# Centro Universitario de la Defensa en la Escuela Naval Militar

**TRABAJO FIN DE GRADO**

*Diseño y desarrollo de una aplicación en Android para geolocalización y comunicaciones en despliegues de Infantería de Marina utilizando una red local*

**Grado en Ingeniería Mecánica**

**ALUMNO:** Luis Rández Aguillo

**DIRECTORES:** Belén Barragáns Martínez

Pablo Sendín Raña

**CURSO ACADÉMICO:** 2019-2020

Universidade Vigo

Centro Universitario de la Defensa
en la Escuela Naval Militar

**TRABAJO FIN DE GRADO**

# *Diseño y desarrollo de una aplicación en Android para geolocalización y comunicaciones en despliegues de Infantería de Marina utilizando una red local*

**Grado en Ingeniería Mecánica**

Intensificación en Tecnología Naval

Infantería de Marina

Universidade Vigo

# RESUMEN

La información para la toma de decisiones en el despliegue de tropas es fundamental a todos los niveles. La posibilidad de contar con información multimedia en tiempo real sobre lo que está pasando en el campo de batalla es un activo de un valor incalculable y para ello es necesario una herramienta que lo permita.

Con esta motivación, se plantea en este TFG el diseño y desarrollo de una aplicación para dispositivos móviles que aborde estas necesidades. Esta aplicación permitirá la transmisión y recepción de información multimedia como puede ser audio y vídeo (*streaming*), un servicio de mensajería (*chat*) así como la geolocalización en tiempo real de todos los participantes, siendo todas ellas funcionalidades críticas para su empleo en las misiones actuales. Se implementará para dispositivos con sistema operativo Android y se desarrollará empleando el lenguaje de programación Kotlin.

Esta aplicación será de especial interés para unidades de Infantería de Marina debido a las misiones que realiza, permitiéndoles mantener ese flujo de información en tiempo real que es fundamental para el correcto desarrollo de las operaciones.

# PALABRAS CLAVE

Aplicación móvil, Android, *Streaming*, *Kotlin*, geolocalización

# AGRADECIMIENTOS

Comienzo agradeciendo a mis dos tutores, Belén Barragáns Martínez y Pablo Sendín Raña por su excelente implicación en la dirección de este trabajo. También a mis compañeros de camareta Alejandro, Raúl y Miguel por su incansable apoyo y motivación. Por último, pero no menos importante agradecer a mi amigo Vali por sus aportaciones.

# CONTENIDO

Contenido ....3

Índice de Figuras ....5

1 Introducción y objetivos ....7

1.1 Contexto ....7

1.2 Motivación de la necesidad ....7

1.3 Objetivo del TFG ....8

1.4 Organización de la memoria ....8

2 Estado del arte ....9

2.1 Necesidad en el ámbito militar de aplicaciones similares ....9

2.2 Aplicaciones similares existentes ....9

2.2.1 Ámbito militar ....9

2.2.2 Ámbito civil ....11

2.3 Selección de tecnologías (*hardware y software*) a emplear ....11

2.3.1 Desarrollo de una aplicación móvil ....11

2.3.2 Motivación de desarrollo en Android ....12

2.3.3 Arquitectura Android ....12

2.3.4 Empleo del IDE de desarrollo y lenguaje de programación ....14

2.3.5 Recursos hardware ....18

2.4 Revisión de protocolos diseñados para comunicaciones multimedia ....19

3 Desarrollo del TFG ....21

3.1 Requisitos de la aplicación ....21

3.2 Diseño de la aplicación ....21

3.3 Codificación de la aplicación ....24

3.3.1 Estructura de directorios de un proyecto Android ....24

3.3.2 Etapas de desarrollo ....31

4 Resultado del TFG: aplicación ATC ....39

4.1 Funcionamiento de la aplicación ....39

4.2 Pruebas de la aplicación ....42

5 Conclusiones y líneas futuras ....43

5.1 Conclusiones ....43

5.2 Líneas futuras ....43

6 Bibliografía ....45

Anexo I: Código de la aplicación .....49

# ÍNDICE DE FIGURAS

Figura 2-1 Operador de los US Marines utilizando el sistema ATAK antes de un salto paracaidista [7] .....10
Figura 2-2 CivTak [6] .....11
Figura 2-3 Estadística de uso de sistemas operativos en móviles en diciembre de 2019 [8].....12
Figura 2-4 Capas de la arquitectura Android [10] .....14
Figura 2-5 Vista *Android* .....16
Figura 2-6 Vista *Project*.....16
Figura 2-7 Ciclo de vida de una actividad [14].....17
Figura 2-8 Xiaomi Mi A2 Lite proporcionado por el CUD.....19
Figura 2-9 Router D-Link DIR-615 [18] .....19
Figura 2-10 Esquema simplificado del funcionamiento de *streaming* en el receptor [20] .....20
Figura 3-1 Pantallas iniciales de la aplicación .....22
Figura 3-2 Pantallas de los dos tipos de comunicaciones .....22
Figura 3-3 Diagrama de flujo que detalla el funcionamiento de la aplicación .....23
Figura 3-4 Niveles API [23].....24
Figura 3-5 *AndroidManifest.xml* de la aplicación desarrollada .....25
Figura 3-6 Web para la obtención de claves API de *Google*.....26
Figura 3-7 Web para la gestión de claves API de *Google* .....26
Figura 3-8 *Layout* tipo texto de *MainActivity* .....28
Figura 3-9 *Layout* tipo diseño de *MainActivity*.....28
Figura 3-10 Fragmento de código de la clase *Comunicaciones* (I).....29
Figura 3-11 Fragmento de código de la clase *Comunicaciones* (II) .....30
Figura 3-12 Código de la clase *CustomInfoWindow*.....30
Figura 3-13 *Clase Participante*.....31
Figura 3-14 Código de *MainActivity* en la que se registra el usuario .....32
Figura 3-15 Pantalla de registro del usuario .....32
Figura 3-16 Código para la obtención de latitud y longitud .....33
Figura 3-17 El círculo azul muestra la ubicación del usuario de la aplicación.....33
Figura 3-18 Diagrama del flujo de paquetes *broadcast* para la actualización de las ubicaciones de los diferentes usuarios.....34
Figura 3-19 Obtención dirección IP del usuario .....34
Figura 3-20 Código para la construcción del mensaje *broadcast*.....35
Figura 3-21 Código para la ejecución del hilo de escucha de mensajes .....35

Figura 3-22 Código para el *parseo* del mensaje *broadcast* .....35
Figura 3-23 Código para la gestión de participantes de la aplicación .....36
Figura 3-24 Función para la discriminación de mensajes *broadcast* y chat .....36
Figura 3-25 Código de la función que ejecuta la actividad *Chat* .....37
Figura 3-26 Código de la función que ejecuta el hilo de escucha de mensajes de chat .....37
Figura 3-27 Función que ejecuta la actividad *Stream* .....38
Figura 3-28 Función que ejecuta la actividad *Stream* .....38
Figura 4-1 Pantalla de registro del usuario .....39
Figura 4-2 Pantalla de mapa en la aplicación .....40
Figura 4-3 Pantalla de mapa con usuario detectado .....40
Figura 4-4 Pantalla de chat entre dos usuarios .....41
Figura 4-5 Imagen de una prueba durante el *streaming* en un pasillo .....41
Figura 4-6 Localización de dos participantes .....42
Figura 5-1 Formato MEDEVAC [28] .....44
Figura 5-2 Formato SPOTREPORT [29] .....44

# 1 INTRODUCCIÓN Y OBJETIVOS

En este primer capítulo se procederá a explicar la contextualización de este TFG, la motivación que ha impulsado su planteamiento, los objetivos que se persiguen en el desarrollo del mismo, así como la estructuración de esta memoria.

## 1.1 Contexto

En el ámbito actual de los conflictos mundiales hay un pilar fundamental, sobre el cual recae un peso importante: las comunicaciones. Este sector ha evolucionado en estas últimas décadas. En el pasado, las comunicaciones eran típicamente de audio, por ejemplo, las llamadas telefónicas o enlaces radio, pero en la actualidad lo usual es poder comunicarnos con vídeo en tiempo real, con las ventajas que ello supone. Esto ha tomado además un peso aún más determinante dentro de las Fuerzas Armadas, no solamente siendo capaces de enviar un flujo de datos, sino consiguiendo la transmisión de información en tiempo real (vídeo y audio) y pudiendo geolocalizar a nuestras unidades sobre el terreno, obteniendo así un conocimiento del entorno mucho más completo.

El empleo de este tipo de comunicaciones para despliegue de tropas ha supuesto un avance muy relevante en los ejércitos, ya que no solamente permite la toma de decisiones a muy alto nivel, también las facilita porque permite ver cómo se está desarrollando el campo de batalla o simplemente cómo evoluciona una operación determinada en tiempo real.

El *streaming* (transmisión de audio y vídeo en tiempo real) aplicado al ámbito militar tiene unos requisitos técnicos que han sido abordados también por aplicaciones en el ámbito civil. Estos requisitos podemos resumirlos en tres principales: la baja latencia, la relación entre calidad y ancho de banda y, por último, el STANAG 4609 [1], que entró en vigor en el año 2009 y describe el formato de intercambio de contenido multimedia para los países que forman parte de la OTAN.

## 1.2 Motivación de la necesidad

La información para la toma de decisiones en el despliegue de tropas es fundamental a todos los niveles. La posibilidad de contar con información multimedia en tiempo real sobre lo que está pasando en el campo de batalla es un activo de un valor incalculable y para ello es necesario una herramienta que lo permita.

Un punto decisivo en la toma de decisiones es el centro de Mando y Control (*C2, Command and Control*) sobre nuestras unidades. Dotarlo de un sistema que permita obtener información multimedia de las unidades desplegadas mejoraría el C2. Precisamente, las mejoras de las comunicaciones en entornos tácticos constituyen hoy en día una de las metas tecnológicas de la Estrategia de Tecnología e

Innovación del Ministerio de Defensa [2]. Por ello se propone desarrollar una aplicación que permita geolocalizar y realizar *streaming* para despliegues en las unidades de Infantería de Marina.

Este nuevo sistema permitiría a las unidades conectadas una comunicación en tiempo real de audio, vídeo y datos, así como una geolocalización continúa representada en un mapa en la pantalla de cada dispositivo.

Además, se pretende que este sistema se ejecute en terminales móviles con sistema operativo Android (teléfonos móviles o tabletas) que cuenten con una cámara como dispositivo de entrada y GPS para transmitir su posición. Estos dispositivos estarán conectados entre sí mediante una red que, en nuestro caso, será una red de área local inalámbrica. Esta aplicación sería totalmente distribuida, se quiere evitar tener un nodo central responsable del sistema, ya que una desconexión o un ataque a este nodo central implicaría la caída de todo el sistema.

## 1.3 Objetivo del TFG

El objetivo de este Trabajo Fin de Grado es el diseño y desarrollo de una aplicación que permita *streaming* en tiempo real de audio y vídeo, y comunicaciones de datos, entre dispositivos Android que están en todo momento geolocalizados entre ellos dentro de una red local. Cada uno de los usuarios de esta aplicación contará con una interfaz que le permitirá geolocalizar al resto de participantes y, a partir de ese instante, iniciar comunicaciones multimedia o de datos.

## 1.4 Organización de la memoria

Este documento se divide en 6 capítulos. En el presente capítulo se ha presentado una contextualización del trabajo, se ha motivado la necesidad y se han presentado los objetivos de este Trabajo Fin de Grado. En el capítulo 2 se realizará una descripción de aplicaciones similares en el ámbito militar, así como la justificación del *software*, del IDE y del lenguaje empleado para desarrollar y diseñar la aplicación. En el capítulo 3 se expone el desarrollo del proyecto, detallando los requisitos previos al desarrollo de la aplicación, también se definen las diferentes etapas de diseño y codificación. En el capítulo 4, correspondiente a la validación y pruebas, se explica como sería la experiencia del usuario cuando emplea la aplicación desarrollada en este proyecto, así cómo las diferentes pruebas realizadas y sus limitaciones. A continuación, en el capítulo 5 se exponen una serie de conclusiones a las que se ha llegado, y se plantean posibles líneas futuras para continuar con el desarrollo de la aplicación. En el capítulo 6 se detalla la bibliografía empleada para la redacción de este documento. Por último, se añade un anexo al final de esta memoria con el código completo de la aplicación.

# 2 ESTADO DEL ARTE

En este segundo capítulo se amplía el contexto, y se estudian aplicaciones similares a la que se pretende crear en este proyecto. También enumeraremos los diferentes recursos de *software* y *hardware* que se han empleado para llevar a cabo el trabajo y la justificación de dicha elección. Se finaliza con la revisión de protocolos diseñados para comunicaciones multimedia.

## 2.1 Necesidad en el ámbito militar de aplicaciones similares

El uso de teléfonos móviles para emitir vídeo en directo en zonas de operaciones ha sido de gran interés para el *Department of Defense* de EEUU, al menos desde el año 2011 [3], cuando comenzó la integración de tecnologías 4G para conseguir una mayor eficiencia en sus sistemas.

En septiembre de 2014, una División de la *US Army Communications Electronics Research and Engineering Center* elevó una petición de información a varias empresas de telefonía móvil [3]. La intención de esta petición fue adaptar la tecnología inalámbrica en los teléfonos que se comercializaban en el momento para proporcionar a los soldados una moderna comunicación segura en el campo de batalla. Los americanos vieron esto como una oportunidad de desarrollar la inteligencia, vigilancia y reconocimiento en sus operaciones. La solución fue el empleo de redes *mesh* [4], aumentando considerablemente su zona de cobertura en el terreno.

Aún con topologías de red que mejoran y amplían los radios de cobertura, se siguen manifestando problemas como la calidad del vídeo que, en ocasiones, se debe sacrificar en beneficio de mejorar la latencia. El empleo de una calidad baja puede tener un impacto determinante en el éxito de la misión.

Otro problema que va de la mano cuando se emplean tecnologías en el ámbito táctico y estratégico es la seguridad: decidir qué operativos o analistas están autorizados a acceder al material sensible y cómo proteger las comunicaciones con protocolos de cifrado.

## 2.2 Aplicaciones similares existentes

### *2.2.1 Ámbito militar*

La aplicación de referencia en este caso es ATAK (*Android Tactical Assault Kit*) [5], una aplicación para *smartphones* desarrollada para dispositivos Android por *US Air Force Research*

*Laboratory* (AFRL). En ocasiones se pueden leer referencias de ATAK como *Android Team Awareness Kit* porque hay diferentes versiones tanto para Windows (WinTAK), como iOS (iTAK) y Android.

Previamente distribuyeron al público una versión civil de ATAK, llamada CivTak [6], a la que se podía acceder de manera gratuita, pero actualmente el enlace de descarga está desactivado.

La distribución de ATAK está limitada a los empleados del gobierno americano y compañías que cumplen los requisitos para obtener la licencia. ATAK ha sido empleado por las fuerzas de los Estados Unidos en Iraq, Afganistán y Siria [7]. Los empleos tácticos que nos proporciona esta aplicación son: planeamiento de misiones, conocimiento del medio en el que operamos y ayudas para apoyo de fuegos.

Las funcionalidades de ATAK pueden resumirse como sigue:

- Puede usarse en diferentes redes (redes móviles, WiFi, y redes tácticas tipo *mesh*).
- Nos permite compartir audio, vídeo, texto, chatear, interactuar y compartir nuestra posición en el terreno.
- Su interfaz pretende dar a conocer al operador un conocimiento preciso sobre lo que está pasando a su alrededor.
- ATAK puede utilizar todos los sensores que integra el *smartphone*: altímetro, barómetro, brújula y GPS.

Las capacidades que tiene ATAK solamente en su versión militar incluyen:

- Herramientas de ayuda al salto en paracaídas, como se aprecia en la Figura 2-1.
- *Targeting*: estudio del método para batir un blanco determinado.
- Geolocalización más precisa que en su modalidad civil.
- Control de frecuencias militares.

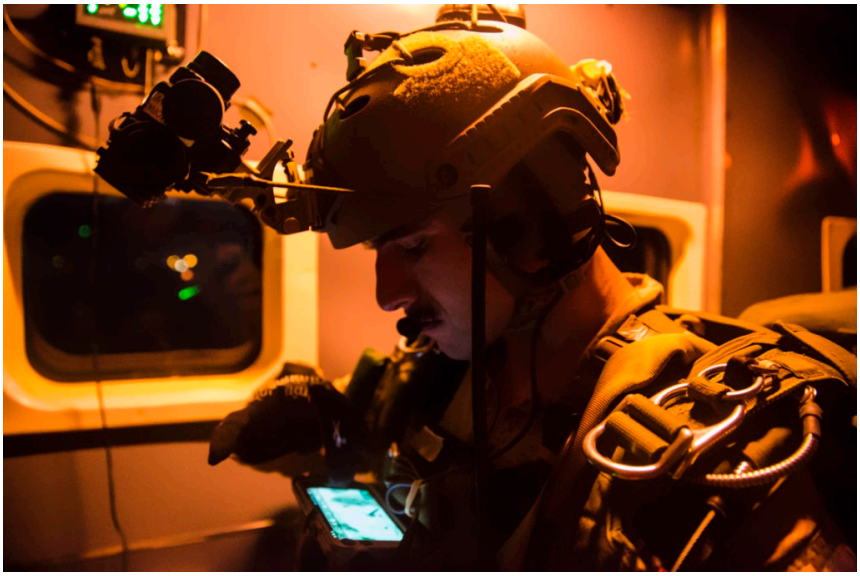


**Figura 2-1 Operador de los US Marines utilizando el sistema ATAK antes de un salto paracaidista [7]**

### *2.2.2 Ámbito civil*

Enlazando con el apartado anterior tenemos una aplicación del mismo desarrollador, CivTak [6] (véase Figura 2-2). Esta aplicación nos proporciona capacidades muy parecidas a ATAK, pero eliminando las que son de carácter militar. Por lo tanto, incluye:

- Mapas tanto online como offline.
- Buscador web.
- Geolocalización.
- Chat.
- Compartir archivos de audio, vídeo, etc.

También proporciona servicios de *streaming*, navegación y control de redes móviles, WiFi y frecuencias radio civiles. Esta aplicación ya no está disponible para su descarga.

**Figura 2-2 CivTak [6]**

## 2.3 Selección de tecnologías (*hardware y software*) a emplear

### *2.3.1 Desarrollo de una aplicación móvil*

Se ha decidido el desarrollo de esta aplicación para dispositivos móviles debido a la funcionalidad que nos ofrecen.

En primer lugar, el participante que se encuentre en zona de operaciones va a necesitar apenas un bolsillo o un adaptador al chaleco para poder portar un dispositivo móvil.

En segundo lugar, los dispositivos móviles de los que disponemos hoy en día nos permiten obtener vídeo y audio con calidad suficiente para cumplir con éxito el objetivo de esta aplicación.

