GRADO EN INGENIERÍA INFORMÁTICA

TRABAJO DE FINAL DE GRADO

DESARROLLO DE UNA APLICACIÓN MÓVIL CON LA FINALIDAD DE PROBAR EL INTERCAMBIO DE MENSAJES UDP ENTRE CLIENTE-SERVIDOR EN ANDROID STUDIO

Autor: Adrián SANCHIS SORIA

Tutor académico:
Pablo BORONAT PÉREZ
Supervisor:
Pablo BORONAT PÉREZ y
Miguel PÉREZ FRANCISCO

Fecha de lectura: 16 de septiembre de 2020

Curso académico 2019 /2020

Resumen

El proyecto redactado en este documento es un trabajo académico dirigido como consecuencia de la interrupción de la estancia en prácticas por la pandemia Covid-19. Este proyecto consiste en el desarrollo de una aplicación móvil en Android Studio para probar el sistema de comunicación UDP entre cliente y servidor. La aplicación móvil se basa en la detección de otros vehículos mientras iniciamos un trayecto con nuestro vehículo. Para el desarrollo del proyecto se emplea la metodología en cascada adaptada a la planificación del proyecto.

En la parte del cliente se envían diferentes mensajes UDP al servidor centralizado, entre estos mensajes se incluye la localización del teléfono móvil. En el servidor centralizado se traducen los mensajes UDP a peticiones HTTP y se envían a un servidor web. Tras realizar la petición HTTP, el servidor centralizado espera la respuesta del servidor web para de nuevo traducirla, esta vez a un mensaje UDP y enviarla al cliente de la aplicación móvil.

Palabras clave

Arquitectura cliente-servidor, mensajes UDP, aplicación móvil Android.

Keywords

Client-server architecture, UDP messages, Android mobile application.

Índice

Capítulo 1

Introducción

1.1. Trabajo previo

Durante la estancia en prácticas en la empresa Cuatroochenta S.L. me dediqué principalmente a adquirir los conocimientos de las tecnologías empleadas en la empresa. Este aprendizaje se desarrolló mediante tutoriales de la tecnología React. El objetivo de estas prácticas era el desarrollo de un motor de formularios para la plataforma Mi Yo Futuro.

El total de horas realizadas en la empresa fueron 90 horas. Este tiempo se puede desglosar principalmente en formación (60 h), también dediqué 10 horas en el diseño de prototipos de las diferentes pantallas. La última semana realicé 20 horas a la implementación de las primeras pantallas.

1.2. Contexto y motivación del proyecto

Como consecuencia del Covid-19, decidí suspender las prácticas en la empresa Cuatroochenta S.L. (1), ya que la empresa no ofreció la posibilidad de seguir las prácticas tutorizadas mediante teletrabajo, y opté por realizar un trabajo académico dirigido propuesto por el tutor Pablo Boronat Pérez.

El proyecto definido en este documento detalla el desarrollo de una aplicación móvil para el intercambio de mensajes UDP (*User Datagram Protocol*) (2) entre un servidor centralizado y un cliente en Android Studio. El protocolo UDP permite la transmisión sin conexión de datagramas. Este protocolo se clasifica en la capa de transporte. Además, no proporciona ningún tipo de control de flujo ni fiabilidad.

La motivación del proyecto principalmente es de carácter formativo, adquirir los conocimientos de las nuevas tecnologías utilizadas durante el proyecto, tanto de Android Studio como el sistema de comunicación de mensajes UDP. A su vez, otro motivo es poder realizar mi primera aplicación móvil.

La utilidad del proyecto es realizar pruebas de implementación de estos sistemas de comunicación (UDP) para comparar diferencias en las prestaciones, proporcionar ejemplos didácticos, o incluso utilizar este proyecto como bibliotecas reutilizables de código abierto. Probaremos este sistema de comunicación desarrollando una aplicación

en Android Studio donde los usuarios finales puedan detectar otros vehículos mientras ellos mismo realizan un trayecto con su propio vehículo.

1.3. Objetivos del proyecto

El principal objetivo de este proyecto consiste en obtener la máxima eficiencia en la implementación de intercambio de mensajes UDP entre un servidor centralizado y una aplicación Android.

El principal objetivo se puede desglosar en varios subobjetivos:

- Adquirir conocimientos de las tecnologías empleadas.
- Implementación *front-end* del cliente de la aplicación en Android Studio.
- Implementación de cliente y servidor del sistema de comunicación UDP.
- Implementación sistema de traducción de los mensajes UDP con un servidor con arquitectura API Rest.
- Emplear una metodología en cascada para el desarrollo del proyecto.
- Obtener una versión de la aplicación móvil en 210 horas empleadas para este trabajo.

El alcance del proyecto es desarrollar una pequeña aplicación móvil donde se pueda probar el sistema de comunicación de mensaje UDP. La aplicación debe poder intercambiar mensajes UDP con un servidor con diferentes tipos de mensajes como:

- Mensaje de alta: recibe confirmación de que está inscrito en el sistema de seguimiento. Si no recibe confirmación repite la operación hasta un número finito de intentos.
- Envía posición de geolocalización del teléfono móvil periódicamente.
- Pide estado actual de alarma periódicamente.
- Almacenar la pareja ID-IP para poder devolver el mensaje al cliente correcto.

El servidor centralizado UDP, además de comunicar con el cliente, debe traducir los mensajes para enviarlos a un servidor HTTP (Hypertext Transfer Protocol) (3)

1.4. Descripción del proyecto

El proyecto definido en este documento detalla el desarrollo de una aplicación móvil para probar el intercambio de mensajes UDP entre un servidor centralizado y un cliente en Android. Estos sistemas de comunicación son una parte de mensajería muy común para teléfonos móviles, por ejemplo, juegos o aplicaciones que usan geolocalización. La característica más importante de estos sistemas es el tiempo real, esto hace que nos proporcione un sistema interactivo.

El proyecto se divide en 2 partes, la parte del cliente y la parte del servidor centralizado. El cliente, como reacción a un evento, por ejemplo, que el usuario pulse un botón en la aplicación, enviará un mensaje UDP al servidor centralizado con los campos necesarios para su manipulación en el servidor. El servidor centralizado se dispondrá a recibir el mensaje UDP y realizar la petición HTTP correspondiente al servidor web dependiendo del contenido del mensaje UDP. El servidor web se desarrolla en paralelo por mi compañero Juan Luis Casañ.

Tras recibir la respuesta del servidor web, el servidor centralizado se dispondrá de nuevo a realizar la traducción, esta vez de la respuesta del servidor web a un mensaje UDP para poder enviarlo de vuelta al cliente. El cliente, tras recibir el mensaje UDP realizará su función en la aplicación, ya sea mostrando el mensaje o alguna alerta informando que se ha realizado la operación.

La situación inicial de la aplicación móvil que se ha implementado para probar el sistema de comunicación tiene la finalidad de que el usuario pueda detectar otros vehículos mientras este está en un trayecto con su propio vehículo. Para ello, cada cierto intervalo de tiempo se debe enviar la posición del vehículo y se debe avisar del estado actual, si existe una alarma o no.

El inicio de la aplicación contiene dos opciones. La primera es registrarse en el servidor, esta opción se ejecuta cada vez que utilizamos la aplicación. El servidor debe devolver el código que identifique al cliente. La otra opción en el inicio de la aplicación es iniciar un nuevo trayecto con un vehículo. Estos dos tipos de mensajes UDP requieren confirmación, es decir, esperan una respuesta durante un cierto tiempo y repiten la petición un número de intentos finito. Si se agotan los intentos, sale un diálogo de alerta indicando el error y dos opciones de reiniciar la aplicación o cerrar la aplicación.

Tras iniciar el trayecto, se envía periódicamente la geolocalización del teléfono móvil al servidor. También pide el estado actual de alarma, si la respuesta por parte del servidor es que existe alarma, se notifica con un aviso sonoro en la aplicación. Esta característica de aviso sonoro se puede configurar en la pantalla de configuración. En dicha pantalla podemos configurar otras opciones de la aplicación.

Cuando finalicemos el trayecto, se envía un mensaje al servidor indicando que el vehículo ya no se debe tener en cuenta, es decir, que borre los datos del trayecto y que no reciba ni el estado de alarma ni envíe actualizaciones de posición del vehículo. Por último, el cliente se puede dar de baja de la aplicación, borrando el código del cliente de la base de datos.

Las tecnologías que vamos a utilizar en este proyecto son:

- **Android Studio**: es el entorno de desarrollo integrado oficial para la plataforma Android. Esta herramienta la utilizamos para desarrollar una pequeña aplicación móvil a modo de prueba del sistema de comunicación UDP. (4)

- **Git**: es una herramienta de control de versiones para proyectos. Principalmente está orientada a código. Esta herramienta la utilizamos para almacenar nuestras versiones del proyecto en el servidor roure.act.uji.es. (5)

- **Postman** (versión de escritorio): permite crear bibliotecas de *endpoints* para ejecutarlos y probar nuestros servicios web. (6)

- **Lenguajes**: Tanto en el cliente de la aplicación como en el servidor centralizado programamos en Java. (7)

- **Pencil**: nos proporciona una herramienta de creación de prototipos GUI gratuita y de código abierto. Esta herramienta la utilizamos para crear los diferentes *mockups*[1] de la aplicación, diseñando las diferentes pantallas que tenga la aplicación. (8)

1.5. Estructura de la memoria

La estructura de esta memoria está distribuida en 5 capítulos. El primer capítulo es la introducción, en este se define el contexto, la motivación y la utilidad del proyecto. Además, también se define los objetivos y descripción del proyecto, concluyendo el capítulo con la estructura de la memoria.

[1] *mockups*: fotomontajes que permite a los diseñadores gráficos y web mostrar al cliente como quedan los diseños antes del desarrollo del proyecto.

Después de este capítulo introductorio, el segundo capítulo trata sobre la planificación del proyecto. En este capítulo, se define la planificación y metodología escogida. A su vez, se estiman los recursos y costes del proyecto.

En el tercer capítulo se detalla el análisis del sistema, el diseño de la arquitectura y la interfaz de usuario. El cuarto capítulo detalla la implementación realizada en el proyecto y las pruebas tanto de verificación como de validación.

Por último, la memoria finaliza con el capítulo de conclusiones. En este capítulo se presenta las conclusiones en varios ámbitos: formativo, profesional y personal.

Capítulo 2

Planificación del proyecto

2.1 Metodología

La metodología de trabajo escogida para la realización de este proyecto ha sido el desarrollo en cascada, también conocido como ciclo de vida de un programa o desarrollo secuencial (9). Esta metodología se basa en un proceso de diseño lineal, el cual ordena las fases del desarrollo del proyecto, de manera que hasta que no se termine una fase no se puede comenzar la siguiente. Al final de cada etapa, se realiza una revisión de la fase antes de comenzar la nueva fase.

En la figura 2.1.1, observamos todas las fases de la metodología en cascada. Las fases son las siguientes:

1. Análisis de los requisitos de software
2. Diseño del sistema
3. Desarrollo e implementación del programa
4. Verificación
5. Mantenimiento

El modelo en cascada funciona muy bien en proyectos pequeños, donde los requisitos de software están correctamente definidos. Debido a que el proyecto a realizar es una pequeña aplicación móvil a modo de demo para probar el sistema de comunicación UDP, creemos que la metodología escogida se adapta adecuadamente a las necesidades de nuestro proyecto. Este modelo es bastante simple, fácil de entender y ejecutar.

Otra gran ventaja que nos aporta este modelo a nuestro proyecto es un amplio esfuerzo de planificación previa, con lo cual vamos a poder estimar el tiempo y los recursos necesarios para cada fase con mayor precisión. Para ello realizaremos un diagrama de Gantt donde expondremos todas las fases, tareas, duración y dependencias que existan entre ellas.

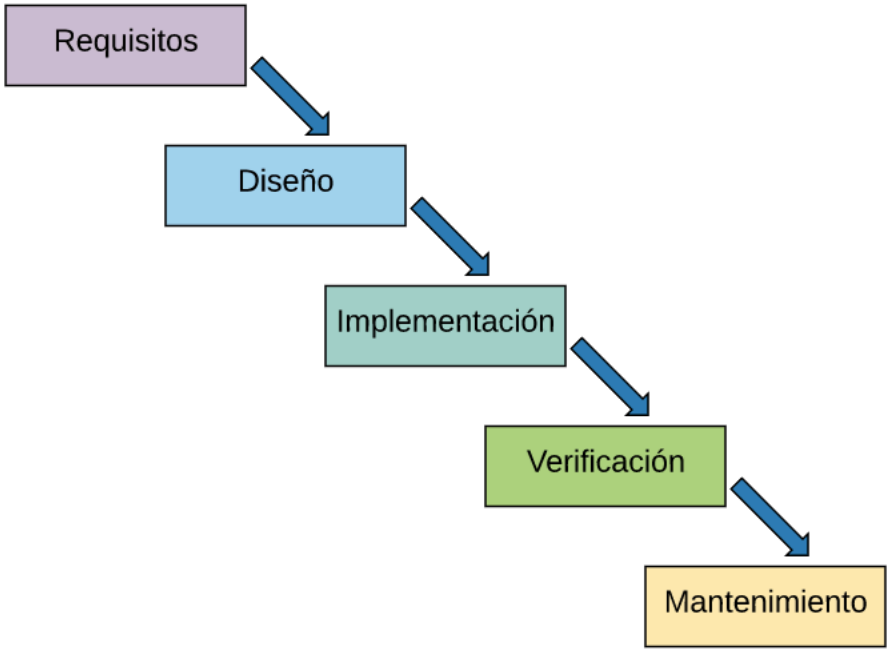


Figura 2.1.1. Metodología en cascada

2.2 Planificación inicial

En una planificación inicial, el proyecto lo hemos dividido en cinco fases: (1) inicio del proyecto, (2) definición del proyecto, (3) diseño, (4) desarrollo técnico del proyecto, (5) validación y pruebas. En la figura 2.2.1 podemos observar la estructura de descomposición del trabajo inicial.

La primera fase es una fase de formación acerca de las tecnologías y herramientas que se utilizaron durante la realización del proyecto. La principal tecnología en la que nos formamos fue Java en Android Studio, ya que toda la parte del cliente se realizó en dicho lenguaje con Android Studio. Vimos tutoriales para poder adquirir los conocimientos de la tecnología, más concretamente de Android de la universidad de Harvard (10). Este tutorial fueron cinco lecciones de una duración de una hora cada una para introducir Android Studio. En cada lección se presentaban conceptos teóricos que iban acompañados de una parte práctica. La parte práctica era el desarrollo de una pequeña aplicación. También vimos ejemplos de los sistemas de comunicación UDP (11), tratados en lenguaje Java, ya que en la parte del servidor también utilizamos este lenguaje. Además, consideramos estos ejemplos de cliente-servidor UDP implementados en Android Studio (12). Por último, tuve que mirar documentación acerca de cómo utilizar los hilos para que la aplicación se pueda ejecutar de manera paralela todos los servicios que le pidamos (13).

