostapd*.

```
cd RTL8188-hostapd-2.0/hostapd
sudo make
```

Y, una vez tengamos de nuevo control de la terminal, instalar el programa:

```
sudo make install
```

Tras la instalación, se creará el binario en la ruta `/usr/local/bin`, se creará un script de arranqe y se creará un fichero de configuración en `/etc/hostapd/hostapd.conf`. Este es el fichero de configuración principal de *hostapd*, el cual deberemos abrir con el editor de texto de nuestra preferencia para editarlo y configurar el servicio.

Hostapd nos ofrece la posibilidad de crear redes seguras protegidas por contraseña, pero el objetivo del proyecto es conseguir comunicar fácilmente el smartphone con la Raspberry, por lo que se configurará para utilizar una red abierta. Los contenidos del fichero quedan, entonces:

```
interface=wlan0
ssid=Raspberry-Pi
hw_mode=g
channel=6
auth_algs=1
wmm_enabled=0
```

Código 4.8 hostadp.conf.

En la primera linea, `interface` representa la interfaz de red a utilizar, en este caso será la interfaz inalámbrica de la Raspberry Pi, wlan0. `ssid` es el nombre de la red inalámbrica, que será incluido en todos los paquetes de la misma para identificarlos como parte de esa red, `hw_mode` configura la frecuencia de la señal (el valor `g` la fija a 2.4Ghz), `channel` indica el canal a utilizar, `auth_algs` indica los algoritmos de autenticación a utilizar (1 = wpa, 2 = wep, 3 = ambos) y `wmm_enabled` configura el soporte para QoS en Wifi, que en este caso no será necesario.

Por último, habremos de editar también el fichero `/etc/default/hostapd`, para indicar que el fichero `/etc/hostapd/hostapd.conf` será el que contiene la configuración del servicio:

```
DAEMON_CONF="/etc/hostapd/hostapd.conf"
```

Código 4.9 /etc/default/hostadp.

4.2.3 node.js

node.js es un entorno (*runtime*) de código abierto de Javascript, construido sobre el motor V8 de Chrome, con la finalidad de construir diferentes herramientas y aplicaciones en el lado servidor. Utiliza una arquitectura basada en eventos no bloqueantes, que provee la posibilidad de entrada/salida asíncronas, lo cual permite optimizar el tráfico y escalabidad de aplicaciones web con numerosas operaciones de entrada/salida.

Los desarrolladores, gracias a node, pueden crear servidores fácilmente escalables sin necesidad de acudir a la programación multi-hilo, utilizando el modelo de eventos, que utiliza *callbacks* para indicar la finalización de una tarea. De hecho, uno de los motivos para la creación de node.js fue el hecho de que la concurrencia es complicada en un gran número de lenguajes de programación y suele conllevar un pobre rendimiento. Es por ello que, unido a la ventaja del extendido uso de Javascript en el mundo de la programación web, la comunidad de node.js le ha hecho extremadamente popular en los últimos años.

Esta gran comunidad ha desarrollado cientos de aplicaciones y librerías de código abierto para node.js, la gran mayoría de ellas alojadas en los servidores de npm (*Node Package Manager*), el gestor de paquetes de node.js que nos permitirá de forma sencilla instalar y desinstalar librerías y resolver dependencias.

Con el fin de instalar y administrar de forma sencilla el entorno node.js (esto es, node.js y npm), haremos uso de **nvm** (*Node Version Manager*). Se trata de un script de bash que tiene la capacidad de manejar múltiples versiones activas de node.js, y permite instalar, ejecutar comandos en una versión de node.js determinada, asignar el valor a la variable PATH para usar una versión determinada de node.js, entre otras muchas funcionalidades. Para instalar nvm, hemos de obtener el código fuente en primer lugar. Para ello hacemos uso del software de control de versiones git, y ejecutando los siguientes comandos, obtendremos nvm en el directorio `/.nvm`:

```
git clone https://github.com/creationix/nvm.git ~/.nvm
cd ~/.nvm
git checkout v0.25.4
```

Habremos ahora de editar los ficheros `/.bashrc` y `/.profile`, añadiendo la línea:

```
source ~/.nvm/nvm.sh
```

al final de ambos. Para ello necesitaremos permisos de super usuario, y podemos hacer uso de cualquier editor de texto. Tras esto, será necesario reiniciar el dispositivo para que los cambios se apliquen:

```
sudo reboot
```

Una vez tengamos la terminal disponible de nuevo, podemos comprobar que la instalación se ha llevado a cabo de forma correcta ejecutando el comando:

```
nvm --version
```

que deberá retornar el valor de la versión que se ha instalado (en nuestro caso, `v0.25.4`).