Por último, pero no menos importante, que el operador esté familiarizado con dispositivos de este tipo también es favorable a la hora de un correcto aprendizaje del funcionamiento de la aplicación.

### *2.3.2 Motivación de desarrollo en Android*

Se ha decidido desarrollar este TFG en plataforma Android debido al porcentaje de usuarios que lo utilizan con respecto a iOS u otros sistemas operativos empleados en teléfonos móviles. Concretamente, el porcentaje de teléfonos móviles a nivel mundial que emplean el sistema operativo Android asciende al 74%, según datos de diciembre de 2019 [8] (véase Figura 2-3).

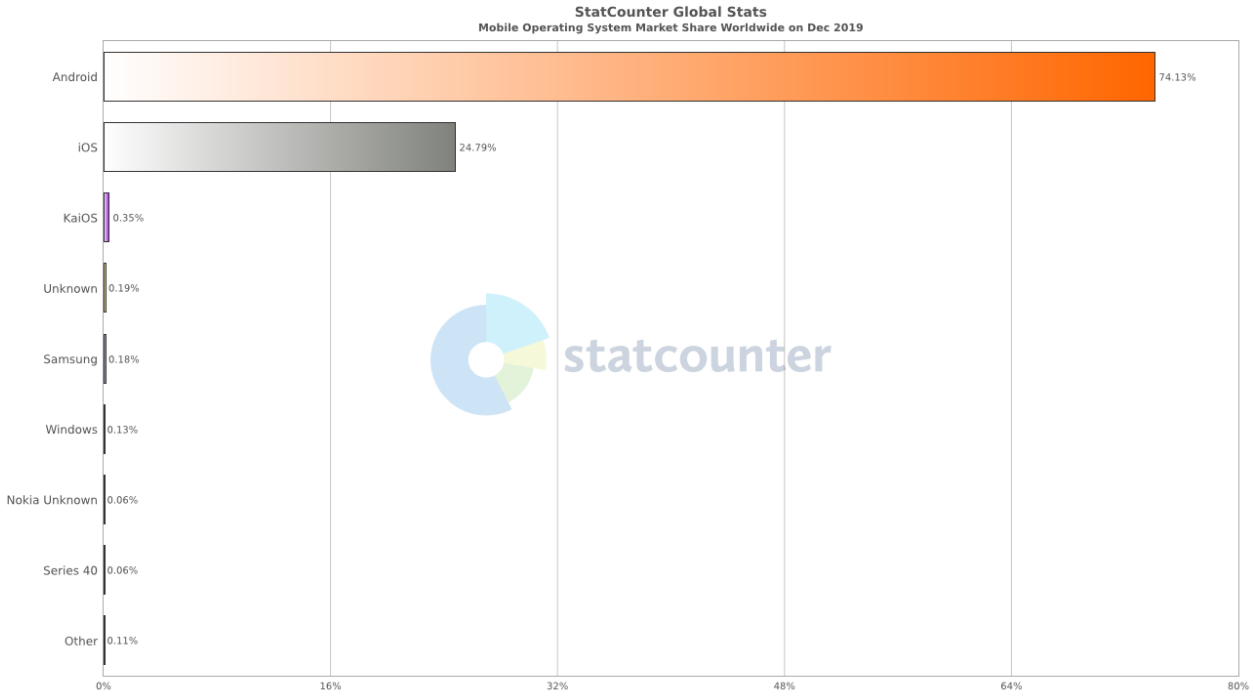



**Figura 2-3 Estadística de uso de sistemas operativos en móviles en diciembre de 2019 [8]**

Android es un sistema de código abierto: esto se ve reflejado en una mayor comunidad de desarrolladores para esta plataforma, lo que se traduce en dos grandes beneficios. El primero es la competencia en el mercado, ya que, a mayor competencia, mayor es la calidad, requiriéndose un mayor esfuerzo a la hora de desarrollar un producto. El segundo beneficio es la facilidad de encontrar a alguien que pueda ayudar con un proyecto desarrollado en este sistema operativo.

La facilidad de distribución de una aplicación en Android también es una ventaja. En Android disponemos de *Google Play* [9] para la descarga de aplicaciones, pero, en el caso de que queramos generar una aplicación privada, y no distribuirla de forma pública, podemos generar un archivo APK (*Android Package*) y distribuirlo a los usuarios que nos interesen.

### *2.3.3 Arquitectura Android*

La arquitectura de Android está compuesta por diferentes capas (ver Figura 2-4), que permiten garantizar que las aplicaciones sean independientes del *hardware*. Se describen a continuación dichas capas, de la inferior a la superior:

- *Linux kernel*: es la base de la plataforma Android. Su función es que tanto el *software* como el *hardware* trabajen de forma correcta y además interactúen entre sí.

- *Hardware Abstraction Layer (HAL)*: la capa de abstracción de *hardware* es la que permite a las capas más altas el acceso a la información del *hardware*.

- *Android Runtime*: el entorno de ejecución de Android es la máquina virtual sobre la que se ejecutan las aplicaciones.

- *Native C/C++ Libraries*: bibliotecas nativas de C/C++, son generadas por los desarrolladores de Android, destinadas a servicios centrales como lo son el *Android Runtime* y el *HAL* basándose en código nativo.

- *Java API Framework*: constituyen el conjunto de funciones del SO Android que están disponibles mediante API (*Application Program Interface*) escritas en *Java*, los principales elementos que lo forman se pueden observar en la Figura 2-4.

- *System Apps*: las aplicaciones del sistema, como pueden ser el calendario, la cámara, mensajería o el navegador, pueden ser empleadas por un desarrollador en su aplicación sin necesidad de compilar esa funcionalidad.

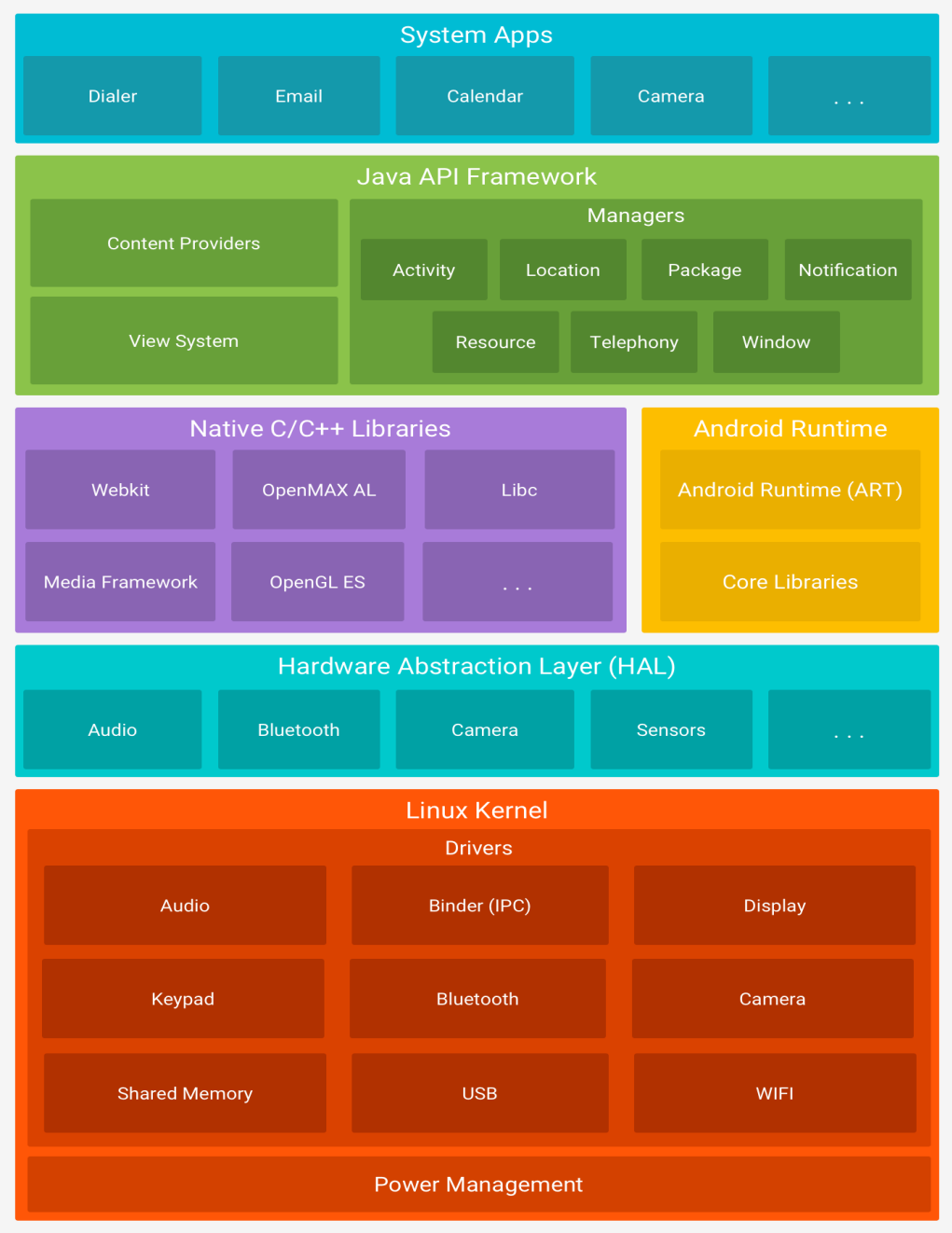



**Figura 2-4 Capas de la arquitectura Android [10]**

### *2.3.4 Empleo del IDE de desarrollo y lenguaje de programación*

Dentro de este apartado entraremos en detalle a presentar el IDE (*Integrated Development Environment*) de desarrollo empleado para realizar la aplicación objetivo de este TFG, así como mostrar las ventajas que tiene respecto del resto. Presentaremos también cómo se estructura un proyecto empleando este IDE. El segundo punto que se desarrolla en este apartado es la presentación del lenguaje de programación empleado, las ventajas respecto de otros lenguajes y por qué se ha tomado la decisión de usarlo.

#### 2.3.4.1 IDE de desarrollo

Android Studio [11] es un IDE de programación basado en IntelliJ [12], para Java y Kotlin, creado por JetBrains en 2013, que reemplazó a Eclipse como el IDE oficial para el desarrollo de aplicaciones para Android.

Sus creadores partieron de la versión *Community* de IntelliJ y añadieron las herramientas y utilidades necesarias para el desarrollo en todo tipo de dispositivos Android: móviles, tabletas, relojes inteligentes, etc.

Las ventajas principales para el empleo de Android Studio son: la incorporación de un editor de código inteligente, una rápida compilación, y la ejecución de la aplicación en un emulador o directamente en el móvil.

Además, gracias a su uso tan extendido, disponemos de un gran abanico de guías, y de *plugins* para la incorporación de nuevas bibliotecas, lo que facilita su aprendizaje.

En cuanto a la estructura de directorios en un proyecto Android, tenemos todos los ficheros correspondientes al proyecto de una manera ordenada. El IDE Android Studio permite al usuario diferentes tipos de visualización: las más destacables son la vista *Android* y la vista *Project*.

En la vista *Android*, se muestran los diferentes ficheros como se ve en la Figura 2-5. En dicha imagen se observa cómo se organizan los módulos y archivos para una navegación más sencilla para el usuario, pero no representa ninguna jerarquía dentro de los archivos. Cada módulo se estructura de la siguiente manera:

- *Manifests:* contiene el archivo *AndroidManifest.xml* [13], donde se almacena información esencial de la aplicación como pueden ser las diferentes actividades que se emplean, la versión de la aplicación y los diferentes permisos que el desarrollador de la aplicación considera necesarios para acceder. Entre todos los datos recogidos en este archivo, los que cabe destacar son:
    - *package*: es el nombre del paquete de nuestra aplicación.
    - *uses-permission*: para habilitar los diferentes permisos a los que debe acceder cada aplicación necesitamos declararlos en este archivo. Ejemplos de estos permisos pueden ser la ubicación, Internet, acceso al estado de la red, cámara, etc.
    - *application*: dentro de esta etiqueta se añaden todas las actividades que tiene cada aplicación. Se le añade el atributo *LAUNCHER* a la actividad que se lanzará primero cuando se ejecute la aplicación.
- *Java:* contiene los diferentes archivos de código fuente escritos en Kotlin.
- *Res:* contiene los recursos de la aplicación, en su mayor parte son ficheros XML. Este módulo se divide en cuatro directorios: *drawable*, *layout*, *mipmap* y *values*, de los cuales cabe destacar *layout* ya que es en el que se almacenan los ficheros que tienen la función de describir el aspecto visual de las diferentes actividades que interactúan con el usuario.

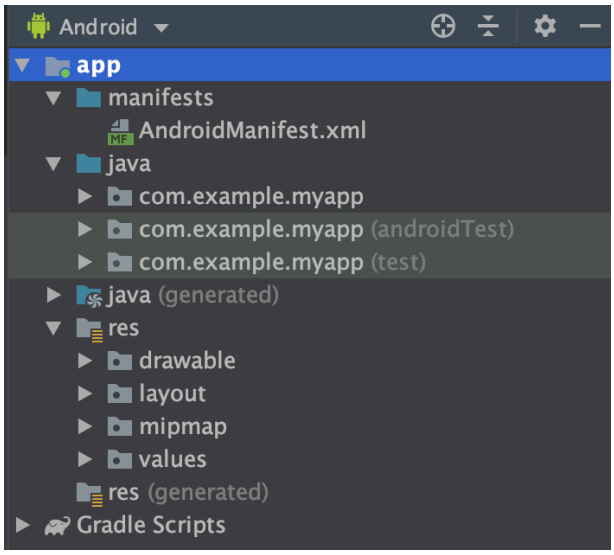



**Figura 2-5 Vista *Android***

En cuanto a la vista *Project* (ver Figura 2-6), se muestran más archivos y directorios que en la vista anterior, entre los que cabe destacar:

- *build*: contiene archivos generados por la compilación. En el directorio *src* se almacenan los recursos descritos en la vista anterior: *AndroidManifest.xml*, *Java* y *res*.
- *build.gradle*: en este fichero se definen las configuraciones de compilación principales, como pueden ser:
    - *compileSdkVersion*: define la versión API empleada para compilar la aplicación.
    - *minSdkVersion*: nivel de API mínimo necesario para poder ejecutar la aplicación.
    - *targetSdkVersion*: nivel de API para el que ha sido diseñada la aplicación.

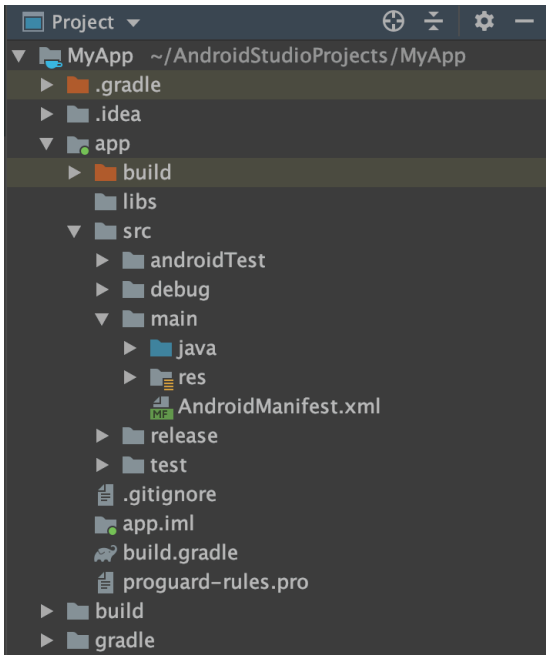



**Figura 2-6 Vista *Project***

Dentro de cada proyecto de Android, los ficheros que empleamos para la interacción con el usuario son básicamente las actividades o *activities* y los archivos XML del *layout*.

Las actividades o *activities* son los componentes de una aplicación que tienen asociada una interfaz gráfica. La mayoría de las aplicaciones, incluida la que se desarrolla en este TFG, contienen varias pantallas o interfaces gráficas, es decir, que incluyen varias actividades. Cuando se crea un nuevo proyecto, se genera por defecto una actividad principal (*MainActivity*), que implementa los métodos y atributos asociados a la primera pantalla que aparece cuando el usuario inicia la aplicación. Luego, cada actividad puede iniciar otra a fin de realizar diferentes acciones: de este modo, hay cierta dependencia entre las actividades de una aplicación.

Existen métodos predefinidos que gestionan el ciclo de vida de una actividad [14], como vemos de forma más detallada en la Figura 2-7.

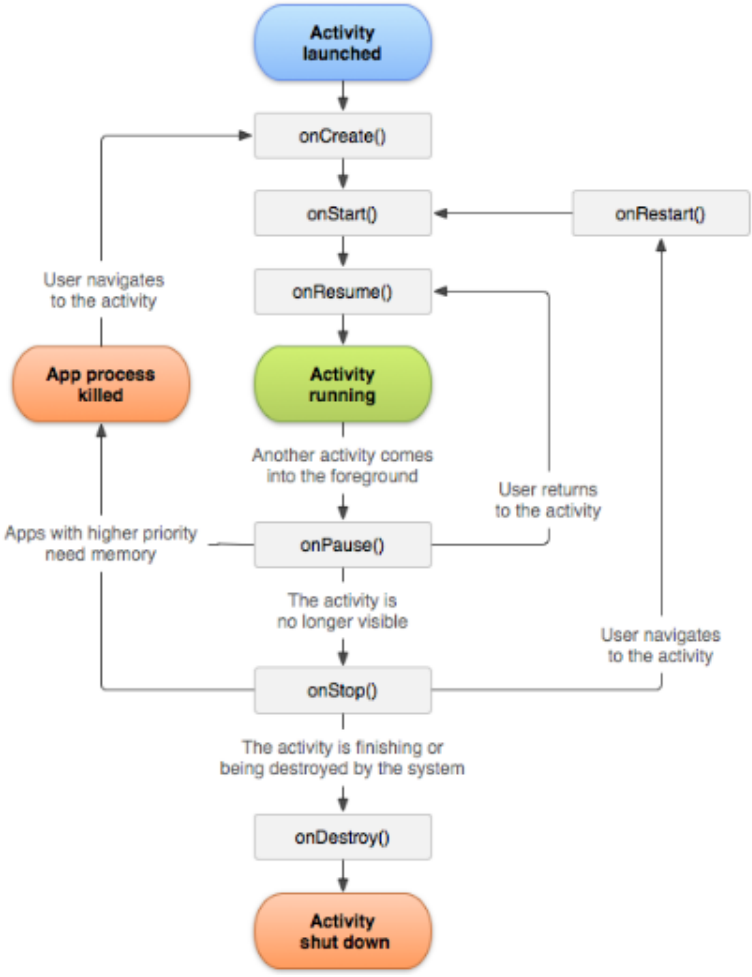



**Figura 2-7 Ciclo de vida de una actividad [14]**

En el momento que se ejecuta una actividad, ésta inicia su ciclo de vida y, por defecto, se ejecutan los siguientes métodos:

- *onCreate*: método llamado cuando se construye la actividad, es el primero en ejecutarse.
- *onStart*: se ejecuta justo después del anterior. En el caso que la aplicación estuviese en segundo plano, se ejecutaría de nuevo cuando se vuelve a encontrar en primer plano.
- *onResume*: es el último método que es llamado antes de mostrar la actividad en pantalla.
- *onPause*: se llama cuando hay otra actividad y además nuestra actividad desaparece de pantalla. Es crítico en caso de querer guardar información antes de que se cierre la actividad.