En la segunda fase, definición del proyecto, se analizó los requerimientos de software y se definió todas las funcionalidades de nuestra aplicación. Para ello realizamos con los

supervisores del trabajo una reunión donde detallamos todos los requisitos y funcionalidades que debía tener la aplicación. Algunos de ellos fueron los siguientes: las *Activities* de la aplicación en Android Studio, los diferentes tipos de mensaje que se intercambian entre cliente y servidor mediante UDP, el tamaño en bytes de los diferentes campos que se envían en los mensajes UDP, las peticiones que debe realizar al servidor web de mi compañero Juan Luis Casañ.

En la fase de diseño, realizamos los *mockups* (diseño de la interfaz de usuario) que tiene la aplicación. Diseñamos todas las pantallas, botones, colores, imágenes y la conexión entre diferentes ventanas de la aplicación. Todo ello con la herramienta de Pencil (8), que nos permite crear nuestros diseños tanto de páginas webs como aplicaciones móviles.

La cuarta fase es a la que dedicamos la mayor parte del tiempo. En esta fase desarrollamos todo el proyecto. La parte *front-end* (interfaz de usuario) del cliente, el intercambio de mensajes UDP de cliente a servidor y viceversa, el servidor centralizado, que es una interfaz de traducción de mensajes UDP a peticiones HTTP y las peticiones HTTP al servidor web.

Por último, en la última fase realizamos pruebas sobre lo implementado en la fase anterior.

Nº	Tareas	Tiempo (horas)	Dependencias
1	**Inicio del proyecto**	**50**	
1.1	Formación Android Studio	30	
1.1.1	Ejemplos Intercambio mensajes UDP	10	
1.1.2	Formación Threads	10	
2	**Definición del proyecto**	**10**	
2.1	Alcance y tareas	2	
2.2	Tutorias	8	
3	**Diseño**	4	2
3.1	MockUps	2	
3.2	Validar diseño	2	
4	**Desarrollo técnico del proyecto**	140	3
4.1	Implementación front-end cliente	60	
4.2	Implementación mensajes UDP de cliente a servidor	20	
4.3	Implementación servidor centralizado (interfaz de traducción)	20	
4.4	Implementación mensajes UDP de servidor a cliente	20	4.2
4.5	Peticiones HTTP	20	4.3
5	**Validación y pruebas**	6	
	Horas totales	210	

Figura 2.2.1. Estructura de descomposición de trabajo (EDT) inicial

Estimamos 50 horas en la primera fase del proyecto, ya que es importante formarnos adecuadamente y adquirir correctamente los conocimientos de las tecnologías para poder luego desarrollar el programa con mayor fluidez. La siguiente fase estimamos 10 horas, ya que las reuniones con el tutor son dos veces por semana de una duración de 30 min. La fase de diseño estimamos 4 horas para realizar todos los *mockups* (diseño de la interfaz de usuario) de la aplicación. La etapa de desarrollo estimamos 140 horas, sobre todo centrando 60 horas en la parte *front-end* (interfaz de usuario) de la aplicación móvil, ya que es la parte más extensa y no tenemos experiencia en la tecnología utilizada. Las otras tareas de esta fase estimamos una duración de 20 horas cada una. La última fase estimo unas 6 horas para probar la funcionalidad de la aplicación.

2.3 Seguimiento del proyecto

En cuanto al control del proyecto, se reservaron dos días a la semana (martes y jueves), para hacer reuniones con los supervisores. Las primeras reuniones tenían la finalidad de fijar los requisitos y funcionalidades de la aplicación. Después, las siguientes reuniones eran informativas, para saber en que fase del proyecto estábamos trabajando, y para recibir una retroalimentación por parte de los supervisores. También con la entrega de los informes quincenales, detallamos el trabajo realizado en esa quincena y el trabajo planeado para la quincena siguiente.

En referencia a la planificación final, añadimos una tarea más en la fase de desarrollo del proyecto. Esta tarea es la implementación de la localización del teléfono móvil de manera periódica. El motivo de añadir esta tarea y desviarnos de la planificación inicial es debido a que esta es de gran importancia para el desarrollo del proyecto, con lo cual hemos decidido dedicarle una tarea propia. La estimación de horas de esta tarea son 20 horas. También hemos reducido la cantidad de horas de la tarea de peticiones HTTP, ya que al tener el servidor web ya implementado por nuestro compañero Juan Luis Casañ, sabemos exactamente qué peticiones realizar al servidor web. La estructura de descomposición del trabajo final se muestra en la figura 2.3.1. Todo el seguimiento del proyecto lo podemos observar en la figura 2.3.3, que representa al diagrama de Gantt. Este diagrama de Gantt corresponde con la planificación real.

Nº	Tareas	Tiempo (horas)	Dependencias
1	**Inicio del proyecto**	**50**	
1.1	Formación Android Studio	30	
1.1.1	Ejemplos Intercambio mensajes UDP	10	
1.1.2	Formación Threads	10	
2	**Definición del proyecto**	**10**	
2.1	Alcance y tareas	2	
2.2	Tutorias	8	
3	**Diseño**	4	
3.1	MockUps	2	
3.2	Validar diseño	2	
4	**Desarrollo técnico del proyecto**	140	1.1
4.1	Implementación front-end cliente	50	
4.2	Implementación mensajes UDP de cliente a servidor	20	3.1
4.3	Implementación servidor centralizado	20	
4.4	Implementación mensajes UDP de servidor a cliente	20	4.2
4.5	Implementación de localización periodicamente	20	
4.6	Peticiones HTTP	10	4.3
5	**Validación y pruebas**	6	
	Horas totales	210	

Figura 2.3.1. Estructura de descomposición de trabajo (EDT) final

Los problemas que han ido surgiendo durante el desarrollo del proyecto han sido la mayoría relacionados con Android Studio, ya que es la primera vez que trabajamos sobre esta tecnología. La resolución de estos problemas ha sido formándose en la tecnología a base de tutoriales y documentación oficial de Android Studio. Una gran ventaja de Android Studio es la gran diversidad de clases y bibliotecas que incorpora y que puedes extender solamente añadiendo las dependencias al fichero *build.gradle,* como podemos observar en la figura 2.3.2. Otra página que nos ha resultado gratificante para la resolución de errores fue *stack overflow* (14)