Una vez instalado correctamente `nvm`, es hora de instalar node.js. Para ello, bastará con ejecutar:

```
nvm install 0.10.26
npm update -g npm
```

Se trata de un proceso sencillo, pues `nvm` es quien se encargará de la instalación por el usuario, teniendo tan sólo que especificar qué versión de node.js se desea instalar. Para utilizar de ahora en adelante la versión recién instalada de node.js, habremos de ejecutar:

```
nvm use 0.10.26
```

A partir de ahora, con tal sólo utilizar el comando `node`, tendremos accesible una consola de node.js, en la que podremos ejecutar comandos o correr nuestro código Javascript.

4.2.4 Configuración adicional y arranque de servicios

No hemos de olvidarnos de configurar, desde la terminal, la interfaz inalámbrica para utilizar la dirección IP configurada en el fichero `/etc/udhcpd.conf`. Para ello, ejecutamos el comando:

```
sudo ifconfig wlan0 192.168.42.1
```

Esto configura la interfaz la interfaz wlan0 para tener la IP que deseamos. Sin embargo, no es un cambio permanente, y se perderá en el próximo reinicio del sistema. Para hacer este cambio permanente y no tener que preocuparnos de la configuración cada vez que la Raspberry se arranque, editaremos el fichero de configuración de interfaces de red, `/etc/network/interfaces` para que incluya:

```
iface wlan0 inet static
  address 192.168.42.1
  netmask 255.255.255.0
  network 192.168.42.0
  broadcast 192.168.42.255

#allow-hotplug wlan0
#wpa-roam /etc/wpa_supplicant/wpa_supplicant.conf
#iface default inet manual
```

Código 4.10 Cambios en /etc/network/interfaces.

Es importante notar que estamos comentando las tres últimas lineas. Comentar `allow-hotplug` previene el inicio de la interfaz cuando el kernel detecte el evento *hotplug* en la interfaz, mientras que comentar la linea `wpa-roam` evita la configuración de de wpa_supplicant (que queda fuera de los objetivos del proyecto).

Tan sólo nos queda ahora arrancar ambos servicios, de nuevo haciendo uso de una terminal:

```
sudo service hostapd start
sudo service udhcpd start
```

Estos cambios pueden también hacerse persistentes, de forma que ambos servicios se inicien al arrancar el sistema operativo, ejecutando:

```
sudo update-rc.d hostapd enable
sudo update-rc.d udhcpd enable
```

`update-rc.d` actualiza los enlances de scripts de arranque sistema `/etc/rc`runlevel`.d/NN`nombre cuyo objetivo es el script `/etc/init.d/`nombre ("NNnombre" es una cadena genérica que puede tomar varios valores, tal y como se explica en los párrafos que siguen).

Será necesario hacer una última modificación. Para que *udhcpd* arranque de manera correcta es necesario que las interfaces de red estén configuradas de forma adecuada (esto es, en el caso que nos ocupa, que `wlan0` tenga asignada la IP estática correspondiente). Para ello, debemos asegurarnos que el servicio `networking` está arrancado cuando lo haga *udhcpd*. Con el fin de hacerlo, deberemos realizar una pequeña modificación a nuestro sistema, creando *symlinks* (enlaces simbólicos) a los scripts de arranque correspondientes a los servicios (localizados en `/etc/init.d`). En primer lugar podemos averiguar el *run level* en el que nuestro sistema opera normalmente haciendo uso del comando:

```
runlevel
```

En el caso de las Raspberries, el resultado es 2. En consecuencia, habremos de acudir al directorio `/etc/rc2.d/` (el directorio cambiará dependiendo del *run level*, si este por ejemplo fuese 3, habríamos de acudir a `/etc/rc3.d/`) y crear los links simbólicos mencionados anteriormente. En una terminal, bastará con ejecutar:

```
sudo ln -s /etc/init.d/networking /etc/rc2.d/K01networking
```

Los enlaces de este directorio siguen una nomenclatura particular, como se puede apreciar. Todos van precedidos por K (*kill*) o S (*start*), y un número, indicando el orden en el que se ejecutarán. Con la ejecución del anterior comando nos aseguramos de que el servicio `networking` será detenido en primer lugar, y tan sólo nos queda asegurarnos de que el script para udhpcd (que ya debería estar presente en el directorio) está precedido de S y un número mayor que 01. En último lugar, volvemos a arrancar el servicio `networking` actualizando el fichero `/etc/rc.local` añadiendo la línea `service networking start`.

