UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

DISEÑO DE UNA PRESENTACIÓN RADAR PARA DISPOSITIVOS MÓVILES EN ANDROID

Trabajo final del grado en Ingeniería Aeroespacial

REALIZADO POR

Martínez Pastor, Arturo José

TUTORIZADO POR

Rodas Jordá, Ángel

CURSO ACADÉMICO: 2020/2021

Resumen

El presente documento recoge todo el proceso, desde las primeras fases de diseño hasta la implementación y los resultados, de la creación de la aplicación «ATvision». Este programa está diseñado para monitorizar el tráfico aéreo desde la pantalla de dispositivos móviles Android. Además, se le ha añadido la capacidad de poder realizar grabaciones y, posteriormente, reproducir la circulación de las aeronaves. De este modo, se generan archivos que pueden ser compartidos por algún medio de mensajería para así ser estudiados y analizados en mayor profundidad en otras plataformas.

Así pues, para su correcta comprensión, es necesario comentar inicialmente las técnicas de programación para Java en Android. Para la generación y manipulación de los mapas del proyecto se emplea la librería «osmdroid» y los datos de las distintas aeronaves se adquieren mediante un receptor de señales ADS-B. En el código se generan un total de 17 clases que trabajan y se comunican para ofrecer las distintas funcionalidades. Funcionalidades que, junto al diseño, contribuyen a crear una aplicación útil y atractiva para aficionados y estudiantes de la ingeniería aeronáutica.

Palabras clave: aplicación, Android, antena, radar, monitorización

Resum

El present document recull tot el procés, des de les primeres fases del diseny fins la implementació i els resultats, de la creació de l'aplicació «ATvision». A més, se li ha afegit la capacitat per poder fer enregistraments i, posteriormente, reproduir la circulació de les aeronaus. D'aquesta manera, es generen arxius que poden ser compartits per algun mitjà de messatgeria per a aixina ser estudiats i analitzats en major profunditat en altres plataformes.

Així doncs, per a la seua correcta comprensió, és necessari comentar inicialmente les tècniques de programació per a Java en Android. Per a la generació i manipulació dels mapes del projecte es fa servir la llibreria «osmdroid» i les dades de les diferents aeronaus s'adquireixen mitjançant un receptor de senyals ADS-B. Al codi es generen un total de 17 classes que treballen i es comuniquen per oferir les diferents funcionalitats. Funcionalitats, que junt amb el disseny, contribueixen a crear una aplicación útil i atractiva per als aficionats i estudiants de l'enginyeria aeronàutica.

Paraules clau: aplicació, Android, antena, radar, monitorització

Índice

1. Introducción 1
   1.1. Descripción del proyecto 1
   1.2. Alcance del trabajo 1
   1.3. Motivación 2
   1.4. Objetivos 3
2. Estado actual de la tecnología 5
3. Recursos empleados 9
   3.1. Java 9
   3.2. Android 11
   3.3. Librería «osmdroid» 15
      3.3.1. MapView 16
      3.3.2. OverlayItem 16
      3.3.3. ItemizedIconOverlay<OverlayItem> 16
4. Monitorización de vuelos 17
   4.1. Evolución de la vigilancia del espacio aéreo 17
   4.2. Sistemas de vigilancia del tráfico aéreo 19
      4.2.1. Radar primario de vigilancia (PSR) 19
      4.2.2. Radar secundario de vigilancia (SSR) 19
      4.2.3. Vigilancia dependiente automática por radiodifusión (ADS-B) 20
   4.3. Dispositivo empleado 22
5. Diseño de la aplicación 24
   5.1. Uso 24
   5.2. Actividades 26
      5.2.1. Actividad principal 27
      5.2.2. Actividad de la biblioteca 29
      5.2.3. Actividad del reproductor 30
      5.2.4. Actividad de la fuente 31
   5.3. Clases 31
   5.4. Programación multihilo 33
6. Implementación 36
   6.1. Comunicación con la antena 36
      6.1.1. Conexión y recepción de los datos 37
      6.1.2. Análisis de los datos 38
      6.1.3. Representación 41
   6.2. Emulador 44
      6.2.1 Grabación 45
      6.2.2. Reproducción 47
   6.3. Interacción con el usuario 48
      6.3.1. Solicitud de ubicación 49
      6.3.2. Mensajes de diálogo 50
      6.3.3. Menú contextual 52

7. Conclusión 54
8. Trabajos futuros...55
Referencias 57
Bibliografía...58
ANEXO I – Detalle de las clases e interfaces de la aplicación..................................59

Índice de abreviaturas

ADS-B	*Automatic Dependent Surveillance – Broadcast* o Sistema de Vigilancia Dependiente Automática – Radiodifusión
API	*Application Programming Interface* o Interfaz de Programación de Aplicaciones
ATC	*Air Traffic Control* o Control del Tráfico Aéreo
ATS	*Air Traffic Services* o Servicios de Tránsito Aéreo
CNS/ATM	*Communication Navegation and Surveillance Systems/Air Traffic Management* o sistemas de Comunicaciones, Navegación y Vigilancia/Gestión del Tránsito Aéreo
FANS	*Future Air Navigation Systems* o Sistemas Futuros para la Navegación Aérea
IDE	*Integrated Development Environment* o Entorno de Desarrollo Integrado
IU	Interfaz de Usuario
IP	*Internet Protocol* o Protocolo de Internet
JRE	*Java Runtime Environment*
JVM	*Java Virtual Machine* o Máquina Virtual Java
MLAT	*Multilateración*
OACI	Organización de Aviación Civil Internacional
OOL	*Object Oriented Language* o Lenguaje Orientado a Objetos
OOP	Object Oriented Programming o Programación Orientada a Objetos
OSM	OpenStreetMap
PSR	*Primary Surveillance Radar* o Radar Primario de Vigilancia
SDK	*Software Development Kit* o Kit de Desarrollo de Software
SSR	*Secondary Surveillance Radar* o Radar Secundario de Vigilancia
TCP	*Transmission Control Protocol* o Protocolo de Control de la Transmisión
UDP	*User Datagram Protocol* o Protocolo de Datagramas de Usuario
UML	*Unified Modeling Language* o Lenguaje Unificado de Modelado
WAM	*Wide Area Multilateration* o Multilateración de Área Amplia

Índice de figuras

Figura 1 Representación del programa SACTA 6
Figura 2 Interfaz del programa flightRadar24 para distintas plataformas 7
Figura 3 Ciclo de vida de una actividad en Android 12
Figura 4 Dos hilos ejecutándose en el mismo procesador 14
Figura 5 Ciclo de vida de un fragmento en Android 14
Figura 6 Cambio de estrategia en la vigilancia aérea como motivo de la introducción de los sistemas CNS/ATM 18
Figura 7 Evolución del porcentaje de aeronaves con tecnología ADS-B en la UE.... 21
Figura 8 Imagen de una antena SBS-1 de Kinetic Electronics 22
Figura 9 Digrama de casos de uso de ATVision cuando la antena se encuentra disponible 25
Figura 10 Digrama de casos de uso de ATVision cuando la antena no se encuentra disponible 26
Figura 11 Las cuatro pantallas de la aplicación 26
Figura 12 Estados de la pantalla principal cuando se interactúa con ella 28
Figura 13 Diagrama de flujo de la actividad de MainActivity 28
Figura 14 Diagrama de flujo de la actividad de EmulatorLibraryActivity 29
Figura 15 Diagrama de flujo de la actividad de EmulatorActivity 31
Figura 16 Notación de las relaciones UML 33
Figura 17 Diagrama simplificado de las clases de ATvision 34
Figura 18 *Threads* de la aplicación y los métodos que los ejecutan 35
Figura 19 Diagrama de actividad del procesado de los mensajes que envía la antena y actualización de la base de datos 40
Figura 20 Los tres iconos empleados en la aplicación para representar los aviones 42
Figura 21 Cambio en la representación de los iconos 43
Figura 22 Diagrama de actiViad del algoritmo de selección/deselección de un icono y la muestra de su información 44
Figura 23 Lógica la de creación de grabaciones 46
Figura 24 Mensaje de solicitud del permiso y ubicación dibujada en el mapa........ 49
Figura 25 Diagrama de flujo de la solicitud del permiso de ubicación........ 50
Figura 26 Icono del botón que solicita el permiso de ubicación 50
Figura 27 Mensaje de alerta que informa de la indisponibilidad de la antena........ 51
Figura 28 Mensajes de diálogo que buscan una confirmación del usuario 52
Figura 29 Menú contextual sobre un fichero de la biblioteca 52
Figura 30 Selección de aplicación para compartir el archivo 53

Índice de tablas

Tabla 1 Resumen del presupuesto del proyecto 4
Tabla 2 Comparación de las especificaciones de los distintos tipos de SSR 20
Tabla 3 Campos de datos que se reciben por medio de un socket desde el servidor Basestation 22
Tabla 4 Campos que transmite cada tipo de mensaje del software Basestation a los sockets que se comunican con él 23
Tabla 5 Clases del proyecto y su función 32
Tabla 6 Campos de datos que se reciben de un archivo del emulador de ATvision 45

1. Introducción

En este primer capítulo del trabajo final de grado se da una primera aproximación al proyecto. Así, se explica en qué consiste, dejando claramente definido su concepto y razón de ser. Para ello, es necesario marcar el alcance del proyecto y las tareas que se han realizado. En esta sección se pretende, también, mostrar la utilidad del estudio y los factores que han motivado a completarlo. Se exponen de una forma clara los objetivos que se han seguido y alcanzado. Por último, resulta de gran importancia comentar también la estrategia detrás del proceso y cada uno de sus pasos.

1.1. Descripción del proyecto

En este trabajo se ha dado solución a la idea de crear una aplicación Android capaz de vigilar el tráfico aéreo. Con este fin, se ha construido un programa capaz de vincularse con un servidor remoto que le comparte datos de los vuelos actuales. Esta técnica le proporciona listas de bytes que necesitan procesamiento. Es necesario conocer el significado de cada valor de la lista para así poder descifrar y almacenar la información recibida. Almacenaje el cual se lleva a cabo en una base de datos local de la que posteriormente se solicitan los objetos que contiene. De este modo, pueden visualizarse el conjunto de los mensajes recibidos en un mapa que se muestre en la pantalla. Además, el usuario puede obtener datos de las aeronaves a tiempo real, pues cada uno de los marcadores permite interactuar sobre él para obtener su información.

Más allá de su función principal, el programa también se ha definido como una grabadora, o un emulador, del tráfico aéreo. Con este fin, se le concede la capacidad de generar archivos de texto en los que los datos de las aeronaves se guardan en un orden determinado y conocido que posteriormente puede ser interpretado. El reproductor cumple la misma función que la representación actual de los vuelos, solo que en esta ocasión la información procede de otra fuente y no es la presente. Por último, el cliente también puede interactuar con cada uno de los ficheros, los cuales se muestran en una biblioteca. Manteniendo pulsado sobre alguno de los elementos se despliega un menú de opciones. Se le ofrece la posibilidad de eliminar, renombrar, compartir y ver la fuente de cada una de las grabaciones.

1.2. Alcance del trabajo

Hay ciertas limitaciones que conducen a la obligación de definir un alcance para el trabajo que se pretende llevar a cabo. Entre ellas las más importantes son las limitaciones respectivas a la naturaleza del proyecto, es decir, a los objetivos de un trabajo final de grado. Al ser tal el tipo del trabajo que se está realizando, así en el tercer

artículo de la normativa marco de trabajo fin de grado y fin de máster de la Universidad Politécnica de Valencia se menciona que «deberá estar orientado a la aplicación y evaluación de competencias asociadas al título para el que se realiza» (punto 1); deben «quedar de manifiesto conocimientos, habilidades y competencias adquiridas por el estudiante a lo largo de sus estudios» (punto 3); «el alcance, contenido y nivel de exigencia de los TFG deberá adecuarse a la asignación de ECTS que dicha materia haya recibido en la memoria de verificación» (punto 6). Además, por este último aspecto se han encontrado restricciones temporales, así como, otras relativas a los conocimientos de los que se disponía al comienzo. Todos estos factores han influido a que el trabajo no pudiese abarcar temas que con mucho gusto se habrían estudiado y añadido. Además, justamente la limitación temporal fue la más determinante a la hora de especificar el alcance del proyecto como punto de partida para su desarrollo. Resulta de suma importancia ser consciente de dónde se parte, hacia dónde se parte y del modo en el que se quiere llegar hasta el destino.

En el apartado anterior se ha descrito concisamente el funcionamiento del proyecto, lo cual no fue fácil de definir inicialmente, pues, si bien el objetivo de monitorizar los vuelos sí que estaba claro, cabía la posibilidad de añadir otras capacidades. Entre ellas, se eligió la funcionalidad como grabadora debido a su utilidad. No obstante, se contemplaron otras posibilidades (ver sección 8).

A pesar de que todo lo mencionado hasta el momento ha sido relativo a la aplicación entre manos, no todo ha sido programación. Resultaba de vital importancia recopilar y considerar información sobre la vigilancia del tráfico aéreo (*air traffic surveillance*). Para no cometer ningún fallo en el planteamiento se debía entender la naturaleza de los datos que se analizaban. Por otra parte, dado que los conocimientos de programación en Java de los que se partía eran, sinceramente, escasos y de Android, trágicamente, nulos, una buena parte del trabajo inicial consistió en adquirir los adecuados para poder enfrentar el proyecto desde un punto de vista amplio y objetivo.

1.3. Motivación

Desde un primer momento el trabajo despertaba gran interés en cuanto a que brindaba una oportunidad para profundizar en las técnicas empleadas para la monitorización de vuelos y para crear una propia. Por otra parte, resulta llamativa su portabilidad, ya gran parte de estos programas no están diseñados como una aplicación para dispositivos móviles. También es de interés que se trate de un proyecto creado con recursos presentes en la Universidad (la UPV) y que emplee los mismos principios que otros *softwares* utilizados para el servicio de control del tráfico aéreo (ATC). La característica de poder grabar vuelos puede resultar interesante para los aficionados de la aviación, pues de esta forma se crea un registro sobre el estado del tráfico aéreo en una determinada fecha. Por otra parte, la opción que se ofrece de compartir dichos archivos es de gran interés académico. De ellos se pueden obtener análisis, gráficas y representaciones que la aplicación no permite y que gozan de gran utilidad.

Como ya se ha indicado antes, el proyecto no solamente conllevaba la creación de la aplicación. Aprender sobre la vigilancia del tráfico aéreo, perseguía el fin de adquirir un contexto teórico con el cual entender la motivación de las primeras invenciones en la historia. Además, permite conocer los cambios que se están dando actualmente en estas metodologías y que tienen una importancia enorme sobre la seguridad y eficiencia de los vuelos. Otro aspecto sobre el que este proyecto brinda ocasión es poder aprender, aplicar y resolver un estudio de programación en Java para Android. De este modo se adquieren competencias muy útiles para el futuro ingeniero y conocimientos que abren numerosas puertas a futuros proyectos.

1.4. Objetivos

En esta sección se materializan los objetivos que se han perseguido con la realización del presente trabajo, algunos de los cuales se han ido introduciendo en los apartados anteriores. Así, atendiendo a su descripción, se encuentra uno muy concreto, importante y principal que es desarrollar una aplicación para dispositivos móviles que permita monitorizar el tráfico aéreo. Su capacidad secundaria define otro objetivo, realizar grabaciones del tráfico aéreo, reproducirlas y llevar a cabo acciones sobre ellas por medio de un programa para dispositivos móviles.

Para alcanzar dichas metas, necesariamente se definieron otras secundarias. De este modo, completando el TFG se pretende obtener una visión general de las tecnologías de posicionamiento de aeronaves, adquirir competencias en lenguaje de programación Java, diseñar y realizar un proyecto de aplicación para Android y crear una herramienta real que se pueda emplear como entrenamiento para estudiantes de aeronáutica y atraiga, al mismo tiempo, a los aficionados.

Más allá de los que se han comentado, la naturaleza de este tipo de trabajo conlleva una serie de objetivos más general y extensa y que, en mayor o menor medida, son independientes del tema principal. Así pues, se adquieren competencias para afrontar la creación y desarrollo de un trabajo de ingeniería de cosecha propia y pensar de forma creativa para encontrar soluciones a problemas que aún no se hayan resuelto o buscar nuevos enfoques a proyectos previos que resulten de interés. Por otra parte, la redacción de la memoria muestra que el estudiante es capaz de redactar un trabajo técnico de gran extensión tanto física como a nivel de conceptos, plasmar fielmente todos los pasos del proyecto de una forma estructurada y transmitir de una forma efectiva la información.

1.5. Presupuesto

A continuación, una vez se han definido todos los aspectos relevantes para entender el programa que se ha desarrollado, parece adecuado estimar el presupuesto de este proceso. Para ser lo más fieles posibles con la realidad se parte de la suposición

de que la cantidad de horas dedicadas ha sido la definida en la guía docente del TFG: 270h. Así pues, para aproximar el coste se consideran tres fuentes principales, el del *hardware*, el del *software* y el de la mano de obra del personal involucrado.

Comenzando con los componentes físicos, han sido necesarios los que se listan:

- Ordenador portátil. A través de él se escribe el código y se recopila información de relevancia. Por otra parte, también sirve para resolver consultas y redactar la memoria que aquí se presenta. El modelo empleado es un Acer *Aspire* 7, con un precio intermedio de 750€ y cuya vida útil se redondea a 5 años. Además, se ha hecho uso de él durante la totalidad de las horas de trabajo, lo que implica un 0.6% de su vida útil.
- Conexión a internet por la fibra para estudiantes de Orange con una mensualidad de 30.95€ [1]. Dado que el trabajo se ha extendido durante un período de seis meses, las horas dedicadas equivalen a 6.25% de la atribución total a la conexión a internet en ese tiempo. De este modo, se ha considerado que a lo largo de las 270h se ha empleado la conexión a internet.
- Una antena SBS-1 de Kinetic Avionic con un precio aproximado de 500€ [2]. Este dispositivo, que se describe en la sección 4.3 resultaba necesario para adquirir las señales de los diferentes vuelos.

A continuación, pasamos al análisis de los programas empleados en la creación:

- Android Studio 4.1.1 es el entorno de desarrollo con el que se ha escrito el código. Es de naturaleza gratuita para todos los usuarios.
- Pack de Microsoft 365 personal. En él se encuentran los programas Word, empleado para la redacción de este documento, y PowerPoint, para la elaboración de la presentación del trabajo. La tarifa anual es de 69€ [3], de la cual el porcentaje de uso es 3.12% si se contempla que se ha empleado en la totalidad de las horas de trabajo del TFG. No obstante, solamente ha ocupado una parte de este tiempo, que grosso modo se estima como un 15%.

Por último, para el cálculo del gasto de la mano de obra, se asume que el salario medio de un desarrollador de Android en España sin experiencia es de 20€/h [4]. Así pues, con todos estos datos se puede calcular el presupuesto. La tabla 1 inferior recoge y presenta los cálculos que se han llevado a cabo para hallarlo y el resultado final.

	Coste (€)	**Uso**	**Total (€)**
Ordenador	750	0.6%	4.5
Internet	183.9	6.25%	11.5
Antena	500	100%	500
Android Studio	0	100%	0
Microsoft 365	69	3.12%	2.15
Programador	5400	100%	5400
		Total	**5918.15**

Tabla 1, resumen del presupuesto del proyecto

2. Estado actual de la tecnología

En este apartado se lleva a cabo una tarea muy importante previa a la ejecución del trabajo propiamente dicho. En él se analizan soluciones que han dado otros autores y creadores a objetivos similares a los del proyecto entre manos. Cada uno de estos trabajos tiene su propio enfoque y se dirige a ser empleado con ciertos usos concretos. En esta segunda sección del trabajo se analizan estos enfoques y las características de cada uno de ellos. Se trata de una tarea técnica de investigación que pretende ofrecer una imagen de la situación de la tecnología respectiva a programas de monitorización de vuelos tanto para dispositivos móviles como para computadoras. De este modo se posibilita la creación de una aplicación que parta del conocimiento de trabajos previos en la materia.

Son numerosos los programas que monitorizan el tráfico aéreo. Los hay que son más especializados y los que son más dirigidas al público general. Algunos son profesionales, otros académicos que sirven para el entrenamiento y otros tienen fines recreativos y de divulgación aeronáutica. En lo relativo a las profesionales, son de interés aquellas herramientas que se emplean para el ATC y el ATFM. Los controladores aéreos necesitan disponer de pantallas en las que se muestre el tráfico fielmente para poder tomar decisiones en consecuencia y asegurar un flujo ordenado de las aeronaves. Esta aplicación fue la primera que surgió de esta tecnología para la aviación civil y es la que ha motivado los mayores cambios técnicos. Debido al objetivo que se persigue con este tipo de programas, presentan características que van más allá del alcance del presente proyecto y tienen un nivel mucho más avanzado de tecnificación, pues requieren de datos fieles a la realidad e ininterrumpidos. No obstante, es de gran utilidad analizar algunas de estas herramientas que se emplean actualmente, para así adquirir una panorámica más amplia del contexto. Entre ellas se encuentran:

- **Artas**. [5] Se trata de un sistema de procesamiento de datos de vigilancia creado y ofrecido por EUROCONTROL. Analiza informes de datos de vigilancia de radares clásicos, Modo-S, WAM y ADS y ofrece a sus usuarios la situación del tráfico aéreo a tiempo real con una alta fiabilidad y precisión. Al tratarse de un sistema distribuido que emplea datos ininterrumpidos de diferentes sensores permite aplicar 5NM de separación mínima. Recibe datos de la posición, velocidad… Pero también de la intención de vuelo por medio del *callsign*, tipo de aeronave… No obstante, este programa no ofrece una presentación de los datos que recoge.
- **SACTA**. [6] Las siglas son Sistema Automatizado de Control Aéreo. Es un sistema distribuido por ENAIRE que se encarga de la gestión del tráfico aéreo. Proporciona una serie de funciones entre las que destacan el procesamiento de información radar y de los planes de vuelo, la concesión de alertas a los controladores cuando una aeronave se desvía de su plan y el intercambio de datos entre las aeronaves y el equipo ATC. Además, toda la información

aeronáutica se presenta en la pantalla. Un ejemplo de dicha presentación se puede observar en la figura 1.