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 29
    buildToolsVersion "29.0.1"
    defaultConfig {
        applicationId "com.example.udpclient"
        minSdkVersion 26
        targetSdkVersion 29
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'androidx.recyclerview:recyclerview:1.1.0'
    testImplementation 'junit:junit:4.13'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test:runner:1.2.0'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
    implementation 'com.google.android.gms:play-services-location:17.0.0'
    compileOnly group: 'org.json', name: 'json', version: '20160810'
}

```

Figura 2.3.2. Fichero build.gradle

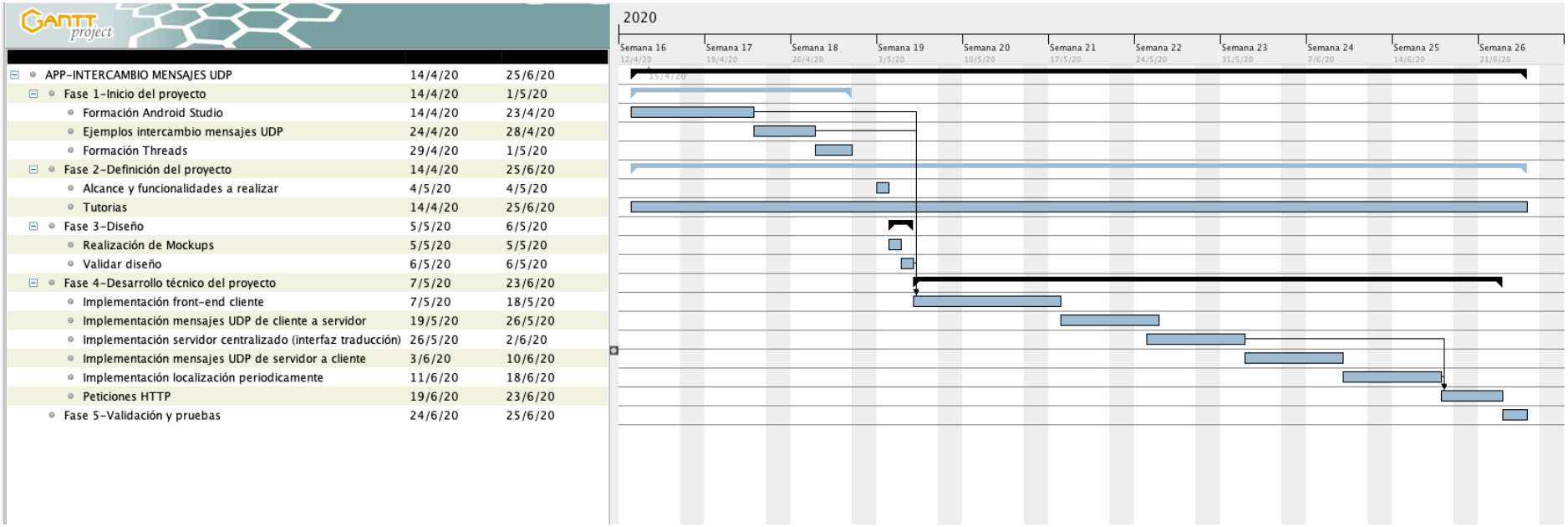


Figura 2.3.3 Diagrama de Gantt

2.4 Estimación de recursos y costes del proyecto

En este apartado vamos a estimar los recursos necesarios para poder desarrollar el proyecto. Los recursos son divididos en dos tipos: tecnológicos y humanos. Los recursos tecnológicos consisten en las herramientas informáticas empleadas para el desarrollo del proyecto y los equipos o material utilizado. Los recursos humanos son las personas que van a desarrollar la aplicación móvil, desde la formación hasta la implementación de la aplicación.

Haciendo referencia a los recursos humanos, la aplicación se realizó solamente por el autor de este documento, Adrián Sanchis Soria: Programador Junior.

En referencia a los recursos tecnológicos, las herramientas informáticas que utilizamos son las siguientes:

1. **Android Studio**: es el entorno de desarrollo integrado oficial para la plataforma Android. Esta herramienta la utilizamos para desarrollar una pequeña aplicación móvil a modo de prueba del sistema de comunicación UDP. Toda la parte de cliente se realizó en Android Studio. Además, incluye un emulador virtual propio de Android Studio para poder visualizar la aplicación.

2. **Git**: es una herramienta de control de versiones para proyectos. Principalmente está orientada a código. Esta herramienta la utilizamos para almacenar nuestras versiones del proyecto en el servidor roure.act.uji.es

3. **Postman** (versión de escritorio): permite crear bibliotecas de *endpoints* para ejecutarlos y probar nuestros servicios web. Realizamos pruebas de llamadas HTTP al servidor web de mi compañero Juan Luis Casañ.

4. **Pencil**: nos proporciona una herramienta de creación de prototipos GUI gratuita y de código abierto. Esta herramienta la utilizamos para crear los diferentes *mockups* de la aplicación, diseñando las diferentes pantallas que tenga la aplicación.

El material empleado para la realización del proyecto es el siguiente:

- Ordenador portátil Apple.
- Monitor Asus.
- Teclado y ratón Logitech.
- Teléfono móvil Redmi Note 4.

Por otra parte, vamos a estimar el coste del proyecto. Para ello, vamos a estimar el coste de los recursos humanos y de los tecnológicos. El coste de un programador junior en la Comunidad Valenciana es de 17.715 euros al año (15). Dividimos el coste medio anual entre el número de meses, (17.715 € / 12 meses). Esto da un resultado 1476,25 euros brutos al mes. Por lo tanto, tras dividir esta cantidad entre el número de horas mensuales que son 160 horas nos da un coste en horas de 9,227 €/hora.

Recurso humano	**Coste (hora)**	**Horas totales**	**Coste Total**
Programador Junior	9,227 €	210	1937,67 €

Tabla 2.4.1. Coste de recursos humanos

Respecto a los costes tecnológicos he de mencionar que no supondría ningún coste por la parte del coste de las herramientas, ya que son todas de licencia gratuita. Lo mismo ocurre con los costes indirectos, al realizar el trabajo dirigido desde casa, el coste es despreciable. Por la parte del material empleado, vamos a suponer que la vida útil de un ordenador y teléfono móvil son 5 años. Este proyecto tiene una duración de 3 meses, con lo cual sería el 5% de 60 meses que equivale a 5 años. El coste del material sería el siguiente:

Equipo	**Coste**
Ordenador portátil Apple (5% vida útil)	87,5 €
Monitor Asus	170 €
Teclado y ratón Logitech	30 €
Teléfono móvil Redmi Note 4 (5% vida útil)	10 €

Tabla 2.4.2. Coste de recursos tecnológicos

El coste total de los recursos tecnológicos son 297,5 euros. En total sumando ambos saldrían a 2.235,17 € el coste total del proyecto.

Capítulo 3

Análisis y diseño del sistema

Este tercer capítulo presenta el tema de análisis y diseño del sistema. En la parte de análisis, se especifican los requisitos y se realiza el modelado del sistema. Por la parte de diseño, tratamos el diseño de la arquitectura, es decir, los componentes del sistema y su interrelación. También indicamos los criterios para realizar la interfaz del sistema.

3.1 Análisis del sistema

En primer lugar, vamos a definir todos los requisitos funcionales que necesita nuestra aplicación móvil. Los requisitos funcionales los dividimos en dos partes, la parte de cliente o usuario final de la aplicación y la parte del servidor centralizado, que es la interfaz de traducción de mensaje UDP a peticiones HTTP.

El cliente tiene los siguientes requisitos funcionales:

- El cliente registra el teléfono móvil. El cliente envía un mensaje UDP de tipo registro al servidor. El mensaje UDP debe contener solamente el tipo de mensaje [TipoRegistro (1 *byte*)].
- El cliente inicia un nuevo trayecto. El cliente envía un mensaje UDP de tipo *start* al servidor. El mensaje UDP debe contener el tipo de mensaje, el id del teléfono y el tipo de vehículo con el que se inicia el trayecto [TipoStart (1 *byte*) | ID (36 caracteres) | TipoVehiculo (2 *bytes*)].
- El cliente cierra la aplicación tras no recibir respuesta del servidor al cabo de varios intentos.
- El cliente reinicia la aplicación tras no recibir respuesta del servidor al cabo de varios intentos.
- El cliente envía la localización del teléfono móvil periódicamente. Tras iniciar un trayecto, se envía un mensaje UDP de tipo posición al servidor. El mensaje debe contener el tipo de mensaje, el id del teléfono, la posición actual del teléfono móvil y una marca de tiempo [TipoPosicion (1 *byte*) | ID (36 caracteres) | Posicion (*longitude* 11 *bytes*, *latitude* 11 *bytes*) | MarcaTiempo (16 *bytes* formato ISO 8601)]. (16)

- El cliente pide el estado actual de alarma. Tras iniciar trayecto, el cliente pide periódicamente el estado actual de la alarma. El cliente envía un mensaje UDP de tipo consulta alarma al servidor. El mensaje debe contener el tipo de mensaje, el id del teléfono, el id de alama y la marca de tiempo [TipoConsultaAlarma (1 *byte*) | ID (36 caracteres) | IDAlarma (1 *byte*) | MarcaTiempo (16 *bytes* formato ISO 8601)].

- El cliente emite un aviso sonoro y muestra una alerta visual si tras consultar el estado de alarma (requisito anterior), la respuesta es afirmativa.

- El cliente puede activar o desactivar las alarmas.

- El cliente puede activar o desactivar el aviso sonoro de alarmas.

- El cliente finaliza el trayecto. El cliente envía un mensaje UDP al servidor de tipo *stop*. El mensaje UDP debe contener el tipo de mensaje y el id del teléfono móvil [TipoStop (1 *byte*) | ID (36 caracteres)].

- El cliente se da de baja de la aplicación, elimina sus datos de esta. El cliente envía un mensaje UDP al servidor de tipo *delete*. El mensaje UDP debe contener el tipo de mensaje y el id del teléfono móvil [TipoDelete (1 *byte*) | ID (36 caracteres)].

El servidor centralizado tiene los siguientes requisitos funcionales:

- El servidor realiza la petición HTTP de registrar el teléfono móvil al servidor web (POST).

- El servidor centralizado devuelve la solicitud de registrarse al cliente. El servidor envía un mensaje UDP de tipo registro al cliente. El mensaje UDP debe contener el tipo de mensaje y el id del teléfono móvil [TipoRegistro (1 *byte*) | ID (36 caracteres)].

- El servidor realiza la petición HTTP de iniciar un nuevo trayecto al servidor web (POST).

- El servidor centralizado devuelve la solicitud de iniciar trayecto al cliente. El servidor envía un mensaje UDP de tipo *start* al cliente. El mensaje UDP debe contener el tipo de mensaje y el id del teléfono móvil [TipoStart (1 *byte*) | ID (36 caracteres)].

- Tras iniciar un trayecto, el servidor centralizado realiza una petición HTTP para añadir un vehículo que tiene alarma (POST).

- El servidor envía y actualiza periódicamente la posición del teléfono móvil al servidor web, realiza una petición (PUT).

- El servidor realiza periódicamente la petición HTTP al servidor web de pedir el estado de alarma (GET).

- El servidor centralizado envía al cliente la respuesta de pedir el estado de alarma al servidor web. El servidor envía un mensaje UDP al cliente de tipo consulta de alarma. El mensaje UDP debe contener el tipo de mensaje, el id del teléfono móvil, el id de alarma, el estado de alarma y una marca de tiempo [TipoConsultaAlarma (1 byte) | ID (36 caracteres) | IDAlarma (1 byte) | alarma SI/NO (2 bytes) | MarcaTiempo (16 *bytes* formato ISO 8601)].

- El servidor realiza una petición HTTP de parar el trayecto al servidor web (DELETE).

- El servidor realiza una petición HTTP de borrar al cliente, es decir, el id del teléfono móvil, al servidor web (DELETE).

- El servidor centralizado envía al cliente un mensaje UDP de tipo *delete*. El mensaje UDP debe contener el tipo de mensaje y el id del teléfono móvil [TipoDelete (1 *byte*) | ID (36 caracteres)].

Todos los requisitos definidos anteriormente se muestran en la figura 3.1.1 que representa al diagrama de casos de uso. En la figura 3.1.2, se muestra el diagrama de clases, el cual describe la estructura del sistema mostrando las clases, operaciones, atributos y relaciones entre los diferentes objetos.

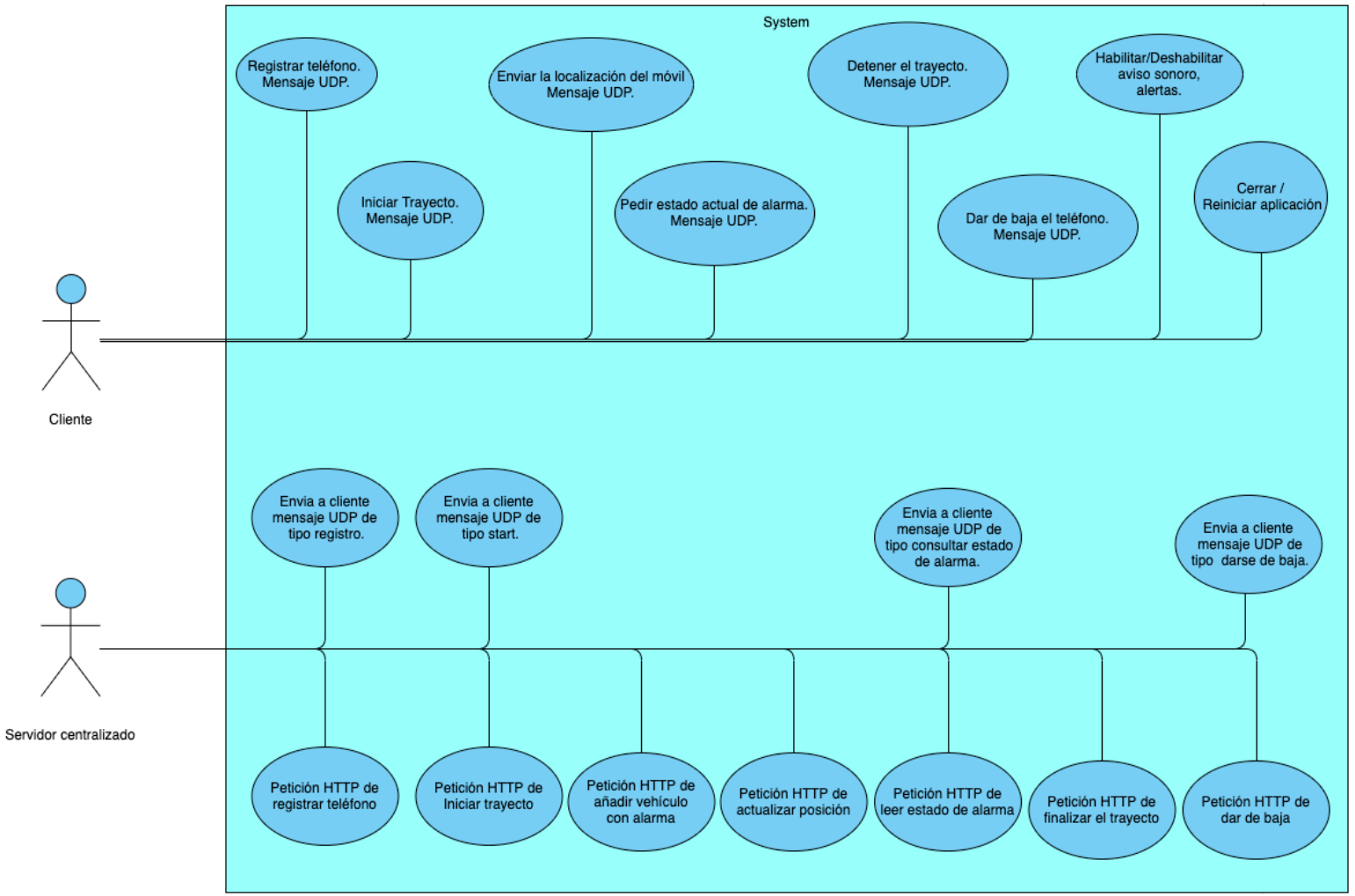


Figura 3.1.1. Diagrama de casos de usos

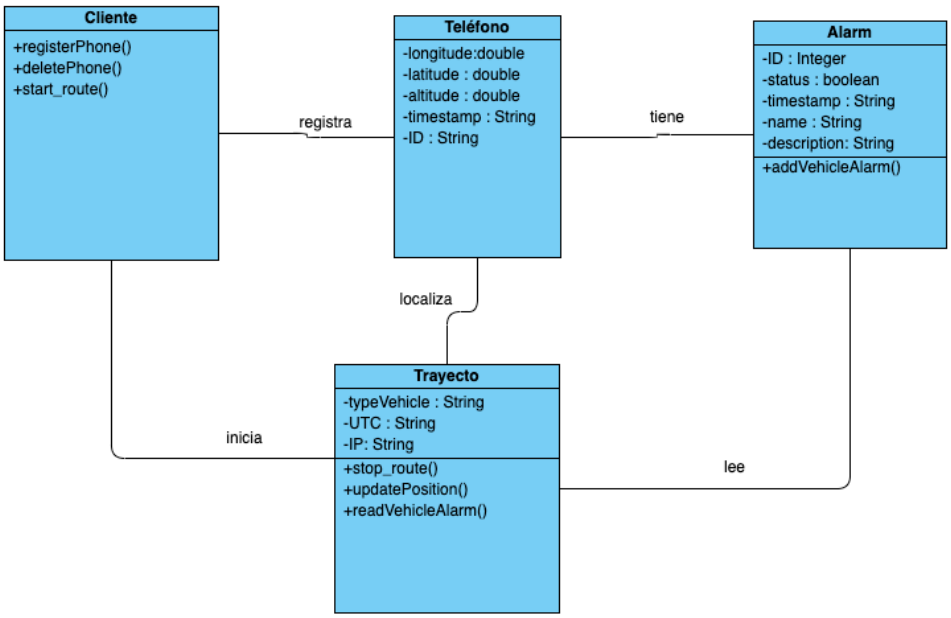


Figura 3.1.2 Diagrama de clases

3.2 Diseño de la arquitectura del sistema

En este apartado se va a definir el diseño de la arquitectura del sistema en función a dos tipos: la arquitectura física y la arquitectura lógica.

La arquitectura física corresponde a una **arquitectura cliente-servidor**. El usuario final se puede comunicar con el servidor centralizado mediante el protocolo UDP, este se basa en el intercambio de datagramas. El servidor centralizado se comunica con el servidor web de mi compañero Juan Luis Casañ mediante los servicios REST (*Representation State Transfer*) (17). El protocolo que utiliza los servicios REST es HTTP sin estado, esto incluye las operaciones CRUD (*CREATE, READ, UPDATE, DELETE*). Además, el formato utilizado para el intercambio de datos entre servidor centralizado y servidor web es JSON (*JavaScript Object Nation*) (18).

Respecto a la arquitectura lógica, corresponde a una **arquitectura de tres capas** (19): capa de presentación, capa de negocio y capa de datos. La capa de presentación es conocida como la interfaz gráfica del usuario final, es la que se presenta en la parte del cliente. La capa de negocio gestiona la lógica de la aplicación, es donde se decide qué se hace con los datos. Esta capa esta conectada a la capa de presentación para recibir las solicitudes del usuario y a la capa de datos para manipular algún dato de la base de datos. La capa de datos es el gestor de la base de datos. En la figura 3.2.1 podemos observar la representación de la arquitectura del sistema.

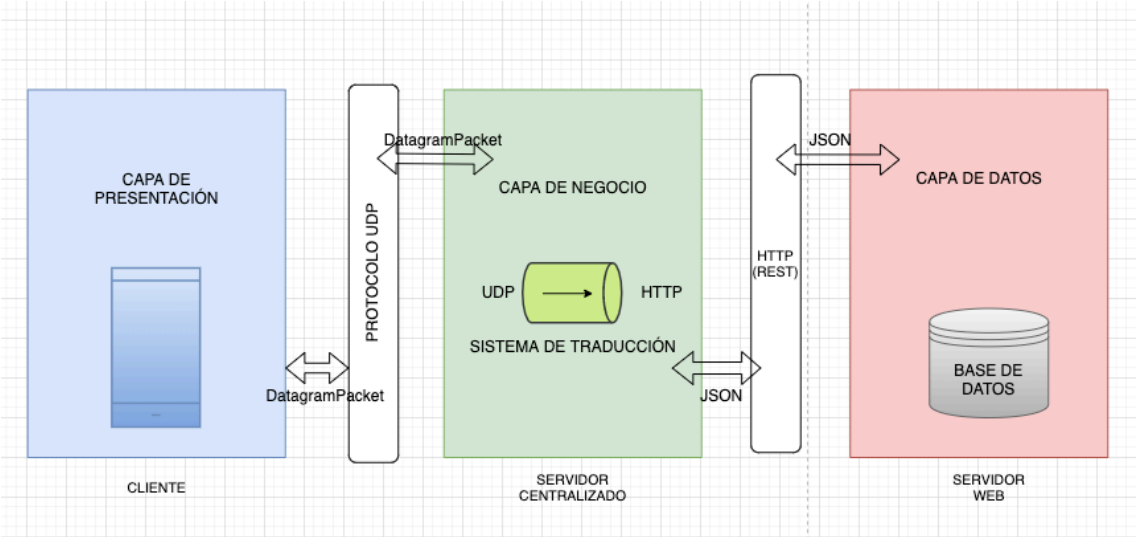


Figura 3.2.1 Arquitectura del sistema completo

3.3 Diseño de la interfaz

En referencia al diseño de la interfaz, se ha intentado seguir unas reglas básicas que todo diseño gráfico debe contener. La primera regla es la coherencia, tanto en diseño y ubicación de los diferentes elementos de una interfaz como de su funcionalidad. Otro criterio que hemos seguido es el diálogo eficiente con el usuario final, es decir, comunicarse con el usuario mediante textos, mensajes de error o colores de forma eficiente. Por último, hemos intentado que sea un diseño de interfaz sencillo y atractivo, con la cantidad suficiente de elementos para no distraer al usuario final.

Previamente hemos realizado cuatro *mockups* que resumen nuestra pequeña demo de la aplicación móvil. Con estas cuatro pantallas de la aplicación hemos podido probar nuestro sistema de comunicación UDP. Los *mockups* se han realizado con la herramienta Pencil (8), totalmente gratuita.

El primer *mockup* es la pantalla principal, donde el cliente se tiene que registrar si no lo está o iniciar trayecto si ya ha registrado su teléfono móvil. El botón de iniciar trayecto está en invisible hasta que se registre el usuario. Tras registrarse, el botón de registro cambia de estado a invisible. La figura 3.3.1 muestra el *mockup* de la pantalla principal.

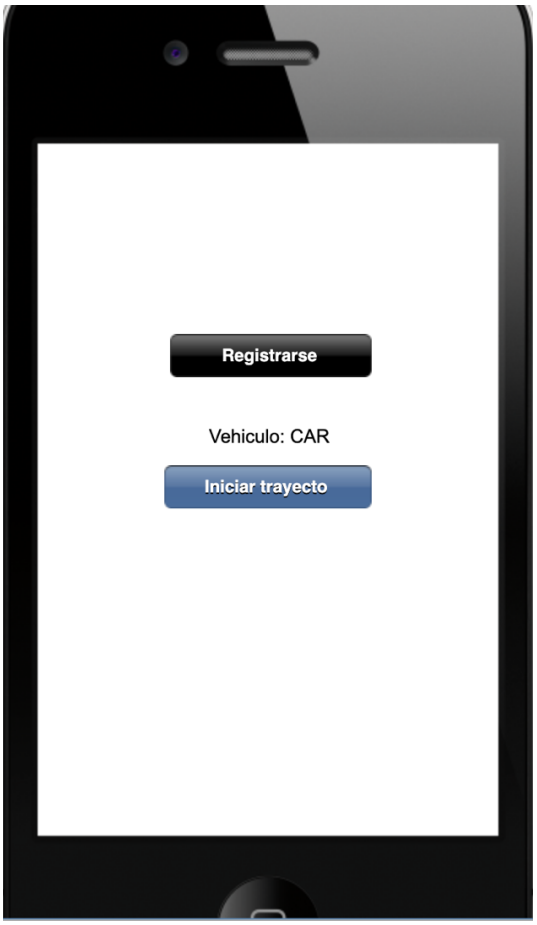


Figura 3.3.1 Boceto de la pantalla de registro y iniciar trayecto

El segundo *mockup* está relacionado con el primero. En este si al cabo de varios intentos de registro o inicio de trayecto no conecta con el servidor, aparece la ventana emergente que le permite al usuario cerrar la aplicación o reiniciar la misma. En la figura 3.3.2 podemos observar este *mockup*.

Figura 3.3.2 Boceto de la ventana emergente

El tercer *mockup* es la pantalla del trayecto, en ella se muestra la localización del teléfono móvil periódicamente. Si el estado de alarma es afirmativo, aparece una señal de advertencia. También, podemos parar el trayecto mediante un botón de color rojo. Este tercer *mockup* se muestra en la figura 3.3.3.

Figura 3.3.3. Boceto de la pantalla de trayecto

Por último, el cuarto *mockup* hace referencia a la pantalla de configuración, donde podemos habilitar o deshabilitar algunas opciones de la aplicación. También en esta pantalla podemos dar de baja el teléfono registrado. Este *mockup* se muestra en la figura 3.3.4.

El botón *switch* de alarmas nos permite decidir si queremos realizar las peticiones de pedir el estado de alarma al servidor web. El botón *switch* de aviso sonoro nos permite decidir si queremos sonido al recibir las alarmas. La siguiente opción que tenemos es el vehículo por defecto, esta opción nos permite cambiar el vehículo con el que iniciamos el trayecto. Es un desplegable con varias opciones de vehículos. Tras configurar todas las opciones disponibles, el botón de aplicar cambios nos guarda el estado de dichas opciones para la siguiente vez que entremos en la aplicación.

Figura 3.3.4. Boceto de la pantalla de configuración

Capítulo 4

Implementación y pruebas

En este capítulo vamos a presentar la implementación y las pruebas que se han hecho en el desarrollo de nuestra aplicación. Para ello, vamos a separar la implementación de la aplicación en dos partes: cliente y servidor centralizado, como hemos comentado anteriormente en la arquitectura del sistema.

4.1. Detalles de implementación

En esta sección se detalla el trabajo de programación realizado en las diferentes partes del sistema. Además, se expone las estrategias o patrones utilizados y las dificultades que hemos tenido a la hora de programar, así como las soluciones adoptadas.

4.1.1 Estructura del proyecto

La estructura principal de la aplicación la hemos separado en tres paquetes: `Client`, `Server` y `Models`. El paquete de cliente contiene el cliente del sistema de comunicación UDP y las diferentes pantallas de la interfaz de usuario. El paquete de servidor contiene el servidor del sistema de comunicación UDP y las diferentes peticiones HTTP que realizamos al servidor web. Por último, el paquete de modelos contiene las clases de los objetos que utilicemos. En la figura 4.1.1.1 podemos observar la estructura general del proyecto.

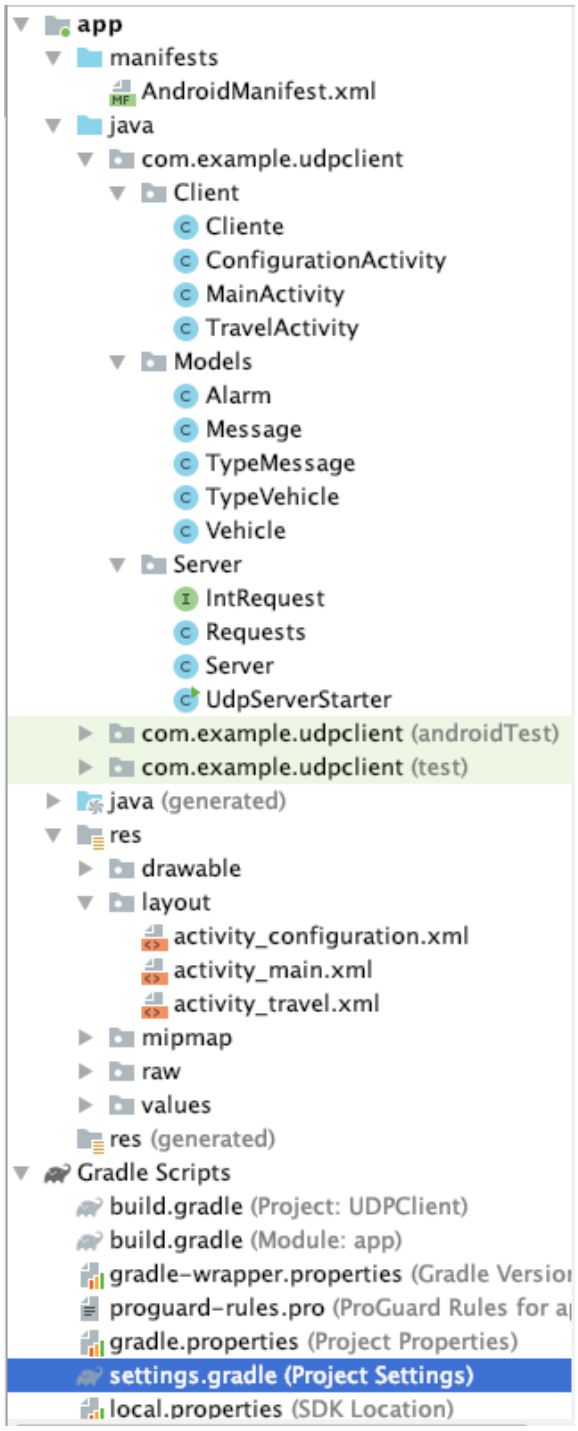


Figura 4.1.1.1 Estructura general del proyecto

En el paquete `Models` definimos las clases necesarias para representar un objeto. En la figura 4.1.1.2 podemos observar la clase `TypeMessage`, donde definimos seis variables estáticas para representar a cada tipo de mensaje. Utilizamos el abecedario para inicializar estas variables, ya que solamente necesitamos 1 *byte* para diferenciar un tipo de mensaje de otro. Cuando enviemos los mensajes UDP de cliente a servidor o viceversa, llamaremos a estas variables para representar el primer *byte* del mensaje UDP.