Habremos de ocuparnos también de que el servidor arranque al hacerlo la Raspberry. Si bien existen diferentes formas de automatizar este proceso en Linux, en este caso se ha optado por la opción de utilizar `upstart`, un reemplazo para el sistema tradicional de iniciar procesos en Linux basado en eventos, que de hecho es de uso extendido en gran parte de las distribuciones Linux basadas en Debian. Fue escrito y desarrollado por Scott James Remnant.

El proceso tradicional de arranque en Linux era responsable de asegurar el estado normal de la máquina tras el encendido, o de terminar correctamente los procesos antes de su apagado. Como resultado, el diseño era estrictamente síncrono, bloqueando las tareas futuras hasta que termine la que se esté procesando en ese momento. Además, las tareas debían definirse *a priori*, lo que limitaba ciertas tareas no relacionadas con el arranque que son indispensables en un ordenador moderno, como la adición de nuevos discos o dispositivos USB, o incluso el firmware de un dispositivo. El modelo de eventos de `upstart` le permite responder a los eventos de manera asíncrona conforme se generan, manejando el arranque de las tareas y servicios durante el arranque y deteniéndolas en el apagado, pero también supervisándolos mientras el sistema está corriendo.

Para instalar `upstart` en nuestra máquina, habremos de ejecutar:

```
sudo apt-get install upstart
```

Una vez instalado, habremos de escribir la configuración de arranque de nuestro servicio. Para ello, creamos el fichero `/etc/init/gpsapi.conf`:

```
start on filesystem and started networking
stop on [06]

setuid pi
setgid pi

respawn

script
  chdir /home/pi/dev/pfc/node
  exec /home/pi/.nvm/v0.10.26/bin/node server.js
end script
```

Código 4.11 /etc/init/gpsapi.conf.

En este fichero estamos indicando a `upstart` bajo qué condiciones debe arrancar y bajo cuáles no, con las dos primeras líneas (en la primera indicamos que el servicio debe arrancar una vez se haya montado el sistema de ficheros y el servicio de red, y en la segunda le indicamos que no debe arrancar para los *runlevels* 0 y 6, *halt* y *reboot*, respectivamente, pues no tiene sentido). A continuación indicamos el usuario y el grupo con el que ejecutar el script (en este caso, se trata del usuario por defecto de Raspbian, Pi), y finalmente le indicamos qué debe ejecutar.

4.3 Servidor web

Para simular la lógica de los drones, se ha decidido desarrollar un pequeño servidor web que correrá en la Raspberry Pi y ofrecerá una sencilla API REST mediante la cual la aplicación de Smartphone podrá solicitar las coordenadas GPS del puesto de auxilio más cercano, así como enviar las suyas propias para solitar ayuda. El *runtime* elegido es **node.js**, construido sobre el motor V8 de Chrome. Utiliza un modelo de entrada/salida no bloqueante, basado en eventos, ligero y eficiente. El *framework* utilizado es **Express**. Se trata de una infraestructura de aplicaciones web node.js mínima y flexible que proporciona un conjunto

sólido de características para las aplicaciones web. Con miles de métodos de programa de utilidad HTTP y middleware a su disposición, la creación de una API sólida es rápida y sencilla, proporcionando además una delgada capa de características de aplicación web básicas, permitiendo un gran rendimiento.

Con el objetivo de escribir los tests, se utilizarán dos de las herramientas más extendidas en Javascript para tal fin: **Mocha**, como *framework* de testing, y **chai**, una librería para comprobaciones (*assertions*).

4.3.1 Desarrollo

Especificaciones

Comencemos, en primer lugar, como se explicó en los fundamentos teóricos, a escribir las especificaciones de nuestro proyecto a alto nivel, sin tener en cuenta de los detalles de la futura implementación. En primer lugar, teniendo en cuenta los objetivos de diseño, parece tener sentido utilizar dos *endpoints* para nuestra API: uno para solicitar las coordenadas GPS del punto de ayuda más cercano (`/help`) y otro para solicitar un rescate (`/rescue`).