- *onStop*: una vez que la aplicación ya no se está ejecutando y el usuario no interactúa con ella se llama a este método.
- *onDestroy*: se llama cuando la actividad se elimina por completo de la memoria de la máquina virtual.

El diseño de cada pantalla y los elementos gráficos (botones, imágenes, mapas…) presentes en ellas están definidos por los *layouts*. Cada actividad tiene asociado un *layout*, implementado en un archivo XML.

#### 2.3.4.2 Lenguaje de programación

Kotlin [15] fue creado por JetBrains, lo que hace que el empleo combinado de este lenguaje con el IDE Android Studio nos ofrezca una sinergia perfecta.

Kotlin es un lenguaje de programación orientado a objetos relativamente moderno (nace en 2011), multiplataforma e interoperable con Java. Es un lenguaje con una sintaxis concisa y concreta, muy valorado entre los programadores y que *Google* ha convertido en su preferido para los desarrollos en Android [16]. Encontramos ciertas ventajas sobre el resto de lenguajes de programación como pueden ser Java o Swift, dos de los lenguajes de programación más conocidos y relevantes en el mundo del desarrollo de aplicaciones para móviles. Estas ventajas son:

- La codificación en Kotlin es más concisa y directa que en Java, es decir, mismo trabajo en menos líneas de código.
- Está diseñado para interoperar con código Java.
- Está respaldado por *Google* y *JetBrains*.

### *2.3.5 Recursos hardware*

Los diferentes recursos *hardware* empleados para el desarrollo de este proyecto son:

- Teléfono móvil Xiaomi Mi A2
- Router WiFi D-Link DIR-615

El teléfono móvil empleado para ejecutar la aplicación y realizar las pruebas es un Xiaomi Mi A2 Lite (ver Figura 2-8). Tiene las siguientes especificaciones técnicas [17]:

- Procesador Qualcomm Snapdragon 625 Octa-Core de 2 GHz.
- Procesador de gráficos Adreno 506.
- RAM de 3GB + 32GB ROM (Ampliable hasta 256GB).
- Cámara dual de 12 + 5 MP.
- Cámara frontal de 5 MP.
- Graba a 1080p/720p a 30fps.
- Batería de 4000 mAh.
- Pantalla de 5,84" FHD (1080x2280).
- Bluetooth 4.2.
- WiFi 2,4G y 5G.
- WiFi Direct y WiFi Display.

**Figura 2-8 Xiaomi Mi A2 Lite proporcionado por el CUD**

El router de banda ancha D-Link DIR-615 (véase Figura 2-9) es el modelo utilizado para desplegar la red inalámbrica empleada para conectar los diferentes dispositivos móviles. Tiene las siguientes especificaciones técnicas [18]:

- Velocidad inalámbrica de hasta 300 Mbps.
- 4 puertos LAN.
- 1 puerto WAN.
- 2,4 GHz.
- Filtrado de direcciones MAC.

**Figura 2-9 Router D-Link DIR-615 [18]**

## 2.4 Revisión de protocolos diseñados para comunicaciones multimedia

Los protocolos de *streaming* nos permiten la visualización inmediata del contenido sin tener que esperar a descargarlo completamente. En 1997 *RealNetwork* fue el primero en emitir un vídeo en

*streaming* [19]. El funcionamiento del *streaming* se representa mediante un esquema simplificado en la Figura 2-10.

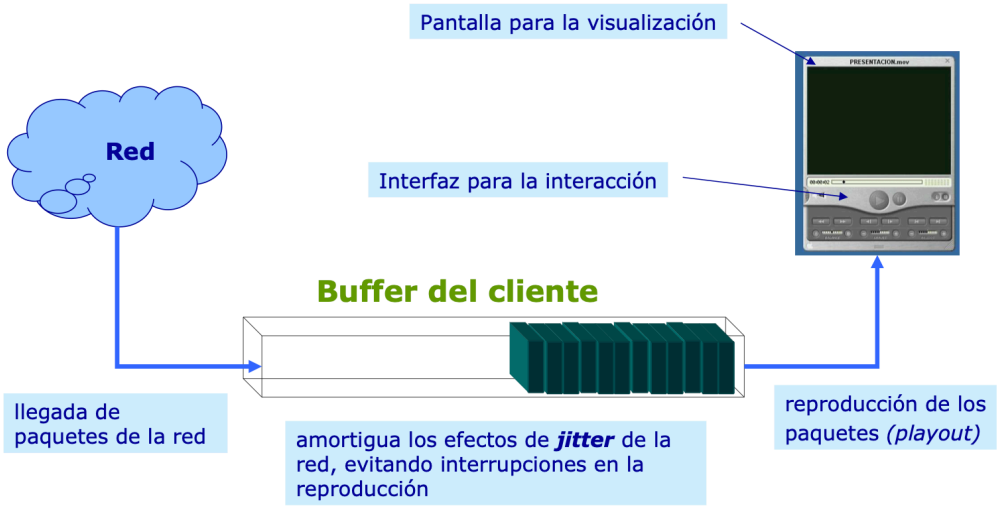



**Figura 2-10 Esquema simplificado del funcionamiento de *streaming* en el receptor [20]**

Entre los diferentes protocolos empleados que podemos encontrar para realizar *streaming*, caben destacar:

- RTP (*Real-Time Transport Protocol*): protocolo de transporte multimedia diseñado para manejar tráfico en tiempo real en Internet. Se implementa sobre UDP, ya que posee menor retardo que TCP. Por tanto, con UDP se gana velocidad a cambio de sacrificar la fiabilidad de TCP. Es por ello que RTP no garantiza la entrega de todos los paquetes, ni la llegada de éstos en el instante adecuado [21].
- RTCP (*Real-Time Transport Control Protocol*): implica la transmisión periódica de paquetes de control a todos los participantes de una sesión. La función principal es proporcionar mecanismos de realimentación para informar sobre la calidad en la distribución de los datos. Esta información se puede utilizar para diagnosticar fallos en la distribución y también para poder construir codificadores adaptables [21].
- RTSP (*Real-Time Streaming Protocol*): es un protocolo a nivel de aplicación de presentación multimedia cliente/servidor, que permite controlar el flujo de audio y vídeo sobre la red IP [20].

# 3 DESARROLLO DEL TFG

En este capítulo se describirá el proceso completo de diseño y desarrollo de la aplicación ATC (Aplicación Táctica del Combatiente), destinando un primer apartado a definir los requisitos que se espera que cumpla la aplicación. Durante toda esta fase de desarrollo y pruebas intermedias (extensible también a la fase de validación que se abordará en el siguiente capítulo) se dispondrá de tres dispositivos móviles y un router WiFi para poder realizar las diferentes pruebas.

## 3.1 Requisitos de la aplicación

Se desea diseñar y desarrollar una aplicación que permita a varios usuarios poder localizarse en tiempo real sobre un mapa, comunicarse a través de un chat y también tener la opción de consumir el vídeo que un usuario esté emitiendo. Se considera que todos los participantes están dentro de la misma red local inalámbrica (en nuestro caso particular, una red WiFi) y este TFG no tomará en consideración cuestiones relativas a la seguridad de las comunicaciones. Para ello, la aplicación debe contar con tres bloques diferenciados entre sí. Uno centrado en la geolocalización de los usuarios en el mapa, otro que permita a dos usuarios comunicarse con mensajes de texto y, por último, un tercero en el que cualquier usuario pueda ver a través de la aplicación el vídeo que está emitiendo otro usuario que esté conectado.

Para ello se requiere que la aplicación disponga de conexión a la red para la utilización de *Google* Maps [22] para permitir representar el mapa en la aplicación.

Además, todo esto se realizará empleando *software* libre.

## 3.2 Diseño de la aplicación

En cuanto al diseño de la aplicación, nos referimos específicamente a dos bloques principales: el diseño de la interfaz y las diferentes *activities* que queremos desarrollar. Podemos observar en los diagramas de las Figuras 3-1 y 3-2 cómo están relacionados.

Una de las primeras etapas a la hora de desarrollar una aplicación móvil consiste en pensar en el diseño de la interfaz, la cual debe ser sencilla a la vez que intuitiva, ya que es el medio a partir del cual el usuario va a interactuar con nuestra aplicación. Por ello se ha intentado incluir el menor número de

pasos posibles, tanto para el registro del usuario como para establecer conexión de chat o vídeo con los demás usuarios.

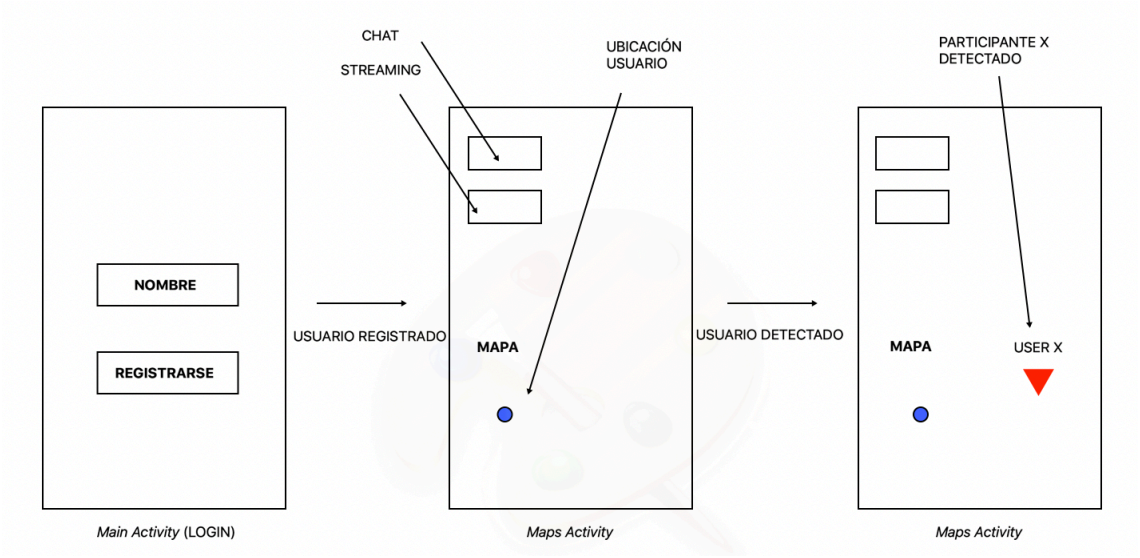



**Figura 3-1 Pantallas iniciales de la aplicación**

Una vez se ha detectado al menos un usuario, se ofrecen dos tipos de comunicaciones: la opción de chat y la opción de *streaming*, que se pueden observar en la Figura 3-2:

**Figura 3-2 Pantallas de los dos tipos de comunicaciones**

Tal y como se observa en las Figuras 3-1 y 3-2, el diseño de la interfaz de la aplicación considera cuatro pantallas:

- Pantalla de registro: en esta pantalla el usuario tiene un campo en el que tiene que introducir su nombre, y pulsando sobre el botón REGISTRARSE pasará a la pantalla de mapa.

- Pantalla de mapa: se mostrará un mapa híbrido (satélite y calles) con la posición del usuario. A medida que otros participantes se registren, irán apareciendo en el mapa mediante marcadores, indicando sus posiciones. Además, esta pantalla tendrá dos botones, uno para iniciar un chat con otro participante y otro para recibir el vídeo y el audio de otro participante. En ambos casos, debe estar previamente seleccionado el participante mediante su marcador.
- Pantalla de chat: en la parte superior de esta pantalla se muestra el nombre o identificador del participante con el que nos estamos comunicando. En la parte inferior, el usuario cuenta con un campo donde introducir el mensaje que desea enviar y un botón para enviarlo. En la parte central aparecerán los mensajes de la conversación (a la izquierda, los recibidos y a la derecha, los enviados).
- Pantalla de *streaming*: consumiremos el vídeo y audio del usuario al que hayamos seleccionado. Toda la pantalla la ocupa la visualización del flujo de vídeo recibido.

La Figura 3-3 presenta el diagrama de flujo que detalla cómo funciona la aplicación desde que se ejecuta hasta que finaliza.

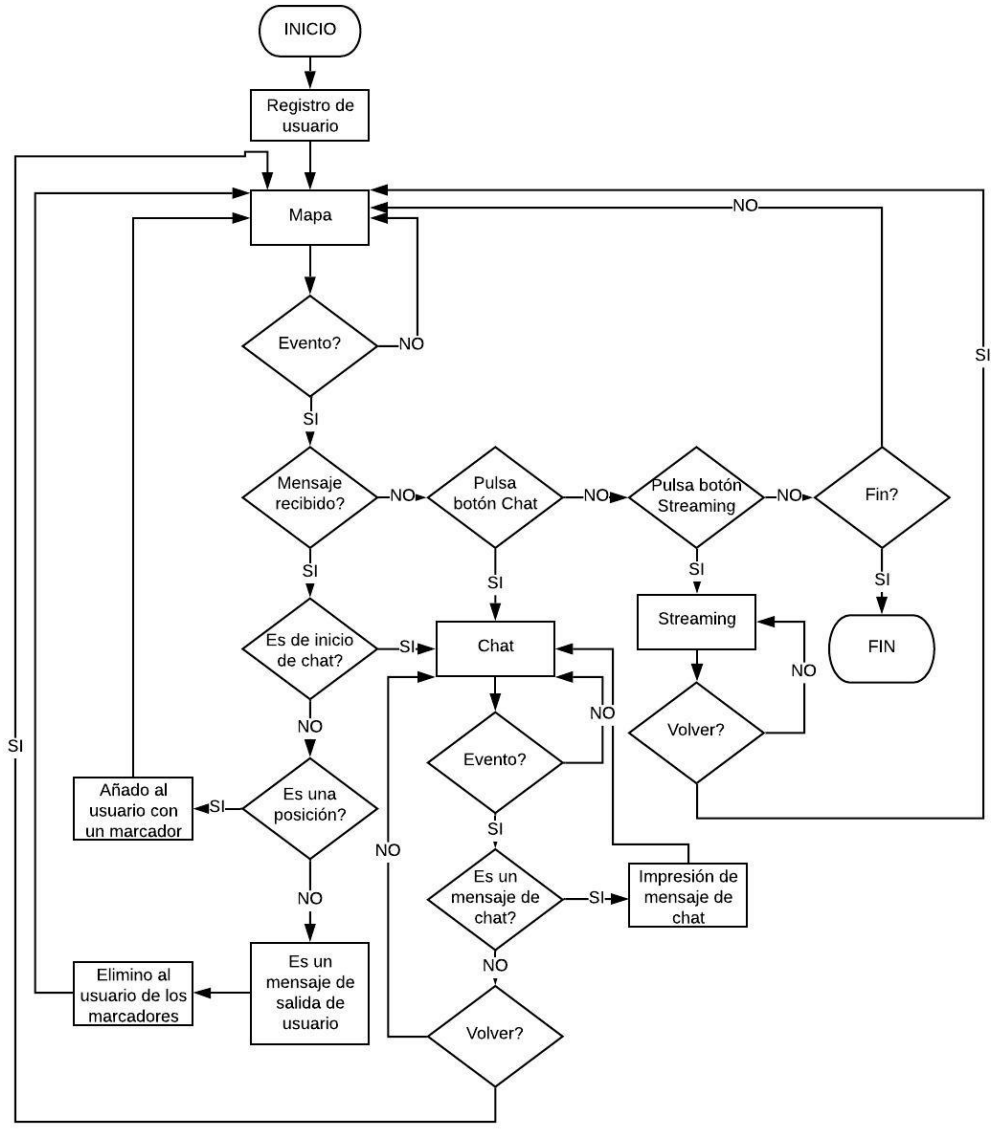



**Figura 3-3 Diagrama de flujo que detalla el funcionamiento de la aplicación**

## 3.3 Codificación de la aplicación

Antes de comenzar a codificar esta aplicación, es necesario elegir la versión del sistema operativo en la que se desea desarrollar el proyecto. Para ello necesitamos conocer los requerimientos de la aplicación y escoger un nivel de API (Figura 3-4) que nos permita optimizarlo para los dispositivos que vamos a emplear. Ya que esta aplicación no está pensada a priori para ser distribuida al público, no necesitamos una API que nos garantice que se pueda emplear en la gran mayoría de dispositivos Android. En este caso, se ha realizado el desarrollo para la versión Android 8.0 (Oreo), nivel de API 26.

| Versión | Nombre Comercial | Nivel de API |
|---|---|---|
| Android 1.0 | *Apple Pie* | 1 |
| Android 1.1 | *Banana Bread* | 2 |
| Android 1.5 | Cupcake | 3 |
| Android 1.6 | Donut | 4 |
| Android 2.0–2.1 | Eclair | 5-7 |
| Android 2.2–2.2.3 | Froyo | 8 |
| Android 2.3–2.3.7 | Gingerbread | 9–10 |
| Android 3.0–3.2.6 | Honeycomb | 11–13 |
| Android 4.0–4.0.5 | Ice Cream Sandwich | 14–15 |
| Android 4.1–4.3.1 | Jelly Bean | 16–18 |
| Android 4.4–4.4.4 | KitKat | 19–20 |
| Android 5.0–5.1.1 | Lollipop | 21–22 |
| Android 6.0–6.0.1 | Marshmallow | 23 |
| Android 7.0 - 7.1.2 | Nougat | 24-25 |
| Android 8.0 - 8.1 | Oreo | 26-27 |
| Android 9.0 | Pie | 28 |

**Figura 3-4 Niveles API [23]**

### *3.3.1 Estructura de directorios de un proyecto Android*

A continuación, se explican los diferentes módulos empleados en el desarrollo del TFG, mostrando ejemplos propios de la aplicación y de la forma en la que afectan al funcionamiento de la misma.

#### *3.3.1.1 AndroidManifest.xml*

Ya se explicó anteriormente que este archivo proporciona al sistema Android la información necesaria para que la aplicación pueda funcionar correctamente.