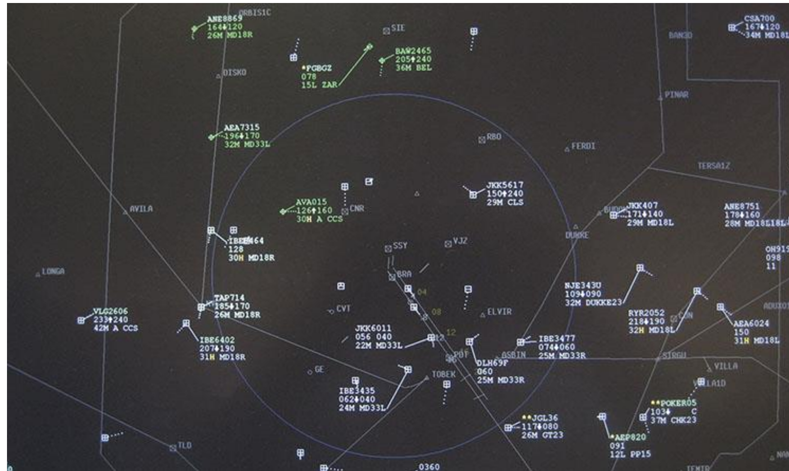

Figura 1, representación del programa SACTA. Fuente: «ENAIRE prueba con éxito una nueva funcionalidad del SACTA en remoto», ITRANSPORT,: Agosto 2020

- **Aurora**. [7] Programa creado por ADACEL® para ofrecer una mejora en la eficiencia del espacio aéreo, así como minimizar el trabajo de los controladores. Incorpora las últimas tecnologías de CNS/ATM y admite señales ADS-B. Actualmente, este programa se emplea en los espacios aéreos de Noruega, Francia, Portugal, Fiji, Nueva Zelanda y Estados Unidos. Se integra con un simulador de tráfico aéreo para entrenar a controladores. Además, integra la información de distintos receptores de datos y la muestra en pantalla.

En este tipo de *software* se basan distintas aplicaciones de aficionados o de enseñanza a otros niveles, como la que se desarrolla en el presente trabajo. Últimamente, estos programas han crecido en popularidad y se han multiplicado en número. A continuación, se presentan los más relevantes y conocidos:

- **FlightRadar24**. [8] Es un servicio de monitorización global de vuelos que proporciona información actualizada de las aeronaves. Combina datos de distintas fuentes como ADS-B, MLAT y radares. Esta información se combina con la de los planes y la situación de los vuelos de la que se dispone a través de las aerolíneas y los aeropuertos. No obstante, su fuente principal de información es ADS-B. FlightRadar24 posee una red de más de 20000 receptores de estas señales alrededor del mundo. MLAT (multilateración) se emplea para monitorizar aquellos aviones que no disponen de esta tecnología ADS-B. Además, si se pierde la visión de un avión del que se conoce su destino porque no hay cubrimiento en su zona, se estima su vuelo hasta dos horas después de la pérdida. Todos estos datos después son mostrados al público general en un mapa como el que se muestra en la figura 2. Se puede acceder a él a través de su página web. Además, disponen de una aplicación para Android y otra para iOS.

- **Virtual Radar Server**. Esta aplicación permite conectarse a un servidor red desde el navegador y muestra el tráfico aéreo dibujado en un mapa. No obstante, es necesario disponer de una radio capaz de recibir transmisiones Modo-S. Fue diseñado para trabajar con los modelos SBS-1 y SBS.3 de *Kinetic Avionic*.
- **Virtual Radar Client (VRC)**. [9] Esta aplicación está diseñada para conectarse con la red virtual de simulación del tráfico aéreo VATSIM. Esta red virtual cuyas siglas significan *Virtual Air Traffic Simulation Network* es una red de «usuarios de simuladores de vuelo que pretenden reproducir del modo más realista posible el tráfico aéreo virtual» (web oficial de VATSIM). Así pues, VRC simula el sistema de radar empleado por los controladores para guiar las aeronaves por sus rutas. Resulta interesante conocer la existencia de este tipo de aplicaciones, no obstante, va más allá del alcance del presente proyecto. Aún así es de interés, pues presenta un origen de los datos que se muestran en el radar que no había sido contemplado aún en ningún ejemplo anterior de los que se han citado.

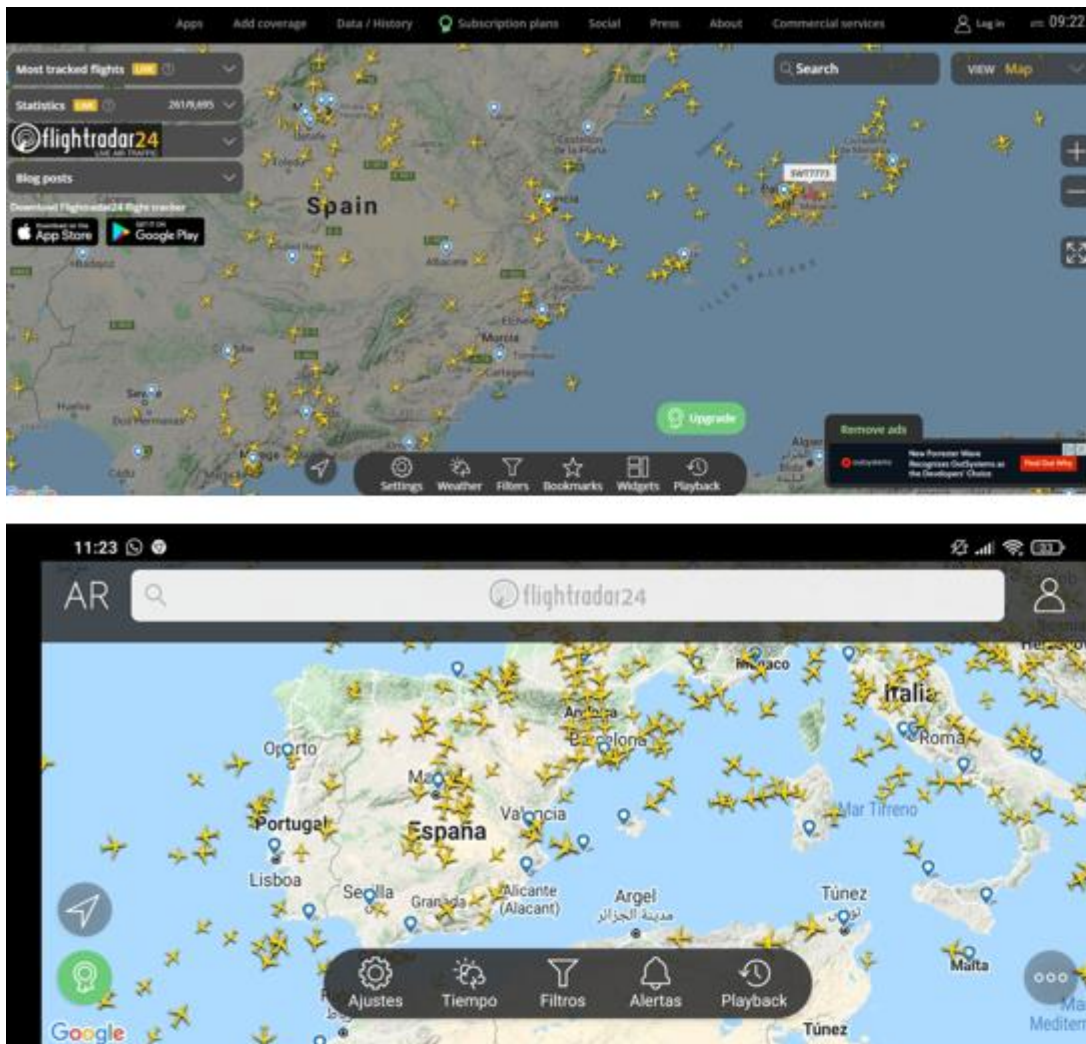

Figura 2, interfaz del programa flightRadar24 para distintas plataformas. La imagen superior es su versión web ejecutada desde un ordenador Windows. La imagen inferior es su app para dispositivos móviles Android

Como se comentaba, hay multitud de software con este fin y se han analizado los que presentan un abanico más amplio de posibilidades debido a lo distinto de la

naturaleza de sus datos y de su objetivo de uso. Otras aplicaciones populares dignas de mención que se asemejan, sobre todo, a flightRadar24 son Plane Finder, FlightAware, ADS-B Exchange o Freedar. Por último, es relevante el análisis de aplicaciones móviles que tengan un comportamiento similar al de la aplicación del proyecto:

- Como se comentaba, **flightRadar24** posee una aplicación móvil, cuya interfaz se observa en la figura 2. A parte de todo lo que ya se ha dicho en el punto anterior sobre este programa, se ha podido observar que la versión Android disponía funcionalidades de gran utilidad, como las que se mencionan. Dispone de un buscador de vuelos por ruta, matrícula, indicativo o aeropuerto. Tiene un modo de realidad avanzada que accede a la cámara del dispositivo y, si se enfoca sobre un avión, ofrece detalles de este. Dispone de detalle a cerca de los vuelos y su historia, los aviones, los aeropuertos… Integra los servicios meteorológicos de los aeropuertos. Incluye una vista tridimensional de realidad virtual del vuelo. En general, se trata de una aplicación muy completa con la que cualquier entusiasta de la aeronáutica puede disfrutar durante horas y ofrece muchas posibilidades más para aquellos que decidan hacerse premium.
- **FlightView free**. Esta aplicación completamente gratuita ofrece la posibilidad de realizar seguimiento de los vuelos que se encuentren haciendo uso de alguno de sus medios de búsqueda. Se puede bien introducir el nombre de la aerolínea, el número de vuelo o los aeropuertos de salida y destino. Muestra una lista con todos los vuelos que cumplen con los criterios de búsqueda y seleccionando uno de ellos, presenta sus detalles y lo dibuja en el mapa. No ofrece una vista de todos los aviones disponibles actualmente en el espacio aéreo, sino que da seguimiento de aquellos que se especifiquen.
- **RadarBox** es similar a FlightRadar24. En esta aplicación también se muestra en un mapa el tráfico aéreo actual a nivel mundial. Como en la otra, existe una versión de pago que aumenta sus posibilidades. Un aspecto muy interesante de la representación de los iconos es que existen diferentes tipos. De esta forma se puede diferenciar el tipo de aeronave según, por ejemplo, la forma en planta de sus alas. Por otra parte, estos iconos se dibujan de diferente tamaño según el del propio avión. También ofrece monitorización de helicópteros y una historia de los vuelos pulsados. Se pueden mostrar en el mapa los receptores de datos de la aeronave de los que hace uso el programa, junto con la cobertura de cada uno de ellos.

3. Recursos empleados

En esta sección del trabajo se describen las herramientas empleadas para construir la aplicación. Como ya se ha mencionado, su diseño es específico para dispositivos Android. Por lo tanto, conviene introducir ciertos conceptos que serán de gran utilidad para la comprensión del resto de la memoria. Se explican desde los términos más elementales hasta los que suponen una cierta especialización. No obstante, todo desde un punto de vista de sentar las bases para abordar la construcción del programa y prestando especial atención en los conocimientos que resultan de una mayor importancia. Cabe mencionar también que el entendimiento de lo que se expone a continuación supuso una parte importante del trabajo inicial, pues se partía de una base más bien escasa de programación para dispositivos móviles. Así pues, lo primero que se debe analizar es la plataforma Java y sus herramientas. A continuación, se describe Android y las características específicas de sus programas. Por último, es de relevancia introducir la librería de desarrollo de Java «osmdroid», la responsable de la generación y manipulación de los mapas presentes en el proyecto.

3.1. Java

Para desarrollar la aplicación, se empleó el entorno de desarrollo integrado (IDE) Android Studio, el oficial de Android. Este programa permite escribir en diferentes lenguajes de programación, de los cuales los dos principales son Java y Kotlin. Tras un análisis de las características y una comparación de los puntos fuertes y débiles de cada uno de ellos, como el que se lleva a cabo en [10], se llegó a la conclusión de que el más conveniente era Java. Esta elección fue tomada atendiendo a criterios tales como los conocimientos previos de desarrollo de software de los que se disponía o los objetivos que se pretendían alcanzar. En este sentido, Java es preferible ya que cuenta con una mayor aplicabilidad y, debido a esta misma popularidad, la comunidad de programadores es mucho mayor. Así, aunque Kotlin presente un mejor rendimiento para el desarrollo en Android, no resultó preferible.

Como bien se ha mencionado, Java es un lenguaje de programación con gran popularidad, de la cual ha gozado desde que fue presentado en 1995. Uno de los aspectos que lo hacen tan valorado es la capacidad de los programas que lo emplean de ejecutarse en plataformas con sistemas operativos tan diversos como Windows, Mac OS, Linux o Solaris. Esta portabilidad es posible gracias a un entorno de ejecución para los programas compilados conocido como *Java Runtime Environment* (JRE). Este entorno de ejecución cuenta con una máquina virtual Java (JVM), que ejecuta las instrucciones de los programas.

Otra de las características que destacan de este lenguaje es su orientación a objetos. Dicho paradigma de programación se distingue del resto por dividir el proyecto en segmentos, conocidos como objetos, que cuentan con una serie de datos y métodos.

Con esta filosofía se consigue una forma de diseñar aplicaciones en la que se optimiza el código, al convertirse en reutilizable y fácil de modificar. En un lenguaje orientado a objetos (OOL), el proceso de escritura comienza con la creación de clases; después estas clases se instancian y se crean los objetos; en último lugar, estos objetos se comunican entre sí. Algunos conceptos básicos de Java que son consecuencia de esta metodología de creación de software son los que se exponen a continuación.

En primer lugar, como ya se ha mencionado, las clases son bloques de código que definen un determinado tipo de objeto. En ellas, por lo tanto, se definen tanto las variables que serán los datos, como sus métodos, también llamados funciones, acciones u operaciones. Toda clase tiene un constructor por medio del cual crear una instancia de ella. Además, se puede emplear uno de los conceptos básicos de la OOP, la encapsulación, cuando se declaran variables y métodos privados, es decir, solo utilizables dentro de la misma clase. Cabe mencionar que para acceder a los datos privados existen unos métodos conocidos como *getters* y *setters*, con los obtener o definir su valor, respectivamente. Por último, es relevante el hecho de que las clases pueden heredar propiedades de otra, implementar una interfaz o sobrescribir métodos de clases que extiendan. Estas últimas propiedades serán explicadas en mayor profundidad posteriormente.

Por otro lado, también existen las variables, utilizadas en un programa Java para almacenar valores. Se definen por un nombre que permite referenciarlas, un tipo y el rango de valores que pueden almacenar. Existen dos tipos, las que referencian datos primitivos y las que lo hacen a objetos. Las primitivas permiten representar valores básicos, tales como números enteros y reales, caracteres y valores booleanos. Por otra parte, los objetos, como ya se ha dicho, representan clases.

Asimismo, se ha mencionado que en Java existen bloques de código conocidos como métodos, los cuales definen funciones que se tienen que realizar. Entre ellos, es de principal importancia el conocido como main(). Este es el que invoca la máquina virtual cuando empieza a ejecutar un programa y el cual a su vez ejecuta llama al resto de las operaciones de la aplicación.

En los lenguajes orientados a objetos, una de las propiedades más importantes que lo definen es que las clases pueden extender o heredar de otras. Si esto tiene lugar, se genera una jerarquía en la que la subclase, o la que hereda, se atribuye los datos y el comportamiento de la superclase. Normalmente, una subclase añade nuevos atributos y métodos que diferencian su comportamiento. Esta propiedad, conocida como herencia, es muy relevante y permite la reutilización del código, ya que se crean clases basadas en otras existentes. Cabe conocer que, en el lenguaje técnico, el hecho de que una clase herede a otra hay veces que se describe como que una desciende a otra.

Siguiendo con las propiedades pilares de la OOP, se señalan otras dos [11]. La primera de ellas, el polimorfismo, permite a un método comportarse de manera diferente en distintas instancias dependiendo del tipo de datos de entrada. La segunda es la conocida como abstracción. Con ella se puede lograr representar las características

principales de una clase sin incluir su comportamiento o significado. Esto permite crear interfaces en Java. Concretamente, una interfaz es una clase completamente abstracta, es decir, sin implementación y en la que solamente se definen métodos abstractos. Por consiguiente, esta herramienta se utiliza para crear subclases que las implementen, comprometiéndose de este modo a implementar todos y cada uno de sus métodos. Esta clase, además, pasará a pertenecer a un nuevo tipo de datos extra, que es el tipo de la interfaz que implementa. Definen, por lo tanto, un patrón de comportamiento específico para un grupo de objetos. Relacionándolo con la herencia, cabe apuntar también que una clase solamente puede extender a otra mientras que puede implementar tantas interfaces como sean necesarias.

Un último concepto que conviene repasar son las colecciones de datos. Collection es una interfaz de Java que permite crear matrices de tamaño variable, en las que se puedan añadir, eliminar y modificar objetos. Los dos tipos de colecciones más utilizados en el desarrollo de la aplicación han sido las clases ArrayList y HashMap. Especial atención se prestará a esta última, ya que, como se mostrará en capítulos posteriores, es la base del análisis de datos que se lleva a cabo en el programa.

3.2. Android

Android es un sistema operativo de código abierto basado en Linux para teléfonos móviles. El hecho de que sea código abierto conlleva, entre otras cuestiones, que cualquier persona con los conocimientos adecuados puede crear nuevas aplicaciones o, incluso, modificar el propio sistema. Por otra parte, también implica que es apto para cualquier tipo de dispositivo móvil y ofrece a los fabricantes la posibilidad basarse en él. Android permite programar, en un entorno de trabajo de Java, aplicaciones sobre una máquina virtual Dalvik (una variación de la máquina de Java con compilación en tiempo de ejecución). Dalvik, junto con las bibliotecas específicas, da a las aplicaciones las funcionalidades de Android [12].

El desarrollo para dispositivos móviles requiere que el programador tenga especial cuidado en determinados aspectos que surgen como consecuencia de las limitaciones del aparato. Se deben tener distintas consideraciones en mente:

- Ser eficiente en cuanto a bucles y cálculos. Optimizar el almacenamiento al máximo.
- Diseñar para pantallas de distintos tamaños y tener en cuenta las más pequeñas.
- Respetar al usuario. No hay que robar el foco a otras aplicaciones y hay que mantener un estilo de interfaz consistente con el sistema y con buena respuesta.
- Intentar que la aplicación se recupere rápidamente al volver a arrancarla y que se mantenga en el último estado en el que estuvo.

Con todo esto, un programador puede lograr crear aplicaciones funcionales en Android tras haberse familiarizado con una serie de conceptos. Los más relevantes de estos conceptos se exponen a continuación.

En primer lugar, los componentes principales de las aplicaciones Android son las actividades. Se puede contar con tantas como sea necesario, donde cada una de ellas representa una pantalla del programa. Consecuentemente, tienen una interfaz de usuario asociada y son entidades independientes. A pesar de esta última característica, todas ellas han de trabajar en conjunto para ofrecer una buena experiencia de uso. Las actividades facilitan las siguientes interacciones entre el sistema y la aplicación:

- Saber qué es en lo que el usuario está centrando su atención para asegurar que el sistema no deja de ejecutar dicho proceso.
- Las actividades paradas son procesos a los que se puede querer volver, por ello se priorizan.
- Además, proveen de un método para que las aplicaciones se comuniquen entre ellas de forma coordinada por el sistema.

Todas las actividades, implementadas como una subclase de AppCompatActivity, tienen que estar declaradas en el archivo de manifiesto (AndroidManifest.xml) del proyecto. En este archivo, también se define cuál será la actividad principal, por la cual la aplicación comenzará a ejecutarse. A diferencia de lo que sucede con los programas en la mayoría de los demás sistemas, en Android el código no comienza con el método principal, sino que responde a distintas fases del ciclo de vida de la actividad principal. Este último concepto introducido es de crucial importancia, toda actividad sigue un ciclo de vida desde que se crea hasta que se destruye. En la figura 3 se observa el ciclo de vida de una actividad y las mencionadas funciones de llamada. Por otro lado, en este mismo archivo de manifiesto se identifican los permisos de usuario que requiere la aplicación, tales como, en el caso de este proyecto, acceso a Internet o a la ubicación del dispositivo. Existen varios tipos de permisos de usuario atendiendo a la protección de los datos. En el ejemplo anterior, el permiso de Internet es de tal naturaleza que no resulta necesaria su consulta al usuario. Sin embargo, la ubicación se solicita y, en caso de ser rechazada la solicitud, ciertas funciones de la aplicación serán invalidadas. También cabe mencionar que como la aplicación empieza a ejecutarse por la actividad principal, es necesario poder comenzar otras actividades. Con este fin existen unos recursos llamados *intents*.

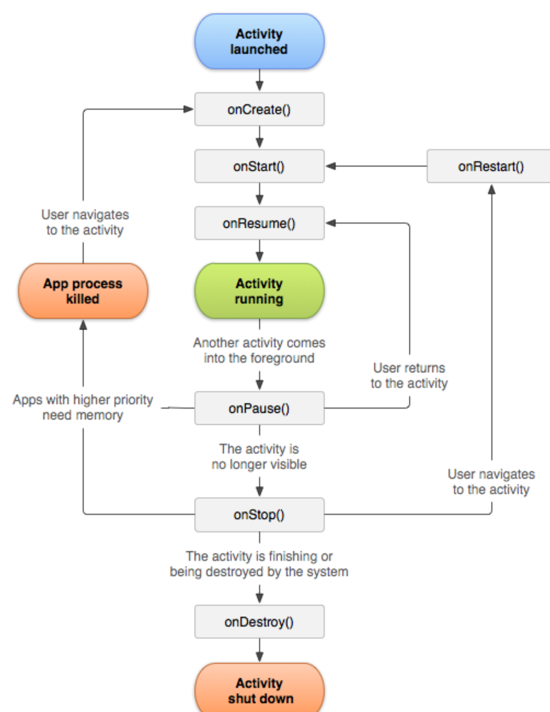

Figura 3, ciclo de vida de una actividad en Android. Fuente de la imagen [13].

Más allá de todo lo visto, una aplicación no es solamente código; requiere recursos independientes de la fuente. Todos ellos se agrupan dentro de la carpeta *res* del proyecto. Destacan los recursos de diseño de las interfaces de usuario (IU), o *layout*. Dichos archivos se escriben en lenguaje XML y definen el aspecto de la pantalla, junto con los distintos elementos que interactúan con el usuario. Cada uno de estos últimos extiende a la clase View, de la que heredan ciertos métodos y atributos comunes. Entre los objetos de vista más relevantes del proyecto están los TextView, Button, ListView o MapView. Otro tipo de recursos dignos de mención son los *values*. En esta carpeta se pueden encontrar archivos *xml* que definen desde los colores o las Strings que la aplicación utiliza, hasta los temas de sus componentes. También existen, entre otros, los recursos de tipo *drawable* o *menu*.

Como bien ya se ha recalcado, al programar para dispositivos móviles hay que tener en cuenta ciertas limitaciones del sistema. Dado que se cuenta con varios procesadores y, sobre todo, debido a la naturaleza del sistema operativo, existe la posibilidad de ejecutar distintas operaciones simultáneamente, para así optimizar el rendimiento de la aplicación. Esto hace que sea indispensable en la gran cantidad de aplicaciones Android la programación multihilo (*multithreading*), en la que procesos se ejecutan asíncronamente. Hay dos razones principales por las que se solicita implementar esta estrategia. La primera de ellas es la ejecución obligatoria de ciertas tareas, como la comunicación TCP o escribir y leer archivos, en segundo plano. La otra es para ejecutar operaciones largas. En ambos casos el objetivo que se persigue es no bloquear la interfaz de usuario y optimizar la usabilidad, es decir, que la aplicación responda rápidamente a las interacciones y que las animaciones sean fluidas.