```java
package com.example.udpclient.Models;

public class TypeMessage {
    public final static String TIPO_REGISTER = "A";
    public final static String TIPO_START = "B";
    public final static String TIPO_POSICION = "C";
    public final static String TIPO_ALARMA = "D";
    public final static String TIPO_STOP = "E";
    public final static String TIPO_DELETE="F";
}
```

Figura 4.1.1.2 Fichero TypeMessage.java

Al enviar el mensaje UDP de iniciar trayecto, nos surgió un problema en el campo tipo de vehículo. El problema fue que, si enviábamos el nombre del vehículo como campo, el mensaje iba a tener diferentes tamaños de *bytes* dependiendo del vehículo seleccionado. Entonces, no podíamos recuperar el campo tipo de vehículo en el servidor, ya que no sabíamos cuantos *bytes* iba a ocupar en el mensaje UDP. Como solución optamos por definir un modelo de tipo de vehículo, donde realizamos las operaciones necesarias para poder enviar el campo tipo de vehículo en el mensaje UDP.

En el modelo tipo de vehículo, definimos cada vehículo con un código de 2 *bytes*, en nuestra demo de momento solamente tenemos 2 vehículos: *car* y *bike*, representando el código 01 y 02 respectivamente. En la interfaz de usuario se representa el nombre del vehículo, mientras que para enviar el mensaje UDP utilizamos el código del vehículo en el campo tipo de vehículo, ya que siempre ocupa 2 *bytes* en el mensaje. Por ello, necesitamos 2 métodos de traducción. El primero, la traducción del nombre del tipo de vehículo al código. Y, el segundo es la traducción de código a nombre del tipo de vehículo. Este modelo lo podemos observar en la figura 4.1.1.3.

```java
package com.example.udpclient.Models;


import java.util.HashMap;

public class TypeVehicle {

    HashMap<String, String> vehicles = new HashMap<~>();  //VEHICULO, CODIGO


    public TypeVehicle(){
        vehicles.put("car", "01");
        vehicles.put("bike","02");
    }

    public String getCodVehicle(String type){
        if(vehicles.containsKey(type))
            return vehicles.get(type);
        return null;
    }

    public String getTypeVehicle(String cod){
        for(String k: vehicles.keySet()){
            String v = vehicles.get(k);
            if (v.compareTo(cod)==0);
                return k;
        }
        return null;
    }


}
```

Figura 4.1.1.3 Fichero TypeVehicle.java

4.1.2 Cliente UDP

En primer lugar, vamos a exponer todo lo relacionado con el cliente. Tanto la clase cliente que envía y recibe mensajes UDP como toda la parte de interfaz de usuario. Esta parte de interfaz de usuario la dividiremos en las diferentes *Activities*[2] que tenga la aplicación móvil.

La comunicación UDP se basa en el intercambio de mensajes datagrama entre cliente y servidor. El fichero `Cliente.java` es donde realizamos el cliente UDP. Este fichero contiene 2 métodos: `send(byte [] m) y receive()`. Todas las líneas de código comentadas en este apartado hacen referencia a la figura 4.1.2.

En el método `send()` se crea el paquete datagrama con el *array* de bytes que se le pasa como argumento, la longitud del *array* de *bytes*, la dirección IP y el puerto del

[2] *Activity*: es un componente principal de la interfaz gráfica de una aplicación Android. Todas las *Activities* tienen un ciclo de vida y, por ello tienen esencialmente cuatro estados: activa, visible, parada, eliminada.

servidor (hemos escogido el puerto 5000) (línea 30). Tras crear el paquete, se envía mediante el *socket* inicializado en el constructor. Se añade un temporizador para que el servidor responda al mensaje enviado (línea 31-33). Si el servidor no responde en el tiempo establecido (5000 ms), saltará la excepción `SocketTimeoutException`, en el método `receive()`. Este método se repite un número finito de veces. Si se agotan los intentos de respuesta, el método `receive()` devuelve `null`. Podemos observar esta excepción en el código desde la línea 52 hasta la 61 de la figura 4.1.2.

En el método `receive()` recogemos la respuesta por parte del servidor centralizado. Para ello, creamos un *array* de tamaño 100 *bytes*. Seguidamente inicializamos el paquete datagrama con el *array* de *bytes* inicializado anteriormente y, recibimos el paquete mediante el *socket* (líneas 43-47). Tras recibir el paquete, cerramos el *socket*, reiniciamos el contador de intentos a 0 y devolvemos la respuesta en un *string* (líneas 49-52).

```java
public class Cliente{

    int attempts = 0;
    int serverPort = 5000;
    DatagramSocket socket;

    Cliente(){
        try {
            socket = new DatagramSocket();
        } catch (SocketException e) {
            e.printStackTrace();
        }
    }
    public void send(byte [] m){
        try{
            InetAddress local = InetAddress.getByName("192.168.0.101");
            DatagramPacket p = new DatagramPacket(m, m.length, local, serverPort);
            socket.setSoTimeout(5000);
            socket.send(p);
            Log.d( tag: "cliente",  msg: "Mensaje enviado a servidor");

        } catch (SocketException e) {
            Log.d( tag: "cliente",  msg: "Error de socket AL ENVIAR", e);
        } catch (UnknownHostException e) {
            Log.d( tag: "cliente",  msg: "Error de servidor", e);
        } catch (IOException e) {
            Log.d( tag: "cliente",  msg: "Error en el envio del paquete", e);
        }
    }
    public String receive(){
        try{
            byte [] buffer = new byte[100];
            DatagramPacket respuesta = new DatagramPacket(buffer, buffer.length);
            socket.receive(respuesta);

            String res = new String(respuesta.getData());
            socket.close();
            attempts=0;
            return res;
        } catch(SocketTimeoutException e) {
            if (attempts==3) {
                System.out.println("INTENTO nº:" + attempts);
                return null;
            }
            else {
                attempts += 1;
                System.out.println("INTENTO :" +attempts);
                receive();
            }
        } catch (SocketException e) {
            Log.d( tag: "cliente",  msg: "Error de socket", e);

        } catch (UnknownHostException e) {
            Log.d( tag: "cliente",  msg: "Error de servidor", e);
        } catch (IOException e) {
            Log.d( tag: "cliente",  msg: "Error en el envio del paquete", e);
        }
        return null;
```

Figura 4.1.2 Clase cliente UDP

4.1.3 MainActivity

En referencia a las *Activities* de la aplicación, exponemos los detalles de programación de 3 de ellas. La primera *Activity* que vamos a tratar de la aplicación móvil es la actividad principal: `MainActivity.java`. Esta clase va a extender a la clase

`AppCompatActivity`, que es la clase base para actividades que desean utilizar algunas de las características principales de Android Studio. En esta clase tendremos principalmente 2 métodos: `onCreate()` y `onActivityResult()`.

El método `onCreate()` es donde se inicializa la actividad. Lo más importante de este método es la llamada a `setContentView()` para asociar la actividad con un recurso de diseño y usar `findViewById()` para asociar los diferentes elementos del recurso de diseño con la clase y poder interactuar con ellos.

En esta actividad vamos a tener como elementos de la interfaz de usuario básicamente tres botones y un campo para mostrar texto: el botón de registrar el teléfono móvil, el botón de iniciar trayecto y el botón de configuración. Cada elemento de la interfaz de usuario lo identificamos con un id diferente para diferenciarlos y, lo asociamos en el método `onCreate()` para poder interactuar con esos elementos en la clase.

Habilitamos o deshabilitamos los botones dependiendo de si el teléfono móvil está registrado o no. Por ello, inicializamos una variable `idClient` a `null`. Entonces, el estado inicial de la actividad tiene solamente el botón de registro en visible y los demás elementos están en invisible o deshabilitados (líneas 57-67 figura 4.1.3.1).