Entre todos los datos recogidos en el archivo de la Figura 3-5, caben destacar los siguientes:

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.ATC">

    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
    <uses-permission android:name="android.permission.RECORD_AUDIO" />
    <uses-permission android:name="android.permission.WAKE_LOCK"/>

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="ATC"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".Chat" >
        </activity>

        <meta-data
            android:name="com.google.android.geo.API_KEY"
            android:value="AIzaSyCwocD7v8VKG0Oyw0nHQ9ljM_EKiWK8mr8" />

        <activity
            android:name=".MapsActivity"
            android:label="ATC" />
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

**Figura 3-5 *AndroidManifest.xml* de la aplicación desarrollada**

- *package*: en este caso sería *ATC* (Aplicación Táctica del Combatiente).
- *uses-permission*: como se puede ver en la Figura 3-5, los permisos que se requieren son:
    - Ubicación: el IDE ofrece dos alternativas: *FINE* o *COURSE LOCATION*. En este caso se ha escogido *FINE* debido a que proporciona datos de ubicación más precisos que la otra alternativa, aunque el consumo de batería sea mayor.
    - Internet.
    - Acceso al estado de la red.
    - Cámara y grabadora.
    - Cambiar estado del WiFi: permite activar o desactivar el WiFi desde la aplicación.
    - *Wake lock*: sería necesario para que la pantalla de nuestro teléfono no se apague mientras se está ejecutando nuestra aplicación.
- *application*: en esta aplicación se le añade el atributo de *LAUNCHER* a la *MainActivity*, lo que quiere decir que será la actividad que se lanzará cuando se ejecute la aplicación.

- *API KEY*: como un caso particular de esta aplicación, debido a que empleamos la API (*Application Program Interface*) de *Google Maps* para poder hacer uso de sus mapas en nuestra *MapsActivity*, necesitamos de la clave de la API para poder habilitarlo. Esta clave se obtiene de forma gratuita desde el enlace que el propio IDE de desarrollo proporciona (ver Figura 3-6), donde se puede observar en la primera casilla *"Maps SDK for Android"*.

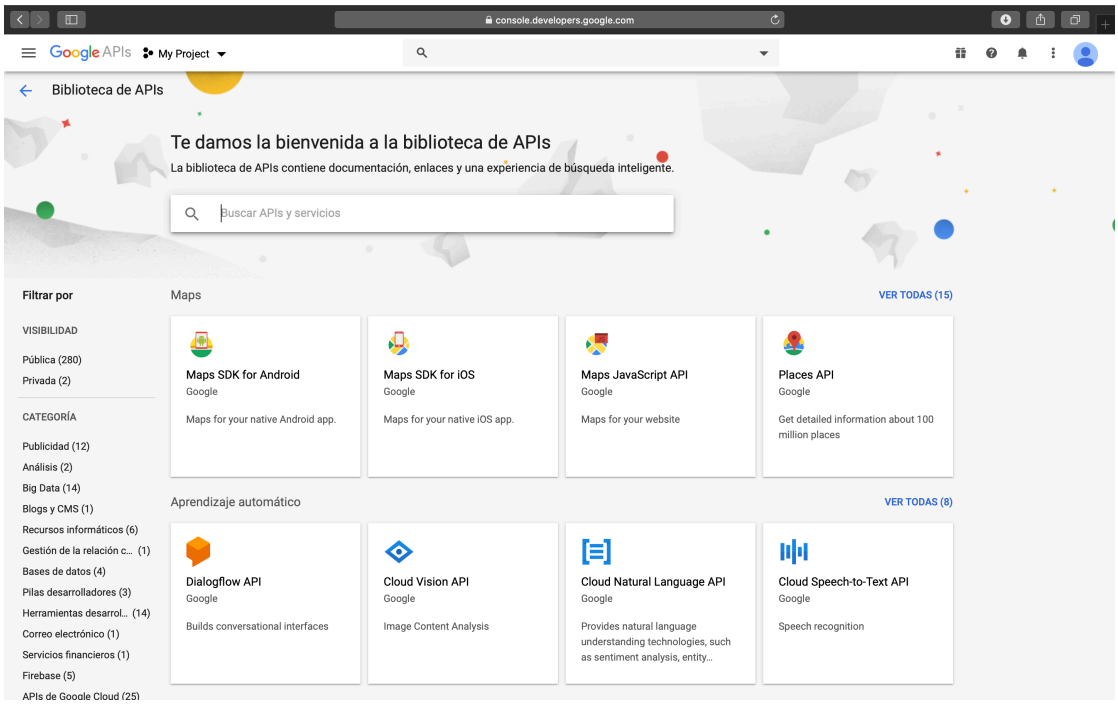



**Figura 3-6 Web para la obtención de claves API de *Google***

Una vez dentro, el usuario puede gestionar sus credenciales (ver Figura 3-7).

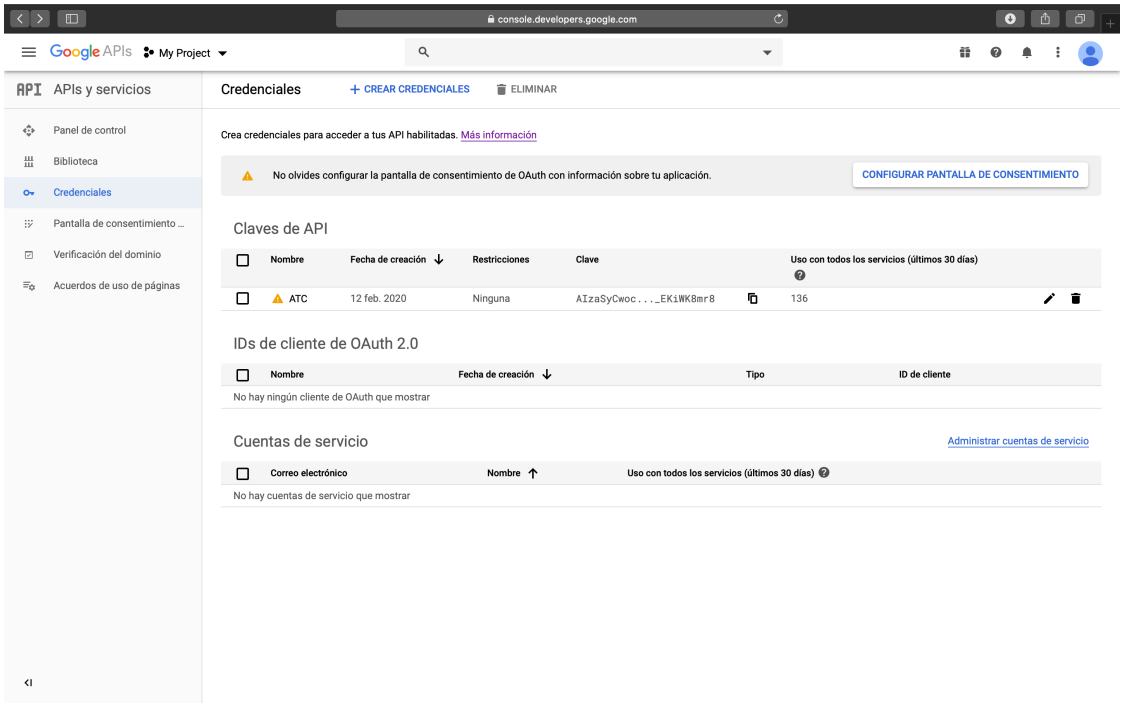



**Figura 3-7 Web para la gestión de claves API de *Google***

#### 3.3.1.2 Actividades

Como se ha visto dentro del apartado 2.3.4.1 del Estado del Arte, las actividades en una aplicación son aquellos componentes que muestran una pantalla al usuario, permitiendo que interactúe con ella.

Desde una actividad podemos iniciar otra actividad y enviar información entre ellas. De este modo, entre las distintas actividades de una aplicación se puede decir que hay una cierta dependencia.

Las actividades dentro la aplicación a desarrollar son las siguientes:

- *MainActivity*: en la que el usuario deberá registrarse con su nombre.
- *MapsActivity*: en esta actividad el usuario podrá ver representada su ubicación en un mapa. Cuando la aplicación detecte que un usuario se ha conectado desde otro dispositivo, lo representará en el mapa y se podrá ver su posición actual. Todas las ubicaciones deben ser actualizadas en tiempo real.
- *Chat*: cuando desde la actividad anterior se seleccione un participante y se pulse sobre el botón CHAT, se iniciará esta actividad para el intercambio de mensajes de texto.
- *Streaming*: cuando desde la actividad de mapa se seleccione un participante y se pulse sobre el botón STREAM, se pasará a esta actividad que permitirá consumir el flujo de audio y vídeo de ese participante.

#### 3.3.1.3 Layouts

Los archivos XML de *layouts* son los que almacenan la información con respecto al diseño gráfico de la aplicación. Estos diseños son los que se aplican a cada una de las diferentes actividades que conforman nuestra aplicación.

El desarrollador tiene a su disposición un gran abanico de posibilidades a la hora de generar el diseño gráfico en la aplicación. A continuación, se definen los distintos elementos gráficos que se han utilizado en esta aplicación:

- *Button*: elemento que detecta pulsaciones del usuario, y que tiene asociado a ese evento código determinado por el programador.
- *ImageView*: muestra una imagen.
- *Toast*: muestra sobre la pantalla un mensaje tipo notificación que dura unos pocos segundos.
- *TextView*: muestra textos que se pueden modificar a través del código.

Todos estos elementos mencionados anteriormente se pueden organizar mediante un *layout* dependiendo de la distribución que queramos emplear:

- *Linear*: coloca los elementos en sentido horizontal o vertical.
- *Relative*: permite especificar la posición entre elementos pudiendo determinar la distancia entre ellos.
- *Constraint*: se comporta exactamente igual que el anterior, la salvedad es que éste lo hace de forma porcentual, adaptándose así a las pantallas de distinto tamaño, ya que guardaría una proporción con las distancias establecidas entre los elementos.

En las Figuras 3-8 y 3-9, se muestran dos imágenes en las que se puede apreciar el *layout* de la actividad principal o *MainActivity*, en la que el usuario deberá registrarse con su ID. En la Figura 3-8 se puede observar el *layout* en formato *XML*, con los diferentes elementos mencionados anteriormente.

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">


    <EditText
        android:id="@+id/editTextId"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_weight="1"
        android:background="@android:color/transparent"
        android:hint="Escribe tu ID"
        android:maxLines="6"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="1.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.499" />

    <Button
        android:id="@+id/btnEnter"
        android:layout_width="184dp"
        android:layout_height="50dp"
        android:layout_gravity="bottom"
        android:clickable="true"
        android:gravity="center"
        android:text="REGISTRARSE"
        android:textSize="14dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.511"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/editTextId"
        app:layout_constraintVertical_bias="0.052" />
```

**Figura 3-8 *Layout* tipo texto de *MainActivity***

Se muestra el *EditText* en el que el usuario procederá a introducir su ID de forma muy intuitiva ya que al iniciar la aplicación dentro de ese cuadro se podrá leer "*Escribe tu ID"*. También vemos cómo se emplea un botón en el que el usuario leerá REGISTRARSE y que deberá pulsar una vez haya tecleado su ID en el *EditText* previamente mencionado.

En la Figura 3-9 se presenta este mismo archivo *XML* del *layout,* pero en la vista de diseño.

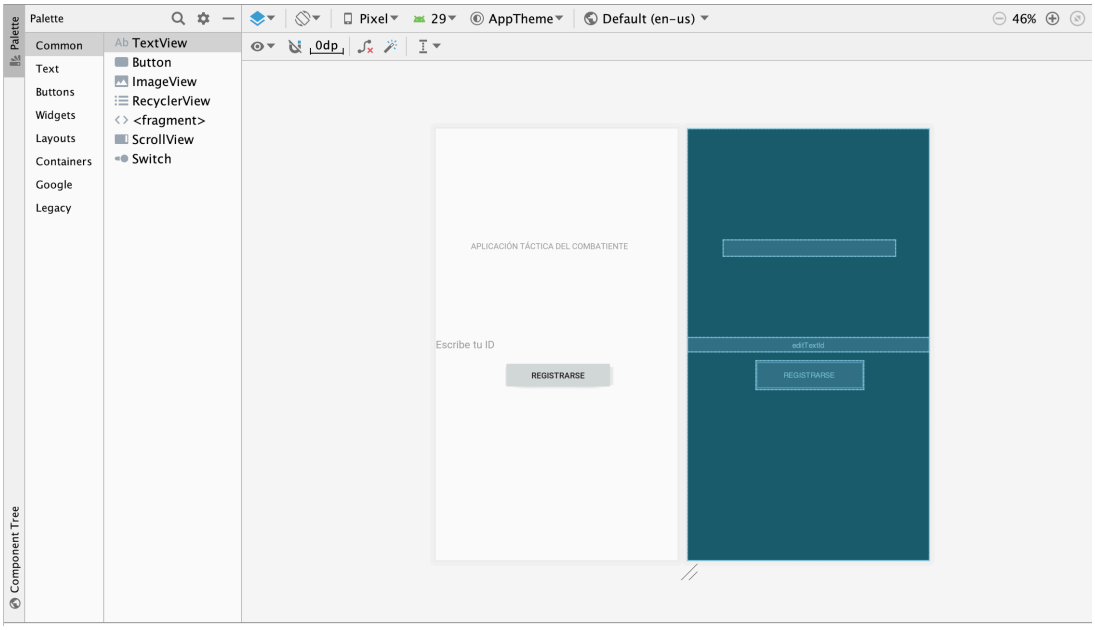



**Figura 3-9 *Layout* tipo diseño de *MainActivity***

Se muestra de forma más intuitiva todo lo mencionado anteriormente, también observando el *TextView* en el que se lee el nombre de la aplicación, Aplicación Táctica del Combatiente. Esta vista de tipo diseño es empleada normalmente para generar la vista gráfica de una forma más cómoda y pudiendo saber antes de compilar cómo se va a mostrar en la aplicación.

### 3.3.1.4 Clases auxiliares

Las clases auxiliares se emplean desde diferentes actividades con la finalidad de poder realizar de forma más cómoda y ordenada el código que podremos reutilizar en la aplicación. De esta manera, se guardan variables y métodos que se emplean varias veces durante el desarrollo de las actividades.

En nuestra aplicación se emplean las diferentes clases auxiliares:

- *Comunicaciones*: en el código que se muestra en la Figura 3-10, perteneciente a la clase *Comunicaciones* se observan dos funciones: *sendUDP* y *checkSocket.* La función *sendUDP* ejecuta código dentro de un *try and catch*, para que *catch* capture cualquier error que se produzca al ejecutar la función.

```
12  /**
13   * Clase que implementa los métodos para enviar y recibir paquetes UDP
14   */
15  class Comunicaciones {
16      // Dirección IP destino (puede ser broadcast)
17      var RemoteHost =""
18      // Puerto destino (por defecto 5554)
19      var RemotePort: Int = 5554
20      // IP local (origen)
21      var ipLocal = ""
22      // Dirección IP del paquete recibido
23      var ipRecibido = ""
24
25      /**
26       * Método para enviar paquetes UDP (envia una cadena recibida como parámetro de entrada)
27       */
28      open fun sendUDP(messageStr: String) {
29
30          // Previene problemas definiendo que esta acción tiene que ser una tarea asíncrona
31          val policy = StrictMode.ThreadPolicy.Builder().permitAll().build()
32          StrictMode.setThreadPolicy(policy)
33
34          try {
35              //Crea el socket para el envío del paquete
36              val socket = DatagramSocket()
37              socket.broadcast = true
38              val sendData = messageStr.toByteArray()
39              // Paquete UDP
40              val sendPacket = DatagramPacket(sendData, sendData.size, InetAddress.getByName(RemoteHost), RemotePort)
41              // Envía el paquete
42              socket.send(sendPacket)
43
44          } catch (e: IOException) {
45              println("Error en envio mensaje UDP "+e.message.toString())
46          }
47      }
48
49      /**
50       * Método para comprobar el estado del socket (true si está libre, false si está en uso)
51       */
52      open fun checkSocket(): Boolean {
53          try {
54              var socket = DatagramSocket(RemotePort)
55              socket.close()
56              return true
57          } catch (e: SocketException) {
58              return false
59          }
60          return false
61      }
62
```

**Figura 3-10 Fragmento de código de la clase *Comunicaciones* (I)**

En el código que se muestra en la Figura 3-11, perteneciente también a la clase *Comunicaciones,* se describe la función *receiveUDP*, en la que se vuelve a ejecutar un *try and catch*. Dentro del bloque

*try* ejecutamos un bucle. Convierte el paquete recibido, que estaba en un array de bytes, a un array de caracteres, siempre que la dirección IP del paquete que recibimos sea diferente a la del dispositivo que está ejecutando la aplicación, además de guardar en la variable *ipRecibido* la dirección IP del paquete. Una vez terminado se cierra el *socket*.

```
63      /**
64       * Método para recibir paquetes UDP (devuelve una cadena recibida como el contenido del paquete)
65       */
66      open fun receiveUDP(broadcast:Boolean):String {
67
68          val charset = Charsets.UTF_8
69          var buffer = ByteArray( size: 2048)
70          var socket: DatagramSocket? = null
71
72          try {
73              //Mantiene un socket abierto escuchando el tráfico UDP destinado a este puerto
74              socket = DatagramSocket(RemotePort)
75              socket.broadcast = true
76
77              val packet = DatagramPacket(buffer, buffer.size)
78              do {
79                  socket.receive(packet)
80                  println(packet.data.toString(charset))
81              } while (packet.address.toString().substring( startIndex: 1) == ipLocal)
82
83              ipRecibido = packet.address.toString().substring( startIndex: 1)
84              return packet.data.toString(charset)
85
86          } catch (e: Exception) {
87              return e.printStackTrace().toString()
88          } finally {
89              // Cierra el socket
90              socket?.close()
91          }
92          // En caso de error devolvemos cadena vacía
93          return ""
94      }
95  }
```

**Figura 3-11 Fragmento de código de la clase *Comunicaciones* (II)**

- *CustomInfoWindow*: esta clase nos sirve para mostrar lo que se desee en la ventana emergente del marcador correspondiente al participante seleccionado. En este caso lo que se hace es simplemente mostrar el nombre del participante seleccionado, para que el usuario sepa quién está conectado y poder establecer comunicaciones.

```
15  /**
16   * Clase para mostrar información cuando se pulsa sobre un marcador
17   */
18  class CustomInfoWindow(val context: Context) : GoogleMap.InfoWindowAdapter {
19
20      // Identificador del participante seleccionado
21      lateinit var participanteSeleccionado: String
22
23      /**
24       * Método llamado para mostrar contenido en la ventada del marcador
25       */
26      override fun getInfoContents(p0: Marker?): View {
27
28          var mInfoView = (context as Activity).layoutInflater.inflate(R.layout.custom_window,  root: null)
29          // Introduce la información que tiene asociada el marcador
30          mInfoView.editTextFormInfoWindow.text = p0!!.title
31          participanteSeleccionado = p0!!.title
32          return mInfoView
33      }
34
35      override fun getInfoWindow(p0: Marker?): View? {
36          return null
37      }
38  }
```

**Figura 3-12 Código de la clase *CustomInfoWindow***

- *Participante*: en esta clase organizamos los diferentes parámetros que diferencian a cada participante. De este modo, cada uno tiene un identificador (ID), una latitud y longitud correspondiente a su ubicación y una dirección IP. Con la función *show* añadimos un marcador en el mapa en la ubicación del participante. El método *update* nos sirve para actualizar la posición de ese participante, borrando el marcador anterior. El método *remove* simplemente elimina el marcador.