Desde el punto de vista del desarrollo, un hilo (*thread*) es la ejecución de instrucciones Java que se llevan a cabo secuencialmente. Una aplicación tiene por lo menos un hilo que define el sentido de la ejecución del código y, si no se generan más, una instrucción se ejecutará cuando ya lo hayan hecho todas las anteriores. Al crear otros subprocesos, el código se separa por distintos caminos que se ejecutan concurrentemente. La forma que emplea el procesador para ejecutar dos hilos al mismo tiempo se puede visualizar en la figura 4. En Java, la programación multihilo se puede llevar a cabo de varias maneras, como mediante la clase AsyncTask. La seguida en este proyecto, debido a que presentaba las características necesarias, es la creación de clases que extendiesen a Thread. En el apartado 5.4 se explica la estrategia multihilo seguida en la aplicación y la figura 18 la esquematiza. A pesar de que la programación multihilo mejora el rendimiento, conlleva ciertas complicaciones. Por una parte, hacer el código más complicado, de forma que se tienen que crear estrategias de ejecución de tareas y se generan problemas de consistencia de las variables. Además, hay que tener en cuenta que está prohibido actualizar elementos de diseño de la IU desde otro hilo que no sea el principal.

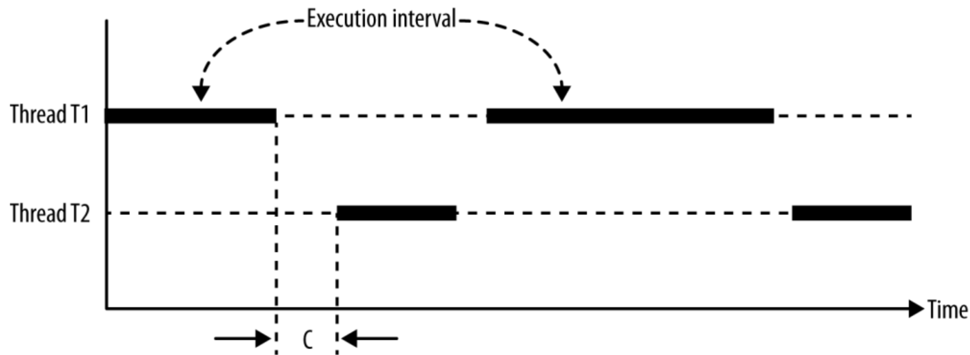

Figura 4, dos hilos ejecutándose en el mismo procesador. Fuente: «Efficient Android Threading»

Un último concepto que conviene introducir, relativo a las estrategias de programación de aplicaciones Android, son los fragmentos. Estas entidades se crean extendiendo a la clase Fragment y funcionan como una «subactividad» con su propia interfaz de usuario. Son una sección modular de una actividad con un ciclo de vida propio (figura 5), sus propios eventos de entrada y que se puede agregar o quitar mientras la actividad se está ejecutando. Estas entidades siempre deben estar alojadas en una actividad, de forma que su ciclo de vida se ve afectado directamente por el de la anfitriona. No ocurre lo mismo al contrario, mientras una actividad se está ejecutando se pueden manipular los distintos fragmentos de forma independiente. Se entiende que la razón por la que fueron introducidos por Android fue admitir diseños de IU más dinámicos y flexibles. Al dividir el diseño en fragmentos se puede modificar su aspecto durante el tiempo de ejecución y conservar estos cambios en una pila de actividades. Dentro de la presente aplicación, se emplean principalmente para crear diálogos de contexto.

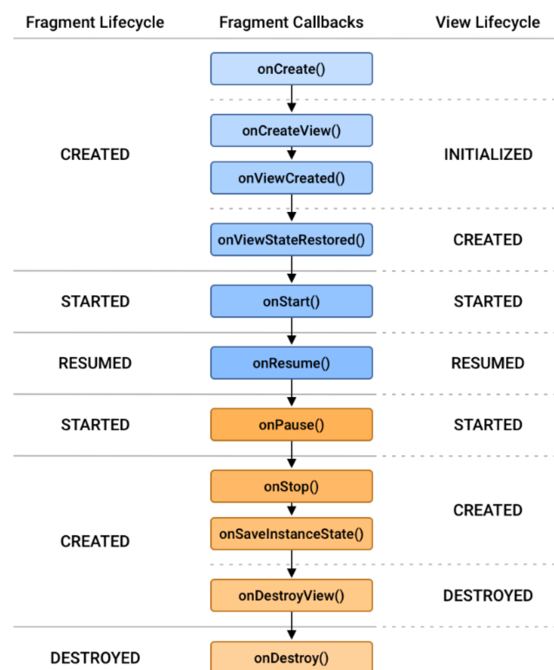

Figura 5, ciclo de vida de un fragmento, sus devoluciones de llamada y su relación con el ciclo de las vistas de dicho fragmento. Fuente de la imagen [13]

3.3. Librería «osmdroid»

«Osmdroid» es una de las librerías adaptadas para Android que genera y trabaja con mapas a partir de la herramienta OpenStreetMap (OSM). OSM es un proyecto de código abierto fundado en 2004 por Steve Coast en el University College London [14]. Surgió con la motivación de distribuir datos geográficos del mundo y crear una comunidad que contribuyese a actualizarlos y mejorarlos para crear un mapa del mundo de libre acceso. Además, existen grupos de desarrolladores de software que participan voluntariamente creando herramientas que permitan poner todos estos datos a disposición de diferentes aplicaciones, plataformas y dispositivos. Se eligió OpenStreetMap como método para generar el mapa principal del proyecto porque, a diferencia de otros mapas como GoogleMaps, no tiene restricciones legales ni técnicas para su uso. Sus mapas se cargan como una serie de mosaicos (*map tiles*), que son típicamente de 256x256 píxeles, de forma que solo se carguen aquellos particulares de la zona concreta en visualización. El *renderizador* más popular en OSM es Mapnik, que es, por otra parte, el que se emplea en la aplicación. Además, para optimizar el procesamiento, los datos representados se distribuyen en una serie de capas superpuestas.

Hay multitud de aplicaciones móviles en Android que utilizan las herramientas puestas a disposición por OSM con objetivos diversos, como se muestra en [15]. Estos programas emplean distintas librerías de desarrollo que ofrecen al programador métodos para poder generar y manipular mapas. Dichas bibliotecas son diversas y sus características, aplicaciones y modo de empleo varían de unas a otras. De entre todas ellas, la elegida fue «osmdroid», pues se trata de una librería cuya API es similar a la de Google Maps V1 y que pretende reemplazar la clase de MapView de Android basándose en OSM. Además, funciona con o sin conexión a Internet y tiene una licencia Apache de uso libre. A pesar de que la información de soporte y ayuda sobre ella es escasa, su página principal de Github ha constituido el principal apoyo para comprender su funcionamiento y resolver dudas. Esta página cuenta con una útil wiki y con un ejemplo de proyecto en el que se muestran posibilidades de uso. Pese a lo escaso de las ayudas, al intentar simular esta librería el comportamiento de la clase MapView de Android, el funcionamiento era similar y para ciertas cuestiones podía seguirse su documentación.

Por defecto, el proyecto no cuenta con esta biblioteca y hay que introducirla manualmente en el archivo que configura el Gradle del proyecto: build.gradle. Gradle [16] es la base del sistema de compilación de Android Studio, una herramienta de automatización de la construcción (*build automation*) de código abierto para la compilación y los procesos asociados. Está diseñado de manera lo suficientemente flexible como para construir casi cualquier tipo de software y funcionar basado en la JVM. Facilita la inclusión de objetos binarios externos o de otros módulos de biblioteca a la compilación en forma de dependencias. Así, para agregar una dependencia al proyecto, se tiene que especificar como *implementation* en el bloque *dependencies* del

archivo build.gradle.xml. A continuación, se explican brevemente las tres clases de esta biblioteca que mayor relevancia han tenido en el proyecto.

3.3.1. MapView

Como ya se ha explicado, el MapView de osmdroid se basa en la clase del SDK de Maps para Android. Desciende de View e implementa dos interfaces. La primera de ellas IMapView le ofrece la funcionalidad como mapa y define métodos para obtener tanto su controlador como las proyecciones. La otra, MultiTouchObjectCanvas, le da cualidades *multitouch*, lo que permite hacer *zoom* pinchando con los dedos o moverse por el mapa arrastrando. Así pues, esta clase es un tipo de vista que muestra un mapa, se puede añadir a la interfaz de una actividad, y, como toda vista, permite la interacción con el usuario. Este mapa se carga desde una fuente de mosaicos (*tiles source*). Entre sus métodos destacan:

- getOverlays(), devuelve una lista con todas las capas del mapa y permite modificarla añadiendo nuevas o cambiando o eliminando las ya existentes.
- invalidate(), invalida toda la vista y la vuelve a dibujar. En la práctica, este método ha sido de gran utilidad para actualizar el mapa después de algún cambio en sus capas.
- getController(), devuelve el controlador del mapa. Este controlador ha sido empleado en el proyecto para centrar el mapa en Valencia al cargarse y con un *zoom* determinado.

3.3.2 OverlayItem

Esta clase describe un punto geográfico con un título y una descripción. Además, se le puede asociar un icono, que será dibujado en la latitud y longitud que se especifique del mapa. El método setMarker() permite definir dicho icono de forma programática, así como cambiarlo durante la ejecución.

3.3.3. ItemizedIconOverlay<OverlayItem>

Crea una capa del mapa a partir de una serie de objetos que extiendan a OverlayItem. Este es el modo por el cual, aunque ya se desarrollará más adelante, se dibujan los iconos de los aviones. Uno de los aspectos que resulta especialmente adecuado de esta forma de dibujar los marcadores es que la clase admite interacción con cada uno de los iconos que dibuja. Así, define un método de llamada de escucha que se ejecuta si se pulsa sobre alguno de ellos. Además, incluye la útil función getItem(), con la que obtener el OverlayItem que se especifique del conjunto de todos los dibujados.

4. Monitorización de vuelos

En este capítulo se pretende ofrecer un contexto sobre la importancia del trabajo que se realiza en la aplicación aplicado en otros ámbitos, como el control del tráfico aéreo. De este modo, se ofrece una visión general sobre la monitorización de vuelos para la vigilancia. Por otra parte, lo que también que se persigue en esta sección es describir las diferentes tecnologías que posibilitan dicha monitorización. Se comenta su historia y evolución y el principio por el que cada una de ellas funciona. Así, finalmente se comenta y explica el método mediante el cual se obtienen los datos para la aplicación. Se justifica por qué resulta adecuado y se introducen los que se recogen y el método por el cual se envían.

4.1. Evolución de la vigilancia del tráfico aéreo

La vigilancia del tráfico aéreo (*air traffic surveillance*) tiene diversas aplicaciones, las cuales han ido creciendo con el paso del tiempo. Cada una de ellas persigue sus propios objetivos y tiene unas necesidades particulares, sobre todo en lo relativo a la precisión de los datos. Por ello, las técnicas empleadas para conocer la posición de las aeronaves varían de unos fines a otros. La aplicación principal y que motiva el desarrollo de nuevas tecnologías es el control del tráfico (ATC) con el objetivo de generar una circulación segura en el espacio aéreo. En la gestión segura y efectiva del tráfico, la monitorización es crucial, pues permite asegurar las separaciones que optimicen el uso de este espacio. Con esta meta, en el decenio de 1980, debido al rápido aumento de las aerolíneas y la cantidad de aeronaves en vuelo, el Consejo de la OACI se planteó el crecimiento sostenido de la aviación civil internacional. Así, como se afirma en la presentación «Evolución ATM» de la CGNA del *Departamento do Controle do Espaço Aéreo de Brasil* y publicada por la OACI:

> Teniendo en cuenta las nuevas tecnologías, determinó que era necesario realizar una evaluación completa y un análisis de los procedimientos y tecnologías al servicio de la aviación civil. En ese momento, hubo un reconocimiento general de que el enfoque que se estaba aplicando con respecto al suministro de servicios de tránsito aéreo (ATS) y al sistema de navegación aérea estaba limitando el crecimiento continuo de la aviación y las mejoras de la seguridad operacional, eficiencia y regularidad de los vuelos.

Siguiendo esta idea, en 1991 la OACI apoyó el concepto de las FANS, un comité especial creado ocho años antes sobre sistemas de navegación aérea del futuro. Después de ser apoyado por el consejo, recibió el nombre de sistemas de comunicaciones, navegación y vigilancia/gestión del tránsito aéreo (CNS/ATM). En su implementación, se comparten ciertas características entre las que destacan la combinación de recursos satelitales y terrestres, una cobertura a nivel global, el empleo de enlaces de datos aire/tierra y de tecnologías digitales y un cierto nivel de

automatización. Por ejemplo, en lo respectivo a la vigilancia aérea, con la introducción de este concepto se pretendía comprender el cambio de estrategia que se puede observar en la figura 6.

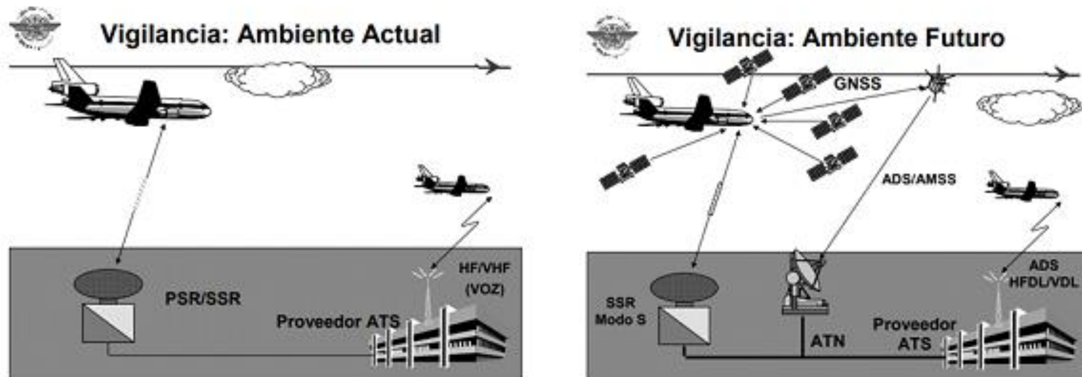

Figura 6, cambio de estrategia en la vigilancia aérea como motivo de la introducción de los sistemas CNS/ATM Figura extraída de «sistemas CNS/ATM de la OACI. Panorámica general» [presentación], https://www.icao.int

Todo lo anterior resulta de gran interés para el desarrollo del proyecto, pues el objetivo que se persigue con la aplicación no es otro que la creación de un método de vigilancia y monitorización aérea, aunque, a diferencia de lo expuesto, en este caso los fines son didácticos y lúdicos. Así pues, cuando se entiende el origen y funcionamiento de estas tecnologías, se consigue conocer la naturaleza de los datos con los que trabaja el programa, lo que brinda de una comprensión de las fases del proyecto.

Uno de los enfoques principales del desarrollo que se muestra en la figura 6 anterior es un abandono de la vigilancia activa hacia una vigilancia pasiva basada en un sistema en el que la aeronave es responsable de proveer los datos de necesarios. En la siguiente sección, se mencionan los mayores beneficios que acarrea dicha transición, siempre haciendo hincapié en las implicaciones en cuanto a seguridad [17].

Los métodos de vigilancia y monitorización del tráfico aéreo empleados hoy en día comprenden una gran variedad de estrategias. Desde radares de alta precisión hasta métodos procedurales de informes verbales de la posición. A continuación, se listan las tecnologías más predominantes y con más importancia, clasificadas atendiendo al tipo de vigilancia en el que cada una se basa:

- Vigilancia independiente
    - PSR (radar primario de vigilancia)
- Vigilancia independiente en colaboración
    - SSR (radar secundario de vigilancia)
        - SSR convencional
        - SSR mono-impulso
        - Modo S SSR
- Vigilancia dependiente automática (ADS)
    - ADS-C (contrato), también conocida como ADS-A (dirigida)
    - ADS-B (radiodifusión)

4.2. Sistemas de vigilancia del tráfico aéreo

En el apartado anterior, se ha mencionado la existencia de distintas técnicas que permiten realizar la actividad de vigilancia del espacio aéreo. Estas herramientas se han ido desarrollando con el paso de los años y conforme incrementaban las solicitaciones de control debido al aumento de la densidad del tráfico. Así, se ha mejorado la exactitud y la eficiencia de los datos obtenidos en un proceso que, aún hoy, sigue evolucionando. A continuación, se definen y explican las tecnologías más relevantes para la captación de información y la monitorización de aeronaves.

4.2.1 Radar primario de vigilancia (PSR)

El PSR es el tipo más antiguo de radar y no necesita de ningún equipo a bordo de la aeronave para funcionar. Su antena rotatoria determina el azimut y rango del vehículo basándose en el reflejo de las radiofrecuencias sobre el fuselaje. Tiene una principal ventaja en cuanto a que no depende del avión [17]. Entre sus limitaciones, además del elevado coste de instalación y mantenimiento, se encuentra el hecho de que no es capaz de identificar la aeronave ni de determinar su altitud.

4.2.2 Radar secundario de vigilancia (SSR)

Se introdujo para superar las limitaciones del radar primario. Ya no es un sistema completamente independiente, ya que requiere equipamiento en la aeronave para que pueda ser monitorizada. No obstante, la función de vigilancia aún es llevada a cabo por el radar. Es por ello por lo que este tipo de vigilancia se conoce con el nombre de independiente en colaboración. En un SSR una antena rotatoria envía interrogaciones al transpondedor colocado en el avión. Este responde a la estación en tierra en un tiempo controlado para que el radar pueda calcular la distancia a la que se encuentra la aeronave a partir del intervalo de tiempo que tarda en recibir la respuesta.

Hay distintos tipos de SSR, entre los que el más común es el Modo A-C. En esta variedad el transpondedor envía un par de datos: el código de identificación (código Modo A) y la altitud. Este método de vigilancia tiene una limitación muy importante, conocida como *garble* o mutilación, que ocurre en áreas con mucha densidad de aviones y en las que algunas aeronaves se encuentran a la sombra de otras. También puede haber otros problemas como, por ejemplo, la congestión de la banda de frecuencias o el hecho de que para un espacio aéreo solamente haya 4096 códigos de identificación disponibles. La limitación de *grable* es superada por el SSR monoimpulso, y en mayor medida por el modo-S. El método al que recurren para hacerlo es el envío de interrogaciones selectivas a las que solamente responde un único avión cada vez. El SSR de este modo interroga a 1030MHz y utilizando modulación por desplazamiento de fase, como se indica en [18]. Este método de modulación permite aumentar la eficiencia y la cantidad de las interrogaciones. Una vez la interrogación llega a la aeronave, la verifica y responde a ella empleando una transmisión de 1090MHz con modulación por posición de pulso. Esta tecnología permite que este tipo radar secundario presente numerosas

ventajas frente a los otros dos. Además, complementa y convive con el modo AC, pues funciona en las mismas frecuencias. En la tabla 2 se comparan los distintos tipos de SSR.

Operaciones	SSR convencional	SSR monoimpulso	SSR modo-S
Respuestas por barrido	20 - 30	4 - 8	1
Precisión de la distancia	230 m	13 m	7 m
Precisión de la derrota	0.08º	0.04º	0.04º
Resolución de la altura	100 ft	100 ft	25 ft
Resistencia a mutilación	Deficiente	Bueno	Mejor
Capacidad de datos (enlace ascendente)	0	0	56 – 1280 bitios
Capacidad de datos (enlace desdendente)	23 bitios	23 bitios	56 – 1280 bitios
Cantidad de claves / direcciones disponibles	4096	4096	> 16 millones

Tabla 2, comparación de las especificaciones de los distintos tipos de SSR

Estos métodos de vigilancia del espacio aéreo presentan varias limitaciones. Principalmente, es una gran desventaja el hecho de que no se pueda mostrar información acerca de la intención de la aeronave. Este es probablemente, junto con la dificultad de proporcionar datos fiables, precisos e ininterrumpidos, el mayor obstáculo para proporcionar un uso efectivo del espacio aéreo que permita acortar las distancias de separación entre los vuelos.

La introducción del Modo-S supone un gran paso adelante al permitir enviar información más allá del identificador y la altitud de la aeronave. Se obtienen datos tales como la velocidad y la derrota que se pueden emplear para inferir el propósito del vuelo. No obstante, aún no resultan idóneos la precisión, el rango y el coste de estos dispositivos.

4.2.3. Vigilancia dependiente automática por radiodifusión (ADS-B)

Una alternativa a la vigilancia independiente activa es la pasiva dependiente [19], en la que no son necesarios los interrogadores. Con esta estrategia la información surge enteramente de la aeronave que la proporciona. Este concepto se conoce también como *Automatic Dependent Surveillance – Broadcast* (ADS-B) o, en castellano, sistema de vigilancia dependiente automática - radiodifusión. El concepto básico es que cada aeronave (aunque también sirve para vehículos terrestres o estaciones fijas) envíe, en intervalos regulares, un mensaje que contenga ciertos campos de información. Entre estos datos que se mandan hay tales como la identificación, la posición, la velocidad o la intención del vehículo. La información de posición puede ser utilizada para simular una vigilancia tipo radar para uso del ATC o, por otras aeronaves, para visualizar el tránsito con fines de anticolisión. El envío se realiza a todos los oyentes a través de un modo de radiodifusión a intervalos especificados. En el otro extremo, existen sistemas capaces de recoger estos mensajes para construir una imagen que monitorice el tráfico aéreo basándose en la posición relativa respecto de su posición.

Este método presenta una ventaja inmediata en cuanto a que su poca complejidad permite reducir el coste del equipo receptor. De esta manera se podría aumentar la cobertura y redundancia, lo que beneficiaría principalmente en áreas remotas donde resulta muy difícil implementar una infraestructura de radares. Por otra parte, ADS-B también ofrece un método mediante el que las aeronaves pueden enviar información acerca de su intención de vuelo. En cuanto a las desventajas, la vigilancia pasiva presenta una muy importante que es, precisamente, esta falta de independencia del sistema de navegación de la aeronave.

Emitir mensajes para este tipo de monitorización de vuelos solicita que se adapten ciertas tecnologías en las aeronaves. Ya se ha redactado normativa aplicable a que ha llevado a un rápido desarrollo del ADS-B, tal y como se puede observar en la figura 7. En especial, la Normativa de Implementación de la Comisión Europea (EU) no. 1207/2011 establece la incorporación y operación obligatoria de vigilancia elemental modo-S para las aeronaves de tráfico general con vuelo instrumental (IFR/GAT). También se impone la implementación y operación de vigilancia avanzada modo-S y *squitters* extendidos de ADS-B a 1090MHz para aeronaves de ciertas características. El término *squitter* hace referencia a las transmisiones automáticas y periódicas que envía un transpondedor modo-S. La aplicabilidad de esta norma tiene fecha de junio de 2020.