```
25  import static com.example.udpclient.Models.TypeMessage.*;
26
27
28  public class MainActivity extends AppCompatActivity{
29
30      private Button buttonRegister;
31      private Button buttonStart;
32      private ImageButton buttonConfiguration;
33      private TextView typeVehicleText;
34      private TextView typeVehicleSpinner;
35      public String idClient = null;
36      public boolean switchAlarm = true;
37      private byte[] buffer;
38      public static final int REQUEST_CODE = 1;
39
40
41
42      @Override
43      protected void onCreate(Bundle savedInstanceState) {
44          super.onCreate(savedInstanceState);
45          setContentView(R.layout.activity_main);
46
47          buttonConfiguration = findViewById(R.id.button_configuration);
48          buttonRegister = findViewById(R.id.button_register);
49          buttonStart = findViewById(R.id.button_start);
50          typeVehicleText = findViewById(R.id.type_vehicle_view);
51          typeVehicleSpinner = findViewById(R.id.type_vehicle_default);
52          typeVehicleSpinner.setText("car");
53
54          StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
55                  .permitNetwork().build());
56
57          if (idClient == null) {  //ESTADO INICIAL
58              buttonRegister.setVisibility(View.VISIBLE);
59              buttonStart.setVisibility(View.INVISIBLE);
60              buttonConfiguration.setVisibility(View.INVISIBLE);
61              typeVehicleText.setVisibility(View.INVISIBLE);
62              typeVehicleSpinner.setVisibility(View.INVISIBLE);
63
64
65          } else {
66              buttonRegister.setVisibility(View.INVISIBLE);
67          }
68
```

Figura 4.1.3.1 Inicialización de variables y estado inicial de la pantalla de la actividad `MainActivity`

En el botón de registro, añadimos un evento que al pulsar sobre él se crea el mensaje UDP de tipo registro. Este mensaje UDP solamente tiene un campo que es el tipo de mensaje. A continuación convertimos el mensaje a un *array* de *bytes* llamando al método `getBytes()`. Seguidamente inicializamos un nuevo objeto `Cliente` y, enviamos el *array* de *bytes* con el método `send(buffer)`. Tras enviar, tenemos que recoger la respuesta UDP que nos llegará del servidor. Si la respuesta recibida por el servidor centralizado es `null`, significa que el cliente no se ha registrado correctamente y aparece un diálogo de alerta que nos indica el error ocurrido y nos da 2 opciones a realizar: cerrar la aplicación o reiniciarla.

En cambio, si el cliente se registró correctamente y el servidor nos devuelve una respuesta adecuada, recogemos el código de cliente realizando un *substring* desde el carácter 1 hasta el carácter 37 sobre la respuesta del servidor. Esto es debido a que un identificador único universal (UUID) ocupa un total de 36 caracteres y sabemos que ocupa esa posición, como hemos definido en los requisitos funcionales. Tras recuperar el código de cliente, habilitamos los demás elementos y deshabilitamos el botón de registrarse. El evento de registro lo podemos observar en la figura 4.1.3.2.

```
70          buttonRegister.setOnClickListener(new View.OnClickListener() {
71              @Override
72              public void onClick(View v) {
73                  String mensaje = TIPO_REGISTER
74                  buffer = mensaje.getBytes(StandardCharsets.UTF_8);
75                  Cliente registerObj = new Cliente();
76                  registerObj.send(buffer);
77                  String res = registerObj.receive();
78
79                  if (res == null) {  //SI ENTRA AQUI ES QUE CLIENTE NO HA IDO BIEN. DEVUELVE NULL
80                      Toast.makeText( context: MainActivity.this,  text: "Cliente no registrado ", Toast.LENGTH_SHORT).show();
81                      AlertDialog.Builder alertTimeOut = new AlertDialog.Builder(v.getContext());
82                      alertTimeOut.setTitle("Error: TimeOut");
83                      alertTimeOut.setMessage("Se ha acabado el número de intentos");
84                      alertTimeOut.setNegativeButton( text: "Cerrar app", (dialog, which) → {
87                              finish();
88                      });
90                      alertTimeOut.setPositiveButton( text: "Reiniciar app", (dialog, which) → {
93                              onRestart();
94                      });
96                      alertTimeOut.show();
97
98                  } else {
99                      idClient = res.substring(1, 37);
100                     System.out.println(idClient);
101                     Toast.makeText( context: MainActivity.this, text: "Cliente registrado ",Toast.LENGTH_SHORT).show();
102                     buttonStart.setEnabled(true);  //HABILITAMOS LOS BOTONES
103                     buttonConfiguration.setEnabled(true);
104                     buttonRegister.setVisibility(View.INVISIBLE);
105                     buttonStart.setVisibility(View.VISIBLE);
106                     typeVehicleText.setVisibility(View.VISIBLE);
107                     buttonConfiguration.setVisibility(View.VISIBLE);
108                     typeVehicleSpinner.setVisibility(View.VISIBLE);
109                 }
110             }
111
112
113         });
114
```

Figura 4.1.3.2. Evento de registro de teléfono móvil

Al pulsar el botón de iniciar trayecto, se produce otro evento, repetimos el patrón anterior de crear el mensaje UDP, esta vez de tipo *start_route* y sus campos correspondientes (definidos en los requisitos). Convertimos el mensaje a un *array* de *bytes* con el método `getBytes()` y enviamos el *array* al servidor mediante el método `send()` del nuevo `Cliente` inicializado. Tras enviarlo, esperamos la respuesta por parte del servidor. Si no llega ninguna respuesta es que algo ha fallado, con lo cual

saltará el diálogo de alerta de nuevo con las opciones de cerrar o reiniciar aplicación. En cambio, si la respuesta del servidor es adecuada extraemos el código del cliente de la respuesta como en el evento anterior e inicializamos un objeto `Intent`[3] con el contexto de la vista y la clase `TravelActivity`. Añadimos al `intent` el código del cliente extrayendo con un *substring* la respuesta del servidor desde el carácter 1 hasta el carácter 37 y la variable switchAlarm. Por último, arrancamos la actividad `TravelActivity`. Este evento lo localizamos en la figura 4.1.3.3 desde la línea 115 hasta la línea 156.

```
115         buttonStart.setOnClickListener(new View.OnClickListener() {
116             @Override
117             public void onClick(View v) {
118                 TypeVehicle typeVehicle = new TypeVehicle();
119                 String mensaje = TIPO_START + idClient + typeVehicle.getCodVehicle(typeVehicleSpinner.getText().toString());   /
120                 System.out.println("MENSAJE A ENVIAR:" +mensaje);
121                 buffer = mensaje.getBytes(StandardCharsets.UTF_8); //getBytes devuelve un nuevo new [] byte en este caso de 39.
122                 Cliente startObj = new Cliente();
123                 startObj.send(buffer);
124                 String res = startObj.receive();
125
126                 if (res == null) {   //SI ENTRA AQUI ES QUE CLIENTE NO HA IDO BIEN. DEVUELVE NULL
127                     AlertDialog.Builder alertTimeOut = new AlertDialog.Builder(v.getContext());
128                     alertTimeOut.setTitle("Error: TimeOut");
129                     alertTimeOut.setMessage("Se ha acabado el número de intentos");
130                     alertTimeOut.setNegativeButton( text: "Cerrar app", (dialog, which) → {
133                             finishAffinity();
134                     });
136                     alertTimeOut.setPositiveButton( text: "Reiniciar app", (dialog, which) → {
139                             Intent intent = new Intent( packageContext: MainActivity.this, MainActivity.class);
140                             startActivity(intent);
141                     });
143                     alertTimeOut.show();
144                 } else {
145                     idClient = res.substring(1, 37);
146
147                     System.out.println("UUID:" + idClient);
148
149                     Intent intent = new Intent(v.getContext(), TravelActivity.class);
150                     intent.putExtra( name: "id", res.substring(1, 37));
151                     intent.putExtra( name: "switchAlarm", switchAlarm);
152                     v.getContext().startActivity(intent);
153                 }
154
155
156             }
157         });
158         buttonConfiguration.setOnClickListener(new View.OnClickListener() {
159             @Override
160             public void onClick(View v) {
161                 Intent intent = new Intent(v.getContext(), ConfigurationActivity.class);
162                 intent.putExtra( name: "id", idClient);
163                 startActivityForResult(intent, REQUEST_CODE);
164
165             }
166         });
167     }
168     @Override
169     protected void onActivityResult(int requestCode, int resultCode, Intent data) {
170         super.onActivityResult(requestCode,resultCode,data);
171
172         if((requestCode == REQUEST_CODE) && (resultCode == RESULT_OK)) {
173             typeVehicleSpinner.setText(data.getStringExtra( name: "type"));
174             switchAlarm = data.getBooleanExtra( name: "alarm", defaultValue: true);
175         }
176     }
177 }
178
```

Figura 4.1.3.3. Eventos de iniciar trayecto, evento del botón de configuración y método `onActivityResult()`

[3] Intent: es un objeto de mensajería que se puede usar para solicitar una acción de otro componente de una aplicación. Su uso más significativo es en lanzamiento de actividades, es la conexión entre dos actividades.

El tercer botón que vamos a exponer es el de la configuración. Tras pulsar sobre él inicializamos un nuevo objeto `Intent` con el contexto de la vista y la clase `ConfigurationActivity`. Añadimos los datos necesarios al `intent`, en este caso solamente el id del cliente. Arrancamos la actividad con el método `startActivityForResult()`, es decir, que de la actividad que arrancamos esperamos recibir un resultado que se comenta a continuación. Este evento se muestra en la figura 4.1.3.3 desde la línea 158 hasta la línea 168.

Por último, vamos a comentar el método `onActivityResult()`. Este consiste en la recuperación de datos entre la actividad de configuración y la actividad principal. La principal función es interconectar los datos enviados desde una actividad secundaria a una actividad principal. En este caso, recuperamos los datos del tipo de vehículo escogido y el resultado del estado del conmutador de alarmas. El método lo podemos observar en la figura 4.1.3.3 en las líneas 169-177.

4.1.4 ConfigurationActivity

La segunda actividad que vamos a exponer es la actividad de configuración. En el fichero `ConfigurationActivity.java` es donde tenemos la clase `ConfigurationActivity`, de nuevo extiende a la clase `AppCompatActivity`. En esta clase tenemos solamente el método `onCreate()`. Dentro de este método, tenemos 5 eventos que se activan al pulsar sobre el botón específico en la interfaz de usuario de esta pantalla**.** Los eventos son los siguientes: selección del vehículo para iniciar trayecto, habilitar o deshabilitar alarmas y avisos sonoros, aplicar cambios y dar de baja el teléfono móvil.

En referencia a la selección del vehículo optamos por usar un *spinner* [4] como elemento seleccionador de los vehículos. Cada opción del *spinner* se añadía a un *array* de *strings*, en nuestra demo añadimos solamente 2 vehículos: *car* y *bike*. También había que iniciar un objeto `ArrayAdapter` con el contexto de la actividad y los vehículos añadidos al *array* y, asociarlo al elemento *spinner* en la interfaz gráfica (líneas 71-76 figura 4.1.4.1.)

El evento de selección de vehículo se produce cuando pulsamos un elemento del *spinner*. Como consecuencia, inicializamos un objeto `SharedPreference.Editor` para garantizar que el valor del *spinner* permanece en un estado cuando se produzca el

[4] Spinner: elemento que ofrece un desplegable en el que poder seleccionar una opción de las múltiples posibles.

evento de aplicar cambios, es decir, que nos guardamos el estado del *spinner* (línea 91-108 figura 4.1.4.1).

```
54      @Override
55      protected void onCreate(Bundle savedInstanceState) {
56          super.onCreate(savedInstanceState);
57          setContentView(R.layout.activity_configuration);
58
59          context = this;
60          id = getIntent().getStringExtra( name: "id");
61
62          buttonConfiguration = findViewById(R.id.button_configuration);
63          titleTextView = findViewById(R.id.title_configuration_text);
64
65          titleTextView.setBackgroundColor(Color.BLACK);
66
67          switchAlarmView = findViewById(R.id.switch_alarm);
68          switchSoundView = findViewById(R.id.switch_sound);
69
70
71          //SELECTOR DE VEHICULOS
72          spinnerVehicle = findViewById(R.id.type_vehicle_spinner);
73          elems.add("coche");
74          elems.add("bici");
75          ArrayAdapter adp = new ArrayAdapter( context: ConfigurationActivity.this, android.R.layout.simple_spinner_dropdown_item, elems);
76          spinnerVehicle.setAdapter(adp);
77
78
79          buttonDeletePhone = findViewById(R.id.button_delete);
80          buttonSaveChanges = findViewById(R.id.button_save_changes);
81
82
83          //GUARDAR VALORES SI APLICAS CAMBIOS
84          SharedPreferences sharedPrefs = getSharedPreferences( name: "switchSound", MODE_PRIVATE);
85          switchSoundView.setChecked(sharedPrefs.getBoolean( key: "switchSound",  defValue: true));
86          sharedPrefs = getSharedPreferences( name: "switchAlarm", MODE_PRIVATE);
87          switchAlarmView.setChecked(sharedPrefs.getBoolean( key: "switchAlarm", defValue: true));
88          sharedPrefs = getSharedPreferences( name: "defaultVehicle", MODE_PRIVATE);
89          spinnerVehicle.setSelection(sharedPrefs.getInt( key: "posicion", defValue: 0));
90
91          spinnerVehicle.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
92              @Override
93              public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
94                  String e = (String) spinnerVehicle.getAdapter().getItem(position);
95                  SharedPreferences.Editor editor = getSharedPreferences( name: "defaultVehicle", MODE_PRIVATE).edit();
96                  int valor = spinnerVehicle.getSelectedItemPosition();
97                  editor.putInt("posicion",valor);
98                  editor.commit();
99
100                 Toast.makeText( context: ConfigurationActivity.this, text: "Seleccionaste: "+ e,Toast.LENGTH_SHORT).show();
101
102             }
103
104             @Override
105             public void onNothingSelected(AdapterView<?> parent) {
106
107             }
108     });
```

Figura 4.1.4.1 Evento de vehículo seleccionado

El siguiente evento por comentar es el de habilitar y deshabilitar las alarmas. Para este evento utilizamos el elemento `Switch` en la interfaz de usuario. Este elemento se basa en un conmutador de únicamente dos estados. El usuario puede pulsar sobre él y el estado del conmutador cambiará. Cuando cambie el estado de este conmutador, se inicializa un `SharedPreference.Editor` de nuevo para almacenar el estado del conmutador. Esta vez se almacena un dato booleano. Este evento lo podemos localizar en la figura 4.1.4.2. desde la línea 111 hasta la línea 128.

Respecto al evento de habilitar y deshabilitar el sonido, repetimos el mismo patrón que el evento anterior. Utilizamos de nuevo otro elemento `Switch` en la interfaz de usuario, ya que se trata únicamente de 2 estados: sonido activado o sonido desactivado. Cuando se produzca el evento, es decir, que se pulse sobre el conmutador, se inicializa otro objeto `SharedPreference.Editor` para almacenar el estado del

conmutador. También, inicializamos un objeto `AudioManager`[5]. Si el conmutador está habilitado, ajustamos el volumen al gusto del usuario. Si el conmutador está deshabilitado, deshabilitamos el sonido. En la figura 4.1.4.2 podemos observar este evento, desde la línea 130 hasta la línea 149.