```
10  /**
11   * Clase que modela al participante en el Mapa
12   */
13  class Participante {
14      /**
15       * Atributos que identifican a un participante:
16       * - identificador
17       * - latitud y longitud (coordenadas GPS)
18       * - dirección IP
19       */
20      lateinit var id: String
21      lateinit var lon: String
22      lateinit var lat: String
23      lateinit var ip: String
24
25      // Elemento marcador
26      lateinit var marcador: Marker
27
28      /**
29       * Método llamado cuando se quiere mostrar un participante en el mapa
30       */
31      open fun show(map: GoogleMap,contexto: Context){
32          // establece la posición GPS
33          val miPosicion = LatLng(lat.toDouble(), lon.toDouble())
34          // crea y añade el marcador al mapa
35          marcador=map.addMarker(MarkerOptions().position(miPosicion).title("$id"))
36          marcador.showInfoWindow()
37      }
38
39      /**
40       * Método llamado para actualizar la posición de un participante
41       */
42      open fun update(latRecibido:String,lonRecibido:String){
43          lat=latRecibido
44          lon=lonRecibido
45          marcador.remove()
46      }
47
48      /**
49       * Método llamado para eliminar un participante del mapa
50       */
51      open fun remove(){
52          marcador.remove()
53      }
54  }
```

**Figura 3-13** ***Clase Participante***

### *3.3.2 Etapas de desarrollo*

En este apartado se van a explicar en detalle las diferentes etapas por las que ha ido avanzando este proyecto, desde una simple pantalla de *login* para el registro del usuario, hasta la actividad que permite consumir el *streaming* de un usuario.

### 3.3.2.1 Pantalla de registro del usuario

Esta es la primera pantalla o actividad que nos encontramos cuando ejecutamos la aplicación, su interacción con el usuario es bastante intuitiva.

El código empleado para esta actividad se muestra en la Figura 3-14.

```
9   class MainActivity : AppCompatActivity() {
10
11      override fun onCreate(savedInstanceState: Bundle?) {
12          super.onCreate(savedInstanceState)
13          setContentView(R.layout.activity_main)
14
15          // Añadimos acciones de evento al botón de registro
16
17          btnEnter.setOnClickListener { it: View!
18
19              // Le pasamos a la actividad 'MapsActivity' el identificador del usuario
20
21              var miId: String = editTextId.text.toString()
22              val intent = Intent( packageContext: this, MapsActivity::class.java)
23              intent.putExtra( name: "Id", miId)
24              startActivity(intent)
25          }
26      }
27  }
28
```

**Figura 3-14 Código de *MainActivity* en la que se registra el usuario**

La vista que le aparecería al usuario se muestra en la Figura 3-15:

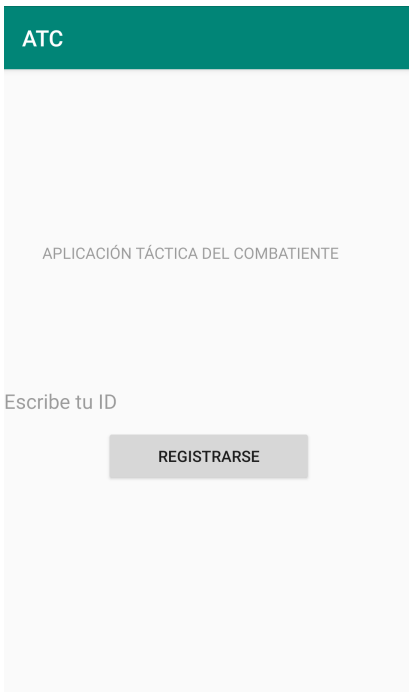



**Figura 3-15 Pantalla de registro del usuario**

### 3.3.2.2 Presentación de la ubicación del usuario integrada en *Google Maps*

La segunda actividad, una vez nos hemos registrado con nuestro identificador (ID), consiste en obtener la posición en tiempo real del dispositivo y representarla en el mapa. Para poder llevarlo a cabo, se ha desarrollado el código que se presenta en la Figura 3-16. Se crea un *callback* para poder actualizar la posición de forma permanente cada vez que es llamado este objeto. En la línea 82 se implementa la representación en el mapa del propio usuario, en las líneas 83 y 84, respectivamente, se añade el botón para acceder de forma rápida a la ubicación actual y el zoom del mapa. En las variables que se declaran en las líneas 88 y 89 como *miLatitud* y *miLongitud* se guardan los correspondientes datos de la posición obtenida.

```
78      callback = object : LocationCallback() {
79
80          override fun onLocationResult(locationResult: LocationResult?) {
81          super.onLocationResult(locationResult)
82              mMap.isMyLocationEnabled = true
83              mMap.uiSettings.isMyLocationButtonEnabled = true
84              mMap.uiSettings.isZoomControlsEnabled = true
85
86              // Obtiene posición GPS actual
87              for (ubicacion in locationResult?.locations!!) { //OBTIENE MI UBICACIÓN ACTUAL
88              miLatitud = ubicacion.latitude.toString()
89              miLongitud = ubicacion.longitude.toString()
90              }
```

**Figura 3-16 Código para la obtención de latitud y longitud**

Una vez se obtienen las coordenadas del usuario, se podrán enviar en un mensaje *broadcast* al resto de participantes, junto con el nombre de usuario (ID) y la dirección IP.

Un ejemplo de cómo esta actividad sería vista por el usuario, se muestra en la Figura 3-17.

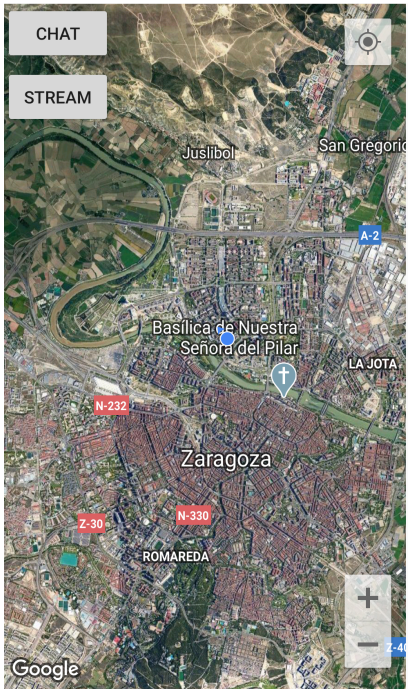



**Figura 3-17 El círculo azul muestra la ubicación del usuario de la aplicación**

### 3.3.2.3 Envío *broadcast* y recepción de ubicaciones y su actualización

Dentro de la actividad de mapas se debe enviar mediante mensajes *broadcast* la posición, el identificador (ID) y también la dirección IP, a la vez que se debe abrir un hilo de escucha para poder recibir los mensajes *broadcast* de otros usuarios que estén conectados, de esta manera se podrán representar en el mapa conociendo el ID y la dirección IP asociados a dicha posición.

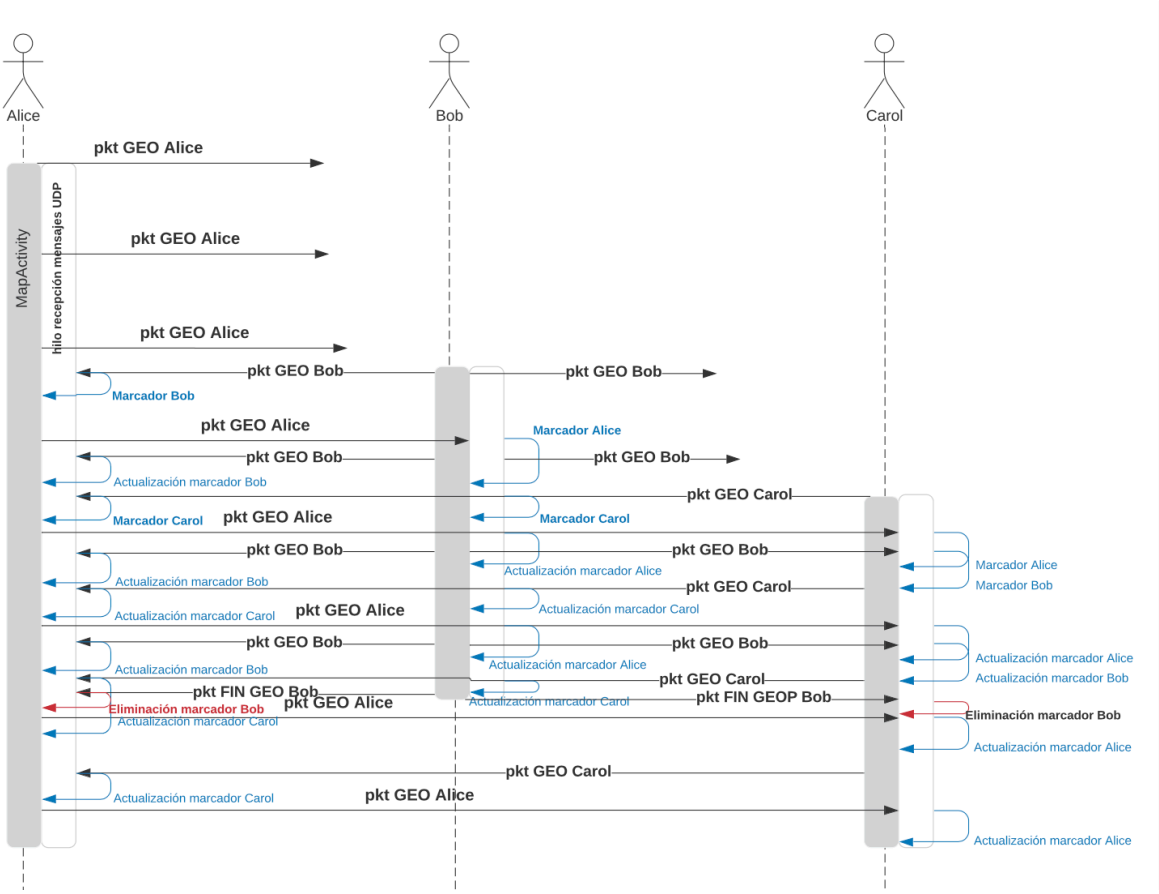



**Figura 3-18 Diagrama del flujo de paquetes *broadcast* para la actualización de las ubicaciones de los diferentes usuarios**

El diagrama mostrado en la Figura 3-18 permite comprender el funcionamiento del proceso de actualización de posiciones de los participantes en la aplicación (*Alice*, *Bob* y *Carol* en este ejemplo). Para ello se distribuyen los mensajes *broadcast* que incluyen el identificador (ID), la dirección IP, la latitud y longitud del participante.

Para obtener la dirección IP del usuario se desarrolla el siguiente código (ver Figura 3-19).

```
// Obtiene ip local
var wifiManager: WifiManager
wifiManager = this.applicationContext?.getSystemService(Context.WIFI_SERVICE) as WifiManager
var iiplocal = wifiManager.connectionInfo.ipAddress
ip = Formatter.formatIpAddress(iiplocal)
```

**Figura 3-19 Obtención dirección IP del usuario**

En este momento, se tienen los cuatro parámetros que debemos enviar mediante mensaje *broadcast*: ID del usuario, latitud, longitud y dirección IP. A continuación, se deben combinar estos cuatro datos y enviarlos, para ello se emplea el código en la Figura 3-20.

```
// Construye el mensaje: ID:latitud:longitud,IP:
var mensajeBroadcast = String()
mensajeBroadcast = miId+":"+miLatitud+","+miLongitud+","+ip+":"
// Envío de mensaje
udpComm.sendUDP(mensajeBroadcast)
}
```

**Figura 3-20 Código para la construcción del mensaje *broadcast***

El siguiente paso consiste en conseguir que cada participante reciba estos mensajes. Para ello se ejecuta un hilo de escucha de mensajes (ver Figura 3-21).

```
// Ejecuta la escucha de mensajes broadcast y de inicio de chat
hiloMensajesBroadcast = Thread {
    receiveBroadcast()
}
hiloMensajesBroadcast.start()
```

**Figura 3-21 Código para la ejecución del hilo de escucha de mensajes**

Una vez se ha recibido el mensaje, se debe descomponer. De esta forma, el usuario que lo ha recibido obtendrá los cuatro parámetros por separado. Para ello se emplea el siguiente código (ver Figura 3-22).

```
val idRecibido = recibido.substring(0, recibido.indexOf( string: ":"))
val latRecibido = recibido.substring(recibido.indexOf( string: ":") + 1, recibido.indexOf( string: ","))
val lonRecibido = recibido.substring(recibido.indexOf( string: ",") + 1, recibido.lastIndexOf( string: ","))
var cadenaAuxiliar = recibido.substring( startIndex: recibido.lastIndexOf( string: ",") + 1)
var ipRecibida =cadenaAuxiliar.substring(0,cadenaAuxiliar.indexOf( string: ":"))
```

**Figura 3-22 Código para el *parseo* del mensaje *broadcast***

En este punto, se han encontrado diversos problemas. Uno de ellos era el error en el envío de direcciones IP en paquetes *broadcast*. Por defecto, el terminal enviaba su dirección IP adjuntando caracteres aleatorios, lo que complicaba su obtención. La solución a este problema fue incorporar caracteres entre los parámetros que se envían y una cadena auxiliar para poder separar el mensaje de forma correcta.

Otro problema se planteó al emplear la red WiFi del edificio de investigación del CUD, en la que se realizaron las primeras pruebas de la aplicación. Se detectó que había *port isolation*, esto implica que los mensajes *broadcast* enviados a través de esa WiFi no se distribuían entre los terminales conectados. Por lo tanto, la gestión del tráfico *broadcast* tiene un impacto en el correcto funcionamiento de la aplicación.

Una vez se han resuelto los problemas comentados anteriormente, se deben clasificar los datos obtenidos para poder registrar el usuario encontrado en nuestra lista de participantes. Para ello se desarrolla un código (ver Figura 3-23) que actúe de la siguiente manera: si el ID recibido en el mensaje

*broadcast* coincide con algún ID ya registrado, actualiza la posición de ese participante y lo representa en el mapa. En el caso de que no encuentre ninguna coincidencia con el ID recibido, se registra al nuevo participante, añadiéndolo a la clase *Participante* y lo representa en el mapa.

Para poder representar a cada participante, de hecho, para poder mostrar mensajes u otros elementos por pantalla necesitamos emplear *runOnUiThread* (ver Figura 3-23). En este caso para acceder a elementos del hilo principal, en caso contrario hay error.

Destacar que, si el participante ya existía previamente, cuando se actualiza su posición se debe borrar el marcador de la ubicación anterior, en caso contrario, se acumularán marcadores en el mapa, lo que dificultará y empeorará el uso de la aplicación.

```
    // Buscamos al usuario, en principio no está en la lista
    var encontrado=0
    for (participante in participantes) {
        // Una vez encontrado
        if (participante.id == idRecibido) {
            // Si la latitud recibida es 100, no se trata de una actualización de posición, es una salida de la aplicación
            if (latRecibido=="100") {
                participante.remove()
            } else {
                // Si lo encontramos actualizamos su posición
                this@MapsActivity.runOnUiThread(java.lang.Runnable {
                    participante.update(latRecibido, lonRecibido)
                    participante.show(mMap,  contexto: this)
                })
            }
            encontrado=1
        }
    }
    // Si no lo hemos encontrado, creamos una entrada para el en la lista de participantes
    // y lo añadimos al mapa como marcador
    if (encontrado==0) {
        var participante = Participante()
        participante.id = idRecibido
        participante.lat = latRecibido
        participante.lon = lonRecibido
        participante.ip = ipRecibida

        this@MapsActivity.runOnUiThread(java.lang.Runnable {
            participante.show(mMap,  contexto: this)
        })
        participantes.add(participante)
    }
}
```

**Figura 3-23 Código para la gestión de participantes de la aplicación**

### 3.3.2.4 Implementación del chat

La actividad de chat se puede ejecutar una vez se ha detectado a otro usuario que esté conectado. Para ello, se ha desarrollado una función que diferencia los mensajes que son de tipo *broadcast* de los que son de chat.

```
fun receiveBroadcast() {

    while (true) {
        // Recibimos el mensaje UDP
        var recibido = udpComm.receiveUDP( broadcast: true)

        // Si es un mensaje de CHAT empieza por un caracter '('
        if (recibido.indexOf( string: "(") != -1) {
            // Se muestra mensaje invitando a iniciar Chat
            this@MapsActivity.runOnUiThread(java.lang.Runnable {
                Toast.makeText( context: this@MapsActivity,  text: recibido+"quiere iniciar chat", Toast.LENGTH_LONG).show()
            })
            return
        }
        // Si estamos en este punto NO es un mensaje de CHAT: se parsea el mensaje recibido: [ID][LAT][LON][IP_RECIBIDA]
        val idRecibido = recibido.substring(0, recibido.indexOf( string: ":"))
        val latRecibido = recibido.substring(recibido.indexOf( string: ":") + 1, recibido.indexOf( string: ","))
        val lonRecibido = recibido.substring(recibido.indexOf( string: ",") + 1, recibido.lastIndexOf( string: ","))
        var cadenaAuxiliar = recibido.substring( startIndex: recibido.lastIndexOf( string: ",") + 1)
        var ipRecibida =cadenaAuxiliar.substring(0,cadenaAuxiliar.indexOf( string: ":"))
```

**Figura 3-24 Función para la discriminación de mensajes *broadcast* y chat**

Una vez se ha detectado un mensaje de chat, se notifica al participante seleccionado que queremos iniciar un chat. En este caso, el usuario que ha recibido dicha notificación, puede acceder al chat o ignorar la alerta.

En el código mostrado en la Figura 3-25, se observa que cuando el usuario pulsa el botón CHAT, se dirige a una nueva actividad, a la que además pasa los siguientes parámetros: ID del usuario con el que va a realizar el chat, su dirección IP y también el ID del propio usuario, que necesitaremos para poder etiquetar los mensajes enviados desde este terminal.

```
// Acciones que se ejecutan al pulsar sobre el botón de CHAT
buttonChat.setOnClickListener { it: View!
    // Buscamos al participante e iniciamos chat con él
    var idChat = ventana.participanteSeleccionado
    for (participante in participantes) {
        if (participante.id == idChat) {
            udpComm.sendUDP( messageStr: "("+miId+")")
            var intent = Intent( packageContext: this@MapsActivity, Chat::class.java)
            intent.putExtra( name: "ID", idChat)
            intent.putExtra( name: "IP", participante.ip)
            intent.putExtra( name: "miId", miId)
            startActivity(intent)
        }
    }
}
```

**Figura 3-25 Código de la función que ejecuta la actividad *Chat***

Cuando ambos usuarios se encuentran en la misma ventana de chat, se ejecuta un hilo de escucha de mensajes tipo chat (ver Figura 3-26), diferente al hilo que escucha mensajes tipo *broadcast*. El motivo por el que el hilo es diferente es debido al problema que se presentó con el *socket* que se empleaba. Lo que implicaba que los mensajes enviados entre terminales no llegaban al puerto de destino. Es por ello que se implementó un *checkSocket* como se puede observar en la Figura 3-26 antes de ejecutar ese nuevo hilo de escucha.