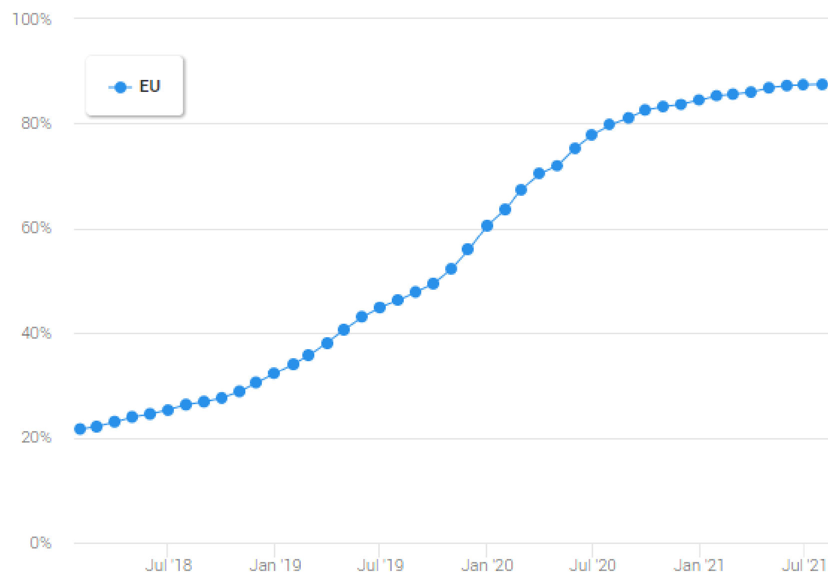

Figura 7, evolución del porcentaje de aeronaves con tecnología ADS-B v2 en la Unión Europea. Fuente de la gráfica: «automatic dependent surveillance – broadcast airborne equipage monitoring. Monitoring overview», Eurocontrol: recurso online [https://www.eurocontrol.int/service/adsb-equipage], accedido el 28/08/2021

4.3. Dispositivo empleado

Después de todo lo visto, lo más conveniente para recibir los datos con los que monitorizar aviones mediante la aplicación resultaría disponer de un receptor de ADS-B. Justamente, el edificio de la Escuela Técnica Superior de Ingeniería del Diseño de la Universidad Politécnica de Valencia dispone de un dispositivo que capta este tipo de señales procedentes de las aeronaves. Además, las procesa y envía la información codificada, atendiendo a una forma específica, a disposición de cualquier cliente que se conecte al servidor. Concretamente, se trata de un receptor de *Kinetic Aviation®* de la familia SBS. Se trata de un producto revolucionario que, con una combinación de hardware y software, permite a los entusiastas y profesionales de la aviación monitorizar los cielos. Con él, se puede obtener una muestra de todos los aviones equipados con Modo-S/ADS-B alrededor de la antena. Esto se realiza por medio de la descodificación de las señales de 1090MHz de sus transpondedores. En la figura 8 se puede observar una imagen de una antena de dicha familia.

Figura 8, imagen de una antena SBS-1 de Kinetic Electronics. Fuente de la imagen: shop.pilotwarehouse.co.uk

Por lo tanto, esta antena recoge los datos que automáticamente envían las distintas aeronaves. No obstante, estos no resultan suficientes, ya que los aviones no transmiten tantos datos como nos interesan debido a que no tienen la necesidad. Aquí entra en juego el software del propio dispositivo, BaseStation [20], que se encarga de acceder a bases de datos y completar la información a partir del código S hexadecimal de registro de cada avión. Como ya se ha comentado, los datos que se recogen pueden ser accedidos por algún cliente que los solicite. Así, cualquier dispositivo móvil puede emplear la API de socket, como se indicó en la sección 3.2 del proyecto, y recibir los datos. Basestation ordena la información de las aeronaves según se muestra en la tabla 3.

Campo 1	Tipo de mensaje, (MSG, STA, ID, SEL o CLK)
Campo 2	Tipo de transmisión, los subtipos de MSG
Campo 3	ID de la sesión, el número de código de la sesión de la base de datos
Campo 4	ID de la aeronave, el número de código de la aeronave en la base de datos
Campo 5	HexIdent, código hexadecimal de identificación de la aeronave en modo-S
Campo 6	FlightID, número de registro del vuelo
Campo 7	Date generated, la fecha en la que se genera el mensaje
Campo 8	Time generated, el momento en el que se genera el mensaje
Campo 9	Date logged, la fecha en el que se envía el mensaje
Campo 10	Time logged, el momento en el que se envía el mensaje
Campo 11	Callsign, un número de ocho dígitos que identifica la aeronave.

Campo 12	Alt, altitud a la que se encuentra. Relativa a 10132.5 mb, no es la altitud AMSL
Campo 13	GroundSpeed, velocidad de avance
Campo 14	Track, derrota. No confundir con el rumbo. Es el ángulo de la trayectoria que sigue la aeronave.
Campo 15	Lat, latitud a la que se encuentra. Este, positivo. Oeste, negativo
Campo 16	Lon, longitud a la que se encuentra. Norte, positivo. Sur, negativo
Campo 17	VerticalRate, la velocidad vertical
Campo 18	Squawk, código del transpondedor modo-A
Campo 19	Alert, bandera que indica si ha cambiado el Squawk
Campo 20	Emergency, bandera que indica que hay código de emergencia
Campo 21	SPI, bandera que indica que el identificador del transpondedor se ha activado
Campo 22	IsOnGround, bandera que indica que el mando de tierra está activo

Tabla 3, campos de datos que se reciben por medio de un socket desde el servidor Basestation

Además, no siempre se envían todos los datos, sino que Basestation dispone de distintos tipos predeterminados de mensajes. Cada uno de ellos pone a disposición del receptor unos datos determinados de los de la tabla 3. Para poder expresar qué datos transmite cada tipo de señal se ha generado la siguiente tabla 4. Es muy importante por lo tanto identificar el tipo de comunicación para así poder distinguir qué valores son los que llegan. La importancia de estas tablas, por lo tanto, tiene que ver con el análisis de los datos en la aplicación, por lo que se recuperan en la sección 6.1.2 del trabajo.

Tipo de mensaje	Campos que genera
MSG 1	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
MSG 2	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 22
MSG 3	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 16, 19, 20, 21, 22
MSG 4	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13, 14, 17
MSG 5	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 19, 21, 22
MSG 6	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 18, 19, 20, 21, 22
MSG 7	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 22
MSG 8	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 22
SEL	1, 3, 4, 5, 6, 7, 8, 9, 10, 11
ID	1, 3, 4, 5, 6, 7, 8, 9, 10, 11
AIR	1, 3, 4, 5, 6, 7, 8, 9, 10
STA	1, 3, 4, 5, 6, 7, 8, 9, 10
CLK	1, 3, 7, 8, 9, 10

Tabla 4, campos que transmite cada tipo de mensaje del software Basestation a los sockets que se comunican con él

5. Diseño de la aplicación

En este apartado se ofrece una visión de la forma en la que la aplicación ha sido creada. En ingeniería, el diseño implica la primera parte de un proyecto, en la que se define el proceso para crearlo, la forma que tendrá y sus distintos elementos. La etapa de diseño de un producto tiene varias fases. Grosso modo podemos diferenciar tres: un diseño conceptual, uno preliminar y otro detallado. El primero de ellos abarcaba la definición del alcance y los objetivos del proyecto, en él se le daba forma a la aplicación y se definía el sentido hacia el que se iban a dirigir los esfuerzos. En el diseño conceptual se entra más en materia. Se definen los métodos en que la aplicación interacciona con el usuario, los distintos casos de uso y actividades que la conformarán. Por último, se detalla el diseño del programa. En esta fase se especifican las distintas clases que lo conforman y algunos de sus métodos. Se define y acota la función de cada uno de estos métodos más importantes y como interaccionan todos ellos para ofrecer una buena experiencia de uso al cliente.

En esta sección también se han empleado distintas gráficas que son norma en los proyectos de desarrollo de software. Estas ayudan a plasmar relaciones y criterios de diseño y resultan de gran utilidad para la correcta y rápida comprensión de la estructura del código.

5.1. Uso

Para comenzar, se da una visión sobre la forma de utilizar la aplicación atendiendo a sus capacidades. En este apartado, la explicación se realiza de forma superficial, a nivel de usuario, sin entrar en detalles a cerca del código o la naturaleza de las respuestas. Como ya se ha introducido a lo largo del proyecto y se concretó en los objetivos (apartado 1.4), las funciones básicas del programa son mostrar la circulación actual de aeronaves y grabar para, posteriormente, reproducir el tráfico aéreo. Así, en una primera instancia, sin conocer aún la naturaleza del código ni el aspecto de la aplicación, se puede diseñar un diagrama de casos de uso donde se esquematice el flujo del trabajo, las funciones de la aplicación y la interacción con el usuario. Dicho diagrama se muestra en la figura 9 de debajo.

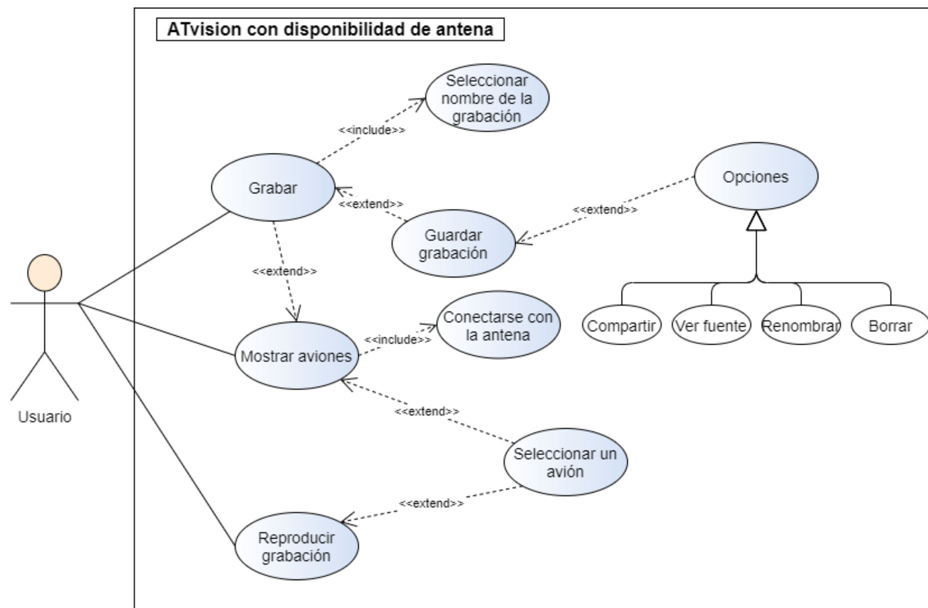

Figura 9, digrama de casos de uso de ATVision cuando la antena se encuentra disponible

En este diagrama se observan las acciones fundamentales de la aplicación, así como los procesos más importantes que se llevan a cabo. Lo más interesante de la figura es que pueden verse las relaciones entre las diferentes acciones. Algunas de ellas tienen lugar automáticamente cuando se realiza la asociada. Estas se mencionan con el nombre «*include*». De esta manera, siempre que se muestren los aviones se tendrá que establecer una conexión con la antena. Al igual, cuando se realiza una grabación, se selecciona su nombre. Otras acciones, por el contrario, se ofrecen al cliente cuando tiene lugar la anterior, pero no se realizan sistemáticamente. Este otro tipo de relaciones son las «*extend*». En este sentido, siempre que se están mostrando aviones o reproduciendo una grabación se pueden seleccionar los iconos que aparecen en pantalla. Además, cuando se hace una grabación, se puede guardar y posteriormente realizar una serie de opciones sobre el archivo. Un aspecto que no se ha comentado de la figura 9 es en lo respectivo al título en el diagrama. En él se lee «con disponibilidad de antena». Esto se debe a que se puede dar la ocasión en la que sea imposible establecer una conexión con el receptor. Entonces, se espera que a pesar de ello la aplicación siga siendo funcional, por lo que resulta necesario considerar dicho caso también.

La figura 10 muestra la forma en la que el programa funciona para resolver dicho problema. En este caso la antena no se encuentra disponible, por lo que cuando se intentan mostrar los aviones salta una alerta que lo avisa. Debido a la imposibilidad de enseñar el tráfico aéreo, tampoco está hábil la función de grabarlo. No obstante, la capacidad de la aplicación como biblioteca y reproductor permanece intacta, ya que en ella no influye el hecho de que que se haya establecido una conexión con el servidor.

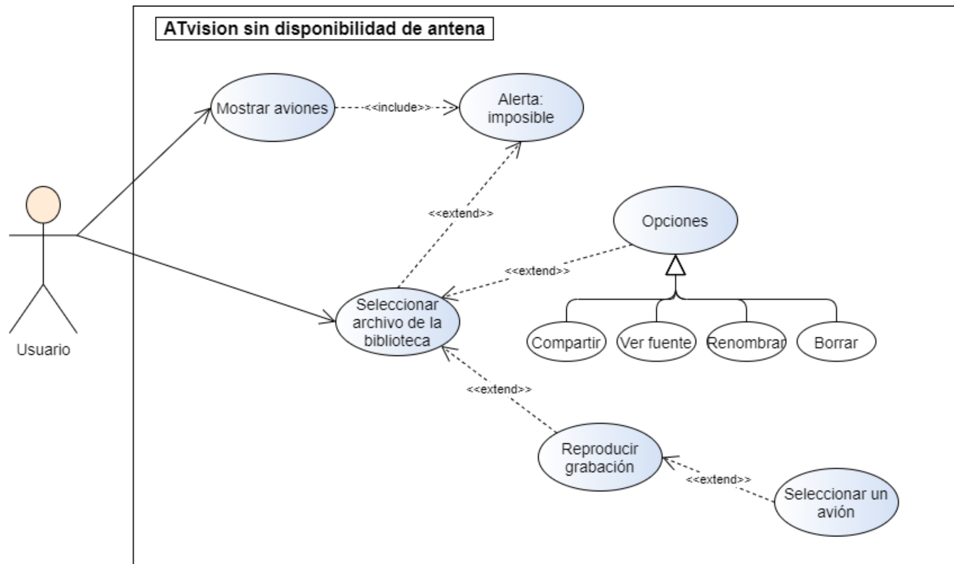

Figura 10, digrama de casos de uso de ATVision cuando la antena no se encuentra disponible

5.2. Actividades

La aplicación cuenta con cuatro actividades, es decir, con cuatro pantallas, cuyo diseño puede observarse en la figura 11: Estas actividades son la base sobre la que se construye la lógica de funcionamiento del proyecto. Cada una de ellas tiene una interfaz de usuario en la que hay ciertos elementos con los que se puede establecer una interacción. Cuando esto ocurre se llama a la función de escucha de la acción correspondiente, en la que hay un código definido que determina la respuesta. Para comprender de una mejor forma este funcionamiento, se entra más en detalle explicando las distintas actividades y la razón de ser de cada elemento de vista, su función y la manera en la que interactúan con el cliente.

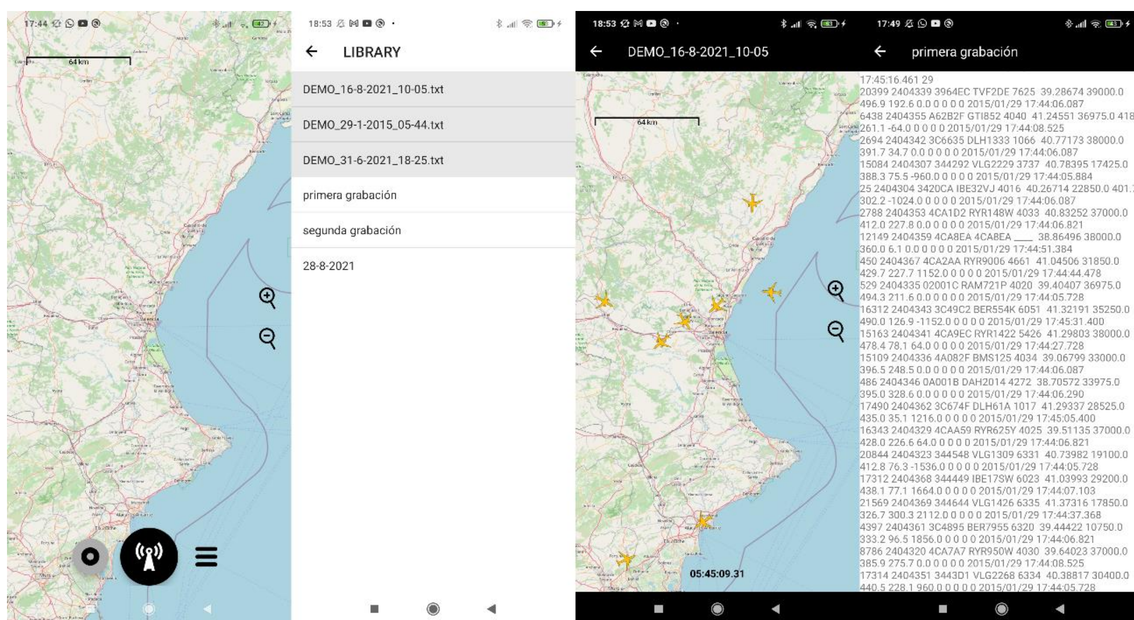

Figura 11, las cuatro pantallas de la aplicación. De izquierda a derecha son: principal, biblioteca, reproductor y fuente

5.2.1. Actividad principal

Cuando se arranca la aplicación, como ya se explicaba en el apartado 3.2, se inicia la actividad principal y se muestra su archivo de *layout*. Entonces, atendiendo a la manera en la que el proyecto ha sido programado, se solicita conocer la ubicación del dispositivo (apartado 6.3.1). Una vez respondido al diálogo de solicitud del permiso, se comienza a interactuar con las distintas funcionalidades de las que cuenta.

Como muestra la figura 11, esta primera actividad cuenta con los siguientes elementos:

- Una vista de mapa MapView heredada de la librería «osmdroid». Dicho componente muestra un mapa que se carga centrado en la ciudad de Valencia, España. Se puede viajar por él haciendo uso de los controles *multitouch*. Esto implica navegar arrastrando la yema del dedo por la pantalla y aumentar o disminuir el tamaño de la imagen pinchando con dos dedos. No obstante, no se puede viajar libremente. Existen ciertos límites en el mapa para solo mostrar la zona del globo en la que la antena tiene cobertura, la que es de interés. Además, se incluyen unos botones de lupa y marcadores representando los distintos aviones y, en la esquina superior izquierda, una barra horizontal que indica la escala. En el capítulo 6.1.3 es explicada la forma en la que se generan estos iconos de las aeronaves y en la que se les otorga la capacidad de ser interactivos de forma que, si el usuario pincha sobre ellos, cambian de color y se muestra su información.
- Un FragmentContainerView. Como bien su nombre implica, este tipo de vistas sirve como contenedor de fragmentos. En particular, en el definido en esta actividad se muestra un fragmento que permite visualizar datos de la aeronave cuyo marcador se encuentre seleccionado, como se puede observar en la figura 12. Por lo tanto, esta vista solo se encuentra visible siempre que haya algún marcador seleccionado.
- Botón de solicitud de la posición. Este botón sirve para conceder el permiso de la ubicación si aún no ha sido concedido.
- Botón de conexión con la antena. Como se puede inferir, presionando este botón, se comienza o termina la conexión con la antena. La recepción de datos se explica en profundidad en las siguientes secciones del trabajo.
- Botón de grabación. Solamente se activa cuando se están recibiendo datos de la circulación aérea. Mientras este botón está bloqueado, se mantiene de color gris. Al desbloquearse, cambia al color rojo (ver figura 12). Entonces, pulsar sobre él permite iniciar o detener una grabación de los aviones que se encuentren en el espacio aéreo durante esos instantes.
- Botón de librería. Es el icono que muestra tres líneas gruesas horizontales. Interactuar con él conduce a la actividad de la librería.

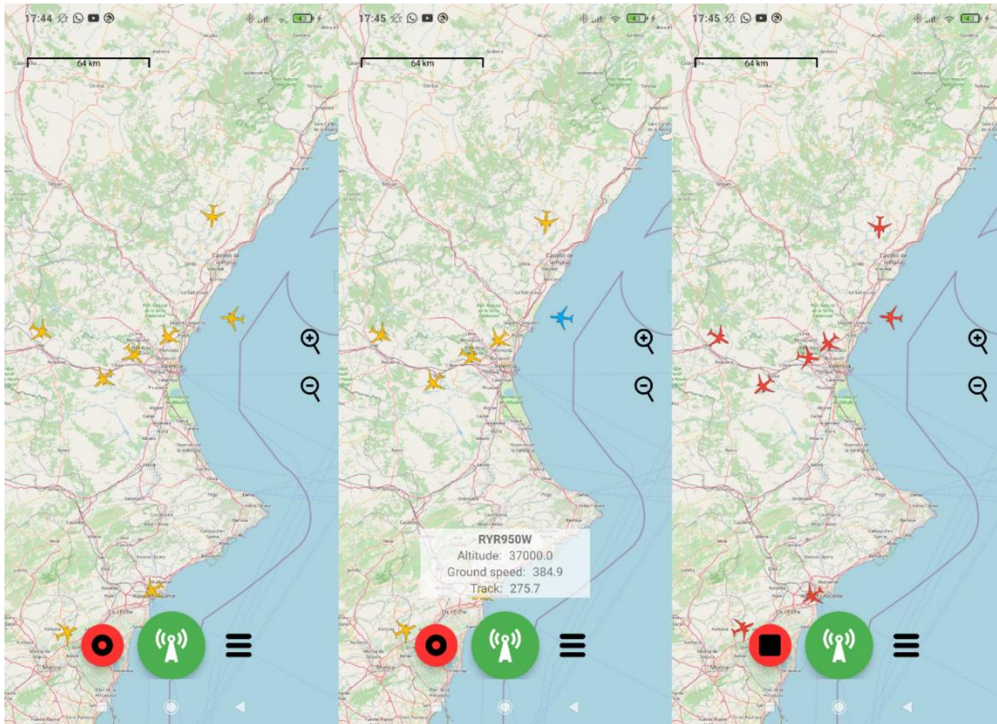

Figura 12, estados de la pantalla principal cuando se interactúa con ella. De izquierda a derecha son: mostrando aviones, un avión seleccionado, grabando

Por lo tanto, en esta actividad se muestra un mapa sobre el que se dibujan los aviones. Para llevar a cabo dicho proceso, es necesario entrar en conexión con la antena por medio de una pulsación sobre el botón correspondiente. Además, cuando se muestra el tráfico, se activa una nueva funcionalidad: la capacidad realizar una grabación de los aviones que se encuentran en ese momento a la visión de la antena. Por otra parte, se puede acceder a la biblioteca de dichas grabaciones interactuando con su icono.