```
110
111         switchAlarmView.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
112             @Override
113             public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
114                 if (isChecked){
115                     SharedPreferences.Editor editor = getSharedPreferences( name: "switchAlarm", MODE_PRIVATE).edit();
116                     editor.putBoolean("switchAlarm",true);
117                     editor.commit();
118
119
120                 }
121                 else{
122                     SharedPreferences.Editor editor = getSharedPreferences( name: "switchAlarm", MODE_PRIVATE).edit();
123                     editor.putBoolean("switchAlarm",false);
124                     editor.commit();
125
126                 }
127             }
128         });
129
130         switchSoundView.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
131             @Override
132             public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
133                 AudioManager audioManager = (AudioManager)getSystemService(Context.AUDIO_SERVICE);
134                 if (isChecked){
135                     SharedPreferences.Editor editor = getSharedPreferences( name: "switchSound", MODE_PRIVATE).edit();
136                     editor.putBoolean("switchSound",true);
137                     editor.commit();
138                     switchSoundView.setSoundEffectsEnabled(true);
139                     audioManager.adjustStreamVolume(AudioManager.STREAM_MUSIC, AudioManager.ADJUST_RAISE, flags: 1);
140                 }
141                 else{
142                     SharedPreferences.Editor editor = getSharedPreferences( name: "switchSound", MODE_PRIVATE).edit();
143                     editor.putBoolean("switchSound",false);
144                     editor.commit();
145                     switchSoundView.setSoundEffectsEnabled(false);
146                     audioManager.adjustStreamVolume(AudioManager.STREAM_MUSIC, AudioManager.ADJUST_MUTE, flags: 1);
147                 }
148             }
149         });
150
151
```

Figura 4.1.4.2 Eventos habilitar/deshabilitar alarmas y sonido

El evento de dar de baja el teléfono se produce cuando se pulsa el botón de "darse de baja". Cuando se pulsa, se crea el mensaje UDP de tipo *delete*. El mensaje UDP de tipo *delete* contiene como campos el tipo del mensaje y id del cliente. El id del cliente lo recuperamos llamando al método `getIntent()` (línea 60 figura 4.1.4.1.). Tras crear el mensaje se convierte a un *array* de *bytes*, se inicia un nuevo objeto `Cliente` y se envía el *array* de *bytes* con el método `send()`. Cuando se reciba la respuesta del servidor, volvemos a la pantalla inicial mediante un objeto `Intent` inicializado a la clase `MainActivity` (líneas 153-171 figura 4.1.4.3).

Por último, en el evento de aplicar cambios inicializamos un nuevo objeto `Intent` y le añadimos como datos el vehículo seleccionado en el *spinner* y el estado del conmutador de alarmas. Tenemos que modificar el resultado a OK para que la primera actividad (`MainActivity`) pueda recuperar los parámetros añadidos al `intent`. En la figura 4.1.4.3 observamos este evento (líneas 175-187). La figura 4.1.4.4 muestra el transito de los datos desde la actividad `ConfigurationActivity` a la actividad

[5] AudioManager: objeto que nos proporciona acceso al volumen y al control del modo timbre.

`MainActivity`. Se recuperan los datos en el método `onActivityResult()` comentado anteriormente.

```
buttonDeletePhone.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {  //BORRAR CLIENTE, sale alerta y volvemos a la pantalla principal ya que no tenemos ID.

        String mensaje = TIPO_DELETE + id;  //MENSAJE TIPO BORRAR CLIENTE
        System.out.println(mensaje);
        buffer = mensaje.getBytes(StandardCharsets.UTF_8);
        Cliente deleteObj = new Cliente();
        deleteObj.send(buffer);
        String res = deleteObj.receive();
        System.out.println("Cliente Borrado: " + res);

        Toast.makeText( context: ConfigurationActivity.this, text: "Cliente borrado ",Toast.LENGTH_SHORT).show();

        Intent intent = new Intent(v.getContext(), MainActivity.class);
        v.getContext().startActivity(intent);

    }
});


buttonSaveChanges.setOnClickListener(new View.OnClickListener() {  //APLICAR CAMBIOS, con los opciones guardadas, regresamos a la pantalla anterior con onBackPressed()
    @Override
    public void onClick(View v) {
        String e = (String) spinnerVehicle.getSelectedItem();
        boolean alarm = switchAlarmView.isChecked();
        Intent intent = new Intent();
        intent.putExtra( name: "type",e);
        intent.putExtra( name: "alarm",alarm);
        setResult(RESULT_OK, intent);
        finish();
        //onBackPressed();
    }
});
```

Figura 4.1.4.3 Eventos de darse de baja y aplicar cambios

Figura 4.1.4.4 Tránsito de datos de `ConfigurationActivity` a `MainActivity`

4.1.5 TravelActivity

La última actividad que vamos a detallar es la actividad del trayecto. En el fichero `TravelActivity.java` tenemos la clase `TravelActivity` que extiende a la clase `AppCompatActivity`. En esta clase básicamente es donde actualizamos la geolocalización del teléfono móvil y realizamos la petición del estado de alarma periódicamente.

En primer lugar, para poder pedir la geolocalización del teléfono necesitamos crear una nueva instancia de `FusedLocationProviderClient` para usar los servicios de geolocalización. La clase `FusedLocationProviderClient` extiende a la clase abstracta `GoogleApi`. Creamos la solicitud de localización. Para ello configuramos el intervalo de actualización de la solicitud de localización a la variable estática `UPDATE_INTERVAL`, que equivale a 10 segundos. El intervalo rápido lo configuramos a la variable `FASTEST_INTERVAL`, equivalente a 5 segundos (método `createLocationRequest()` en las líneas 166-170 de la figura 4.1.5.1.) Tras crear la solicitud, comienza la actualización de localización. La instancia `fusedLocationClient` llama al método `requestLocationUpdates` con la solicitud de localización y un *callback* de localización como parámetros (método `starLocationUpdate()` en las líneas 186-191 de la figura 4.1.5.1.). El *callback* está inicializado en el método `onCreate()`. Cabe mencionar que la actualización de la geolocalización del teléfono móvil solamente funciona si los permisos adecuados están importados en el fichero AndroidManifest.xml. Los permisos son los siguientes: `ACCESS_FINE_LOCATION` y `ACCESS_COARSE_LOCATION`.

```
165
166     protected void createLocationRequest() {
167         locationRequest = LocationRequest.create();
168         locationRequest.setInterval(UPDATE_INTERVAL);
169         locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
170         locationRequest.setFastestInterval(FASTEST_INTERVAL);
171
172     }
173
174
175
176
177     @Override
178     protected void onResume() {
179         super.onResume();
180         if (requestingLocationUpdates) {
181             startLocationUpdates();
182         }
183     }
184
185
186     private void startLocationUpdates(){
187         if (checkLocationPermission())
188             fusedLocationClient.requestLocationUpdates(locationRequest, locationCallback, looper: null);
189         else
190             Log.d( tag: "gps",  msg: "EN resumen-La app no tienen los permisos necearios: " );
191     }
192
193     @Override
194     protected void onPause() {
195         super.onPause();
196         stopLocationUpdates();
197     }
198
199     private void stopLocationUpdates(){
200         fusedLocationClient.removeLocationUpdates(locationCallback);
201
202
203     }
204
```

Figura 4.1.5.1 Métodos `createLocationRequest()`, `onResume()`, `startLocationUpdates()`, `onPause()` y `stopLocationUpdates()` de la actividad `TravelActivity`

En el *callback* de localización, como podemos observar en la figura 4.1.5.2 (líneas 90-107), tenemos un método `onLocationResult()` donde mostramos en la interfaz de usuario las actualizaciones de geolocalización periódicas del teléfono móvil según el

intervalo de tiempo configurado. Para ello, modificamos los elementos de texto de la interfaz de usuario de longitud y latitud.

```
69      @Override
70      protected void onCreate(Bundle savedInstanceState) {
71          super.onCreate(savedInstanceState);
72          setContentView(R.layout.activity_travel);
73          ActivityCompat.requestPermissions( activity: this, new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, requestCode: 1);
74
75          fusedLocationClient = LocationServices.getFusedLocationProviderClient( activity: this);
76
77          imageAlert = findViewById(R.id.image_alert);
78          imageAlert.setVisibility(View.INVISIBLE);
79          lastLonTextView = findViewById(R.id.last_longitude_text_view);
80          lastAltTextView = findViewById(R.id.last_altitude_text_view);
81          lastLatTextView = findViewById(R.id.last_latitude_text_view);
82
83          buttonStopTravel = findViewById(R.id.button_stop_travel);
84
85          id = getIntent().getStringExtra( name: "id");
86          switchAlarm = getIntent().getBooleanExtra( name: "switchAlarm", defaultValue: true);
87
88          createLocationRequest(); //Creamos la solicitud de Localización
89
90          locationCallback = new LocationCallback() {
91              @Override
92              public void onLocationResult(LocationResult locationResult) {
93                  super.onLocationResult(locationResult);
94                  if (locationResult == null) {
95                      return;
96                  }
97                  for (Location location : locationResult.getLocations()) {
98                      System.out.println("Ultima localización: " + location);
99                      lastLonTextView.setText(formato.format(location.getLongitude()));
100                     lastLatTextView.setText(formato.format(location.getLatitude()));
101
102                     updatePosition();  //CLIENTE UDP ENVIA ACTUALIZACION POSICIONç
103                     getStateAlarm(); //CLIENTE PIDE ESTADO DE ALARMA
104
105                 }
106             }
107         };
108
109
110         buttonStopTravel.setOnClickListener((v) → {
113                 onPause();
114                 String mensajeStop = TIPO_STOP + id;
115                 buffer = mensajeStop.getBytes(StandardCharsets.UTF_8);
116                 Cliente stopObj = new Cliente();
117                 stopObj.send(buffer);
118                 Toast.makeText( context: TravelActivity.this, text: "Trayecto terminado ",Toast.LENGTH_SHORT).show();
119                 onBackPressed();
120
121         });
123     }
124
```

Figura 4.1.5.2 Método `onCreate()` de la clase `TravelActivity`

Seguidamente realizamos las llamadas a los métodos `updatePosition()` y `getStatealarma()` que se encargarán de realizar el mensaje UDP correspondiente y enviarlo al servidor. El método `updatePosition()` se encarga de crear el mensaje UDP de tipo posición, con los diferentes campos requeridos. Tras crearlo repetimos el mismo patrón que las anteriores veces, lo convertimos a un *array* de *bytes* con el método `getBytes()` y lo enviamos al servidor inicializando un nuevo objeto `Cliente`. Este método lo podemos observar en la figura 4.1.5.3 desde la línea 125 hasta la línea 133.

En el otro método `getStateAlarm()`, si el conmutador de alarma está activado creamos el mensaje UDP de tipo alarma, con todos los campos correspondientes a este tipo de mensaje. En el campo del código de alarma establecemos como estático el código de alarma a 1, de momento solamente utilizaremos una alarma. De nuevo,

convertimos el mensaje a un *array* de *bytes* y lo enviamos al servidor centralizado. Recuperamos el estado de alarma en la respuesta por parte del servidor. Si el estado de alarma es afirmativo, se mostrará la imagen de alerta en la interfaz de usuario y se avisará con sonido de la alerta si el conmutador de sonido esta activado. Este método se muestra en la figura 4.1.5.3 en las líneas 135-153.

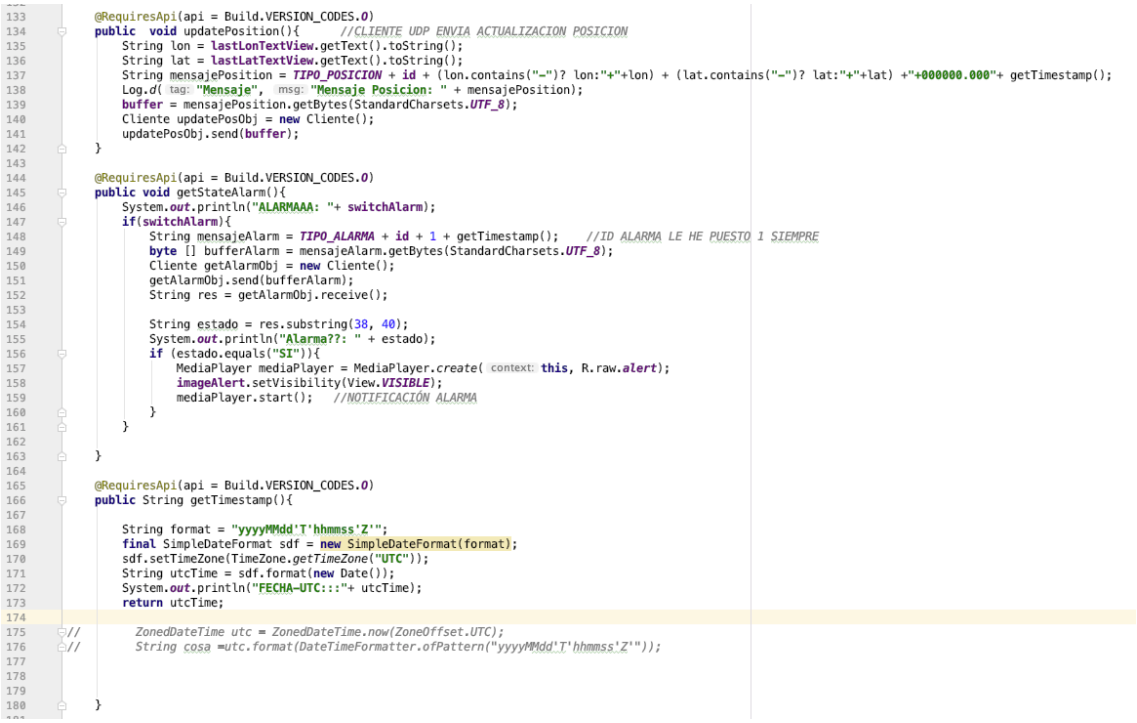


Figura 4.1.5.3 Métodos `updateLocation()`, `getStateAlarm()` y `getTimestamp()` de la actividad `TravelActivity`

Por último, vamos a hablar del botón de parar el trayecto. Al pulsar sobre este elemento en la interfaz de usuario, llamamos al método `onPause()` para que pare todas las actualizaciones de localización del teléfono móvil. También en este evento, se crea el mensaje UDP de tipo *stop_route*, con todos sus campos correspondientes (*stop_route* y id). El id del cliente lo recuperamos llamando al método `getIntent()` (línea 85 de la figura 4.1.5.2.). Seguimos el mismo patrón utilizado con todos los mensajes UDP, convertimos el mensaje a un *array* de *bytes* y lo enviamos al servidor centralizado mediante el método `send()`. Tras enviarlo, volvemos a la actividad anterior mostrando una notificación de que el trayecto ha sido terminado. Este evento lo podemos localizar en la figura 4.1.5.2 entre las líneas 110 y 123.

4.1.6 Servidor UDP

El fichero que define el servidor UDP es `UdpServerStarter.java`. Este fichero contiene la clase `UdpServerStarter`. Esta clase es donde se recibe el mensaje datagrama del cliente para luego tratar con él. Para ello, se inicializa un objeto `DatagramSocket` con la misma IP que el cliente y el mismo puerto. Recordemos que

elegimos el puerto 5000. Inicializamos un mensaje datagrama y lo recibimos con el método `receive()` del socket. Tras recibir el mensaje inicializamos un nuevo `Thread`[6] de un nuevo objeto `Server` pasándole como parámetros el mensaje datagrama y el *socket*. En la figura 4.1.6.1 se muestra la clase `UdpServerStarter`.