```
if (comm.checkSocket()) {  //EJECUTAMOS EL HILO DE ESCUCHA DE MENSAJES
    Thread {
        while (true) {
            receiveChat()
        }
    }.start()
}
```

**Figura 3-26 Código de la función que ejecuta el hilo de escucha de mensajes de chat**

### *3.3.2.5* Implementación del *streaming*

Entre las diferentes opciones que se han valorado para la implementación de la función de *streaming*, destacan:

- El empleo de una biblioteca escrita en *Java* que ofrece soporte para *streaming* empleando RTSP [24].
- Librería de *Bambuser* [25], muy simple y con buen funcionamiento, pero la dependencia de su servidor y que el flujo de vídeo y audio se esté enviando fuera de la red local, fueron factores determinantes para declinar su elección.

- Librería de RTMP [26], presenta un esquema algo más complejo que los otros dos porque hace uso de un servidor externo para recibir los flujos de vídeo y audio.

Se ha optado por la primera opción, empleando el protocolo RTSP, ya que nos permite controlar el flujo de audio y vídeo sobre la red IP. Para poder implementarlo, se ha optado por el empleo de una función que ejecute desde nuestra aplicación los servicios que proporciona la aplicación *VLC media player* [27] para poder reproducir el vídeo.

Cuando se ejecuta por primera vez *MapsActivity*, se abre también un hilo de ejecución con un servidor de *streaming* preparado (ver Figura 3-27)

```
// Se crea un servicio RTSP en el puerto 1935
rtspServerCamera1 = RtspServerCamera1(surfaceView,  connectCheckerRtsp: this,  port: 1935)
surfaceView.holder.addCallback( callback: this)
```

**Figura 3-27 Función que ejecuta la actividad *Stream***

En este caso, cabe destacar como punto negativo que, al crear la *surfaceView,* el dispositivo está reproduciendo lo que graba con la cámara principal de forma continua, esto implica un consumo considerable de la batería. Aunque esto es transparente para el usuario, ya que esa *surfaceView* va a estar cubierta por la interfaz gráfica de la aplicación.

Cuando se pulsa el botón STREAM, se ejecuta el código que muestra la Figura 3-28. Una vez se ha seleccionado el usuario que está conectado y del que se desea ver lo que está emitiendo, la aplicación abre un enlace, formado por la dirección IP del usuario seleccionado y el puerto 1935, que es el que se emplea por defecto. Mediante el protocolo RTSP se podrá consumir el vídeo y el audio. Cabe destacar en este bloque que, gracias a la aplicación *VLC*, es posible reproducir ese *stream*, por ello se incorpora un *Intent* para su lanzamiento.

```
    // Acciones que se ejecutan al pulsar sobre el botón de STREAMING
    streaming.setOnClickListener { it: View!
        // Buscamos al participante e iniciamos el consumo del streaming de él utilizando VLC
        var idStreaming = ventana.participanteSeleccionado
        for (participante in participantes) {
            if (participante.id == idStreaming) {
                val uri: Uri = Uri.parse( uriString: "rtsp://"+participante.ip+":1935")
                var intent = Intent(Intent.ACTION_VIEW,uri)
                intent.setPackage("org.videolan.vlc")
                startActivity(intent)
            }
        }
    }
}
```

**Figura 3-28 Función que ejecuta la actividad *Stream***

# 4 RESULTADO DEL TFG: APLICACIÓN ATC

Este capítulo presenta un breve manual de usuario de la aplicación, así como se muestran los principales resultados de las pruebas realizadas a la aplicación desarrollada en este TFG.

## 4.1 Funcionamiento de la aplicación

En este apartado se observa cómo funciona la aplicación que se ha desarrollado. Para ello se muestran capturas de pantalla de uno de los diferentes dispositivos en los que se instaló la aplicación, mostrando en primera persona la experiencia de uso de la aplicación y cómo interacciona con otros dispositivos que están conectados.

La primera pantalla vista por el usuario cuando ejecuta la aplicación sería la vista en el apartado 3.3.2.1, ver Figura 4-1.

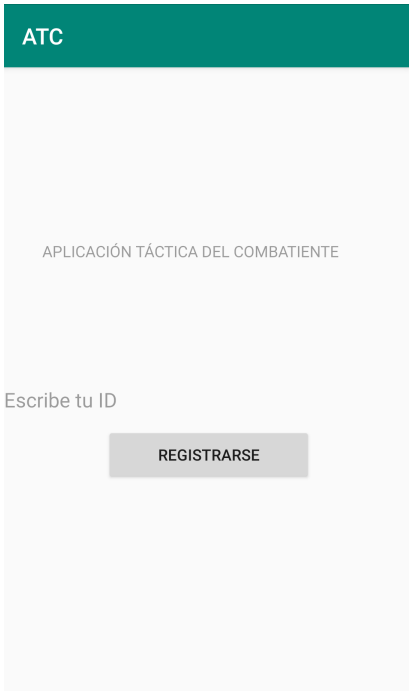



**Figura 4-1 Pantalla de registro del usuario**

Una vez nos hemos registrado con nuestro identificador de usuario (ID), se mostrará la pantalla de la Figura 4-2, donde se visualiza con un punto azul la localización actual del usuario. La ilustración es similar a la del apartado 3.3.2.2.

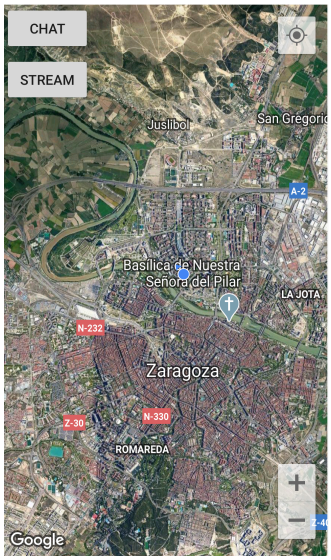



**Figura 4-2 Pantalla de mapa en la aplicación**

Nos encontraremos en esta pantalla hasta que se conecte otro participante. En este ejemplo (ver Figura 4-3) vemos cómo se ha detectado al usuario *Bob* y su ubicación se ha representado con un marcador de color rojo.

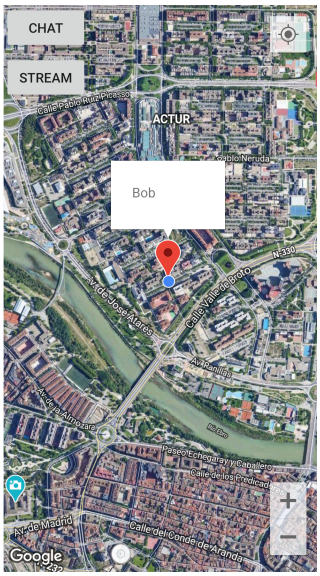



**Figura 4-3 Pantalla de mapa con usuario detectado**

Una vez nos encontramos en este punto, tenemos dos opciones:

- Establecer una conversación de chat. En la Figura 4-4 se observa una captura de pantalla de una conversación entre dos usuarios. En el caso del ejemplo son dos usuarios: patrulla A y patrulla B. Como se aprecia en la Figura 4-4, la captura de pantalla se hizo desde el terminal correspondiente a la patrulla A.

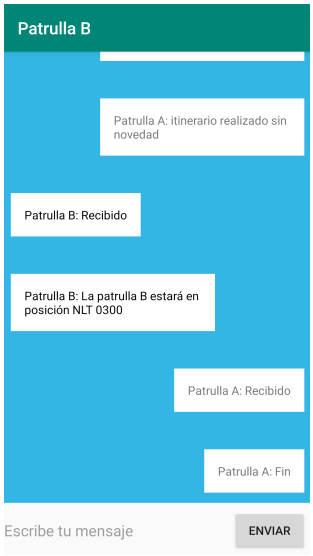



**Figura 4-4 Pantalla de chat entre dos usuarios**

- Si pulsamos sobre el botón STREAM una vez hemos seleccionado al usuario de interés, consumiremos el vídeo y audio de su cámara principal. Un ejemplo de una prueba durante un *streaming* en un pasillo cualquiera (Figura 4-5) se puede observar que el teléfono de la izquierda está consumiendo lo que el teléfono de la derecha está grabando.

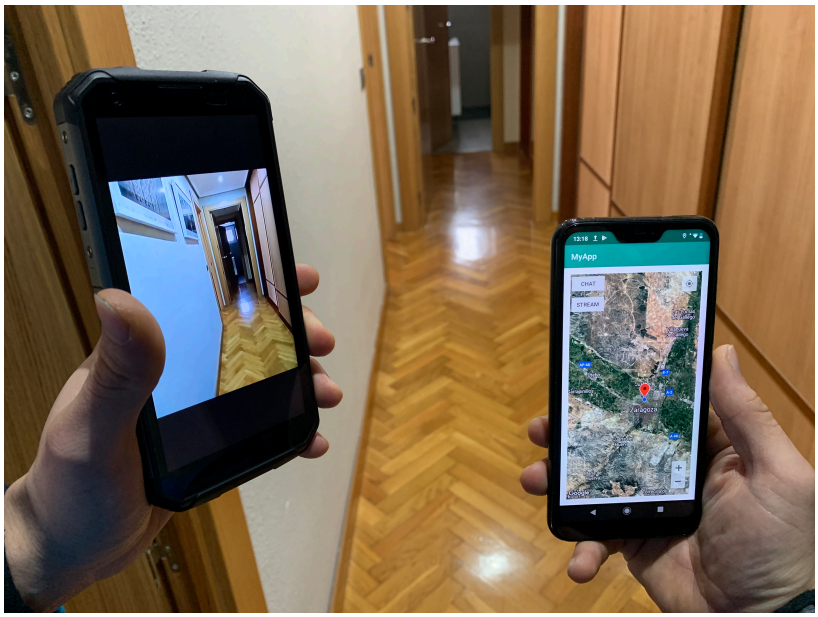



**Figura 4-5 Imagen de una prueba durante el *streaming* en un pasillo**

Por último, si el usuario espera a que se conecte otro participante, se representaría de la siguiente forma (ver Figura 4-6), siendo el propio usuario representado con un punto azul y los otros dos participantes representados con marcadores de color rojo.

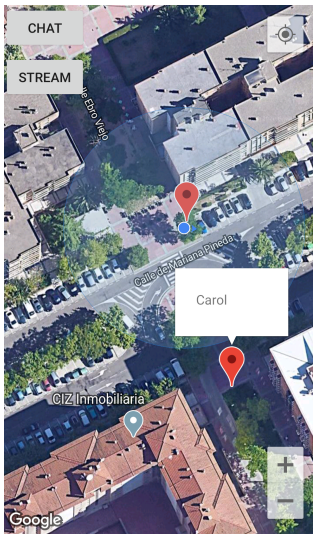



**Figura 4-6 Localización de dos participantes**

## 4.2 Pruebas de la aplicación

La aplicación ha sido depurada en diferentes etapas, puesto que no se ha avanzado a la siguiente etapa de desarrollo mientras no se eliminasen todos los errores de la etapa previa. De este modo, se siguieron etapas de depuración independientes para el desarrollo de las funcionalidades de geolocalización, de envío de mensajes *broadcast*, de mensajería de texto (chat) y de *streaming*.

Las principales dificultades detectadas en cada etapa se describen a continuación:

- Durante la realización de las primeras pruebas de la aplicación se detectó que había *port isolation* en la red WiFi del edificio de investigación del CUD. Esto implica que los mensajes *broadcast* enviados a través de esa WiFi no se distribuían entre los terminales conectados. La solución adoptada fue cambiar a otra red que no tuviera *port isolation*.
- Error en el envío de direcciones IP en paquetes *broadcast*. El terminal adjunta caracteres aleatorios al final de la dirección IP, lo que complicaba su obtención una vez llegaba el mensaje a otro terminal. La solución a este problema fue incorporar caracteres entre los parámetros que se envían y una cadena auxiliar para poder detectar y así, separar el mensaje de forma correcta.
- En la etapa de implementación de chat se planteó un nuevo problema. Debido a que no se comprobaba previamente el puerto (*socket*) donde se envían los mensajes de chat, éstos no llegaban al otro terminal. La solución fue modificar el código que define el hilo de escucha de mensajes de tipo chat e implementar una función que comprobaba el puerto antes de poder ejecutar el hilo de escucha.

La aplicación se ha probado en diferentes escenarios, con diferentes redes WiFi y se puede concluir que se comporta acorde a lo esperado. Asimismo, las diferentes pruebas que se han llevado a cabo para comprobar el funcionamiento de la aplicación se ejecutaron en un espacio reducido debido a la limitación del alcance de la red WiFi utilizada con un único *Access Point.*

# 5 CONCLUSIONES Y LÍNEAS FUTURAS

En este capítulo se exponen las conclusiones que se han obtenido durante la realización del TFG y se propondrán algunas posibles líneas futuras del proyecto.

## 5.1 Conclusiones

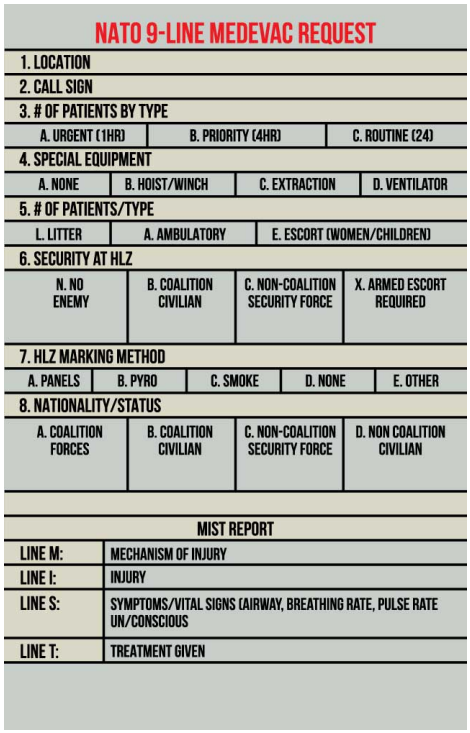
El resultado final es una aplicación cuyo objetivo es que el usuario pueda comunicarse a través de un chat o consumir un flujo de audio y vídeo de otro participante que esté conectado al mismo tiempo en la misma red WiFi usando la misma aplicación. A esto le añadimos una geolocalización continua del resto de participantes conectados. En una vista general de la aplicación, conseguimos obtener un conocimiento más preciso de nuestro entorno. La aplicación ATC se ha desarrollado con la finalidad de emplearse en despliegues de Infantería de Marina. En los escenarios actuales en los que este cuerpo está operando, una aplicación de estas características proporciona al soldado que se encuentra en una patrulla, en un puesto de observación o bien en un puesto de mando, un conocimiento mucho más ampliado y preciso del entorno, beneficiando de esta manera la toma de decisiones. Revisando el apartado 1.3 de esta memoria, se puede concluir que se ha alcanzado satisfactoriamente el objetivo principal de este TFG.

## 5.2 Líneas futuras

Respecto al desarrollo futuro de esta aplicación se propone lo siguiente:

- Mejorar la interfaz y presentación de la aplicación para hacerla más vistosa e intuitiva.
- Considerar el empleo de protocolos de seguridad para las comunicaciones.
- Añadir las siguientes funcionalidades:
    - Enviar mensajes predeterminados que se desarrollan de forma frecuente en las operaciones actuales: MEDEVAC [28] (ver Figura 5-1) y SPOTREPORT [29] (ver Figura 5-2). El primer mensaje consta de un formato que se emite cuando se ha producido una baja y es necesaria su evacuación. El segundo es un mensaje que emite una unidad cuando observa algo que es necesario enviar al mando, ya que requiere de información esencial que puede tener efecto en un planeamiento u operación que se esté llevando a cabo.
    - Enviar paquetes a otros participantes que contengan una localización (coordenadas) y que pueda representarse en el mapa.
    - Conseguir emplear mapas descargados previamente para no emplear la API de *Google Maps* [22] y de esta forma no depender de ella. Esto permitiría incluso poder funcionar sin conexión a Internet.

- Desarrollar esta aplicación para otros sistemas operativos, como puede ser iOS.
- Desarrollar una red *mesh* [4] dedicada a esta aplicación que consiga ampliar su radio de acción, ya que actualmente está dentro del alcance de una red WiFi al uso.



| NATO 9-LINE MEDEVAC REQUEST | | | |
|---|---|---|---|
| 1. LOCATION | | | |
| 2. CALL SIGN | | | |
| 3. # OF PATIENTS BY TYPE | | | |
| A. URGENT (1HR) | B. PRIORITY (4HR) | C. ROUTINE (24) | |
| 4. SPECIAL EQUIPMENT | | | |
| A. NONE | B. HOIST/WINCH | C. EXTRACTION | D. VENTILATOR |
| 5. # OF PATIENTS/TYPE | | | |
| L. LITTER | A. AMBULATORY | E. ESCORT (WOMEN/CHILDREN) | |
| 6. SECURITY AT HLZ | | | |
| N. NO ENEMY | B. COALITION CIVILIAN | C. NON-COALITION SECURITY FORCE | X. ARMED ESCORT REQUIRED |
| 7. HLZ MARKING METHOD | | | |
| A. PANELS | B. PYRO | C. SMOKE | D. NONE / E. OTHER |
| 8. NATIONALITY/STATUS | | | |
| A. COALITION FORCES | B. COALITION CIVILIAN | C. NON-COALITION SECURITY FORCE | D. NON COALITION CIVILIAN |

| MIST REPORT | |
|---|---|
| LINE M: | MECHANISM OF INJURY |
| LINE I: | INJURY |
| LINE S: | SYMPTOMS/VITAL SIGNS (AIRWAY, BREATHING RATE, PULSE RATE UN/CONSCIOUS |
| LINE T: | TREATMENT GIVEN |

**Figura 5-1 Formato MEDEVAC [28]**

| SPOT REPORT/SALUTE | |
|---|---|
| LINE | ITEM |
| 1 | Size |
| 2 | Activity |
| 3 | Location |
| 4 | Unit/Uniform |
| 5 | Time observed |
| 6 | Equipment |

**Figura 5-2 Formato SPOTREPORT [29]**

# 6 BIBLIOGRAFÍA

[1] «STANAG 4609, acuerdo entre países OTAN que define el formato para el contenido multimedia intercambiado.,» [En línea]. Available: https://gwg.nga.mil/misb/docs/nato_docs/STANAG_4609_Ed3.pdf. [Último acceso: 22 enero 2020].

[2] M. Defensa. [En línea]. Available: tecnologiaeinnovacion.defensa.gob.es/es-es/Estrategia/HojasDeRuta/Paginas/C4I.aspx. [Último acceso: 2 abril 2020].

[3] «Artículo de streamingmedia.com que describe el estado actual de las tecnologías streaming en el ámbito militar,» [En línea]. Available: https://www.streamingmedia.com/Articles/Editorial/Featured-Articles/Video-in-the-War-Zone-The-Current-State-of-Military-Streaming-101310.aspx. [Último acceso: 22 enero 2020].

[4] «Red mesh: definición, funcionamiento y qué implica su implementación.,» [En línea]. Available: https://www.xataka.com/especiales/redes-wifi-mesh-que-son-como-funcionan-y-por-que-pueden-mejorar-tu-red-wifi-en-casa. [Último acceso: 16 marzo 2020].

[5] «ATAK,» [En línea]. Available: https://federallabs.org/successes/awards/awards-gallery/2018/android-based-team-awareness-kit-–-atak. [Último acceso: 22 enero 2020].

[6] «CIVTAK,» [En línea]. Available: https://www.takciv.org. [Último acceso: 22 enero 2020].

[7] «ATAK: definición y funcionalidades,» [En línea]. Available: https://tacticalblueprint.com/what-is-atak/. [Último acceso: 22 enero 2020].

[8] «Estadística de empleo móviles Android,» [En línea]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201912-201912-bar. [Último acceso: 22 enero 2020].

[9] «Google play: definición, servicios y para qué sirven,» [En línea]. Available: https://www.xatakandroid.com/sistema-operativo/que-servicios-google-play-sirven. [Último acceso: 16 marzo 2020].

[10] «Niveles SO Android,» [En línea]. Available: https://developer.android.com/guide/platform?hl=es-419. [Último acceso: 10 marzo 2020].

[11] «Android Studio,» [En línea]. Available: https://developer.android.com/studio/intro?hl=es-419. [Último acceso: 22 enero 2020].

[12] «IntelliJ IDEA,» [En línea]. Available: https://www.jetbrains.com/es-es/idea/. [Último acceso: 22 enero 2020].

[13] «Android manifest,» [En línea]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro?hl=es-419. [Último acceso: 10 marzo 2020].

[14] «Ciclo de vida de una actividad en Android,» [En línea]. Available: https://medium.com/@Emmitta/ciclo-de-vida-de-una-actividad-android-f30f8f2d1256. [Último acceso: 10 marzo 2020].

[15] «Kotlin,» [En línea]. Available: https://kotlinlang.org. [Último acceso: 22 enero 2020].

[16] «Preferencias de Kotlin sobre Java,» [En línea]. Available: https://www.xatakandroid.com/programacion-android/no-hara-falta-aprender-java-para-programar-android-kotlin-pasa-a-ser-preferido-google. [Último acceso: 8 marzo 2020].

[17] «Características de Xiaomi Mi A2 Lite,» [En línea]. Available: https://www.mi.com/es/mi-a2-lite/. [Último acceso: 22 enero 2020].

[18] «Router dlink,» [En línea]. Available: https://eu.dlink.com/es/es/products/dir-615-wireless-n-300-router. [Último acceso: 4 abril 2020].

[19] «Diferentes protocolos streaming (RTMP, RTSP, RTP),» [En línea]. Available: https://www.usastreams.com/blog-tecnologia/2578/protocolos-de-streaming-hls-rtmp-rtsprtp-mpeg-ts/. [Último acceso: 15 marzo 2020].

[20] F. J. S. Alonso, «Tecnologías de streaming,» [En línea]. Available: http://www.atc.uniovi.es/atc/teleco/5tm/archives/8streaming.pdf. [Último acceso: 15 marzo 2020].

[21] «Protocolos streaming,» [En línea]. Available: https://prezi.com/gi4voktj6m-m/servicios-de-streaming/. [Último acceso: 4 abril 2020].

[22] «Google Maps: definición y para qué sirve,» [En línea]. Available: https://neoattack.com/neowiki/google-maps/. [Último acceso: 16 marzo 2020].

[23] «Niveles API en Android,» [En línea]. Available: https://stackoverflow.com/questions/8629687/android-api-level-vs-android-version/8629727#8629727. [Último acceso: 16 marzo 2020].

[24] «Librería RTSP empleada para la implementación del streaming,» [En línea]. Available: https://github.com/pedroSG94/RTSP-Server. [Último acceso: 20 febrero 2020].

[25] «Empleo de librería de Bambuser para implementación de streaming,» [En línea]. Available: https://bambuser.com/docs/broadcasting/android-kotlin/. [Último acceso: 20 febrero 2020].

[26] «Librería servidor RTMP para implementación de streaming,» [En línea]. Available: https://github.com/TakuSemba/RtmpPublisher. [Último acceso: 20 febrero 2020].

[27] «Aplicación VLC,» [En línea]. Available: https://www.videolan.org/index.es.html. [Último acceso: 16 marzo 2020].

[28] «Formato mensaje MEDEVAC,» [En línea]. Available: https://blog.refactortactical.com/blog/how-to-call-in-a-9line-medevac-and-mist-report/. [Último acceso: 16 marzo 2020].

[29] «Formato mensaje SPOTREP,» [En línea]. Available: https://www.armystudyguide.com/content/Leadersbook_information/Combat_Leaders_Guide/miscellaneous-spot-report-2.shtml. [Último acceso: 16 marzo 2020].

# ANEXO I: CÓDIGO DE LA APLICACIÓN

En este anexo se adjunta todo el código de las diferentes actividades y clases empleadas para el desarrollo de la aplicación desarrollada en este TFG.

Primero se adjuntan las tres actividades.

*MainActivity:*

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Añadimos acciones de evento al botón de registro

        btnEnter.setOnClickListener {

            // Le pasamos a la actividad 'MapsActivity' el identificador del usuario

            var miId: String = editTextId.text.toString()
            val intent = Intent(this, MapsActivity::class.java)
            intent.putExtra("Id", miId)
            startActivity(intent)
        }
    }
}
```