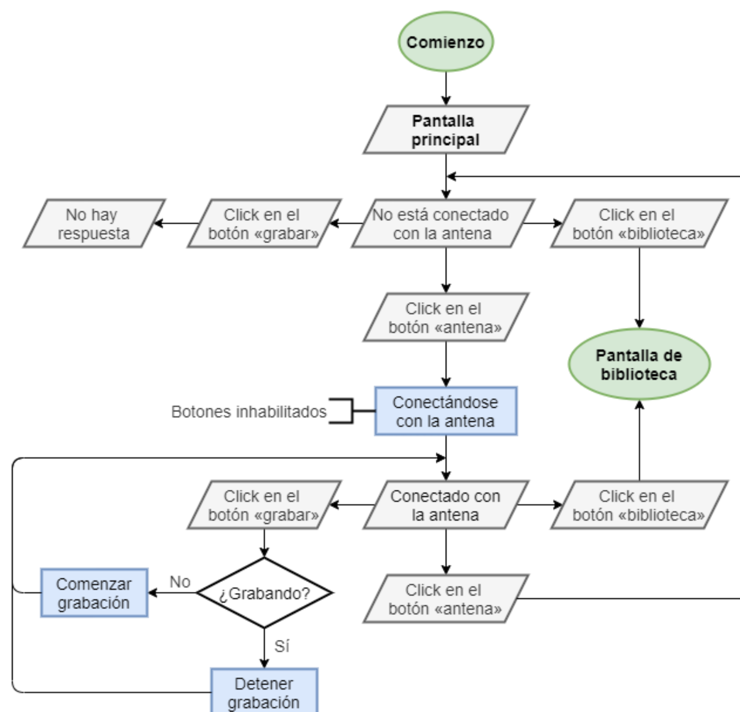

Figura 13, diagrama de flujo de la actividad de MainActivity

5.2.2. Actividad de la biblioteca

Esta pantalla, que en el código se denomina EmulatorLibraryActivity, representa la librería de todos los archivos de texto en los que se guardan las grabaciones del espacio aéreo. Como bien se indicaba, para acceder a esta actividad desde la principal se interactúa con el botón de librería. Entonces, se crea un Intent mediante el cual pasar a la siguiente pantalla, cuyo aspecto se muestra en la figura 11. Los elementos de diseño de la interfaz son los que aquí se mencionan:

- Una Toolbar o barra de herramientas donde aparece el título de esta pantalla, «librería», y un botón de retroceso que permite volver a la actividad anterior.
- Un ListView o vista de lista con los nombres de las demos. Estos archivos vienen dados en la instalación de la aplicación. Para diferenciarlos de los generados por el usuario, el fondo de cada una de las filas de esta lista es de un tono gris claro. Si se pulsa sobre alguno los nombres, se accede a la siguiente actividad del proyecto, el reproductor de dicha grabación.
- Otro ListView con los nombres de todos los archivos guardados por el usuario. Cada uno de ellos admite dos tipos de acciones, una pulsación corta o uno larga.

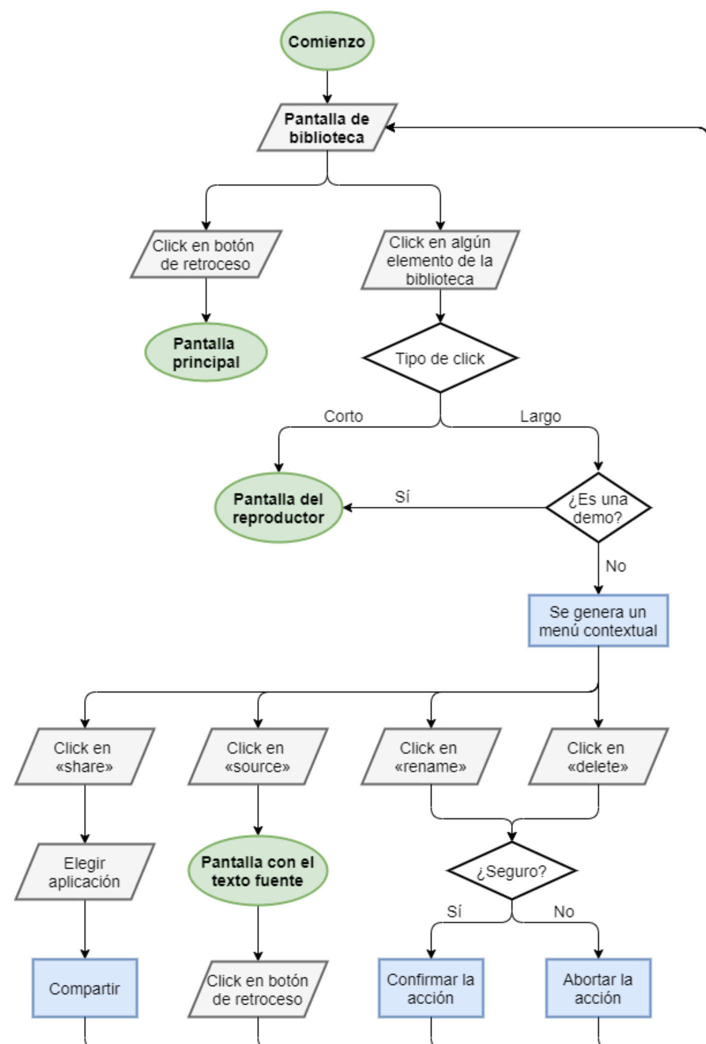

Figura 14, diagrama de flujo de la actividad de EmulatorLibraryActivity

En cuanto a los dos tipos de acciones, es necesario explicar más en profundidad la respuesta a cada una de ellas. Si se realiza una pulsación corta sobre el elemento, se abre, como sucedía con las demos, el reproductor. Si, por el contrario, se interactúa por medio de una presión prolongada, se abre un menú contextual en el que se ofrecen las siguientes opciones sobre el elemento seleccionado (figura 29):

- *Rename*: cambiar el nombre del archivo seleccionado por uno nuevo al gusto del usuario.
- *Delete*: eliminar el archivo permanentemente.
- *Share*: compartirlo mediante una de las aplicaciones para compartir archivos de texto, según elija el usuario.
- *Source*: se genera una actividad en la que se puede el archivo de texto que se emplea como fuente para el reproductor.

Todo lo relativo a las opciones que genera este menú contextual será detallado en mayor profundidad en la sección correspondiente, la sección 6.3.3. La figura 14 ilustra el proceso mencionado.

5.2.3. Actividad del reproductor

Por lo tanto, si se realiza una pulsación breve sobre alguno de los elementos de la biblioteca, se llama a la siguiente actividad, el reproductor. Como su nombre indica, en esta plataforma se reproducen las grabaciones del espacio aéreo. Al igual que la actividad principal, también admite la interacción con los iconos. Además, se puede observar en la figura 11 que la interfaz de usuario de esta pantalla es muy similar a la principal, ya que de nuevo cuenta con un mapa como elemento central. Como la pantalla anterior, incluye una barra de herramientas con su título. Para un mayor análisis, se listan todos y cada uno de los elementos que componen la interfaz. No se incluyen las vistas MapView y el FragmentContainerView, que se definen igual y realizan la misma función que los descritos en el punto 3.1.1:

- Una Toolbar en la que, como en la actividad de la librería, aparece el título de esta pantalla y un botón de retroceso. Esta vez, el título es el nombre del archivo que se reproduce, por lo que cambia en función de este.
- Antes de la reproducción y después de ella, toda la pantalla está ocupada por un botón de *play/replay*, lo que evita que se interaccione de otra manera con la actividad que no sea pulsarlo. Cuando esta acción tiene lugar, se da comienzo o se vuelve a iniciar la reproducción.
- Un TextView en el que hay un texto en negrita. Se va mostrando y actualizando el tiempo real al que se tomaron las diferentes imágenes del tráfico aéreo con el siguiente formato:

  hora:minuto:segundo.milésima

Aunque resulte sencillo, se esquematiza el funcionamiento de esta pantalla en la figura 15. Además de los elementos de la interfaz, otra diferencia con respecto a la

principal es en lo relativo a su tema. Mientras que en la otra las barras de navegación y de estado eran transparentes, en esta ocasión son negras. De esta forma es más fácil e intuitivo distinguir que se está en el emulador.

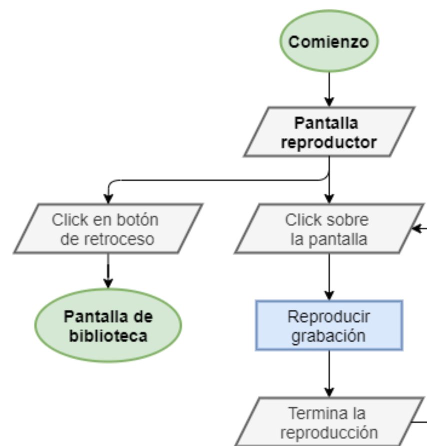

Figura 15, diagrama de flujo de la actividad de EmulatorActivity

5.2.4. Actividad de la fuente

Por otro lado, cabe recordar que, si en vez de una pulsación corta se realizaba una larga sobre alguno de los archivos de la biblioteca, era posible abrir el menú contextual y seleccionar distintas opciones sobre él. Concretamente, si entre ellas se actúa sobre «*source*», se inicia la última de las actividades del proyecto que queda por describir y que es objeto de esta sección. Los elementos del archivo *layout* no son más que una Toolbar de la misma forma que la de la actividad del reproductor y un ScrollView con un TextView en el que se cargan los distintos archivos de texto. Por otro lado, la presencia de la vista de *scroll* permite el desplazamiento a lo largo del texto con un simple arrastre hacia arriba o debajo. El diagrama de comportamiento de esta actividad es muy simple y breve y se encuentra incluido dentro de la figura 14.

5.3. Clases

Como se explicaba en el apartado 3.1, las clases constituyen la pieza básica de la programación orientada a objetos. Es por ello por lo que tienen un gran papel en el diseño del proyecto. Entender su naturaleza y función resulta de gran importancia, ya no solo para comprender el funcionamiento de la aplicación y su implementación, sino para evitar errores a la hora de crear código por una falta de entendimiento. A medida que se avanza en el trabajo de programación, todo se comienza a complicar y las relaciones se vuelven confusas. En ese momento es muy importante mantener presente en la mente qué hace qué. La tabla 15 es un resumen de las distintas clases que componen la aplicación y su función básica. Se muestra de la forma más resumida posible, no obstante, una mayor visión sobre cada una de ellas se presenta en el Anexo

I. De las cinco últimas de esta tabla todavía no se ha comentado nada, están relacionadas con los mensajes que se le muestran al usuario y se mencionan en la sección 6.3.

MainActivity	Define el comportamiento de la pantalla principal
EmulatorLibraryActivity	Define el comportamiento de la pantalla de la librería
EmulatorActivity	Define el comportamiento de la pantalla del reproductor
ReadSourceActivity	Define el comportamiento de la pantalla en la que se muestra la fuente de una grabación
AntennaReceiver	Se conecta con la antena y recibe sus mensajes
AntennaEmulator	Reproduce las grabaciones
Antenna Recorder	Realiza las grabaciones
Traffic	Representa los aviones y sus atributos
PlaneMarker	Representa los marcadores que se dibujan en el mapa
TrafficMap	Es la base de datos del programa, donde se guardan todos los marcadores representados actualmente en el mapa
TrafficCleaner	Borra de la base de datos los aviones que ya no están activos
PlanesMap	Define métodos que realizan acciones sobre el mapa en la aplicación
TrafficInfoFragment	Crea el recuadro en el que se muestran los datos de la aeronave seleccionada
ConnectionAlertDialog	Define el mensaje que salta cuando ha habido un error de conexión con la antena
GuardarGrabacionDialog	Diseña el diálogo que aparece cuando se para una grabación
DeleteDialog	Diseña el diálogo que se muestra cuando hay intención de eliminar un archivo
RenameDialog	Diseña el diálogo que se muestra cuando hay intención de renombrar un archivo

Tabla 5, clases del proyecto y su función

Además, resulta conveniente señalar las relaciones que se establecen entre ellas. Como se vio en el capítulo anterior, desde la actividad principal se llama a la de la biblioteca y, desde esta, a las del reproductor y la del texto fuente. De esta forma, se establece una relación de dependencia entre estas clases que representan las actividades, por la cual sin MainActivity no existiría EmulatorLibraryActivity y sin EmulatorLibraryActivity no lo harían ni EmulatorActivity ni ReadSourceActivity. Además, desde la actividad principal se establece conexión con la antena y se comienzan las grabaciones, lo que establece otras dos dependencias de AntennaReceiver y AntennaRecorder con esta clase. Cuando se ha creado un vínculo con el servidor y se empieza a recibir información, se crean objetos de Traffic, cada uno de los cuales se corresponde con un PlaneMarker. Todos estos objetos forman la base de datos TrafficMap. Además, cabe señalar que todos los marcadores se dibujan por medio de la clase PlaneMap, por lo que debe conocerlos. Por otra parte, esta clase depende de la actividad principal, pues es la que dibuja y manipula su mapa. Al mismo tiempo que se comienzan a dibujar aviones se tienen que analizar cuáles ellos se tendrían que borrar,

TrafficCleaner tiene una relación estrecha con TrafficMap. Una última clase que entra en juego en el momento en el que se tiene la representación del tráfico aéreo es TrafficInfoFragment. Debido a su naturaleza, debe estar relacionada con un único objeto de Traffic y depende de MainActivity, pues desde ahí se actualiza y controla su ciclo de vida.

Por otra parte, analizando las relaciones que conlleva la clase EmulatorActivity, la cual cumple la otra gran función de la aplicación, se puede ver que son bastante similares a las de la actividad principal. Por un lado, cuando se comienzan a representar los iconos, se genera un AntennaEmulator y un TrafficCleaner. Estas dos clases crean objetos Traffic que a su vez se convierten en PlaneMarker y son representados en el mapa. Por otro lado, igual que ocurría antes, solo que con AntennaReceiver, las dos clases necesitan mantener una relación estrecha con TrafficMap, a donde envían y de donde reciben datos.

Todas estas relaciones se pueden ver esquematizadas en el siguiente diagrama de clases de diseño, figura 17. Se trata de un diagrama estándar en el diseño de sistemas de software que sigue el lenguaje UML (lenguaje unificado de modelado). No obstante, se encuentra simplificado, pues no se añade información a cerca de las variables ni de los métodos de cada una de las distintas clases. Aún así, resulta de gran ayuda para entender las relaciones que hay entre ellas. El significado de los conectores de la figura también sigue este estándar y se detalla en la figura 16.

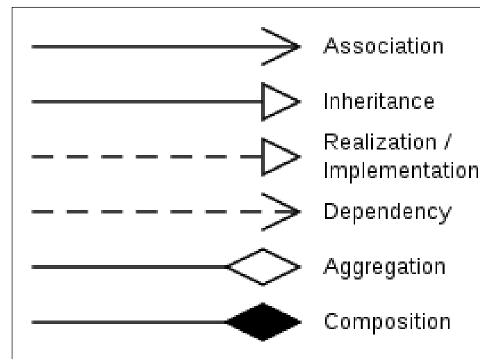

Figura 16, notación de las relaciones UML. Fuente de la imagen: «Class Diagram», Wikipedia [online]. Consultado el 26/08/2021

5.4. Programación multihilo

En la sección 3.2 del trabajo se explicó lo que suponía una estrategia de programación asíncrona o multihilo. Además, se mencionaban las ventajas que suponía esta metodología y las características de los dispositivos móviles que la convertían en necesaria. A pesar de sus ventajas, una de las partes más complicadas de la programación es cuando se tiene que escribir código que se ejecuta en distintos subprocesos. En una primera instancia, al iniciar una aplicación, el sistema crea un subproceso de ejecución que se denomina «principal». Este es muy importante porque está a cargo de distribuir eventos a los elementos de vista correspondientes de la interfaz de usuario. También es aquí donde la aplicación interactúa con estos componentes. Debido a esto, el hilo principal también se suele llamar hilo de la interfaz de usuario (IU) [13]. Por diseño, se espera que una app cree, use y destruya los objetos de la interfaz en este hilo

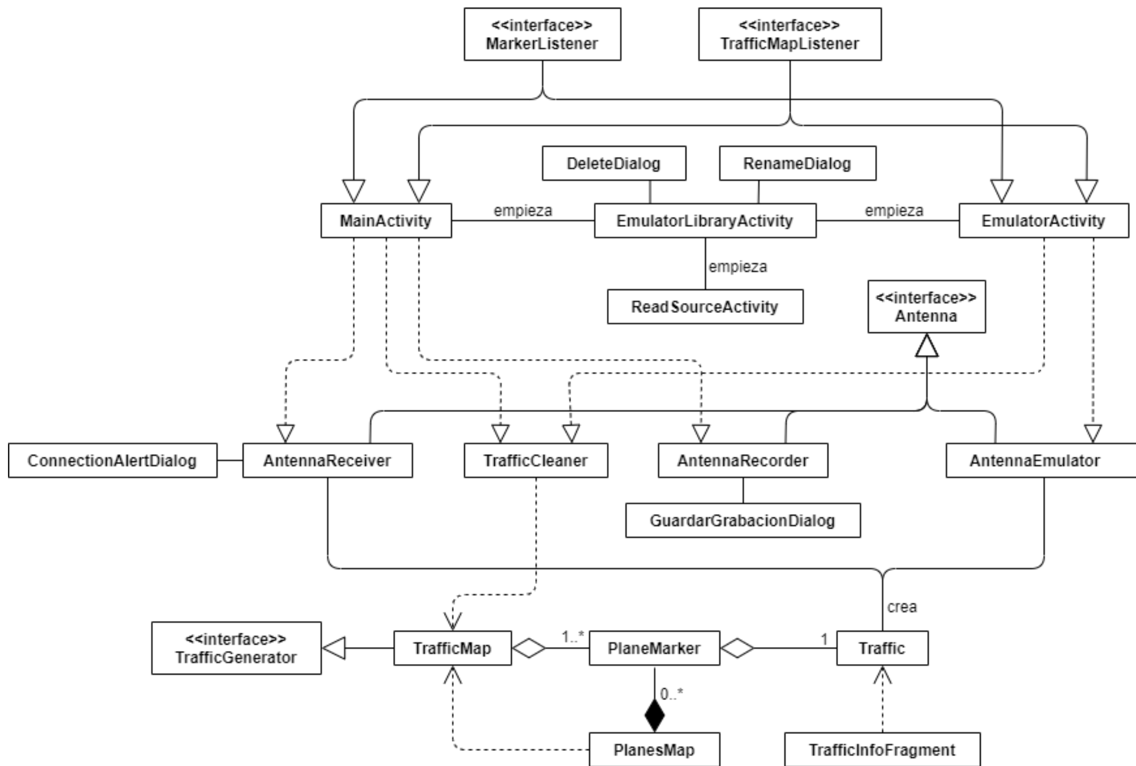

Figura 17, diagrama simplificado de las clases de ATvision

Debido a la cualidad de este *thread* como gestor de los eventos sobre los elementos de la interfaz de usuario, si todo se realiza en él, las operaciones prolongadas como acceder a la red bloquean toda la pantalla. Por consiguiente, este modelo de subproceso único puede generar un rendimiento deficiente, pues cuando se bloquea no se pueden distribuir los eventos, ni siquiera los de dibujo. Cuando esto ocurre, desde la perspectiva del usuario, la aplicación no responde e incluso se llega a abortar su ejecución. En el presente apartado se entra en detalle en la solución adoptada. La figura 18 inferior esquematiza las clases y los eventos que generan cada uno de los subprocesos. Una primera aproximación consiste en listar los distintos hilos existentes en la aplicación y las tareas que se llevan a cabo en ellos.

- Empezando por el hilo principal. En él, como ya se ha mencionado se definen y manipulan todos los elementos de interfaz de las distintas pantallas. Además, responde a los eventos de llamada de cada uno de ellos. Por otra parte, en este subproceso también se asignan valores importantes, tales como los iconos de las aeronaves, el directorio de las grabaciones o el FragmentManager. Otra operación que se realiza es la comprobación y, en su caso, solicitud del permiso de ubicación. Por último, modifica las capas del mapa, controla el fragment que muestra la información de las aeronaves seleccionadas y da comienzo a otros subprocesos.
  Subproceso de recepción de datos. Crea una vinculación con la antena y comienza un bucle de recepción y análisis de cadenas de información. Entrando más en detalle en este bucle, después de que lleguen los datos, con ellos se crea o actualiza un objeto Traffic que se añade a la base de datos del proyecto. Tras

esta última acción, y todavía dentro de este hilo, la capa de los iconos del mapa es actualizada y se informa al evento de escucha de que ha habido una actualización TrafficMap. Finalmente, cuando se decide salir del bucle se cierra la comunicación con el servidor.

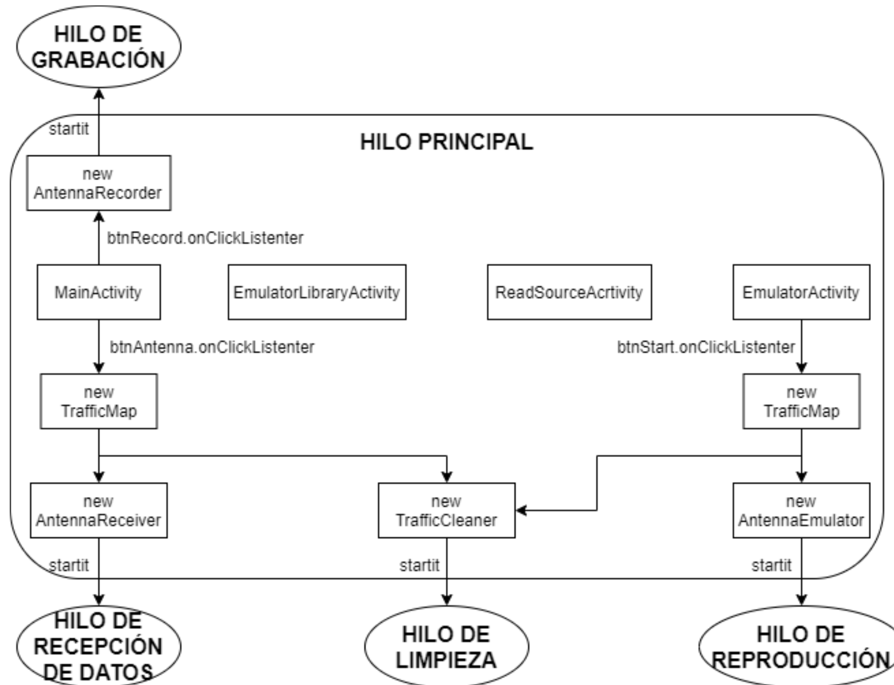

Figura 18, threads de la aplicación y los métodos que los ejecutan

- Subproceso de grabación. Lo primero que se lleva a cabo es originar un nuevo archivo en el que guardar los datos. Entonces, se genera un escritor para él y se comienza un bucle encargado de realizar capturas del tráfico aéreo cada cierto tiempo y transcribirlas en el documento de texto. Como se verá en el apartado 6.2.1, esta transcripción ha de hacerse atendiendo a un protocolo. Por último, al terminar la grabación se genera un diálogo que pregunta al usuario cómo desea guardar el archivo.
- Subproceso de reproducción. Es el encargado de reproducir las grabaciones. Con este fin, su primera tarea es identificar el archivo y crear una instancia de él con la que poder generar un lector. Después, comienza otro bucle que, esta vez, lo que hace es recorrer el archivo línea a línea. Cuando termina de leer una, se genera un procesamiento que crea o actualiza un objeto Traffic y lo añade a la base de datos. Una vez termina de recorrer las líneas relativas a la captura de un instante concreto, se actualiza el dibujo de todos los aviones sobre el mapa. Como ocurría en el subproceso de recepción de datos, todas las actualizaciones en el TrafficMap se comunican al TrafficMapListener.
- Subproceso de limpieza. Este último hilo tiene la función de eliminar los aviones de los cuales ya no se producen actualizaciones. Cada medio segundo se invoca un método que recorre, por medio de un iterador, la base de datos y comprueba cuanto hace que no se actualiza cada objeto Traffic. Si este periodo de tiempo es superior al minuto, dicha aeronave se elimina del HashMap.