```
12
13  public  class UdpServerStarter{
14
15      public  static void main(String[] args){
16          int serverPort =5000;
17
18          byte[] buffer = new byte[100];
19
20          try {
21              InetAddress serverIp = InetAddress.getByName("192.168.0.111");
22              DatagramSocket socket = new DatagramSocket(serverPort,serverIp);
23              ///POSIBLE IP
24              while (true){
25                  System.out.println("Esperando mensaje");
26                  DatagramPacket p = new DatagramPacket(buffer, buffer.length);
27                  socket.receive(p);
28                  System.out.println("Mensaje recibido");
29                  Thread h = new Thread(new Server(p,socket));
30                  h.start();
31              }
32          } catch (UnknownHostException e) {
33              e.printStackTrace();
34          } catch (SocketException e) {
35              e.printStackTrace();
36          } catch (IOException e) {
37              e.printStackTrace();
38          }
39      }
```

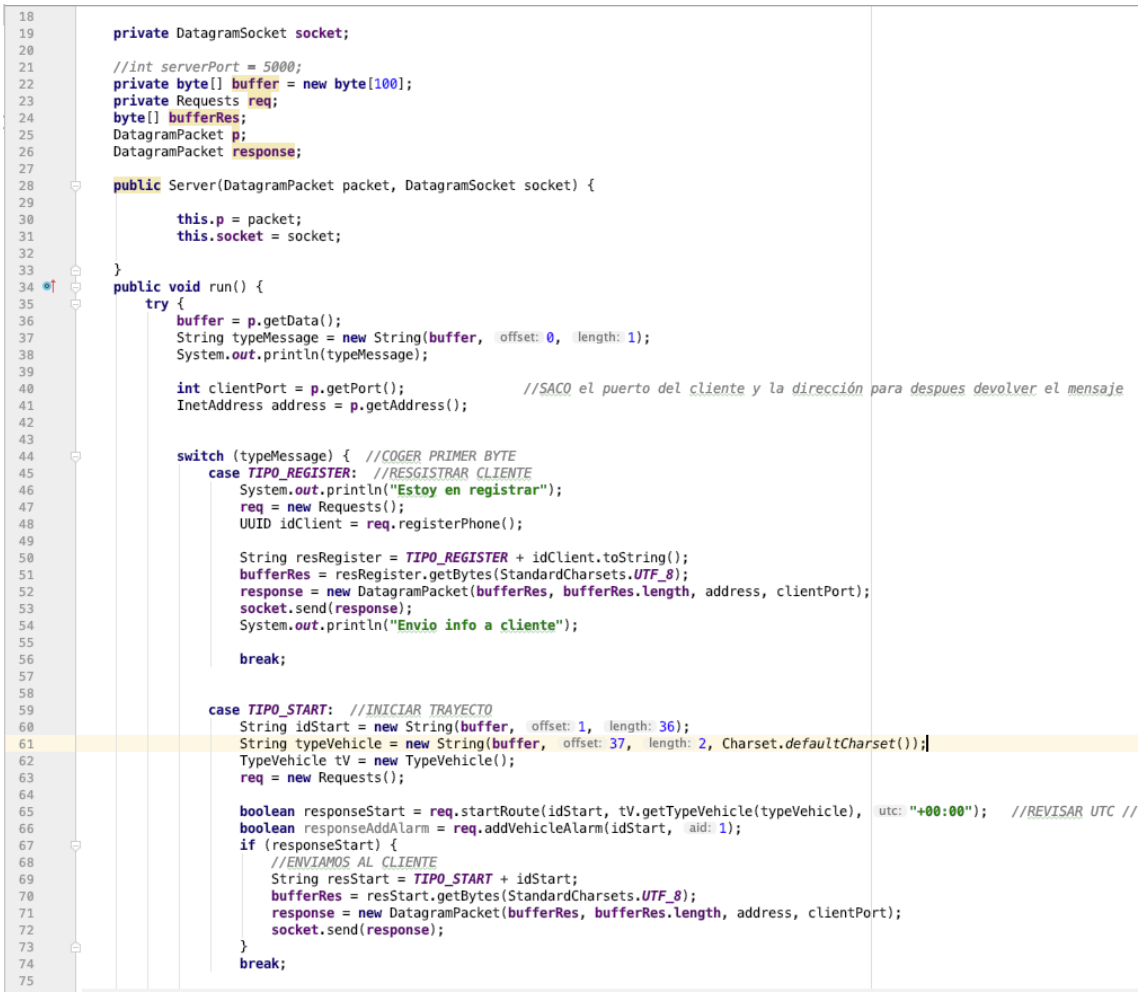
Figura 4.1.6.1. Servidor UDP

4.1.7 Servidor centralizado

El servidor centralizado es donde nos lleva la clase `UdpServerStarter`. En este servidor recibimos el mensaje datagrama por parte del cliente y según qué tipo de mensaje, realizamos unas peticiones HTTP u otras. Este servidor lo exponemos en el fichero `Server.java`, que contiene la clase `Server` que extiende a la clase `Thread`. Este fichero es la interfaz de traducción de mensajes UDP a peticiones HTTP, la interfaz de interconexión entre el cliente y el servidor web. La clase `Server` únicamente tiene un método que es el de ejecución. (`run ()`).

[6] Thread: es un trozo de código que puede ser ejecutado al mismo tiempo que otro trozo de código. La finalidad de los hilos es realizar varias tareas simultáneamente en el mismo proceso del sistema operativo.

En primer lugar, tenemos que extraer el primer *byte* del mensaje datagrama. Este *byte* es el que va a definir el tipo del mensaje UDP. Utilizamos una estructura de control de `switch` para separar bloques de código según el tipo de mensaje extraído.



```
18
19      private DatagramSocket socket;
20
21      //int serverPort = 5000;
22      private byte[] buffer = new byte[100];
23      private Requests req;
24      byte[] bufferRes;
25      DatagramPacket p;
26      DatagramPacket response;
27
28      public Server(DatagramPacket packet, DatagramSocket socket) {
29
30              this.p = packet;
31              this.socket = socket;
32
33      }
34      public void run() {
35          try {
36              buffer = p.getData();
37              String typeMessage = new String(buffer,  offset: 0,  length: 1);
38              System.out.println(typeMessage);
39
40              int clientPort = p.getPort();              //SACO el puerto del cliente y la dirección para despues devolver el mensaje
41              InetAddress address = p.getAddress();
42
43
44              switch (typeMessage) {  //COGER PRIMER BYTE
45                  case TIPO_REGISTER:  //RESGISTRAR CLIENTE
46                      System.out.println("Estoy en registrar");
47                      req = new Requests();
48                      UUID idClient = req.registerPhone();
49
50                      String resRegister = TIPO_REGISTER + idClient.toString();
51                      bufferRes = resRegister.getBytes(StandardCharsets.UTF_8);
52                      response = new DatagramPacket(bufferRes, bufferRes.length, address, clientPort);
53                      socket.send(response);
54                      System.out.println("Envio info a cliente");
55
56                      break;
57
58
59                  case TIPO_START:  //INICIAR TRAYECTO
60                      String idStart = new String(buffer,  offset: 1,  length: 36);
61                      String typeVehicle = new String(buffer,  offset: 37,  length: 2, Charset.defaultCharset());
62                      TypeVehicle tV = new TypeVehicle();
63                      req = new Requests();
64
65                      boolean responseStart = req.startRoute(idStart, tV.getTypeVehicle(typeVehicle),  utc: "+00:00");   //REVISAR UTC //
66                      boolean responseAddAlarm = req.addVehicleAlarm(idStart,  aid: 1);
67                      if (responseStart) {
68                          //ENVIAMOS AL CLIENTE
69                          String resStart = TIPO_START + idStart;
70                          bufferRes = resStart.getBytes(StandardCharsets.UTF_8);
71                          response = new DatagramPacket(bufferRes, bufferRes.length, address, clientPort);
72                          socket.send(response);
73                      }
74                      break;
75
```

Figura 4.1.7.1 Constructor de la clase `Server` y los casos de la estructura `switch`: TIPO_REGISTER, TIPO_START

En cada caso de la estructura `switch` inicializamos un nuevo objeto `Request`, para poder realizar las peticiones al servidor web. Extraemos los campos necesarios del mensaje UDP para pasarle dichos campos como parámetros a los métodos que realizan las peticiones HTTP. Tras realizar las peticiones HTTP, recogemos las respuestas por parte del servidor web y las convertimos de nuevo a mensaje datagrama para enviarlas al cliente.

El paso de datos del servidor centralizado al cliente se realiza de la misma manera que de cliente a servidor. Primero, creamos el mensaje UDP de servidor a cliente correspondiente (definido en los requisitos funcionales). Convertimos el mensaje a un *array* de *bytes* e inicializamos un mensaje datagrama con el *array* de *bytes*, el puerto del cliente y la IP como parámetros. Por último, enviamos el mensaje datagrama al cliente mediante el método `send()` del *socket*. Cabe mencionar que el puerto del cliente y la

IP los almacenamos en una variable cuando recibimos el mensaje datagrama del cliente, en las líneas 40 y 41 de la figura 4.1.7.1 podemos observar dicho almacenamiento.

En la figura 4.1.7.1 se muestra los casos de la estructura `switch` `TIPO_REGISTER` y `TIPO_START`. En la figura 4.1.7.2 podemos observar los casos de la estructura `switch` `TIPO_POSICION`, `TIPO_ALARMA`, `TIPO_STOP` y `TIPO_DELETE`.

```
75
76              case TIPO_POSICION:
77                  req = new Requests();
78                  String idUpdate = new String(buffer,  offset: 1,  length: 36);
79                  double lon = Double.valueOf(new String(buffer,  offset: 37,  length: 11));
80                  double lat = Double.valueOf(new String(buffer,  offset: 48,  length: 11));
81                  double alt = Double.valueOf(new String(buffer,  offset: 59,  length: 11));
82                  String timestamp = new String(buffer,  offset: 70,  length: 16);
83                  req.updatePosition(idUpdate, lon, lat, alt, timestamp);
84                  break;
85
86              case TIPO_ALARMA:
87                  //PETICION HTTP
88                  req = new Requests();
89                  String id = new String(buffer,  offset: 1,  length: 36, Charset.defaultCharset());
90                  System.out.println("ID:" + id);
91                  int idAlarm = Integer.parseInt(new String(buffer,  offset: 37,  length: 1));
92                  System.out.println("Id Alarm:" + idAlarm);
93                  String t = new String(buffer,  offset: 38,  length: 16);
94                  System.out.println("Time:" + t);
95                  boolean resStatus = req.updateVehicleAlarm(id,idAlarm,t);
96                  boolean res = req.getVehicleAlarm(id, idAlarm, t);   //SOLICITUD WEB  idclient, idAlarm, timestamp
97                  System.out.println("Resultado:" + res);
98
99                  //ENVIAMOS AL CLIENTE
100                 String resGetAlarm = TIPO_ALARMA + id + idAlarm + (res == true ? "SI" : "NO") + t;   //SI LA RESPUESTA ES TRUE, AL
101                 bufferRes = resGetAlarm.getBytes();
102                 response = new DatagramPacket(bufferRes, bufferRes.length, address, clientPort);
103                 socket.send(response);
104                 break;
105
106             case TIPO_STOP:   //PARAR TRAYECTO
107                 String idStop = new String(buffer,  offset: 1,  length: 36, Charset.defaultCharset());
108                 req = new Requests();
109                 boolean responseStop = req.stopRoute(idStop);
110                 System.out.println(responseStop);
111                 break;
112
113
114             case TIPO_DELETE:  //BORRAR CLIENT
115                 req = new Requests();
116                 String idDelete = new String(buffer,  offset: 1,  length: 36, Charset.defaultCharset());
117                 System.out.println("ID: " + idDelete);
118                 boolean responseDelete = req.deletePhone(idDelete);
119                 System.out.println(responseDelete);
120
121                 if (responseDelete) {
122                     //ENVIAMOS A CLIENTE
123                     String resDelete = TIPO_DELETE + idDelete;
124                     bufferRes = resDelete.getBytes();
125                     response = new DatagramPacket(bufferRes, bufferRes.length, address, clientPort);
126                     socket.send(response);
127                     break;
128                 }
129
130
131         }
132
```

Figura 4.1.7.2 Casos de la estructura `swicth`: `TIPO_POSICION`, `TIPO_ALARMA`, `TIPO_STOP` y `TIPO_DELETE` de la clase `Server`

4.1.8 Peticiones HTTP

En este apartado vamos a exponer las diferentes peticiones HTTP que realizamos al servidor web. Para ello, definimos una interfaz con los métodos que contendrán dichas peticiones HTTP, como podemos observar en la figura 4.1.8.1.

Las implementaciones de los métodos de la interfaz las realizamos en el fichero `Requests.java`. En este fichero implementamos las siguientes peticiones HTTP:

- Registro de teléfono móvil (POST)
- Inicio de trayecto (POST)
- Finalización de trayecto (DELETE)
- Dar de baja el teléfono móvil, eliminar id de cliente. (DELETE)
- Actualización de la posición (PUT)
- Inserción de alarma que tiene vehículo (POST)
- Pedir el estado de alarma de un vehículo (GET)
- Actualización de estado de alarma (PUT)

```
5  public interface IntRequest {
6      public UUID registerPhone();
7      public boolean startRoute(String id, String typeVehicle, String utc);
8      public boolean stopRoute(String id);
9      public boolean deletePhone(String id);
10     public boolean updatePosition(String id, double lon, double lat, double alt, String time);
11     public boolean addVehicleAlarm(String id, int aid);
12     public boolean getVehicleAlarm(String id, int idAlarm, String time);
13     public boolean updateVehicleAlarm(String id, int idAlarm, String time);
14 }
```

Figura 4.1.8.1 Interfaz de solicitudes HTTP

En todas las peticiones tendremos la misma base de URL (*Uniform Resource Locator*): http://127.0.0.1:5000, que es donde está inicializado el servidor web. Todas las peticiones tienen el mismo patrón de código. Para cada petición se va a utilizar una URL diferente, es decir, la URL base comentada anteriormente más la URL especifica para cada petición. Las URL de las peticiones son las siguientes:

- Registrar teléfono ➔ **base URL + server/phone/** (línea 31 figura 4.1.8.2.)
- Iniciar trayecto ➔ **base URL + server/phone/start_route/ + id** (línea 79 figura 4.1.8.3)
- Parar trayecto ➔ **base URL + server/phone/stop_route/ + id** (línea 119 figura 4.1.8.4)
- Borrar teléfono ➔ **base URL + server/phone/ + id** (línea 156 figura 4.1.8.5)
- Actualizar posición ➔ **base URL + server/phone/start_route/ + id + ?pos=true** (línea 194 figura 4.1.8.6)
- Añadir alarma de vehículo ➔ **base URL + server/vehicle/alarm?pid= +id+ &aid= + aid** (línea 235 figura 4.1.8.7)

- Pedir estado de alarma de un vehículo ➔ **base URL + server/vehicle/alarm?pid + id + &aid= idAlarm** (línea 273 figura 4.1.8.8)

- Actualizar estado de alarma de un vehículo ➔ **base URL + server/vehicle/alarm?pid + id + &aid= idAlarm** (línea 315 figura 4.1.8.9)

La estrategia para cada petición es la misma. Primero, se inicializa una conexión HTTP con la URL definida mediante un objeto `HttpURLConnection`, se deja abierta la conexión. A continuación, se modifica el método de solicitud dependiendo de la petición. Los métodos pueden ser los siguientes: POST, PUT, GET, DELETE. Además, también se modifica las propiedades de la solicitud (línea 34 figura 4.1.8.2). En caso de que la petición sea un método POST o PUT se debe escribir un objeto `JSONObject` sobre la conexión establecida. En el objeto JSON es donde le pasamos algunos datos para que el servidor web interactúe con ellos. Este caso lo podemos observar entre las líneas 37 y 40 de la figura 4.1.8.2.

Para recibir la respuesta del servidor web, necesitamos inicializar un objeto `BufferedReader`[7] para poder leer el resultado del servidor web. Leemos las líneas del objeto `BufferReader` y las insertamos en un objeto `StringBuilder`[8]. Seguidamente, transformamos el resultado del objeto `StringBuilder` a un objeto `JSONObject`, con la finalidad de poder leer los datos más cómodamente. Por último, devolvemos los datos requeridos por cada petición. Las devoluciones de las peticiones son las siguientes:

- Registro de teléfono móvil (POST) ➔ ID cliente

- Inicio de trayecto (POST) ➔ Boolean

- Finalización de trayecto (DELETE) ➔ Boolean

- Dar de baja el teléfono móvil, eliminar id de cliente. (DELETE) ➔ Boolean

- Actualización de la posición (PUT) ➔ Boolean

- Inserción de alarma que tiene vehículo (POST) ➔ Boolean

[7] BufferedReader: lee el texto de una secuencia de entrada de caracteres, almacenando caracteres en un buffer para proporcionar una lectura eficiente de caracteres.

[8] StringBuilder: similar a la clase `String`, almacena cadena de caracteres.

- Pedir el estado de alarma de un vehículo (GET) ➔ Boolean
- Actualización de estado de alarma (PUT) ➔ Boolean