*MapsActivity:*

```
/**
 * Actividad encargada de mostrar el mapa, posicionar el usuario en él, enviar y recibir
mensajes de geolocalización,
 * iniciar el servidor de streaming para atender a las peticiones de posibles clientes, y
pasar mediante la pulsación
 * de los correspondientes botones a las actividades de CHAT y STREAMING
 */
class MapsActivity : AppCompatActivity(), OnMapReadyCallback, ConnectCheckerRtsp,
View.OnClickListener, SurfaceHolder.Callback {

    private lateinit var mMap: GoogleMap
    var fusedLocationClient: FusedLocationProviderClient? = null
    var locationRequest: LocationRequest? = null
    var callback: LocationCallback? = null
    // Lista de los participantes con los cuales poder comunicarse
    val participantes = mutableListOf<Participante>()

    // Variables propias del usuario
    var miLatitud = ""
    var miLongitud = ""
    var ip = ""
    var miId: String = ""

    // Objeto para comunicaciones UDP
    lateinit var udpComm: Comunicaciones
    // Objeto para comunicaciones RTSP
    private var rtspServerCamera1: RtspServerCamera1? = null
    // Objeto para ventana de información de participante
    lateinit var ventana: CustomInfoWindow
    // Thread para recibir mensajes broadcast
    lateinit var hiloMensajesBroadcast: Thread
```

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    window.addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON)
    setContentView(R.layout.activity_maps)

    // Obtiene el parámetro ID que le pasa la actividad Main
    val objetoIntent: Intent = intent
    miId = objetoIntent.getStringExtra("Id")

    // Se crea un servicio RTSP en el puerto 1935
    rtspServerCamera1 = RtspServerCamera1(surfaceView, this, 1935)
    surfaceView.holder.addCallback(this)

    val mapFragment = supportFragmentManager.findFragmentById(R.id.map) as
SupportMapFragment
    mapFragment.getMapAsync(this)
    fusedLocationClient = FusedLocationProviderClient(this)
    inicializarLocationRequest()
    /**
     * Objeto llamado de forma periodica para el envío de mensajes de geolocalización
     * y para actualizar la posición propia en el mapa
     */
    callback = object : LocationCallback() {

        override fun onLocationResult(locationResult: LocationResult?) {
            super.onLocationResult(locationResult)

            mMap.isMyLocationEnabled = true
            mMap.uiSettings.isMyLocationButtonEnabled = true
            mMap.uiSettings.isZoomControlsEnabled = true

            // Obtiene posición GPS actual
            for (ubicacion in locationResult?.locations!!) { //OBTIENE MI UBICACIÓN ACTUAL
                miLatitud = ubicacion.latitude.toString()
                miLongitud = ubicacion.longitude.toString()
            }
            // Construye el mensaje: ID:latitud:longitud,IP:
            var mensajeBroadcast = String()
            mensajeBroadcast = miId+":"+miLatitud+","+miLongitud+","+ip+":"
            // Envío de mensaje
            udpComm.sendUDP(mensajeBroadcast)
        }
    }

        // Se crea el objeto de comunicaciones UDP
        udpComm = Comunicaciones()

        // Obtiene ip local
        var wifiManager: WifiManager
        wifiManager = this.applicationContext?.getSystemService(Context.WIFI_SERVICE) as
WifiManager
        var iiplocal = wifiManager.connectionInfo.ipAddress
        ip = Formatter.formatIpAddress(iiplocal)
        //Asigna la ip local al atributo del objeto de comunicaciones
        udpComm.ipLocal = ip
        // Asigna como ip remota la dirección broadcast de una red local con máscara /24
        udpComm.RemoteHost = ip.substring(0, ip.lastIndexOf(".")) + ".255"


        // Ejecuta la escucha de mensajes broadcast y de inicio de chat
        hiloMensajesBroadcast = Thread {
            receiveBroadcast()
        }
        hiloMensajesBroadcast.start()

        // Acciones que se ejecutan al pulsar sobre el botón de CHAT
        buttonChat.setOnClickListener {
            // Buscamos al participante e iniciamos chat con él
```

```
            var idChat = ventana.participanteSeleccionado
            for (participante in participantes) {
                if (participante.id == idChat) {
                    udpComm.sendUDP("("+miId+")")
                    var intent = Intent(this@MapsActivity, Chat::class.java)
                    intent.putExtra("ID", idChat)
                    intent.putExtra("IP", participante.ip)
                    intent.putExtra("miId", miId)
                    startActivity(intent)
                }
            }
        }

        // Acciones que se ejecutan al pulsar sobre el botón de STREAMING
        streaming.setOnClickListener {
            // Buscamos al participante e iniciamos el consumo del streaming de él
utilizando VLC
            var idStreaming = ventana.participanteSeleccionado
            for (participante in participantes) {
                if (participante.id == idStreaming) {
                    val uri: Uri = Uri.parse("rtsp://"+participante.ip+":1935")
                    var intent = Intent(Intent.ACTION_VIEW,uri)
                    intent.setPackage("org.videolan.vlc")
                    startActivity(intent)
                }
            }
        }
    }


    /**
     * Función que recibe mensajes UDP y decide si es un mensaje de invitación a inicio de
Chat,
     * si es un mensaje de nuevo participante, si es de actualización de posición o si es
de
     * un participante que deja la aplicación.
     */
    fun receiveBroadcast() {

        while (true) {
            // Recibimos el mensaje UDP
            var recibido = udpComm.receiveUDP(true)

            // Si es un mensaje de CHAT empieza por un caracter '('
            if (recibido.indexOf("(") != -1) {
                // Se muestra mensaje invitando a iniciar Chat
                this@MapsActivity.runOnUiThread(java.lang.Runnable {
                    Toast.makeText(this@MapsActivity, recibido+"quiere iniciar chat",
Toast.LENGTH_LONG).show()
                })
                return
            }
            // Si estamos en este punto NO es un mensaje de CHAT: se parsea el mensaje
recibido: [ID][LAT][LON][IP_RECIBIDA]
            val idRecibido = recibido.substring(0, recibido.indexOf(":"))
            val latRecibido = recibido.substring(recibido.indexOf(":") + 1,
recibido.indexOf(","))
            val lonRecibido = recibido.substring(recibido.indexOf(",") + 1,
recibido.lastIndexOf(","))
            var cadenaAuxiliar = recibido.substring(recibido.lastIndexOf(",") + 1)
            var ipRecibida =cadenaAuxiliar.substring(0,cadenaAuxiliar.indexOf(":"))

            // Buscamos al usuario, en principio no está en la lista
            var encontrado=0
            for (participante in participantes) {
                // Una vez encontrado
```

```
                if (participante.id == idRecibido) {
                    // Si la latitud recibida es 100, no se trata de una actualización de
posición, es una salida de la aplicación
                    if (latRecibido=="100") {
                        participante.remove()
                    } else {
                        // Si lo encontramos actualizamos su posición
                        this@MapsActivity.runOnUiThread(java.lang.Runnable {
                            participante.update(latRecibido, lonRecibido)
                            participante.show(mMap, this)
                        })
                    }
                    encontrado=1
                }
            }

            // Si no lo hemos encontrado, creamos una entrada para el en la lista de
participantes
            // y lo añadimos al mapa como marcador
            if (encontrado==0) {
                var participante = Participante()
                participante.id = idRecibido
                participante.lat = latRecibido
                participante.lon = lonRecibido
                participante.ip = ipRecibida

                this@MapsActivity.runOnUiThread(java.lang.Runnable {
                    participante.show(mMap, this)
                })
                participantes.add(participante)
            }
        }
    }

    /**
     * Configura la petición de ubicación cada 10 segundos
     */
    private fun inicializarLocationRequest() {

        locationRequest = LocationRequest()
        locationRequest?.interval = 1000
        locationRequest?.fastestInterval = 500
        locationRequest?.priority = LocationRequest.PRIORITY_HIGH_ACCURACY
    }

    /**
     * Muestra un mapa de tipo híbrido (satélite y calles)
     */
    override fun onMapReady(googleMap: GoogleMap) { //CREA UN MAPA TIPO HÍBRIDO
(SATÉLITE+CALLES)
        mMap = googleMap
        mMap.mapType = GoogleMap.MAP_TYPE_HYBRID
        obtenerUbicacion()
        ventana = CustomInfoWindow(this)
        mMap.setInfoWindowAdapter(ventana)
    }

    private fun obtenerUbicacion() {
        fusedLocationClient?.requestLocationUpdates(locationRequest, callback, null)
    }

    private fun detenerActualizacionUbicacion() {
        fusedLocationClient?.removeLocationUpdates(callback)
    }


    /**
     * Método ejecutado en el inicio de la Activity
```

```
     */
    override fun onStart() {
        super.onStart()
        obtenerUbicacion()
    }

    /**
     * Si la actividad no está en primer plano detenemos el envío de pkts de
geolocalización
     */
    override fun onPause() {
        super.onPause()
        detenerActualizacionUbicacion()
    }


    override fun onDestroy() {
        super.onDestroy()
        Toast.makeText(this@MapsActivity, "Salida", Toast.LENGTH_SHORT).show()
    }


    /**
     * Método llamado cuando el elemento 'surface' se crea:
     * se comprueba que el servidor está parado y listo para iniciarse.
     * En caso contrario se muestra un mensaje de error.
     */
    override fun surfaceCreated(surfaceHolder: SurfaceHolder) {
        try{
            if (!rtspServerCamera1!!.isStreaming) {
                if (rtspServerCamera1!!.isRecording || rtspServerCamera1!!.prepareAudio()
&& rtspServerCamera1!!.prepareVideo()) {
                    rtspServerCamera1!!.startStream()
                } else {
                    Toast.makeText(this, "Error preparing stream, This device cant do it",
Toast.LENGTH_SHORT)
                        .show()
                }
            }
        } catch (e: Exception) {
            Toast.makeText(this, e.message, Toast.LENGTH_SHORT).show()
        }
    }


    /** Métodos que debemos */
    override fun onNewBitrateRtsp(bitrate: Long) {
    }

    override fun onConnectionSuccessRtsp() {
    }

    override fun onConnectionFailedRtsp(reason: String) {
        runOnUiThread {
            rtspServerCamera1!!.stopStream()
        }
    }

    override fun onDisconnectRtsp() {
    }

    override fun onAuthErrorRtsp() {
    }

    override fun onAuthSuccessRtsp() {
    }
```

```
    override fun onClick(v: View?) {
    }

    override fun surfaceChanged(holder: SurfaceHolder?, format: Int, width: Int, height:
Int) {
    }

    override fun surfaceDestroyed(holder: SurfaceHolder?) {
    }
}
```

*Chat:*

```
//Clase que coloca en el layout los mensajes recibidos
class ChatFromItem(val text: String): Item<GroupieViewHolder>() {

    override fun bind(viewHolder: GroupieViewHolder, position: Int) {
        viewHolder.itemView.textview_from_row.text = text
    }
    override fun getLayout(): Int {
        return R.layout.chat_from_row
    }
}
//Clase que coloca en el layout los mensajes enviados
class ChatToItem(val text: String): Item<GroupieViewHolder>(){

    override fun bind(viewHolder: GroupieViewHolder, position: Int) {
        viewHolder.itemView.textview_to_row.text = text
    }
    override fun getLayout(): Int {
        return R.layout.chat_to_row
    }
}

class Chat : AppCompatActivity() {
    lateinit var comm: Comunicaciones
    lateinit var ip :String
    lateinit var id:String
    lateinit var myid:String

    var adapter = GroupAdapter<GroupieViewHolder>()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_chat)

        val objetoIntent: Intent =intent

        ip = objetoIntent.getStringExtra("IP")
        id = objetoIntent.getStringExtra("ID")
        myid = objetoIntent.getStringExtra("miId")

        comm= Comunicaciones()
        comm.RemoteHost=ip

        comm.sendUDP("("+ip+")")

        recycler_view.adapter = adapter

        comm.RemotePort=6545
        supportActionBar?.title = id //COLOCA EN LA CABECERA DE LA ACTIVIDAD EL NOMBRE DEL
PARTICIPANTE CON EL QUE ESTOY CHATEANDO
```

```
        if (comm.checkSocket()) {  //EJECUTAMOS EL HILO DE ESCUCHA DE MENSAJES
            Thread {
                while (true) {
                    receiveChat()
                }
            }.start()
        }
    }

    fun clickButtonSend(view: View) { //ENVÍA EL MENSAJE ESCRITO EN EL EDITTEXT

        val editText = findViewById<EditText>(R.id.editText)

        var message = editText.text.toString()

        val chatMessage ="$myid: $message"
        adapter.add(ChatToItem(text = chatMessage))

        comm.sendUDP(message)

        editText.setText("")
    }

     fun receiveChat() { //RECIBE EL MENSAJE DEL CHAT Y LO GUARDA EN EL ADAPTER PARA
IMPRIMIRLO EN PANTALLA

         var mensaje_recibido = comm.receiveUDP(true)
                this@Chat.runOnUiThread(java.lang.Runnable {

                    val chatMessage = "$id: $mensaje_recibido"
                    adapter.add(ChatFromItem(text = chatMessage))

                })
    }
}
```

A continuación, se adjuntan las clases auxiliares.

*Participante:*