6. Implementación

Tras adquirir una visión del diseño del proyecto, sus componentes y sus funciones, en este capítulo de la memoria se detalla cómo se han resuelto las distintas tareas a nivel de código. Así pues, se describe la implementación del proceso de comunicación con la antena y representación de los iconos en el mapa. Posteriormente, se expone el funcionamiento del emulador o grabadora y reproductora. Por último, se explica algo que aún no se ha mencionado, la interacción con el usuario. En este último apartado se comenta la forma en la que se generan los distintos mensajes que guían al cliente por la aplicación con el fin de proporcionarle una buena experiencia de uso.

6.1. Comunicación con la antena

En primer lugar, se entra en materia en la función principal del proyecto y la que definió su alcance en una primera instancia. Este subapartado abarca todo lo relativo al vínculo creado con el servidor que transmite los datos de las diferentes aeronaves, lo que recoge desde su conexión hasta la forma en la que se reciben los datos. Posteriormente, estos datos son analizados y se genera un almacenamiento interno en el que se guardan los que resultan de interés. Por último, la representación de esta información en la pantalla es el fin que se persigue en todo este proceso, por lo que es digno de un análisis en profundidad.

Antes de entrar en materia en cada uno de los apartados mencionados conviene mencionar el proceso que se siguió hasta alcanzar el objetivo final de esta sección. Así pues, a medida que se comprendía la forma de actuar se realizaron distintas aplicaciones de prueba, cada una de las cuales realizaba lo siguiente:

- La primera de ellas establecía una conexión con el servidor cada vez que se pulsaba un botón. Entonces, solamente recibía una línea de información y la presentaba en la pantalla dividida por campos.
- La segunda funcionaba de la misma forma en cuanto a que solamente se comunicaba con la antena cada vez que se pulsaba sobre el botón. No obstante, introducía la novedad de que los datos se iban guardando. Así se le ofrecía al usuario una lista en la que se podían ver todos los aviones que se habían recibido. Al pulsar sobre uno de ellos se mostraba la información separada por campos que se había recibido de él por última vez
- Por último, ya se creó un código más sofisticado. Este proyecto realizaba una conexión ininterrumpida en la que se procesaban los datos y se actualizaba un almacenamiento interno. Del mismo modo que la anterior, se le ofrecía al usuario una lista para elegir el avión del que quería visualizar la información, la cual se iba actualizaban automáticamente.

6.1.1. Conexión y recepción de los datos

Para poder recibir datos, la aplicación establece una conexión con la antena, descrita en el punto 4.3 de este trabajo, que se encuentra en el edificio de la ETSID en la Universidad Politécnica de Valencia. Por lo tanto, tras haber entendido el objetivo del proyecto y haber adquirido los principios de programación en Android, el siguiente paso consistía en hallar un método que realizase dicha conexión.

El principio para resolver el problema es la programación mediante la API Socket. Esta herramienta permite a una gran cantidad de programas acceder a los servicios de comunicación por la red, lo cual habilita a que dispositivos separados por una gran distancia puedan comunicarse y compartir información. Como se explica de forma extensa en el libro de K. L. Calvert y M. J. Donahoo «TCP/IP Sockets in Java, practical guide for programmers», los datos que se transmiten viajan en forma de secuencias de bytes, conocidas como paquetes, y estructuradas según un protocolo concreto. Un protocolo es un acuerdo entre los programas que intercambian información y que da significado a cada uno de los paquetes. TCP/IP es una colección de protocolos (*protocol suite*) que reúne los empleados en la red. De ellos, los principales son el Protocolo de Internet (*Internet Protocol*), IP, el Protocolo de Control de la Transmisión (*Transmission Control Protocol*), TCP, y el Protocolo de Datagramas de Usuario (*User Datagram Protocol*), UDP. Por lo tanto, por medio de la API Socket, las aplicaciones pueden acceder a los servicios que proporcionan UDP y TCP.

El protocolo de red IP resuelve el problema de transmitir paquetes de información al destino por medio de un canal servidor-servidor. En cuanto a UDP y TCP, ambos se construyen sobre este servicio proporcionado por IP, pero de distinta forma. Así, ofrecen diferentes transportes según la necesidad del sistema. No obstante, a pesar de las diferencias, tienen una función en común: dirigir. Si bien, IP transmite paquetes a los servidores (*hosts*), TCP y UDP los llevan a una aplicación en específico. Ambos protocolos utilizan direcciones conocidas como números de puerto para identificar este programa concreto dentro de un servidor. En contraste con UDP, TCP está diseñado para detectar y recuperarse de los errores para permitir un sistema fiable de flujo de bytes. Así, antes de intercambiar información, dos programas deben establecer una comunicación TCP con una serie de mensajes de autenticación entre los dos dispositivos.

Recapitulando, para establecer una conexión TCP/IP como la de la aplicación entre manos, es necesario especificar dos datos. En primer lugar, la dirección del servidor. Estas direcciones IP son números binarios de 32 bit que se suelen mostrar como un *string* de 4 números decimales separados por puntos (por ejemplo 10.1.2.3). No obstante, también existe otra notación por nombres, que resulta más conveniente por dos razones. La primera de ellas es la mayor facilidad para los humanos de recordar un nombre que un código numérico. La segunda, los nombres proporcionan un cierto nivel de abstracción que protege de cambios en la dirección IP. Siguiendo adelante, una vez se conoce y se ha especificado la dirección del servidor, el otro dato que hace falta para establecer una conexión TCP es el puerto, un número de 16 bit entre 1 y 65535.

La API de Socket no es más que una abstracción que Java proporciona para que las aplicaciones puedan vincularse con un servidor, y de este modo poder enviar y recibir datos. Así pues, una instancia de esta clase representa una conexión TCP vinculada. Este es el método que se ha seguido en la aplicación y ha permitido que el programa, el cliente, pueda recibir información de la antena, el servidor. Para que el cliente comience la comunicación, se siguen los siguientes pasos:

1. Instanciar a la clase Socket. El constructor establece una conexión TCP tras haber especificado el servidor remoto (servantena.etsid.upv.es) y el puerto (30002).
2. Comenzar la comunicación empleando los flujos de entrada y salida del *socket*, conocidos como InputStream y OutputStream. A partir del objeto socket, es posible obtener el de entrada aplicando el método getInputStream(). Un flujo es simplemente una secuencia ordenada de bytes y el hecho de que sea de entrada lo define para poder ser leído.
3. A partir del InputStream, se crea un InputStreamReader y de este, un BufferedReader. El lector se genera con el objetivo de transformar el flujo de bytes en un flujo de caracteres de texto. Dicho de otro modo, el trabajo que realiza InputStreamReader es decodificar los bytes. De esta forma, cada vez que se invoca al método read(), se leen uno o más bytes de la cadena. BufferedReader se instancia con el fin de optimizar la eficiencia. [13]. Esta clase incluye el método readLine(), que permite leer líneas de bytes completas, es decir, hasta el delimitador «\n».

Una vez se termina de transmitir información, es necesario cerrar todos los puertos. Con esta finalidad existe el método close(), parte de la interfaz de Java Closeable.

Todas las tareas anteriormente descritas las realiza en segundo plano la clase AntennaReceiver. La lectura de información comienza cuando el usuario pulsa sobre el botón de la antena en la actividad principal (figura 13). Una vez se configura el lector comienza un bucle que no se detiene hasta que el usuario lo especifica volviendo a presionar sobre dicho botón. En ese bucle, el objeto BufferedReader llama al método readLine() anteriormente comentado y posteriormente, esta línea se procesa. Con esta metodología se consigue crear o actualizar un objeto Traffic que identifica el avión cuyos datos han sido leídos y define sus atributos.

6.1.2. Análisis de los datos

Por consiguiente, tras leer una línea de bytes el programa la analiza y mantiene actualizados una serie de objetos Traffic. Esto es posible gracias al método messageTCP() de la clase TrafficMap (para más detalle consultar el Anexo I).

En primer lugar, cabe definir la clase TrafficMap, su importancia dentro de la aplicación y su utilidad. Esta clase es la central del proyecto, por ella pasa y en ella se almacena toda la información. Constituye la base de datos donde se recogen todas las

aeronaves y, como tal, debe permanecer actualizada. A ella acuden todos los métodos que necesitan conocer el tráfico aéreo actual ya sea para representarlo, compararlo con otros valores o grabarlo. Esta naturaleza como fuente de almacenamiento se la otorga el hecho de extender a ConcurrentHashMap. Esto resulta especialmente conveniente por un par de motivos: esta clase desciende de la clase abstracta Map e implementa la interfaz ConcurrentMap.

Map se define como un objeto que mapea (relaciona) una serie de claves y una colección de valores de forma que a cada clave le corresponda un valor concreto. Además, tiene unos métodos que resultan de gran ayuda. Entre ellos destacan:

- get(), que devuelve el valor con el que se relaciona la clave que se le pasa como argumento.
- put(), añade el nuevo par clave-valor que se le especifique a la base de datos.
- remove(), elimina una pareja clave-valor determinada
- replace(), que toma como argumento una clave y un valor y actualiza el valor asociado a la clave dada por el nuevo que se le haya pasado.

Implementar a ConcurrentMap lo define como un tipo de mapa que proporciona seguridad entre hilos y evita problemas de concurrencia. Los efectos en cuanto a la consistencia de memoria son al igual que otras colecciones concurrentes. Las acciones en un hilo previas a añadir un objeto en un ConcurrentMap como clave o valor ocurren antes a aquellas acciones que conllevan el acceso o la eliminación de este objeto del ConcurrentMap en otro hilo [13]. Además, también se implementa a Serializable. Esta interfaz no define ningún método ni ninguna clase, simplemente sirve para identificar a un objeto como Serializable. Este hecho permite que java pueda convertir un objeto a una cantidad de bytes y pueda luego recuperarlo

Por lo tanto, TrafficMap tiene un par de colecciones en las que cada miembro de una (claves) permite identificar un término de la otra (valores). En este caso, las claves son el código hexadecimal de la aeronave en forma de String y los valores son objetos de la clase PlaneMarker, que representan el marcador de cada avión dibujado en el mapa y almacena todos los datos de circulación que se hayan transmitido. Así, se puede crear la base de datos de la aplicación en la que se almacena el estado de cada avión y se puede acceder a él o modificarlo tan sencillamente como conociendo su identificador.

Se había mencionado en la sección anterior que una vez que una línea de los datos enviados por la antena es leída, se introduce en messageTCP(). Este método realiza diversas acciones sobre la información transmitida. Lo primero es separar la información y almacenarla en un vector de String. Esta tarea la facilita el hecho de que en la cadena inicial cada campo está separado del siguiente por un carácter «,», una coma. Así, empleando *regex* y el método split() de la clase String se separa dicho mensaje inicial en el vector buscado. En la tabla 3 presentada con anterioridad se resumen todos los datos que envía la antena y en el orden en que se transmiten, así es posible saber en qué posición del vector se encuentra cada propiedad. De esta forma, lo primero que se realiza una vez que se ha obtenido el vector de valores es comprobar si el avión del cual

se han obtenido los datos ya existe en el ConcurrentHashMap. Para llevar a cabo dicho trabajo, se identifica el código hexadecimal y se usa el método get(), que, en caso de que la aeronave no exista, devolverá un valor nulo. En dicha situación, se crea un nuevo marcador del avión. Se instancia primero a Traffic y luego a PlaneMarker y este último, se añade a la base de datos. Por el contrario, si existía la aeronave, se actualiza su marcador, que reemplaza al anterior en TrafficMap. Esta lógica se esquematiza en la figura 19, que la representa por medio de un diagrama de actividad creado, de nuevo, siguiendo la norma UML.

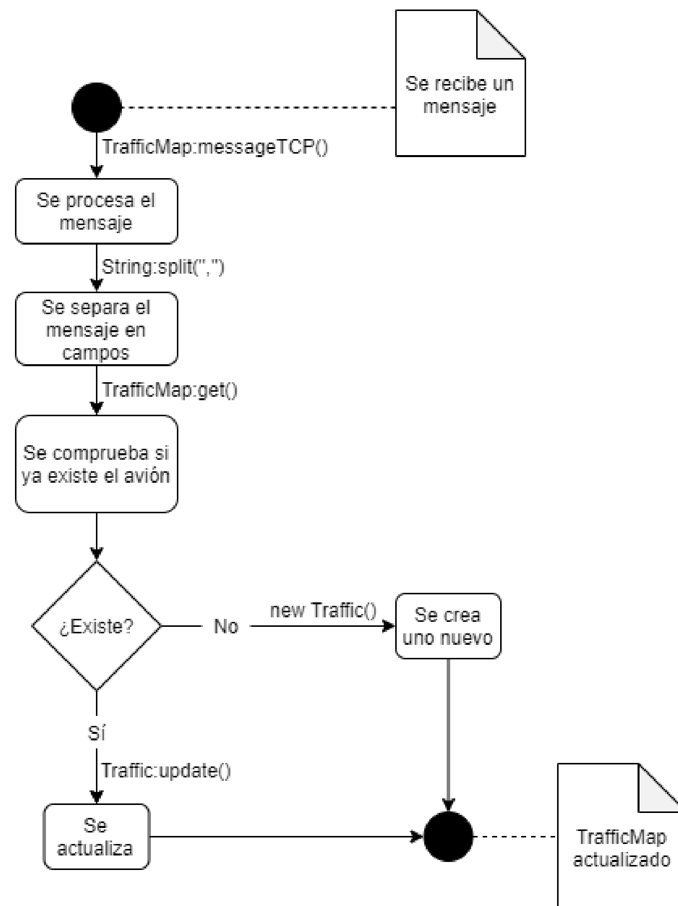

Figura 19, diagrama de actividad del procesado de los mensajes que envía la antena y actualización de la base de datos

Otro punto de interés en cuanto al análisis de los datos enviados por medio de la antena es la lógica mediante la cual un objeto de Traffic puede actualizarse. Ya se vio en el punto 4.3 que existen diversos tipos de mensajes que se pueden recibir en función de los datos que hayan sido transmitidos (tabla 4). En consecuencia, lo primero que se realiza es identificar este tipo de mensaje con el que se trabaja. Dicho dato es un campo de la cadena de bytes recibida, más en concreto, el primer valor, por lo que resulta fácil de hallar. Viene dado en forma de String y puede ser «MSG», «SEL», «ID» o «AIR». En el caso de que se trate de un «MSG» habrá también un subtipo asociado, que se encuentra de nuevo presente en la cadena String, pero esta vez en segundo lugar. Cuando está identificado, es sencillo realizar un switch() que opere ciertas acciones en función de cuál sea el mensaje transmitido.

Una última parte del análisis de los datos de la aplicación que resulta de gran importancia para su correcto funcionamiento es el limpiador. Este, que se materializa en la clase TrafficCleaner y se implementa con el método cleanLostFlights() de TrafficMap, es un bucle que corre en segundo plano y se inicia al mismo tiempo que la recepción de la información de la antena. Actúa como un observador que retira las aeronaves inservibles, es decir, de las que no se obtiene ninguna información en un tiempo, con el fin de que no estorben. Es importante, pues puede haber vehículos cuya información fue en un momento recibida pero que ya no se encuentran al alcance de la antena. En ese caso, es necesario identificarlos y borrarlos, pues ya no resultan de interés y serían una carga tanto visual como a nivel de rendimiento. De esta forma, cada medio segundo (ha sido especificado de este modo) el bucle que se está ejecutando asíncronamente llama a la función de borrar los vuelos perdidos. En ella, se recorren todos miembros de la base de datos, por medio de un enumerador. Uno a uno se comprueba el momento de su última actualización y se compara con el actual. La diferencia no ha de ser de más de sesenta segundos, o lo que es lo mismo, un avión no puede llevar más de un minuto fuera del alcance de la antena. En dicho caso, el icono se elimina del mapa y el marcador, de la base de datos.

6.1.3. Representación

Con lo que ya se ha analizado, es posible generar una base de datos que incluya los valores de los vuelos del espacio aéreo cubierto por la antena. La siguiente misión en el desarrollo de la aplicación consiste en crear una manera a través de la cual dicha información pueda ser representada visualmente y mantenerse actualizada. El presente apartado pretende explicar el enfoque adoptado, las herramientas empleadas y las dificultades encontradas.

Ya se ha mencionado que el recurso que se emplea en el proyecto para la generación de mapas es la biblioteca «osmdroid», basada en OpenStreetMap. Los atractivos de ella son que permite sustituir la clase MapView de GoogleMaps por un MapView propio, así como generar, añadir o eliminar capas de los mapas. En concreto, ItemizedIconOverlay crea una capa (*overlay*) de iconos (OverlayIcon) que, además, escuchan eventos de pulsación sobre cada uno de ellos individualmente, y pueden producir respuestas en dichos casos. Esta cualidad resultaba especialmente atractiva, pues a parte de generar una capa donde todos los aviones fuesen dibujados, se pretendía que al clicar sobre alguno de ellos se pudiese seleccionar.

PlanesMap, una clase creada con el fin de mejorar la legibilidad y reciclabilidad del código, define métodos que realizan todas las tareas involucradas en la interacción con el mapa (más detalle sobre esta clase en el Anexo I). Ella es la encargada de dibujar los iconos. Para ello, junto a métodos como drawScale() o drawZoomButtons(), los cuales dibujan la escala y los botones de zoom respectivamente, se definió la función drawPlanes(). Su argumento es un vector con todos los marcadores, el cual se obtiene a partir de la base de datos y se introduce en el constructor de ItemizedIconOverlay. Junto

con esta colección también es necesario introducir un recurso *drawable*, que represente el icono de todos los aviones por defecto, y un *listener* para detectar las interacciones del usuario sobre cada uno de estos iconos. La imagen empleada para el marcador por defecto es un B787 de color amarillo, figura 20.

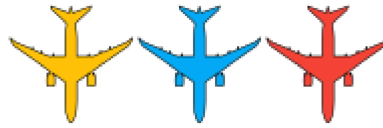

Figura 20, los tres iconos empleados en la aplicación para representar los aviones. Sin seleccionar (izquierda). Seleccionado (centro). Grabando (derecha)

De todas las pruebas hechas, la solución que dio menos problemas y mejores resultados, sobre todo en lo relativo a la fluidez de los cambios en la imagen, fue dibujar todos los aviones de nuevo, toda la capa de ItemizedIconOverlay, con cada actualización de la base de datos. Se asume que este no es el método más eficiente; de hecho, uno de los trabajos futuros (sección 7) es la búsqueda de formas que lo realicen con un mejor rendimiento. No obstante, los resultados obtenidos resultan satisfactorios para lo que se pretende. Atendiendo a esto, se convierte en necesario volver a dibujar todos los aviones cada vez que se actualice TrafficMap. Por lo tanto, en el método messageTCP(), que se explicaba en la sección anterior, siempre que termina de actualizarse la base de datos, después de toda la lógica, se ejecuta drawPlanes().

Para introducir la capacidad de seleccionar un avión se emplearon, como ya se ha mencionado, las funciones de llamada a eventos sobre los marcadores, es decir, los Listeners. No obstante, antes de esto fue necesario introducir una nueva variable pública en PlanesMap que identificase la aeronave que se encontrase seleccionada. Con esta herramienta, podía implementarse un algoritmo que se ejecutase con la pulsación de algún marcador y determinase, por medio de una simple comparación de los identificadores de las aeronaves, si se ha pulsado el que estaba seleccionado. En caso de ser así se deseleccionaría y ya no se mostrarían sus datos en el FragmentMapContainer. Si, por el contrario, no hubiese ningún avión seleccionado o se interactuase sobre uno nuevo, habría que tener en cuenta el hecho de que no puede haber más de uno seleccionado al mismo tiempo. En tal caso, esta nueva aeronave sería la única que la variable determinase como tal y cuyos datos se mostrasen en pantalla.

A continuación, surgieron dos problemas. El primero de ellos consistía en que los métodos descritos permitían identificar una aeronave seleccionada, pero la interfaz de usuario no incorporaba ninguna manera de distinguir dicho avión. Con este fin, se introdujo un nuevo icono cuyo aspecto era el mismo que el automático, pero de color azul (figura 20). El cambio al nuevo icono no se podía llevar a cabo en la función de respuesta a una pulsación. Si se realizaba de dicha manera el marcador solamente se distinguiría durante la iteración en la que tuviese lugar el evento, pues al volver a cargarse los aviones, el código no tendría memoria de cuál se había seleccionado y el icono recuperaría el color amarillo habitual. En este sentido, se añadió después de la creación de la capa, un bucle para recorrer todos los marcadores individualmente, para lo que resultó de gran utilidad el método getItem() explicado en la Sección 2.3.3.

Entonces, se compararía el nombre de cada marcador con el del avión seleccionado guardado en la variable correspondiente, lo que habilitaba para cambiar su icono.

El segundo de los retos que surgió a la hora de representar los datos en el mapa fue el siguiente. De momento, ya se había logrado representar todos los aviones, seleccionar uno y distinguirlo. No obstante, era necesario llevar a cabo el cambio que se muestra en la figura 21. En ella, la representación de la derecha es, evidentemente, más intuitiva y funcional. Como puede observarse, los iconos se alinean con la derrota de la aeronave, de forma que el morro apunta hacia donde se mueve. Esta rotación se realiza en el bucle que se ha descrito antes, en el que se recorren uno a uno los marcadores para comprobar cuál es el seleccionado. Con la misma filosofía, cuando se tiene identificado un marcador, se puede leer su derrota y llamar a un método privado de la clase PlanesMap, llamado rotateDrawable(). Sus argumentos lo componen el icono y el ángulo que se ha de rotar. Como herramienta se emplea una subclase de Drawable conocida como RotateDrawable que incorpora los métodos setFromDegrees() y setToDegrees. Este par de funciones resultaron muy útiles para el fin que se buscaba, pues en ellas se especifican los ángulos de inicio y de fin de la rotación respectivamente.