El primero de ellos, `/help` recibirá como parámetros las coordenadas GPS del cliente, y en función de las mismas, devolverá las coordenadas GPS del punto de ayuda más cercano, tras realizar los cálculos pertinentes. En caso de que no se manden las coordenadas GPS, o de que estas no sean válidas, el servidor deberá devolver un error.

El otro *endpoint*, `/rescue`, también recibirá como parámetros las coordenadas GPS del cliente. De nuevo, si estas no existen o son erróneas, deberá devolver un código de estado de error, y en caso contrario, deberá devolver un código de estado "OK" así como un pequeño mensaje para el usuario indicando que la ayuda está en camino. En caso de que dicho usuario ya haya realizado una petición (donde se identificará al usuario por su dirección IP), devolverá un mensaje de OK añadiendo también la fecha y hora en las que realizó la petición, así como el tiempo que ha transcurrido desde entonces.

Preparación

Una vez definidas las especificaciones, antes de comenzar con la implementación, utilizaremos el gestor de paquetes de node.js **npm** para inicializar el proyecto.

```
npm init
```

De esta manera podremos crear, con una sencilla interfaz, el contenido básico del fichero `package.json`. Este fichero, que debe seguir el formato JSON, contiene información sobre el proyecto (tales como el nombre y la versión, una descripción, un enlace al sistema de seguimiento de error o *bug-tracker* del proyecto o una dirección de correo a la que reportar los errores, información sobre la licencia con la que se publica el módulo, un campo para indicar el punto de entrada del programa, una referencia al repositorio donde se aloja el código, scripts, etc), así como la lista de dependencias del mismo (nombre y versión de las mismas).

El siguiente paso es instalar las dependencias necesarias. Npm distingue entre dos tipos de dependecias: *dependencies*, las cuales son indispensables para poder ejecutar el programa, y *devDependencies* (dependencias de desarrollo), las cuales sólo son necesarias durante la fase de desarrollo del proyecto. Por lo tanto, los paquetes `mocha` y `chai` serán dependencias de desarollo, mientras que `express` y `underscore` serán dependencias necesarias para que el programa funcione correctamente.

```
npm install --save-dev mocha
npm install --save-dev chai
npm install --save express
npm install --save underscore
```

Los parámetros `-save` y `-save-dev` sirven para que npm incluya de forma automática las dependencias al fichero `package.json`, el cual ahora deberá tener un contenido como el que sigue (tras añadir un script para ejecutar los tests utilizando mocha):

```
{
  "name": "gps-api",
```

```
  "version": "1.0.0",
  "description": "simple api to return fake gps coordinates",
  "main": "index.js",
  "scripts": {
    "test": "./node_modules/.bin/mocha test/*.spec.js"
  },
  "author": "Roberto Espejo",
  "license": "",
  "devDependencies": {
    "chai": "^3.5.0",
    "mocha": "^3.2.0",
    "request": "^2.79.0"
  },
  "dependencies": {
    "express": "^4.14.0",
    "body-parser": "^1.15.2",
    "underscore": "^1.8.3"
  }
}
```

Código 4.12 package.json.

La estructura del directorio del proyecto será la siguiente:

```
/
├── scripts/
├── node_modules/
├── test/
│   └── server.spec.js
├── package.json
└── server.js
```

Creamos por tanto los ficheros necesarios y nos disponemos a comenzar con la implementación.

Implementación

Ahora que está todo preparado, podemos comenzar con la implementación del código, como se ha comentado anteriormente, siguiendo la metodología TDD:

- En primer lugar, elegimos una especificación. Por ejemplo, tomemos el primero de los endpoints, `/help`. Se especifica que, ante una llamada sin parámetros, el servidor deberá devolver un código de error.
- A continuación, debemos escribir una prueba (test). Utilizaremos los bloques `describe` para dar contexto a los tests, y la directiva `it` para describir cada uno de los propios casos de prueba (*test cases*). En este caso se pretende comprobar que, ante una llamada sin parámetros, el servidor responde con un código de error http 400 (*Bad request*, Petición errónea). Haremos uso por tanto de la librería *request* para realizar esta llamada, y, mediante las comprobaciones de *chai*, nos aseguraremos de que la respuesta del servidor es la esperada.

```
describe('GPS API', function() {

  describe('Help', function() {

    var url = 'http://localhost:3000/help';

    it('should return an error if no parameters provided', function(done) {
    request(url, function(error, response, body) {
          expect(response.statusCode).to.equal(400);
          expect(body).to.equal(JSON.stringify({
          msg: 'GPS coordinates need to be provided. Please try again.'
            }));
          done();
        });
  });
});
```

```
});
```