```
class Participante {
    /**
     * Atributos que identifican a un participante:
     * - identificador
     * - latitud y longitud (coordenadas GPS)
     * - dirección IP
     */
    lateinit var id: String
    lateinit var lon: String
    lateinit var lat: String
    lateinit var ip: String
    // Ventana donde se muestra información del participante
    lateinit var customInfoWindow : CustomInfoWindow
    // Elemento marcador
    lateinit var marcador: Marker

    /**
     * Método llamado cuando se quiere mostrar un participante en el mapa
     */
    open fun show(map: GoogleMap,contexto: Context){
        // establece la posición GPS
        val miPosicion = LatLng(lat.toDouble(), lon.toDouble())
        // crea y añade el marcador al mapa
        marcador=map.addMarker(MarkerOptions().position(miPosicion).title("$id"))
        marcador.showInfoWindow()
    }
```

```
    /**
     * Método llamado para actualizar la posición de un participante
     */
    open fun update(latRecibido:String,lonRecibido:String){
        lat=latRecibido
        lon=lonRecibido
        marcador.remove()
    }

    /**
     * Método llamado para eliminar un participante del mapa
     */
    open fun remove(){
        marcador.remove()
    }
}
```

*CustomInfoWindow:*

```
class CustomInfoWindow(val context: Context) : GoogleMap.InfoWindowAdapter {

    // Identificador del participante seleccionado
    lateinit var participanteSeleccionado: String

    /**
     * Método llamado para mostrar contenido en la ventada del marcador
     */
    override fun getInfoContents(p0: Marker?): View {

        var mInfoView = (context as
Activity).layoutInflater.inflate(R.layout.custom_window, null)
        // Introduce la información que tiene asociada el marcador
        mInfoView.editTextFormInfoWindow.text = p0!!.title
        participanteSeleccionado = p0!!.title
        return mInfoView
    }

    override fun getInfoWindow(p0: Marker?): View? {
        return null
    }
}
```

*Comunicaciones:*

```
class Comunicaciones {
    // Dirección IP destino (puede ser broadcast)
    var RemoteHost =""
    // Puerto destino (por defecto 5554)
    var RemotePort: Int = 5554
    // IP local (origen)
    var ipLocal = ""
    // Dirección IP del paquete recibido
    var ipRecibido = ""

    /**
     * Método para enviar paquetes UDP (envia una cadena recibida como parámetro de
entrada)
     */
    open fun sendUDP(messageStr: String) {

        // Previene problemas definiendo que esta acción tiene que ser una tarea asíncrona
        val policy = StrictMode.ThreadPolicy.Builder().permitAll().build()
        StrictMode.setThreadPolicy(policy)

        try {
            //Crea el socket para el envío del paquete
```

```
            val socket = DatagramSocket()
            socket.broadcast = true
            val sendData = messageStr.toByteArray()
            // Paquete UDP
            val sendPacket = DatagramPacket(sendData, sendData.size,
InetAddress.getByName(RemoteHost), RemotePort)
            // Envía el paquete
            socket.send(sendPacket)

        } catch (e: IOException) {
            println("Error en envio mensaje UDP "+e.message.toString())
        }
    }

    /**
     * Método para comprobar el estado del socket (true si está libre, false si está en
uso)
     */
    open fun checkSocket(): Boolean {
        try {
            var socket = DatagramSocket(RemotePort)
            socket.close()
            return true
        } catch (e: SocketException) {
            return false
        }
        return false
    }

    /**
     * Método para recibir paquetes UDP (devuelve una cadena recibida como el contenido del
paquete)
     */
    open fun receiveUDP(broadcast:Boolean):String {

        val charset = Charsets.UTF_8
        var buffer = ByteArray(2048)
        var socket: DatagramSocket? = null

        try {
            //Mantiene un socket abierto escuchando el tráfico UDP destinado a este puerto
            socket = DatagramSocket(RemotePort)
            socket.broadcast = true

            val packet = DatagramPacket(buffer, buffer.size)
            do {
                socket.receive(packet)
                println(packet.data.toString(charset))
            } while (packet.address.toString().substring(1) == ipLocal)

            ipRecibido = packet.address.toString().substring(1)
            return packet.data.toString(charset)

        } catch (e: Exception) {
            return e.printStackTrace().toString()
        } finally {
            // Cierra el socket
            socket?.close()
        }
        // En caso de error devolvemos cadena vacía
        return ""
    }
}
```

*RTSPserver:*

```
class RtspServer(context: Context, private val connectCheckerRtsp: ConnectCheckerRtsp,
  val port: Int) {

  private val TAG = "RtspServer"
  private lateinit var server: ServerSocket
  val serverIp = getServerIp(context)
  var sps: ByteArray? = null
  var pps: ByteArray? = null
  var vps: ByteArray? = null
  var sampleRate = 32000
  var isStereo = true
  private val clients = mutableListOf<Client>()
  private var thread: Thread? = null

  fun startServer() {
    thread = Thread {
      server = ServerSocket(port)
      while (!Thread.interrupted()) {
        Log.i(TAG, "Server started $serverIp:$port")
        try {
          val client =
            Client(
              server.accept(), serverIp, port, connectCheckerRtsp, sps, pps, vps,
sampleRate,
              isStereo
            )
          client.start()
          clients.add(client)
        } catch (e: SocketException) {
          Log.e(TAG, "Error", e)
          break
        } catch (e: IOException) {
          Log.e(TAG, e.message ?: "")
          continue
        }
      }
      Log.i(TAG, "Server finished")
    }
    thread?.start()
  }

  fun stopServer() {
    clients.forEach { it.stopClient() }
    clients.clear()
    thread?.interrupt()
    try {
      thread?.join(100)
    } catch (e: InterruptedException) {
      thread?.interrupt()
    }
    thread = null
    if (!server.isClosed) server.close()
  }

  fun sendVideo(h264Buffer: ByteBuffer, info: MediaCodec.BufferInfo) {
    clients.forEach {
      if (it.isAlive && it.canSend) {
        it.rtspSender.sendVideoFrame(h264Buffer.duplicate(), info)
      }
    }
  }

  fun sendAudio(aacBuffer: ByteBuffer, info: MediaCodec.BufferInfo) {
    clients.forEach {
      if (it.isAlive && it.canSend) {
        it.rtspSender.sendAudioFrame(aacBuffer.duplicate(), info)
```

```
      }
    }
  }

  fun setVideoInfo(sps: ByteBuffer, pps: ByteBuffer, vps: ByteBuffer?) {
    this.sps = getData(sps)
    this.pps = getData(pps)
    this.vps = getData(vps)  //H264 has no vps so if not null assume H265
  }

  private fun getData(byteBuffer: ByteBuffer?): ByteArray? {
    return if (byteBuffer != null) {
      val bytes = ByteArray(byteBuffer.capacity() - 4)
      byteBuffer.position(4)
      byteBuffer.get(bytes, 0, bytes.size)
      bytes
    } else {
      null
    }
  }

  private fun getServerIp(context: Context): String {
    val wm = context.applicationContext.getSystemService(Context.WIFI_SERVICE) as
WifiManager
    return InetAddress.getByAddress(ByteArray(4) { i ->
      wm.connectionInfo.ipAddress.shr(i * 8).and(255).toByte()
    }).hostAddress
  }

  internal class Client(private val socket: Socket, serverIp: String, serverPort: Int,
    connectCheckerRtsp: ConnectCheckerRtsp, private val sps: ByteArray?,
    private val pps: ByteArray?, private val vps: ByteArray?, private val sampleRate: Int,
    isStereo: Boolean) : Thread() {

    private val TAG = "Client"
    private var cSeq = 0
    private val output = BufferedWriter(OutputStreamWriter(socket.getOutputStream()))
    private val input = BufferedReader(InputStreamReader(socket.getInputStream()))
    val rtspSender = RtspSender(connectCheckerRtsp)
    private val protocol: Protocol = Protocol.UDP
    private val commandsManager =
      ServerCommandManager(
        serverIp,
        serverPort,
        socket.inetAddress.hostAddress
      )
    var canSend = false

    init {
      commandsManager.setIsStereo(isStereo)
    }

    override fun run() {
      super.run()
      Log.i(TAG, "New client ${commandsManager.clientIp}")
      while (!interrupted()) {
        try {
          val request = commandsManager.getRequest(input)
          cSeq = commandsManager.getCSeq(request) //update cSeq
          if (cSeq == -1) { //If cSeq parsed fail send error to client
            output.write(commandsManager.createError(500, cSeq))
            output.flush()
            continue
          }
          val action = request.split("\n")[0]
          Log.i(TAG, request)
```

```
          val response = commandsManager.createResponse(action, request, cSeq)
          Log.i(TAG, response)
          output.write(response)
          output.flush()

          if (action.contains("play", true)) {
            rtspSender.setSocketsInfo(protocol, commandsManager.videoClientPorts,
              commandsManager.audioClientPorts)
            rtspSender.setVideoInfo(sps, pps, vps)
            rtspSender.setAudioInfo(sampleRate)
            rtspSender.setDataStream(socket.getOutputStream(), commandsManager.clientIp)
            rtspSender.setVideoPorts(commandsManager.videoPorts[0],
commandsManager.videoPorts[1])
            rtspSender.setAudioPorts(commandsManager.audioPorts[0],
commandsManager.audioPorts[1])
            rtspSender.start()
            canSend = true
          }
        } catch (e: SocketException) { // Client has left
          Log.e(TAG, "Client disconnected")
          break
        } catch (e: Exception) {
          Log.e(TAG, "Unexpected error")
        }
      }
    }

    fun stopClient() {
      canSend = false
      rtspSender.stop()
      interrupt()
      try {
        join(100)
      } catch (e: InterruptedException) {
        interrupt()
      } finally {
        socket.close()
      }
    }
  }
}
```

*RTSPserverCamera1:*

```
class RtspServerCamera1 : Camera1Base {

  private val rtspServer: RtspServer
  constructor(surfaceView: SurfaceView, connectCheckerRtsp: ConnectCheckerRtsp, port: Int)
: super(
    surfaceView) {
    rtspServer = RtspServer(surfaceView.context, connectCheckerRtsp, port)
  }
  constructor(textureView: TextureView, connectCheckerRtsp: ConnectCheckerRtsp, port: Int)
: super(
    textureView) {
    rtspServer = RtspServer(textureView.context, connectCheckerRtsp, port)
  }

  @RequiresApi(api = Build.VERSION_CODES.JELLY_BEAN_MR2)
  constructor(openGlView: OpenGlView, connectCheckerRtsp: ConnectCheckerRtsp, port: Int) :
super(
    openGlView) {
    rtspServer = RtspServer(openGlView.context, connectCheckerRtsp, port)
  }

  @RequiresApi(api = Build.VERSION_CODES.JELLY_BEAN_MR2)
  constructor(lightOpenGlView: LightOpenGlView, connectCheckerRtsp: ConnectCheckerRtsp,
```

```
    port: Int) : super(lightOpenGlView) {
    rtspServer = RtspServer(lightOpenGlView.context, connectCheckerRtsp, port)
}

@RequiresApi(api = Build.VERSION_CODES.JELLY_BEAN_MR2)
constructor(context: Context, connectCheckerRtsp: ConnectCheckerRtsp, port: Int) : super(
    context) {
    rtspServer = RtspServer(context, connectCheckerRtsp, port)
}

fun setVideoCodec(videoCodec: VideoCodec) {
    videoEncoder.type =
        if (videoCodec == VideoCodec.H265) CodecUtil.H265_MIME else CodecUtil.H264_MIME
}

fun getEndPointConnection(): String = "rtsp://${rtspServer.serverIp}:${rtspServer.port}/"

override fun setAuthorization(user: String, password: String) { //not developed
}

fun startStream() {
    super.startStream("")
}

override fun prepareAudioRtp(isStereo: Boolean, sampleRate: Int) {
    rtspServer.isStereo = isStereo
    rtspServer.sampleRate = sampleRate
}

override fun startStreamRtp(url: String) { //unused
}

override fun stopStreamRtp() {
    rtspServer.stopServer()
}

override fun getAacDataRtp(aacBuffer: ByteBuffer, info: MediaCodec.BufferInfo) {
    rtspServer.sendAudio(aacBuffer, info)
}

override fun onSpsPpsVpsRtp(sps: ByteBuffer, pps: ByteBuffer, vps: ByteBuffer?) {
    val newSps = sps.duplicate()
    val newPps = pps.duplicate()
    val newVps = vps?.duplicate()
    rtspServer.setVideoInfo(newSps, newPps, newVps)
    rtspServer.startServer()
}

override fun getH264DataRtp(h264Buffer: ByteBuffer, info: MediaCodec.BufferInfo) {
    rtspServer.sendVideo(h264Buffer, info)
}

/**
 * Unused functions
 */
@Throws(RuntimeException::class)
override fun resizeCache(newSize: Int) {
}

override fun shouldRetry(reason: String?): Boolean = false

override fun reConnect(delay: Long) {
}

override fun setReTries(reTries: Int) {
}
```

```
  override fun getCacheSize(): Int = 0
  override fun getSentAudioFrames(): Long = 0
  override fun getSentVideoFrames(): Long = 0
  override fun getDroppedAudioFrames(): Long = 0
  override fun getDroppedVideoFrames(): Long = 0
  override fun resetSentAudioFrames() {
  }
  override fun resetSentVideoFrames() {
  }
  override fun resetDroppedAudioFrames() {
  }
  override fun resetDroppedVideoFrames() {
  }
}
```

*ServerCommandManager:*

```
class ServerCommandManager(private val serverIp: String, private val serverPort: Int,
  val clientIp: String?) : CommandsManager() {

  private val TAG = "ServerCommandManager"
  var audioPorts = ArrayList<Int>()
  var videoPorts = ArrayList<Int>()

  fun createResponse(action: String, request: String, cSeq: Int): String {
    return when {
      action.contains("options", true) -> createOptions(cSeq)
      action.contains("describe", true) -> createDescribe(cSeq)
      action.contains("setup", true) -> {
        if (loadPorts(request)) createSetup(cSeq) else createError(500, cSeq)
      }
      action.contains("play", true) -> createPlay(cSeq)
      action.contains("pause", true) -> createPause(cSeq)
      action.contains("teardown", true) -> createTeardown(cSeq)
      else -> createError(400, cSeq)
    }
  }

  private fun loadPorts(request: String): Boolean {
    val ports = ArrayList<Int>()
    val portsMatcher =
        Pattern.compile("client_port=(\\d+)(?:-(\\d+))?",
Pattern.CASE_INSENSITIVE).matcher(request)
    if (portsMatcher.find()) {
      portsMatcher.group(1)?.toInt()?.let { ports.add(it) }
      portsMatcher.group(2)?.toInt()?.let { ports.add(it) }
    } else {
      Log.e(TAG, "UDP ports not found")
      return false
    }
    val trackMatcher = Pattern.compile("trackID=(\\w+)",
Pattern.CASE_INSENSITIVE).matcher(request)
    if (trackMatcher.find()) {
      val track = trackMatcher.group(1)?.toInt()
      if (track == 0) { //audio ports
        audioPorts.clear()
        audioPorts.add(ports[0])
        audioPorts.add(ports[1])
      } else { //video ports
        videoPorts.clear()
        videoPorts.add(ports[0])
        videoPorts.add(ports[1])
      }
    } else {
      Log.e(TAG, "Track id not found")
      return false
    }
```

```
    Log.i(TAG, "Video ports: $videoPorts")
    Log.i(TAG, "Audio ports: $audioPorts")
    return true
  }

  fun getCSeq(request: String): Int {
    val cSeqMatcher =
        Pattern.compile("CSeq\\s*:\\s*(\\d+)", Pattern.CASE_INSENSITIVE).matcher(request)
    return if (cSeqMatcher.find()) {
      cSeqMatcher.group(1)?.toInt() ?: -1
    } else {
      Log.e(TAG, "cSeq not found")
      return -1
    }
  }

  @Throws(IOException::class, IllegalStateException::class, SocketException::class) fun
getRequest(
    input: BufferedReader): String {
    var request = ""
    var line: String? = input.readLine()
    while (line != null && line.length > 3) {
      request += "$line\n"
      line = input.readLine()
    }
    return request
  }

  private fun createStatus(code: Int): String {
    return when (code) {
      200 -> "200 OK"
      400 -> "400 Bad Request"
      401 -> "401 Unauthorized"
      404 -> "404 Not Found"
      500 -> "500 Internal Server Error"
      else -> "500 Internal Server Error"
    }
  }

  fun createError(code: Int, cSeq: Int): String {
    return "RTSP/1.0 ${createStatus(
      code)}\r\n" + "Server: pedroSG94 Server\r\n" + "Cseq: $cSeq\r\n\r\n"
  }

  private fun createHeader(cSeq: Int): String {
    return "RTSP/1.0 ${createStatus(200)}\r\n" + "Server: pedroSG94 Server\r\n" + "Cseq:
$cSeq\r\n"
  }

  private fun createOptions(cSeq: Int): String {
    return createHeader(cSeq) + "Public: DESCRIBE,SETUP,TEARDOWN,PLAY,PAUSE\r\n\r\n"
  }

  private fun createDescribe(cSeq: Int): String {
    val body = createBody()
    return createHeader(
      cSeq) + "Content-Length: ${body.length}\r\n" + "Content-Base:
$serverIp:$serverPort/\r\n" + "Content-Type: application/sdp\r\n\r\n" + body
  }

  private fun createBody(): String {
    val bodyAudio = Body.createAacBody(trackAudio, sampleRate, isStereo)
    val bodyVideo = Body.createH264Body(trackVideo, "Z0KAHtoHgUZA", "aM4NiA==")
    return "v=0\r\n" + "o=- 0 0 IN IP4 $serverIp\r\n" + "s=Unnamed\r\n" + "i=N/A\r\n" +
"c=IN IP4 $clientIp\r\n" + "t=0 0\r\n" + "a=recvonly\r\n" + bodyAudio + bodyVideo + "\r\n"
  }
```

```
  override fun createSetup(cSeq: Int): String {
    return createHeader(
      cSeq) + "Content-Length: 0\r\n" + "Transport:
RTP/AVP/UDP;unicast;destination=$clientIp;client_port=8000-8001;server_port=39000-
35968;ssrc=46a81ad7;mode=play\r\n" + "Session: 1185d20035702ca\r\n" + "Cache-Control: no-
cache\r\n\r\n"
  }

  private fun createPlay(cSeq: Int): String {
    return createHeader(
      cSeq) + "Content-Length: 0\r\n" + "RTP-Info: url=rtsp://$serverIp:$serverPort/\r\n" +
"Session: 1185d20035702ca\r\n\r\n"
  }

  private fun createPause(cSeq: Int): String {
    return createHeader(cSeq) + "Content-Length: 0\r\n\r\n"
  }

  private fun createTeardown(cSeq: Int): String {
    return createHeader(cSeq) + "\r\n"
  }
}
```