Figura 21, cambio en la representación de los iconos. En la imagen izquierda no se rotan, mientras que en la derecha se alinean con la dirección de su trayectoria

Otro aspecto de la representación de los datos es el ya tantas veces mencionado FragmentContainerView que aparece cuando se selecciona un avión y muestra algunos de sus valores más importantes. Esta vista de la interfaz de usuario no es más que un contenedor al que se le puede asociar un fragmento con un *layout* propio que se muestre. Este fragmento viene definido por la clase TrafficInfoFragment y se compone de cuatro líneas de información. En la primera de ellas, resaltado en negrita, está el título, el *callsign* del avión. Por debajo suyo se encuentran escritos los siguientes datos en orden descendiente: la altitud, la velocidad de tierra y la derrota. Alcanzar el objetivo deseado conllevó ciertos retos similares a los que se encontraron cuando se pretendía seleccionar un avión y distinguirlo. Para identificar el avión seleccionado, se accede a la

variable de PlanesMap que almacena su nombre. Para responder ante una pulsación del usuario sobre un avión, la función de respuesta a dicho evento llama a showInfo() o hideInfo() del MarkerListener según corresponda. Por último, para mantener actualizada la información que se muestra, cada vez que se actualiza un avión se comprueba si se está mostrando su información. En caso positivo, también se actualizan sus valores en TrafficInfoFragment. Para poder entender mejor la lógica de este último comportamiento, la figura 22 muestra un diagrama de actividad del algoritmo.

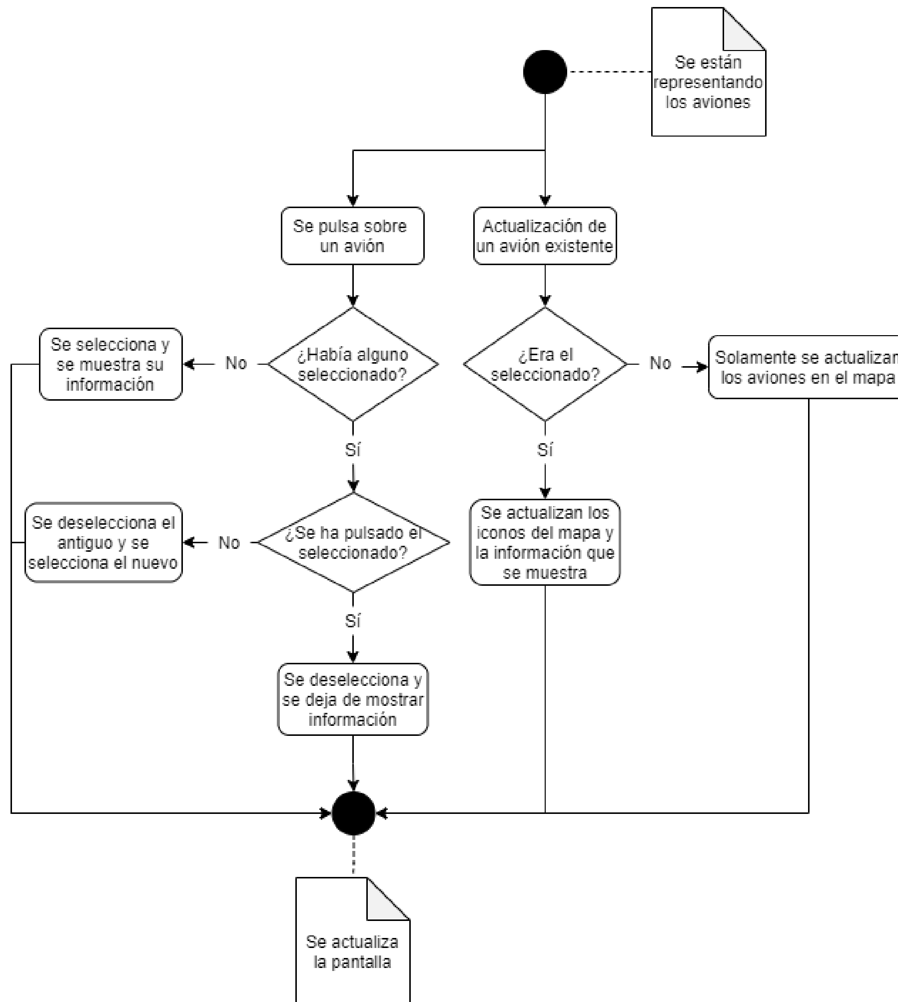

Figura 22, diagrama de activiad del algoritmo de selección/deselección de un icono y la muestra de su información

6.2. Emulador

Una vez explicada la forma en la que se ha implementado la función principal de la aplicación, se ha de abordar el método de conseguir su otro gran objetivo. La parte de la aplicación que se ha denominado con el término «emulador» es la que graba el espacio aéreo, genera y almacena documentos y los reproduce en el caso de que se solicite. De esta forma, en esta sección se comenzará explicando el código detrás de la grabadora y la creación de los documentos. Posteriormente, se analiza el reproductor, que como ya se verá, no difiere en gran cantidad de lo expuesto en el apartado anterior.

6.2.1. Grabación

En la figura 13 se observa que cuando hay vinculación con la antena, se habilita un botón que permite grabar el espacio aéreo. Esta capacidad de la aplicación ya fue introducida desde el principio de la presente memoria, cuando se describía su funcionamiento general en el apartado 1.1. En esta sección se entra en detalle en la generación de dichos documentos, una primera parte del emulador.

Las grabaciones se guardan en un archivo de texto (*.txt*) que describe el espacio aéreo. Por lo tanto, lo primero que es necesario crear para que estos documentos puedan ser interpretados es un protocolo con el que describir la forma en la que se estructura el texto y el significado de cada dato. De este modo, al igual que ocurría cuando se conectaba con la antena, cada línea de datos que llegue a la aplicación podrá ser procesada y el archivo, reproducido. La filosofía es la misma, solo que esta vez los datos, en lugar de ser enviados por un servidor externo, los envía un documento interno. Así bien, en una primera instancia el archivo se divide en varios bloques o párrafos, cada uno de los cuales describe el tráfico aéreo en un momento en concreto. Es como si se tomase una foto de los aviones que se encontraban circulando en un instante concreto, este párrafo describiría dicha foto. En un nivel inferior, estos bloques se dividen en distintas líneas de información. Cada una de ellas, salvo la primera, representan una aeronave en concreto. La primera, no obstante, ofrece dos datos distintos: el momento en el que se toma la captura y la cantidad de aviones que había en ese determinado instante. Especialmente el segundo de ellos resulta de gran importancia para el reproductor, como se comentará más adelante en el apartado 6.2.2. Entonces, a excepción de esta primera línea, cada una del resto define un avión en específico. Lo hacen de una manera similar a la que la antena emplea para enviar datos de las aeronaves. Se divide la información del vuelo en campos y se ordena de una forma concreta y conocida, la cual se observa en la tabla 6. No obstante, en este caso los campos no se separan entre ellos por una coma, sino por un espacio en blanco, aunque a nivel práctico es lo mismo.

Campo 1	AircraftID, número de registro de la aeronave
Campo 2	FlightID, número de registro del vuelo
Campo 3	HexIdent, código hexadecimal de identificación de la aeronave en modo-S
Campo 4	Callsign, un número de ocho dígitos que identifica la aeronave.
Campo 5	Squawk, código del transpondedor modo-A
Campo 6	Lon, longitud a la que se encuentra. Norte, positivo. Sur, negativo
Campo 7	Lat, latitud a la que se encuentra. Este, positivo. Oeste, negativo
Campo 8	Alt, altitud a la que se encuentra. Relativa a 10132.5 mb, no es la altitud AMSL
Campo 9	GroundSpeed, velocidad de avance
Campo 10	Track, derrota. No es el rumbo.
Campo 11	VerticalRate, la velocidad vertical
Campo 12	Alert, bandera que indica si ha cambiado el Squawk
Campo 13	Emergency, bandera que indica que hay código de emergencia
Campo 14	SPI, bandera que indica que el identificador del transpondedor se ha activado
Campo 15	IsOnGround, bandera que indica que el mando de tierra está activo
Campo 16	Date, la fecha en la que se genera el mensaje
Campo 17	Time, el momento en el que se genera el mensaje

Tabla 6, campos de datos que se reciben de un archivo del emulador de ATvision

Con el objetivo de generar de manera intuitiva y simple una de estas líneas para una aeronave concreta, se crea el método getInfo() en la clase Traffic. Invocándolo, se genera y devuelve un único String que recoge toda la información previamente mencionada y en el orden establecido.

Una vez sentadas estas bases, se analiza lo que sucede a nivel del código cuando se pulsa el botón de grabar (y este no está inhabilitado). El proceso se muestra esquemáticamente en la figura 23. Desde la actividad principal se crea un objeto de AntennaRecorder. Antes de comenzar a grabar, se genera un nuevo archivo de texto en un directorio y con un nombre predefinidos. El directorio es una carpeta dentro del almacenamiento interno de la aplicación en el dispositivo, específicamente, la carpeta se llama «Emulator». Por otra parte, el nombre que se le da al archivo es el instante en el que se creó. A continuación, se da comienzo a un bucle en segundo plano. En él, cada cierto tiempo, que por defecto es un segundo, se realiza una captura del espacio aéreo, es decir, se discretiza la circulación de las aeronaves. Esto es posible gracias al método getTraffics() de la base de datos, el cual devuelve una lista con todos los objetos Traffic presentes. Es importante darse cuenta de que la longitud de esta lista indica el número de aeronaves en el espacio aéreo en ese preciso instante. Entonces, se comienza a escribir el párrafo. Como ya se ha mencionado, la primera línea es distinta, pues incluye el tiempo actual seguido de la cantidad de aviones. Posteriormente, se recorre el vector y se van escribiendo los datos de cada avión en el orden que indica la tabla 6 y empleando, como bien se señalaba, el método getInfo().

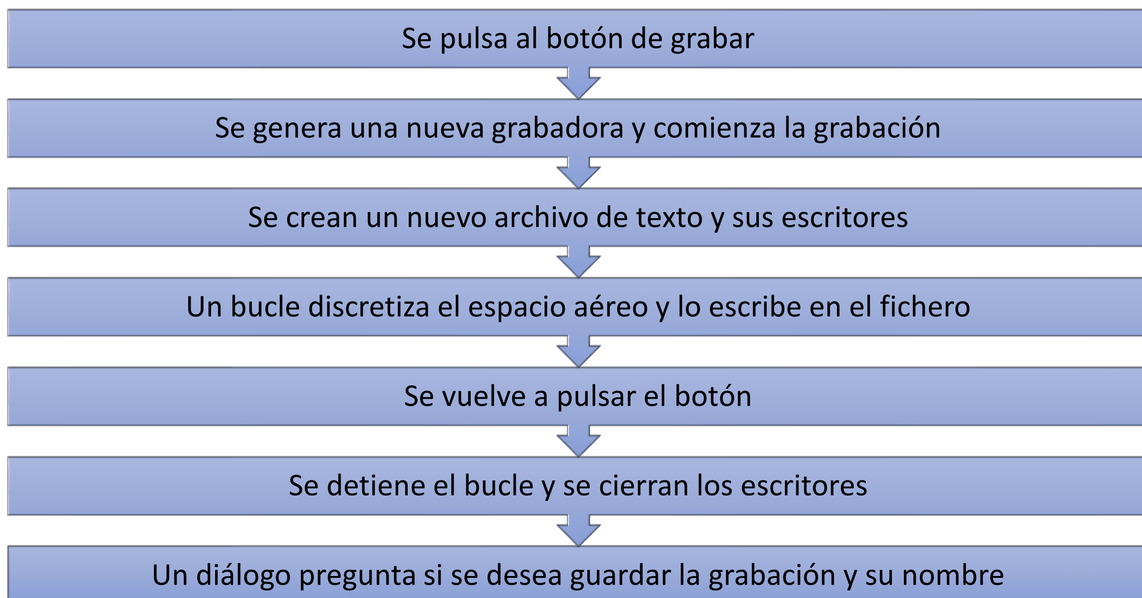

Figura 23, lógica la de creación de grabaciones

En Java, los directorios y archivos se representan de forma abstracta por medio de la clase File. En ella, existe el método createNewFile(), que, como indica su nombre, crea un nuevo fichero en el directorio que se le especifique. Por otra parte, esta clase es el argumento que necesita el constructor de FileWriter, a partir del cual se puede

construir un BufferedWriter. Esta última clase es el escritor que se ha empleado para añadir líneas de texto por medio de sus métodos createNewLine() y write().

Por último, se muestra en la figura 23 que volviendo a pulsar el botón con el que se comenzó la grabación, se detiene, se termina el bucle y se cierran los escritores. Entonces, aparece un mensaje en la pantalla (figura 28) que pregunta al usuario cómo le gustaría llamar al archivo. Por otra parte, también puede decidir si rechaza la grabación, para lo que pulsará sobre «Delete» en vez de «Save». No obstante, el funcionamiento de este diálogo, junto con otros presentes en la aplicación, se presentan en profundidad en la sección 6.3.2.

6.2.2. Reproducción

La siguiente parte del emulador conlleva la reproducción de las grabaciones, adquiridas conforme al método que se acaba de detallar o bien instaladas con la aplicación. En cualquier caso, el proceso es el mismo y muy similar (se vuelve a incidir en este aspecto) a la forma en la que se representaban los datos transmitidos por la antena. Del mismo modo que ocurría en dicho caso, en la pantalla del reproductor se crea un mapa sobre el cual se dibujan los distintos iconos. Además, de nuevo estos marcadores tienen la capacidad de ser seleccionados, conforme a la lógica que se describía en el punto 6.1.3 y que se plasma gráficamente en la figura 22.

Más allá de estas similitudes, cuando se carga esta pantalla, aparece un botón de *play* que la ocupa al completo. Pulsándolo se da comienzo a la lectura del archivo. Entonces, se crea una nueva base de datos TrafficMap que, a su vez genera un AntennaEmulator. Es en esta última clase es en la que se identifica el archivo a reproducir y se crea un lector para él. Si bien, para escribir sobre el fichero se empleaba un BufferedWriter, ya que presentaba un buen rendimiento, en este caso, para su lectura se utiliza un BufferedReader. Resulta que es el mismo tipo de lector que para las líneas de bytes de la antena. Así pues, es de gran utilidad el método que se comentaban en el apartado correspondiente, readLine(), mediante el cual se lee una línea del documento.

Pese a los puntos en común que se han destacado entre la recepción de datos de la antena y del fichero, la naturaleza del bucle que adquiere la información difiere. En este caso, se cuenta con la ventaja de que se conoce a priori la cantidad de aviones en un momento en concreto. Esto rebaja la carga de trabajo de la aplicación, pues no se tiene que dibujar el conjunto de aeronaves cada vez que se actualiza una de ellas. Al contrario, debido a este conocimiento, los iconos sobre el mapa se actualizan una vez se ha recorrido todo el conjunto y se han adaptado sus datos para el momento en concreto que se está representando. Este hecho hace que no sea del todo necesaria la existencia de una base de datos de la naturaleza de TrafficMap y, por supuesto, con un buen código se puede prescindir de TrafficCleaner. No obstante, debido a las limitaciones del trabajo estas dos últimas características no se adoptaron (se dejan como

trabajo futuro, sección 8), pues complicaban el código y, a pesar de no implementarlas, los resultados obtenidos cumplían con las solicitaciones de la aplicación.

Así pues, el conocimiento inicial de la cantidad de aviones conlleva la creación de un bucle interno en el principal que recorre el documento. En este nuevo se leen línea a línea todas las aeronaves de una cierta captura. Después de adquirir la información relativa a una de ellas, se procesa. Para ello, se llama a messageFILE(), método de TrafficMap. En él se lleva a cabo el mismo proceso que en messageTCP(), recordando que en este caso los campos de información se ordenan de distinta forma (tabla 6):

1. Se separa la línea por campos, teniendo en cuenta que cada uno de ellos se separa del siguiente con un espacio en blanco. De esta forma se crea una lista de String con los diferentes datos.
2. Se comprueba la existencia del avión que se está procesando en la base de datos.
3. Si no existía se crea uno nuevo y se añade. Si existía, se actualiza el que había y se comunica al TrafficMapListener.

Según lo que se ha comentado, el archivo se reproduce por medio de dos *loops*. Volviendo a la analogía de los párrafos en la que cada párrafo representa un instante de tiempo y cada línea una aeronave concreta, el bucle externo avanza párrafo a párrafo, mientras que el interno línea a línea. Así, el interno se detiene cuando llega a la última línea de un párrafo, mientras que el externo lo hace con el último párrafo del documento. Cuando esto último tiene lugar, se cierra el lector, se visualiza un botón de *replay* y se le comunica a la actividad que ya se ha acabado la reproducción. Con ello, se persigue el objetivo de deseleccionar cualquier avión que estuviese seleccionado y dejar de mostrar sus datos.

6.3. Interacción con el usuario

En este último apartado con respecto a la implementación de ATvision se analiza un aspecto sobre el que poco se ha comentado a lo largo del trabajo hasta ahora. Se trata de la interacción con el usuario. No obstante, antes de nada, resulta necesario definir y acotar lo que se entiende por «interacción con el usuario». Desde un punto de vista léxico, el término significa comunicación entre un cliente y otro ente. Este otro ente puede ser desde una persona hasta un objeto. En el caso en el que se está tratando el término, se refiere a un dispositivo móvil. Por lo tanto, en este sentido, para que la interacción resulte efectiva debe haber un entendimiento entre persona y dispositivo y un intercambio eficaz de información. Así pues, en la programación Android, la reciprocidad con el usuario de una app se establece por dos métodos básicos: la interfaz de usuario y la navegación.

A pesar de todo lo que se ha comentado, en este apartado el término se aborda de una manera más acotada. Así pues, se analizan aquellos mensajes que esperan una respuesta inmediata del usuario para que la aplicación pueda seguir su curso bien sea

por haber recibido información importante o por haberla mostrado. Además, se aborda el método por el que se le solicita el permiso de ubicación al usuario. Por último, una funcionalidad que caracteriza a la aplicación es poder realizar distintas acciones sobre las grabaciones. Este conjunto de opciones también se analiza en esta sección.

6.3.1. Solicitud de ubicación

La primera tarea que tiene lugar la primera vez que se ejecuta ATvision es la solicitud de los permisos de ubicación. Si se le conceden, se dibuja en el mapa una nueva capa en la que se muestra la ubicación del dispositivo, tal y como se puede ver en la figura 24. En este subapartado se repasa el código detrás de este mensaje que se muestra al usuario.

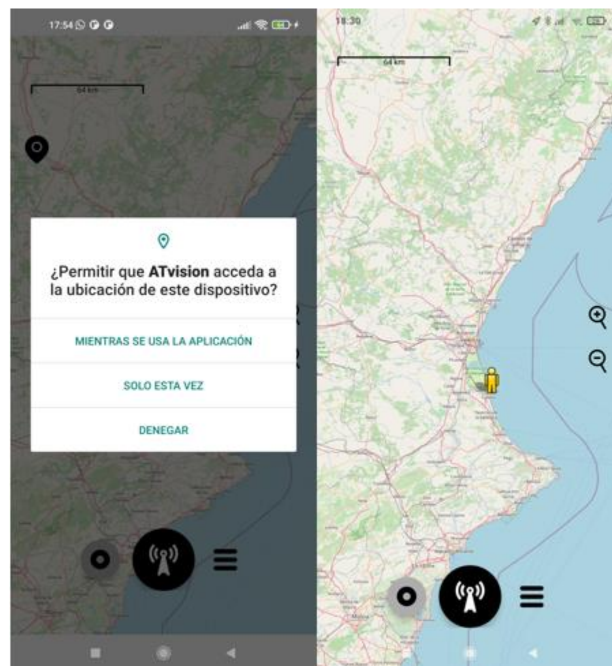

Figura 24, mensaje de solicitud del permiso y ubicación dibujada en el mapa

Tanto en la actividad principal, como en la del reproductor aparece un mapa en el que se dibuja la circulación aérea. Es por ello, que resultaría de gran interés para el usuario poder visualizar su posición actual con el fin de compararla con la de las aeronaves. En este sentido, cuando se crea la vista de cada una de estas actividades se comprueba si el permiso de ubicación está concedido. Esto es posible mediante un método estático de ContextCompat con el nombre de checkSelfPermission(). En él se introduce como argumento el permiso cuya concesión se quiere comprobar. En el caso que concierne de la aplicación entre manos, se puede referenciar por medio de la variable estática Manifest.permission.ACCESS_FINE_LOCATION. Así pues, tal y como se muestra en el diagrama de la figura 25, si no se ha dado la conformidad de acceder a la ubicación del dispositivo, se pregunta por ella. Esto es realizado de manera automática por Android mediante el método requestPermissions(). Si, por el contrario, sí que se había autorizado, se crea y añade al mapa la capa con la ubicación. Cuando se muestra

el diálogo que pide conocer la localización, se muestran tres opciones al usuario. La primera de ellas es aceptar la autorización, la segunda es denegarla y la tercera conlleva, a demás de la no conformidad, el deseo de que no vuelva a ser preguntado.

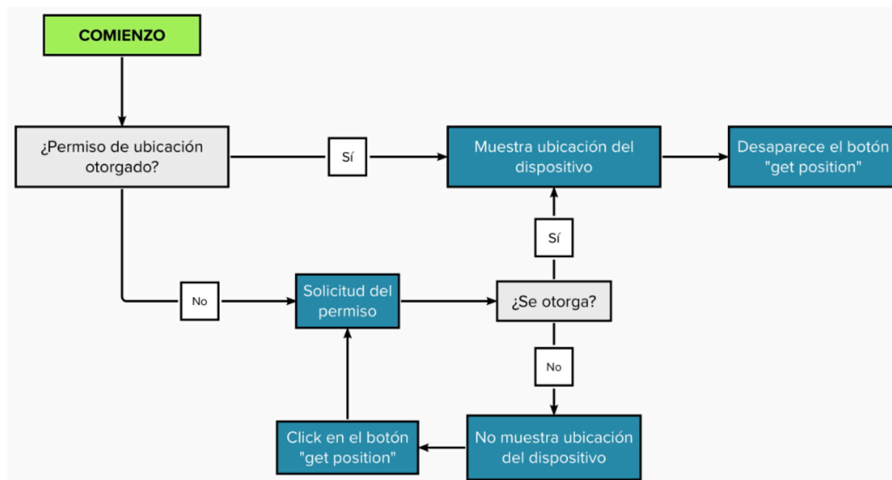

Figura 25, diagrama de flujo de la solicitud del permiso de ubicación

Por último, es necesario conocer la respuesta del usuario. Para ello se sobrescribe un método de AppCompatActivity: onRequestPermissionsResult. En él se crea un switch con el que se puede saber qué es lo que se ha seleccionado. Así pues, si ha aceptado, se crea y añade la ubicación del dispositivo al mapa. No obstante, en el caso de que se rechace, se muestra en la esquina superior izquierda de la pantalla principal un botón como el que se observa en la figura 26. Al interactuar con este icono, se vuelve a solicitar la ubicación y si esta vez se acepta, hace que desaparezca.

Figura 26, icono del botón que solicita el permiso de ubicación

6.3.2. Mensajes de diálogo

Un ejemplo de estos mensajes puede verse en la figura 27. En ella, se le muestra información importante al usuario y no se le ofrece posibilidad de elegir la respuesta. Su existencia fue introducida en la figura 10. En ella, se exponía que si se intentaba establecer un vínculo con la antena cuando no estaba disponible había de mostrarse un mensaje que alertase de este hecho. En él se informa de la no disponibilidad del servidor y no da más opción que emplear el emulador si se quiere seguir disfrutando de la aplicación. Así pues, pulsando sobre el botón «ACCEPT» el usuario es trasladado por medio de un Intent a la actividad de la librería, donde las capacidades funcionan como es habitual.