- Verificamos ahora que la prueba falla. Como nuestro código aún no está implementado, es imposible que el test pase. Ejecutamos los tests utilizando el comando que provee npm (y que configuramos en el fichero `package.json`):

```
npm test
```

Obtendremos la siguiente salida en la consola:

```
> gps-api@1.0.0 test /home/rob/pfc/node
> mocha test/*.spec.js

  GPS API
    Help
      1) should return an error if no parameters provided

  0 passing (60ms)
  1 failing

  1) GPS API Help should return an error if no parameters provided:
     Uncaught TypeError: Cannot read property 'statusCode' of undefined
      at Request._callback (test/server.spec.js:13:25)
      at self.callback (node_modules/request/request.js:186:22)
      at Request.onRequestError (node_modules/request/request.js:845:8)
      at Socket.socketErrorListener (_http_client.js:310:9)
      at emitErrorNT (net.js:1276:8)
      at _combinedTickCallback (internal/process/next_tick.js:74:11)
      at process._tickCallback (internal/process/next_tick.js:98:9)

npm ERR! Test failed. See above for more details.
```

Esto se debe a que aún no tenemos implementado el servidor, y por lo tanto, el test no es capaz de recibir ninguna respuesta.

- Escribir la implementación: Escribir el código más sencillo que haga que la prueba funcione. En este caso, editamos el fichero `server.js` e incluimos la lógica.

```
var express = require('express');
var app = express();

app.get('/help', function(req, res) {

  var keys = Object.keys(req.query);

  if (!_validateArguments(keys)) {
    res.status(400).send({
      msg: 'GPS coordinates need to be provided. Please try again.'
    });
  }

});

app.listen(3000);

_validateArguments = function (Arguments) {
  return Arguments.length !== 0;
};
```

Se ha definido un método `_validateArguments`. En Javascript no existen métodos o atributos privados como en otros lenguajes de programación orientados a objetos, por lo que, por convención de código, los nombres de los métodos o atributos que deberían ser privados y no utilizados fuera del objeto donde están definidos se preceden con una barra baja (_) para así indicarlo. Si ahora ejecutamos el servidor:

```
node server.js
```

podemos comprobar que el test pasará y por tanto ya tenemos nuestro código implementado. Bastaría con repetir este proceso hasta conseguir implementar todas las especificaciones. El hecho de tener tests también nos ayuda a la hora de mantener el código: si durante el proceso de implementación del resto de las mismas los tests anteriores dejan de pasar, sabremos que algo no está comportándose como es debido.

El resto del código sigue este mismo patrón de desarrollo, y se adjunta en su totalidad en los apéndices. Con el fin de agilizar el comentario del mismo, se ofrecerá a continuación una descripción más general del resto del código, sin entrar en tanto detalle sobre cómo ha afectado TDD al desarrollo del mismo.

GpsCalculator.js Antes de hablar del resto del servidor, haremos mención al módulo GpsCalculator. Un módulo en Javascript no es más que una pequeña unidad de código reutilizable, que suele utilizarse para agrupar métodos con un fin común. En este caso, el propósito de nuestro módulo GpsCalculator es agrupar toda la lógica relacionada con el cálculo y validación de las coordenadas GPS. El módulo exporta (mediante la directiva `exports`) dos métodos: *getNearestPoint* y *validateCoordinates*, y posee otros dos métodos privados que no exporta (como indica su nombre prefijado de _). Para el cálculo de la distancia del punto de ayuda más cercano, la función `getNearestPoint` acepta como parámetros las coordenadas originales y una lista de los puntos de ayuda, sobre los cuales itera llamando al método privado `getNearestPoint`, quien aplica la llamada **fórmula de Haversine** (que ya se explicó en los fundamentos teóricos) para calcular la distancia entre dos puntos definidos por coordenadas de latitud y longitud. Finalmente, getNearestPoint devuelve aquel lugar cuya distancia sea menor respecto a las demás al punto dado.