```java
public class Requests implements IntRequest {

    private String baseUrl = "http://127.0.0.1:5000/";

    @Override
    public UUID registerPhone() {
        StringBuilder resultado = new StringBuilder();
        try {
            JSONObject  jsonBody = new JSONObject();   ///????PREGUNTAR
            jsonBody.put( name: "type",  value: "car");
            URL url = new URL( spec: baseUrl+"server/phone/");
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod("POST");
            connection.setRequestProperty("Content-Type", "application/json");
            connection.setDoOutput(true);

            DataOutputStream wr = new DataOutputStream(
                    connection.getOutputStream());
            wr.writeBytes(jsonBody.toString());
            wr.close();

            BufferedReader rd = new BufferedReader(new InputStreamReader(connection.getInputStream()));
            String linea;
            while ((linea = rd.readLine()) != null) {
                resultado.append(linea);
            }
            rd.close();

            JSONObject res = new JSONObject(resultado.toString());
            System.out.println("Pid: "+ res.getJSONObject("data").getString( name: "pid"));
            String id = res.getJSONObject("data").getString( name: "pid");
            return UUID.fromString(id);

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (JSONException e) {
            e.printStackTrace();
        }
        return null;

```

Figura 4.1.8.2 Petición de registro de teléfono móvil

```java
public boolean startRoute(String id, String typeVehicle, String utc) {
    StringBuilder resultado = new StringBuilder();
    try {

        JSONObject jsonBody = new JSONObject();
        jsonBody.put( name: "utc", utc);
        jsonBody.put( name: "type",typeVehicle);

        System.out.println(jsonBody.toString());
        URL url = new URL( spec: baseUrl+"server/phone/start_route/"+id);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("POST");
        connection.setRequestProperty("Content-Type", "application/json");
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Language", "en-US");

        //ENVIAR request
        DataOutputStream wr = new DataOutputStream(
                connection.getOutputStream());
        wr.writeBytes(jsonBody.toString());
        wr.close();

        BufferedReader rd = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String linea;
        while ((linea = rd.readLine()) != null) {
            resultado.append(linea);
        }
        rd.close();

        JSONObject res = new JSONObject(resultado.toString());
        System.out.println("success: "+ res.getBoolean( name: "success"));
        return  res.getBoolean( name: "success");

    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (JSONException e) {
        e.printStackTrace();
    }
    return false;
}
```

Figura 4.1.8.3. Petición de iniciar trayecto

```java
@Override
public boolean stopRoute(String id) {
    StringBuilder resultado = new StringBuilder();

    try {

        URL url = new URL( spec: baseUrl+"server/phone/stop_route/"+id);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("DELETE");
        connection.setRequestProperty("Content-Type", "application/json");
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Language", "en-US");

        BufferedReader rd = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String linea;
        while ((linea = rd.readLine()) != null) {
            resultado.append(linea);
        }
        rd.close();

        JSONObject res = new JSONObject(resultado.toString());
        System.out.println("Resultado: "+res);
        return res.getBoolean( name: "success");

    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (JSONException e) {
        e.printStackTrace();
    }

    return false;
}
```

Figura 4.1.8.4. Petición de parar el trayecto

```java
@Override
public boolean deletePhone(String id) {

    StringBuilder resultado = new StringBuilder();

    try {

        URL url = new URL( spec: baseUrl+"server/phone/"+id);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("DELETE");
        connection.setRequestProperty("Content-Type", "application/json");
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Language", "en-US");

        BufferedReader rd = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String linea;
        while ((linea = rd.readLine()) != null) {
            resultado.append(linea);
        }
        rd.close();

        JSONObject res = new JSONObject(resultado.toString());
        System.out.println("Resultado: "+res);
        return res.getBoolean( name: "success");

    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (JSONException e) {
        e.printStackTrace();
    }

    return false;
}
```

Figura 4.1.8.5 Petición HTTP de darse de baja

```java
@Override
public boolean updatePosition(String id, double lon, double lat, double alt, String timestamp) {
    StringBuilder resultado = new StringBuilder();
    try {
        JSONObject  jsonBody = new JSONObject();
        jsonBody.put( name: "lon",  lon);
        jsonBody.put( name: "lat", lat);
        jsonBody.put( name: "alt", alt);
        jsonBody.put( name: "timestamp", timestamp);
        URL url = new URL( spec: baseUrl+"server/phone/"+id+"?pos=true");
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("PUT");
        connection.setRequestProperty("Content-Type", "application/json");
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Language", "en-US");

        //ENVIAR JSON
        DataOutputStream wr = new DataOutputStream(
                connection.getOutputStream());
        wr.writeBytes(jsonBody.toString());
        wr.close();

        BufferedReader rd = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String linea;
        while ((linea = rd.readLine()) != null) {
            resultado.append(linea);
        }
        rd.close();

        JSONObject res = new JSONObject(resultado.toString());
        System.out.println("Resultado Position: "+res);
        return res.getBoolean( name: "success");

    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (JSONException e) {
        e.printStackTrace();
    }

    return false;
}
```

Figura 4.1.8.6 Petición HTTP de actualizar posición

```java
@Override
public boolean addVehicleAlarm(String id, int aid) {

    StringBuilder resultado = new StringBuilder();
    try {

        URL url = new URL( spec: baseUrl+"server/vehicle/alarm?pid="+id+"&aid="+aid);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("POST");
        connection.setRequestProperty("Content-Type", "application/json");
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Language", "en-US");

        BufferedReader rd = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String linea;
        while ((linea = rd.readLine()) != null) {
            resultado.append(linea);
        }
        rd.close();

        JSONObject res = new JSONObject(resultado.toString());
        System.out.println("Resultado: "+ res);
        System.out.println("success: "+ res.getBoolean( name: "success"));
        return res.getBoolean( name: "success");


    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (JSONException e) {
        e.printStackTrace();
    }
    return false;
}
```

Figura 4.1.8.7 Petición HTTP de añadir alarma de vehículo

```java
@Override
public boolean getVehicleAlarm(String id, int idAlarm, String timestamp){  //QUE DEVUELVE


    StringBuilder resultado = new StringBuilder();
    try {

        URL url = new URL( spec: baseUrl + "server/vehicle/alarm?pid="+id+"&aid="+idAlarm);
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("GET");
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Type",
                "application/json");


        BufferedReader rd = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String linea;
        while ((linea = rd.readLine()) != null) {
            resultado.append(linea);
        }
        rd.close();

        System.out.println("RESULTADO: " + resultado.toString());
        JSONObject res = new JSONObject(resultado.toString());
        boolean status;
        if(res.getJSONObject("data").getBoolean( name: "status")){
            status = true;
        }
        else{
            status = false;
        }
        System.out.println("Estado: "+ status);
        return status;
    } catch (IOException e) {
        e.printStackTrace();

    } catch (JSONException e) {
        e.printStackTrace();
    }
    return false;
}
```

Figura 4.1.8.8 Petición HTTP de pedir estado de alarma

```
307
308     public boolean updateVehicleAlarm(String id, int idAlarm, String t) {
309         StringBuilder resultado = new StringBuilder();
310         try {
311             JSONObject  jsonBody = new JSONObject();
312             jsonBody.put( name: "status",   value: true);
313             jsonBody.put( name: "timestamp", t);
314
315             URL url = new URL( spec: baseUrl + "server/vehicle/alarm?pid="+id+"&aid="+idAlarm);
316             HttpURLConnection connection = (HttpURLConnection) url.openConnection();
317             connection.setRequestMethod("PUT");
318             connection.setDoOutput(true);
319             connection.setRequestProperty("Content-Type",
320                     "application/json");
321
322             //ENVIAR JSON
323             DataOutputStream wr = new DataOutputStream(
324                     connection.getOutputStream());
325             wr.writeBytes(jsonBody.toString());
326             wr.close();
327
328             BufferedReader rd = new BufferedReader(new InputStreamReader(connection.getInputStream()));
329             String linea;
330             while ((linea = rd.readLine()) != null) {
331                 resultado.append(linea);
332             }
333             rd.close();
334
335             JSONObject res = new JSONObject(resultado.toString());
336             if(res.getBoolean( name: "success")){
337                 return true;
338             }
339             else{
340                 return false;
341             }
342         } catch (IOException e) {
343             e.printStackTrace();
344
345         } catch (JSONException e) {
346             e.printStackTrace();
347         }
348         return false;
349     }
```

Figura 4.1.8.9 Petición HTTP de actualizar estado de alarma

4.2. Verificación y validación

En este apartado vamos a exponer las pruebas de verificación y validación que hemos realizado para comprobar que el funcionamiento del programa es el adecuado.

En referencia a la verificación, el mismo entorno de desarrollo Android Studio nos proporciona un emulador que simula un teléfono móvil. Toda la parte de la interfaz de usuario podíamos comprobar si funcionaba correctamente por el mencionado emulador. En caso de que no se mostrara la acción, botón o resultado en la interfaz de usuario, pasamos a comprobar el código línea a línea e imprimiendo por pantalla los datos necesarios para encontrar el error.

Por otra parte, para la realización de las pruebas de peticiones HTTP utilizamos la herramienta Postman (6), esta herramienta nos permite crear bibliotecas de *endpoints* para ejecutarlos y probar nuestros servicios web. Enviamos objetos JSON a la URL indicada para comprobar que el servidor web recibía los datos enviados y nos devolvía las respuestas correctas.

Las pruebas de validación han sido ir comprobando que todas las pantallas funcionaban correctamente en el emulador y que todos los requisitos definidos en el capítulo 3 han sido implementados correctamente. Tras la comprobación en el emulador, pasamos al teléfono móvil. Instalamos la aplicación en nuestro teléfono móvil y comprobamos que todas las pantallas, los botones y las peticiones HTTP al servidor web funcionaban correctamente.

Capítulo 5

Conclusiones

En este apartado vamos a presentar conclusiones en varios aspectos: ámbito formativo, ámbito profesional y ámbito personal. Además, también comentamos el trabajo futuro y las mejoras que se pueden realizar al proyecto.

En referencia al **ámbito formativo**, el desarrollo de este proyecto nos ha servido para adquirir los conocimientos básicos acerca de la tecnología Android Studio. A su vez, hemos asentado conocimientos del lenguaje Java. Respecto a las herramientas que hemos utilizado, hemos aprendido a utilizar nuevas herramientas como Postman o Pencil. Por último, hemos puesto en práctica la metodología en cascada para la planificación y seguimiento del proyecto.

En el **ámbito profesional** no podemos exponer la experiencia profesional en la empresa, ya que este proyecto es un trabajo académico dirigido por el tutor Pablo Boronat. Aun así, hemos podido aprender las pautas y las fases para desarrollar un proyecto, gracias a la metodología utilizada.

Respecto al **ámbito personal**, este proyecto fue la primera aplicación móvil que implementé. Al ser la primera, puede que haya errores o formas distintas de programar el desarrollo de la aplicación. Gracias a este proyecto me ha despertado un gran interés acerca de este mundo. Como consecuencia, me gustaría seguir aprendiendo sobre las aplicaciones móviles, cabe la posibilidad que me inscriba en un máster de desarrollo de aplicaciones móviles en Android y iOS en un futuro cercano.

Por último, las **mejoras** que se pueden hacer al proyecto en un futuro son las siguientes:

- Administración de alarmas (crear, borrar, actualizar)
- Autenticación del usuario
- Implementación de altitud en los parámetros de localización

Bibliografía

1. Cuatroochenta. [Citado el: 25 de julio de 2020.] https://cuatroochenta.com/.

2. UDP. [Citado el: 25 de julio de 2020.] https://es.wikipedia.org/wiki/Protocolo_de_datagramas_de_usuario#:~:text=User%20Datagram%20Protocol%20(UDP)%20es,y%20la%20capa%20de%20aplicaci%C3%B3n..

3. HTTP. [Citado el: 25 de julio de 2020.] https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto#:~:text=El%20Protocolo%20de%20transferencia%20de,en%20la%20World%20Wide%20Web..

4. Android Studio. [Citado el: 25 de julio de 2020.] https://developer.android.com/studio.

5. Git. [Citado el: 25 de julio de 2020.] https://git-scm.com/.

6. Postman. [Citado el: 25 de julio de 2020.] https://www.postman.com/.

7. Java. [Citado el: 25 de julio de 2020.] https://www.java.com/es/.

8. Pencil. [Citado el: 25 de julio de 2020.] https://pencil.evolus.vn/.

9. Desarrollo en cascada. [Citado el: 25 de julio de 2020.] https://es.wikipedia.org/wiki/Desarrollo_en_cascada.

10. Harvard. [Citado el: 25 de julio de 2020.] https://cs50.harvard.edu/college/2019/fall/tracks/mobile/android/.

11. discoduroderoer. [Citado el: 25 de julio de 2020.] https://www.discoduroderoer.es/ejemplo-conexion-udp-clienteservidor-en-java/.

12. UDP-Github. [Citado el: 25 de julio de 2020.] https://github.com/sauravpradhan/UDP-Packet-Sender-and-Receiver.

13. Hilos. [Citado el: 25 de julio de 2020.] https://javadesdecero.es/avanzado/hilos-multihilos-ejemplos/#:~:text=Hilos%20en%20Java%20con%20Ejemplos,-Avanzado%20Dejar%20un&text=Multithreading%20es%20una%20caracter%C3%ADstica%20de,livianos%20dentro%20de%20un%20proceso..

14. stackoverflow. [Citado el: 25 de julio de 2020.] https://stackoverflow.com/.

15. Indeed. [Citado el: 24 de julio de 2020.] https://es.indeed.com/salaries/programador-junior-Salaries,-Comunitat-Valenciana.

16. ISO 8601. [Citado el: 25 de julio de 2020.] https://es.wikipedia.org/wiki/ISO_8601.

17. REST. [Citado el: 25 de julio de 2020.] https://openwebinars.net/blog/que-es-rest-conoce-su-potencia/.

18. JSON. [Citado el: 25 de julio de 2020.] https://www.json.org/json-es.html.

19. Capas. [Citado el: 25 de julio de 2020.] https://instintobinario.com/arquitectura-en-tres-capas/.