server.js El código del servidor, **server**, hace uso del módulo GpsCalculator mediante la directiva `require`. Esto hace que en el contexto de server.js esté disponible un objeto GpsCalculator que provee los métodos que nuestro módulo exporta, como se ha comentado anteriormente, lo cual nos permite utilizarlo en las rutas `/help` y `/rescue` para validar las coordenadas GPS. El diseño de estos dos *endpoints* ya ha sido explicado cuando se dieron sus especificaciones en cuanto a qué devolver, pero merece la pena destacar el hecho de que `/help` se encarga también de guardar en un vector de objetos las peticiones recibidas cuyo resultado ha sido 200, con el fin de saber si se ha procesado ya una petición desde dicha dirección IP. De este modo, la próxima vez que se reciba una petición, se podrá comprobar si existe ya en dicho vector para cambiar el mensaje de respuesta que el servidor ofrecerá al cliente. Otro aspecto a destacar en el desarrollo del servidor es la implementación de otra ruta de la que no se había hablado hasta ahora: `/requests`. Este *endpoint* se ha implementado con dos fines: para monitorizar el estado de las peticiones de rescate recibidas hasta el momento, con propósitos de depuramiento, así como para que cualquier equipo de rescate pueda acceder a esta información y dirigirse a ayudar al usuario. De nuevo con propósitos de depuramiento se ha implementado una funcionalidad extra, y es que esta ruta acepta un parámetro *flush* para vaciar la lista de peticiones de ayuda recibidas.

4.4 Puesta en marcha y pruebas

Tras la configuración de las Raspberries, tan sólo será necesario encenderlas para poner todo en marcha, pues los scripts de arranque configurados se encargarán de iniciar todos los procesos necesarios en la máquina una vez esta esté en marcha. En cuanto a la aplicación Android, será suficiente con instalar el archivo `.apk` generado desde el IDE para tener la aplicación disponible y funcionando.

Con respecto a la distribución de las Raspsberries, se ha situado una a 2 metros del dispositivo móvil (Raspberry B, cuyo SSID es Rob-RPB) y otra a una distancia de 10 metros (Raspberry A, cuyo SSID es Rob-RPA).

Se realizarán a continuación una serie de pruebas en las que se pretende comprobar el correcto funcionamiento tanto de la aplicación Android como del servidor web:

- Prueba 01: Se arrancará la aplicación con el WiFi desconectado y el servicio de localización desactivado.

- Prueba 02: Se arrancará la aplicación con el WiFi desconectado y el servicio de localización activado, y se realizará una petición de rescate.
- Prueba 03: Se arrancará la aplicación con el WiFi conectado y el servicio de localización activado, y se realizará una petición para obtener el punto de ayuda más cercano.
- Prueba 04: Se desconectará el WiFi tras realizar al menos una petición.

4.4.1 Prueba 01

Para esta prueba se iniciará la aplicación con el servicio WiFi desconectado y el servicio de localización también desactivado. El usuario será bienvenido con el botón de "Start Scanning" y, tras hacer click en él, será informado de que el servicio WiFi está desactivado y está en proceso de ser activado mediante un *toast*.

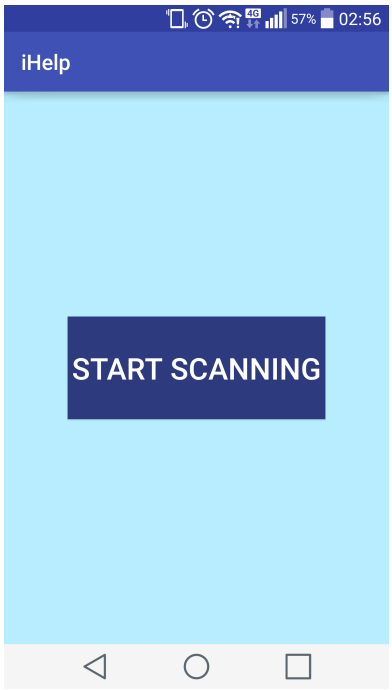

Figura 4.5 Prueba 01. Botón para empezar el escaneo de redes inalámbricas.

En este momento, el usuario recibirá un mensaje diciendo que el servicio de localización debe ser activado para poder utilizar la aplicación. Al hacer click en *Location settings* el usuario será redirigido a los ajustes de localización del teléfono, donde podrá activar dicho servicio.

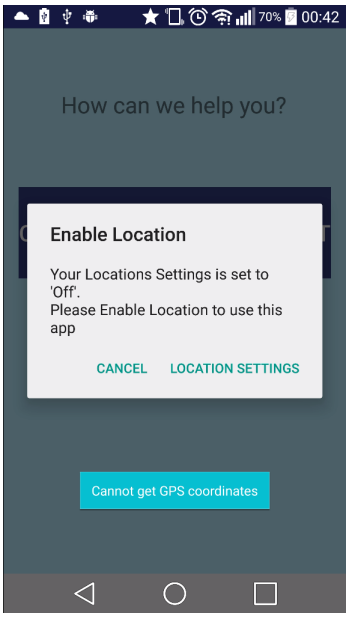

Figura 4.6 Prueba 01. Alerta para activar el servicio de localización.

Inmediatamente al regresar a la aplicación, la interfaz de usuario estará deshabilitada y un mensaje nos informará de que no tenemos coordenadas GPS, pues el dispositivo aún no ha sido capaz de obtener la posición del satélite.

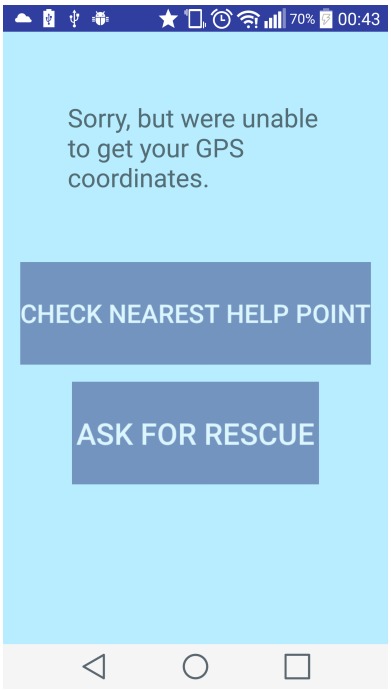

Figura 4.7 Prueba 01. Interfaz de usuario dehabilitada.

4.4.2 Prueba 02

Para esta prueba se iniciará la aplicación con el servicio WiFi desconectado y el servicio de localización activado. El usuario será de nuevo bienvenido con el botón de "Start Scanning" y, tras hacer click en él, como en el caso anterior, será informado de que el servicio WiFi está desactivado y está en proceso de ser activado mediante el mismo *toast* de la prueba anterior.

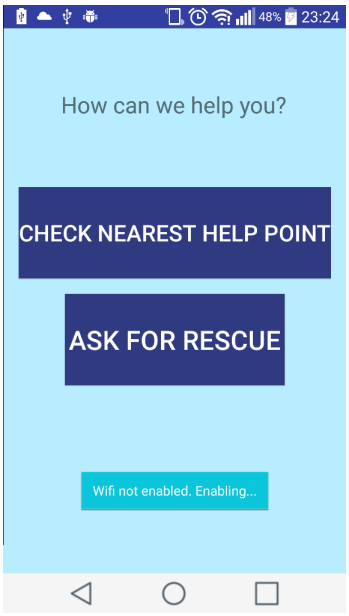

Figura 4.8 Prueba 02. *Toast* activando el Wifi.

Acto seguido, tras conectar a la red que se recibe con mayor intensidad (la que proviene de la Raspberry B), podemos observar la primera diferencia con el caso anterior. Los botones en este caso aparecen activados. Pasamos entonces a hacer click en el botón *Ask for rescue*, y la aplicación nos mostrará el mensaje del servidor informándonos de que la ayuda está en camino.

Figura 4.9 Prueba 02. Mensaje del servidor tras pedir un rescate.

Las coordenadas GPS han llegado al servidor y éste nos ha devuelto el mensaje de ayuda. Podemos comprobar en el servidor que el fichero log contiene una entrada correspondiente a nuestra llamada:

```
[RESCUE] Received request on /RESCUE from 192.168.42.14, params: 51.55721578 −0.98982799
[RESCUE] Returning 200: { msg: Worry not, help is on the way.}
```

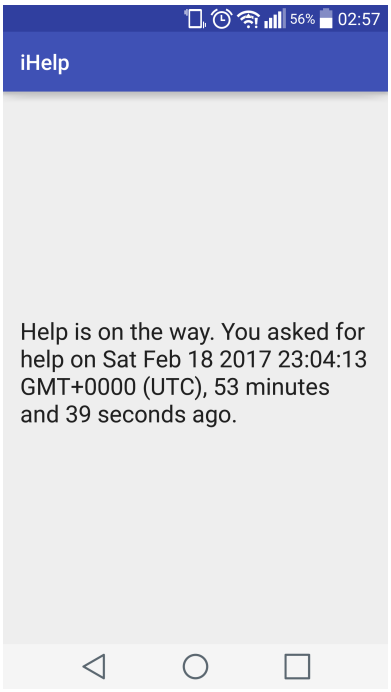

Figura 4.10 Prueba 02. Mensaje del servidor tras pedir un rescate por segunda vez.

Si volvemos ahora a la pantalla anterior, haciendo uso de la navegación de Android, y volvemos a utilizar el botón, obtendremos esta vez una respuesta diferente, y es que el servidor habrá registrado la llamada anterior, de forma que ahora indicará cuándo se realizó la llamada anterior y el tiempo que ha transcurrido desde entonces. De nuevo podemos comprobar en el log del servidor:

```
[RESCUE] Received request on /RESCUE from 192.168.42.14, params: 51.55721578 −0.98982799
[RESCUE] Returning 200 to an already existing request
```

4.4.3 Prueba 03

Para esta prueba se iniciará la aplicación con el servicio WiFi conectado y el servicio de localización activado. Al igual que en los casos anteriores, el usuario será de nuevo bienvenido con el botón de "Start Scanning". Tras hacer click en él, será informado de la conexión a la red inalámbrica (de nuevo RPB), y al contrario que en los casos anteriores, no verá el *toast* informado de que el WiFi está desactivado. Una vez de nuevo en el menú principal, haremos click esta vez en el botón de obtener el punto de ayuda más cercano.

La aplicación nos mostrará la respuesta del servidor, indicando un nombre identificativo del punto de ayuda más cercano junto a sus coordenadas GPS, así como una indicación de la distancia a la que se encuentra y en qué dirección.

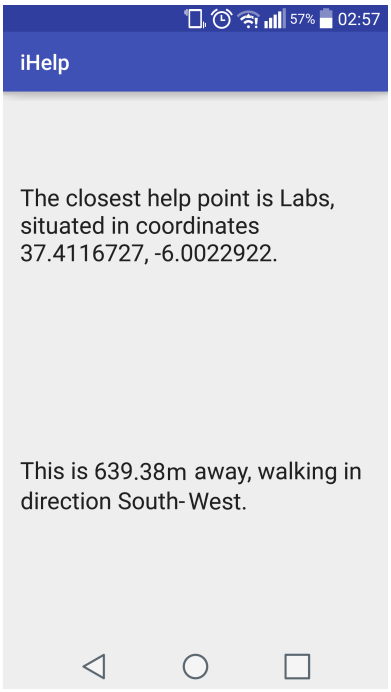

Figura 4.11 Prueba 03. Mensaje del servidor tras solicitar el punto de ayuda más cercano a nuestra posición.

4.4.4 Prueba 04

Para esta última prueba se iniciará de nuevo la aplicación con el servicio WiFi conectado y el servicio de localización activado. Tras escanear redes como en las pruebas anteriores, se procederá a desconectar el WiFi conel fin de simular un error en la red, o bien una desconexión real por parte del usuario. A continuación, al tratar de hacer click en alguno de los dos botones del menú principal, se puede observar cómo la aplicación se reinicia mostrando un *toast* al usuario informándole de la pérdida de conexión.

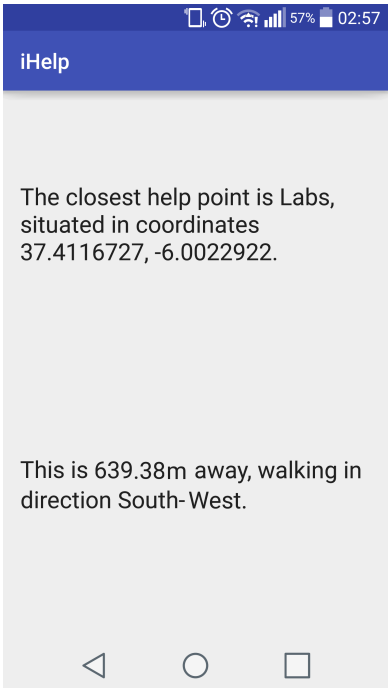

Figura 4.12 Prueba 04. *Toast* informando al usuario de la pérdida de conexión.