A nivel de código, el mensaje se crea por medio de la clase ConnectionAlertDialog, la cual extiende a DialogFragment, una clase que identifica objetos de fragmento que muestran diálogos. Por el hecho de pertenecer a ella, se debe sobrescribir el método onCreateDialog(). Este funciona como onCreate() para las actividades, por lo que en él se asigna la interfaz de usuario del diálogo y se define la respuesta a los eventos de click sobre sus botones. Resulta de gran interés la subclase de AlertDialog, Builder. Es ella la que configura el aspecto mediante el método setView() si se emplea una interfaz que no es la automática. En el caso de que se emplee la interfaz por defecto y no una definida por el programador, los métodos setTitle(), setMessage() y setNeutralButton() permiten personalizar lo que se enseña y así definir el mensaje como en la figura 27. Por último, existe la función dismiss(), que se llama desde dentro del diálogo para indicar que se cierra. Lo más habitual es invocarla como respuesta a la pulsación sobre un botón y tras algún proceso previo o, incluso, sin proceso previo. Otro método que tienen los objetos de diálogo es show(), por el cual se visualizan.

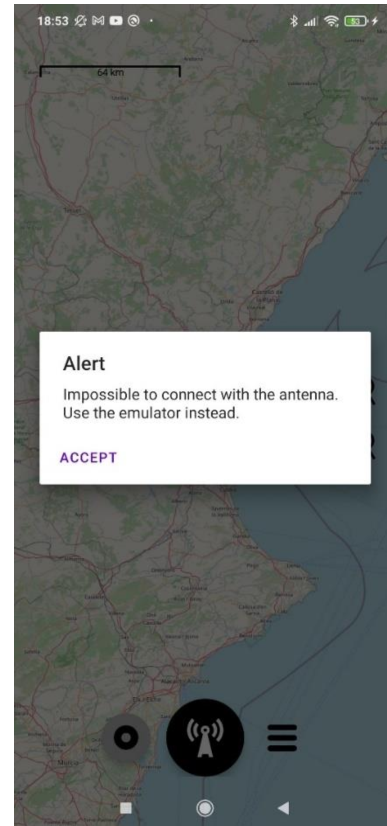

Figura 27, mensaje de alerta que informa de la indisponibilidad de la antena

A parte de la que se ha analizado, hay otras tres ventanas de diálogo que se muestran al usuario para pedirle que confirme algún dato y se pueden ver recogidas en la figura 28. A primera vista, se observa que la interfaz de este mensaje no se parece a la anterior, no es la automática que genera el constructor. Esto se realizó para dar un énfasis especial al mensaje de alerta, que es el único con esa apariencia, la cual además es fácilmente reconocible pues es común a muchas aplicaciones de Android. Sobre estos tres otros mensajes, previamente ya se ha ilustrado la forma en la que encajan en la lógica de la aplicación.

El de la izquierda se enseña cuando se termina una grabación. Pide al usuario que introduzca el nombre deseado para guardarla, aunque también puede decidir rechazarla y eliminarla. Debido a esto, se crea dentro de la clase AntennaRecorder cuando se sale del bucle y antes de cerrar los puertos. El constructor de este fragmento de diálogo, cuya clase es GuardarGrabacionDialog, necesita como argumento el nombre del archivo que se acaba de grabar. Este nombre, dado que aún no ha habido opción de cambiarlo es el automático y que corresponde al día y la hora en la que se ha creado. El hecho de pasar el objeto File al diálogo es porque dentro se define la respuesta a los eventos de *click* sobre sus botones. De esta forma, si se selecciona «DELETE», se llama a file.delete(), mientras que si es «ACCEPT» la opción elegida, se ejecuta file.renameTo() para cambiar el nombre al que haya escrito en el TextView. Los otros dos diálogos recogen cada uno una de las funcionalidades del que se acaba de explicar y que se encuentran en mayor detalle en el Anexo I.

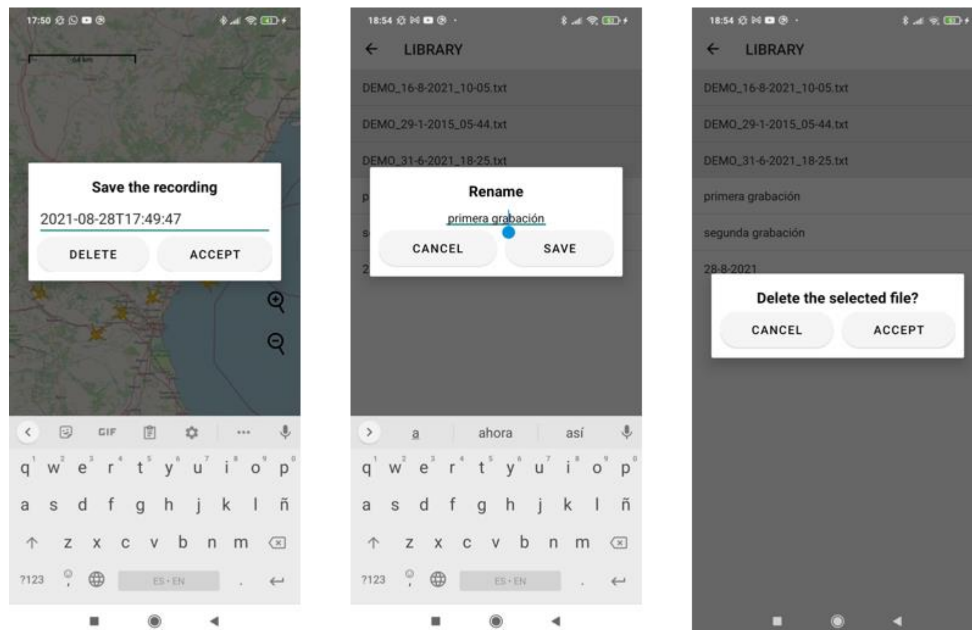

Figura 28, mensajes de diálogo que buscan una confirmación del usuario.

6.3.3. Menú contextual

Por último, se expone la funcionalidad de la aplicación que marca la diferencia en cuanto a la experiencia de usuario, a la vez que concede un nivel satisfactorio de usabilidad. Se trata del menú que, como se contempla en la figura 29, muestra distintas opciones al usuario sobre las grabaciones de la biblioteca.

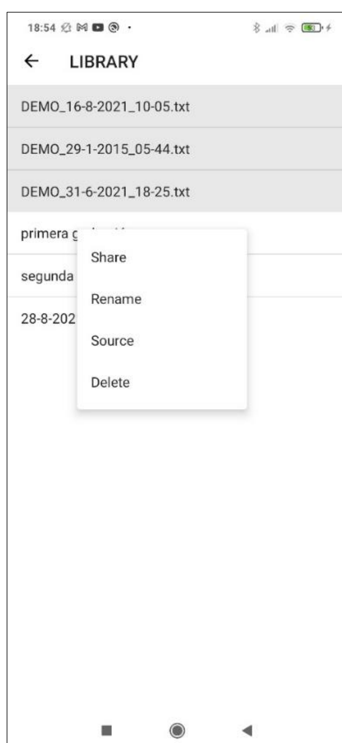

Figura 29, menú contextual sobre un fichero de la biblioteca

En primer lugar, cabe detallar la forma en la que se programa dicho menú para mostrarse sobre un elemento en concreto de la vista de lista. Con este fin, AppCompatActivity tiene una función pública llamada registerforContextMenu(), que recibe como argumento un objeto View y le asigna un menú contextual el cual se muestra con una pulsación prolongada sobre él. Así pues, para conferir a este desplegable las capacidades para las que se diseña, se sobrescriben onCreateContextMenu() y onContextItemSelected(). El primero de ellos responde al evento de creación y en él ocurre la importante acción de identificar el documento de la lista para el que se ha creado. Es necesario realizar esto porque, por medio de la estrategia que se ha comentado, las opciones se registran para un View, que en el presente caso es el ListView, es decir, toda la lista. Para llevar a cabo la especificación, se crea un AdapterContextMenuInfo, una clase de gran utilidad debido a que tiene una variable pública que indica la posición, dentro de la lista, en la que se genera el menú.

A continuación, es necesario que se responda ante las distintas entradas de la persona que interacciona con él. Esta es la utilidad del segundo de los métodos que se sobrescribe. OnContextItemSelected() permite obtener el número identificador de la selección, el cual se puede comparar fácilmente con el código de cada uno de los elementos del menú. En consecuencia, por medio de un *switch* es posible hallar la opción que desea llevarse a cabo:

- Si se selecciona «delete» o «rename» la intención es borrar o renombrar el archivo, respectivamente. Entonces, se solicita confirmar la acción por medio de los diálogos que se mencionaban en la sección 6.3.2 anterior y pueden verse en la figura 28.
- Si se pulsa sobre «share», lo que se desea realizar es compartir el archivo. Entonces, a nivel de código se genera un *intent*. Para ello se emplea la clase IntentBuilder, subclase de ShareCompat. Se trata de un ayudante para generar una acción de envío de archivos. Algunos de sus métodos útiles son setType() y setStream(). En el primero de ellos se especifica el tipo del archivo que se comparte, que en la presente aplicación es *txt*. El segundo, define el archivo que se comparte. Entonces, cuando se ejecuta el *intent* se muestra una interfaz (figura 30) para que el usuario elija la aplicación con la que desea completar la acción.
- En último lugar, se puede actuar sobre «source». Esta opción lleva a otra pantalla en la que se lee la fuente que genera la grabación. Dicha pantalla se trata de ReadSourceActivity, por lo que para empezar esta actividad se utiliza un nuevo *intent* que lleve a ella e incluya el nombre del documento que se solicita mostrar. Este nombre es recibido en la nueva actividad y a partir de él se crea un objeto File. Entonces, se crean un flujo de entrada que recoge los bytes del archivo, y otro de salida, en el que se escriben. Se va recorriendo el fichero línea a línea, de forma que se pasen todos los datos al OutputStream. Cabe mencionar que a veces este proceso conlleva un periodo de tiempo perceptible por el usuario. Para que el cliente no caiga en la errónea creencia de que la aplicación se ha bloqueado, se genera un ProgressBar que le transmite que está pensando. Una vez se alcanza el final del fichero, se interpretan los bytes y se escriben en el TextView de la interfaz.

Figura 30, selección de aplicación para compartir el archivo

8. Conclusión

En las páginas previas se ha detallado el proceso de la creación de la aplicación ATvision. Se han comentado las dificultades encontradas y la forma en la que el estudiante ha sabido superarlas para crear un programa capaz de monitorizar y grabar el tráfico aéreo. Los pasos que se siguieron, si bien en un comienzo se encontraban difuminados, poco a poco fueron cobrando claridad y marcaron un camino en el que cada proceso llevaba al siguiente y cada implementación se basaba en las previas. Ver materializándose como, poco a poco, las líneas de código cobraban vida en el dispositivo móvil dotaban de una inspiración y una energía con la cual poder superar los contratiempos que iban surgiendo por el camino.

De este modo, mediante la completa elaboración del proyecto se ha obtenido un conocimiento profundo de las técnicas que emplean los controladores aéreos para monitorizar el tráfico. Se ha podido entender la importancia de buscar alternativas y nuevas tecnologías que permitan un completo y seguro desarrollo de la industria aeronáutica. Además, se han estudiado las aplicaciones a nivel comercial que tiene esta tecnología y se ha podido ver como un estudiante de ingeniería, con los conocimientos de final de carrera, ya está preparado para desarrollar proyectos con fines lucrativos. El presente trabajo también ha servido para indagar en la programación Java y Android y para adquirir capacidades muy valiosas en el mundo laboral actual. Además, se ha disfrutado mientras se expandían los conocimientos en estas áreas y a medida que se adquiría un nivel de conocimientos que, a través del programa del grado de Ingeniería Aeroespacial de la UPV seguido por el alumno, no podrían haber sido adquiridos.

A lo largo del recorrido académico del autor se han empleado recursos de enseñanza que despertaban un gran interés y que resultaban realmente útiles para comprender ciertos conceptos. Es por ello por lo que se ha disfrutado en la creación de un software que puede ayudar a futuros estudiantes de este mismo modo. Se ha creado una *app* capaz con tales fines y de fácil acceso portátil. Además, también puede despertar el interés de los entusiastas y aficionados a conocer la circulación de las aeronaves por el espacio aéreo.

En último lugar, afrontar un proyecto de tales dimensiones es una experiencia que aún no se había llevado a cabo y de la que se han podido extraer numerosas y valiosas enseñanzas. Así pues, se han adquirido competencias relativas a la organización de un proyecto, a la definición de sus objetivos y a su planificación y administración del tiempo. El pensamiento crítico y creativo ha sido esencial para encontrar soluciones a cada uno de los problemas que han ido surgiendo por el camino. Por otro lado, la redacción de la memoria ha terminado de definir a este proceso de aprendizaje como integral, en el que el estudiante ha podido desarrollarse en multitud de ramas del saber.

9. Trabajos futuros

Un trabajo de diseño e implementación de software como el que aquí se ha presentado siempre conlleva un desarrollo constante. Son numerosas las actualizaciones que se crean para cada programa periódicamente. Así pues, una aplicación con un cierto ímpetu de reconocimiento no deja de crecer para ofrecer nuevas funcionalidades y mejorar la experiencia de sus usuarios. Por otra parte, un creador con ambición de que su producto se diferencie siempre encontrará maneras para mejorarlo y adaptarlo a nuevas necesidades.

Por ello, el presente trabajo podría haber adquirido nuevos enfoques, haber evolucionado por otros caminos o haber indagado en su propia forma. Pero había que concentrar los esfuerzos hacia los requerimientos iniciales para construir una aplicación útil y capaz de alcanzar los objetivos para los que se ideó. No obstante, una vez alcanzado dicho punto se abre un mundo de oportunidades para el futuro desarrollo del programa. Los más relevantes de estos enfoques se listan en el presente capítulo.

En primer lugar, se encuentran las carencias a nivel del código que se ha expuesto en las páginas anteriores. Especialmente, se han recalcado la ineficiente forma en la que la representación se actualiza cada vez. Por otra parte, tampoco es óptima la forma en la que se recorre el archivo grabado y se actualiza la base de datos a partir de él. En cuanto a lo primero, en futuros trabajos que soliciten un mejor uso del código, se puede crear un método en el que solamente se vuelva a pintar el marcador cuyos valores se han actualizado con una transmisión de la antena. Así, se evitaría tener que representar alrededor de setenta iconos cada vez, número que se vería reducido a la unidad. El segundo aspecto también podría suponer una considerable reducción de la cantidad de trabajo. Dado que se recorre todo un párrafo del documento y se dibujan todos sus aviones al mismo tiempo cuando se termina de recorrer, no es necesaria la existencia de una base de datos como TrafficMap para el emulador y, mucho menos, el limpiador TrafficCleaner. Como alternativa, se podría crear una lista nueva cada vez con todos los aviones y representarse.

Estos son solo dos ejemplos de toda la limpieza y orden que podría realizarse con respecto al código. Otro aspecto que podría ofrecer una característica de gran interés es respecto al método en el que el grabador está configurado. Se podría escribir el código de tal forma que se pudiese captar el tráfico aéreo incluso sin estar en la aplicación. A continuación, se mencionan nuevas capacidades que se le podrían otorgar al proyecto para aumentar su atractivo hacia los usuarios, las cuales no son pocas, si bien solamente se definen las más relevantes.

- Posibilidad de poder modificar el servidor y el puerto con el que se establece conexión y del que se recogen datos. Esta técnica permitiría extender los clientes a cualquier usuario que disponga de un receptor de señales Modo-S y ADS-B. Solamente sería necesario configurar la aplicación para la nueva antena y se mostrarían las aeronaves que cubre.

- Como extensamente se ha mencionado a lo largo de esta memoria, la utilidad de compartir los archivos en los que se graba el tráfico aéreo es poder analizarlos posteriormente en un software que lo facilite. De este modo se pueden obtener gráficas y estadísticas que resultan de gran interés tanto para los estudiantes como para los entusiastas. Se piensa que sería de gran valor que la aplicación tuviese la capacidad propia de realizar algunos de estos análisis y mostrar sus resultados.
- Poder seleccionar más de una aeronave y mostrar datos relativos entre ambos vuelos. Esta modificación sería fácilmente implementable modificando el algoritmo de drawPlanes() y trafficUpdate(). Por otra parte, los datos relativos se podrían obtener por medio de una nueva clase que con sus métodos realizase los cálculos.
- Aumentar los datos de las aeronaves, así como crear una historia del vuelo de esta. Un atributo común entre muchos de los programas de monitorización de vuelos para el público genérico es que, cuando se selecciona una aeronave, aparece el recorrido que ha realizado. Además, se hallan otros datos que la aplicación recoge desde bases de datos, como bien pueden ser la procedencia y el destino del vuelo o características físicas del avión. Además, se podría habilitar el acceso a otra base de datos para recoger y mostrar sobre el mapa aeropuertos y radioayudas para la navegación.
- Poder llevar a cabo distintos servicios en segundo plano sin estar mostrándose la aplicación. Por ejemplo, se podría reescribir el código de la grabadora del tráfico aéreo para seguir con las grabaciones, aunque el usuario no esté dentro del programa. Otro caso que se podría desarrollar es analizar los vuelos que pasan por la situación del dispositivo o que el cliente ha indicado que son de interés. De esta forma, se habrán de mostrar notificaciones para indicar que la grabadora está en marcha, que hay un vuelo cerca o datos relevantes a cerca de los vuelos de interés.
- Realidad aumentada. Esta funionalidad, con la que cuentan aplicaciones Android como SkyMap o FlightRadar24, conlleva una interacción directa con la cámara del dispositivo. De esta forma reconoce los objetos que se están enfocando, que en el caso que aquí se desarrolla son aviones volando en el aire. Entonces, el código reconoce estos aviones por medio de un cálculo que consiste en hallar hacia que dirección se está enfocando y desde donde se está haciendo. Para obtener el vector en la dirección en que se está enfocando sería necesario acceder a los sensores internos del dispositivo. Entre ellos se encuentra el de gravedad, que ofrece el vector gravedad en un sistema de referencia fijado al dispositivo móvil. De este modo se puede comparar el vector entre cierto avión y el usuario con el vector que indica hacia dónde se señala.

Referencias

[1] «Nuestras tarifas». *Orange* [en línea].
Disponible en: https://www.orange.es/promociones-ofertas. Accedido el 10 de agosto de 2021.

[2] «Kinetic SBS-1 Virtual Radar». *Pilot warehouse* [en línea].
Disponible en: http://shop.pilotwarehouse.co.uk/. Accedido el 10 de agosto de 2021.

[3] «Microsoft 365 Personal». *Microsoft* [en línea].
Disponible en: https://www.microsoft.com/. Accedido el 10 de agosto de 2021.

[4] «¿Cuál es el precio hora de un programador?». *Kaira* [en línea].
Disponible en: https://kaira.es/cual-es-el-precio-hora-de-un-programador/. Accedido el 10 de agosto de 2021.

[5] «Air traffic management surveillance tracker and server». *EUROCONTROL* [en línea].
Disponible en: https://www.eurocontrol.int/product/artas. Accedido el 11 de agosto de 2021.

[6] «SACTA». *ENAIRE* [en línea].
Disponible en: https://www.enaire.es/servicios/gestion_de_transito_aereo. Accedido el 11 de Agosto de 2021.

[7] «Aurora». Adacel [en línea].
Disponible en: https://www.adacel.com/air-traffic-management. Accedido el 11 de agosto de 2021

[8] «How flight tracking works». *FlightRadar24* [en línea].
Disponible en: https://www.flightradar24.com/how-it-works. Accedido el 11 de agosto de 2021.

[9] Rodrigo A., Cabrales A., «Virtual radar client (VRC 1.1) manual del usuario. Introducción», *VRC.*

[10] S. Bose, A. Kundu, M. Mukherjee y M. Banerjee, «A comparative study: Java vs Kotlin programming in Android application development», International Journal of Advanced Research in Computer Science, vol. 9, no. 3, mayo-junio 2018.

[11] P. Chawla. «OOP with C++».7

[12] M. Báex, A. Borrego, J. Cordero, L. Cruz, M. González, F. Hernándex, D. Palomero, J. R. de Llera, D. Sanz, M. Saucedo, P. Torralbo, A. Zapata, «Introducción a Android», UCM: E.M.E. Editorial

[13] «Guía para desarrolladores de Android». Android [online].
Disponible en: https://developer.android.com/docs?hl=es-419. Accedido el 27 de agosto de 2021.

[14] M. Haklay y P. Weber, «OpenStreetMap: User-Generater Street Maps», *IEEE Pervasive Computing*, vol. 7, no. 4, pp. 12-18, octubre-diciembre 2008.

[15] «Android». *Open Street Map wiki* [en línea]
Disponible en: https://wiki.openstreetmap.org/wiki/Android. Accedido el 27 de agosto de 2021.

[16] «Gradle User Manual», versión 7.2
[17] R. Andrew, «Air Traffic Surveillance – Building Capacity and Enhancing Safety», *ResearchGate Measurement and Control*
[18] W. Stamper, «Understanding mode S technology», *RF Design*: diciembre 2005
[19] H. V. Sudarshan, «Sistema mundial ATM, tecnología y operaciones en taller sobre el desarrollo de un caso de negocio para la implantación de los sistemas CNS/ATM», Lima, 2008
[20] «Kinetic SBS-1 Mk2 Real Time Virtual Radar». *Javiation* [en línea]. Disponible en: https://www.javiation.co.uk/sbs-1.html. Accedido el 28 de agosto de 2021.

Bibliografía

Göranson. *Efficient Android threading*. 1ª edición. Sebastopol: O'Reilly Media, 2014.
J. M. L. de Guevara. *Fundamentos de programación en Java*. Facultad de Informática Universidad Complutense de Madrid: Editorial EME.
J. Sánchez. *Java 2*. 2004
K. L. Calvert y M. J. Donahoo. *TCP/IP Sockets in Java, practical guide for programmers*. EEUU: Academic Press, 2002

Anexo I

Detalle de las clases e interfaces de la aplicación

Este anexo sirve como profundización en la naturaleza del código del proyecto. Se entra, desde un punto de vista más técnico, en los aspectos más importantes de la programación de las distintas clases e interfaces. El objetivo a es que sirva como complemento a todo lo redactado con anterioridad y ayude a comprender mejor ciertos aspectos de las explicaciones. Con este objetivo, se comienza describiendo brevemente cada clase e interfaz. A continuación, se exponen aspectos de relevancia sobre la forma en la que su código está estructurado, haciendo hincapié en las funciones de los distintos métodos que componen cada una de ellas. En primer lugar, se mencionan las actividades que componen la aplicación. Posteriormente, sus interfaces. Seguidamente, se detallan otras clases cuya función es notoria en la lógica del programa. Es importante recalcar que algunas de estas clases se basan en ciertos programas de la asignatura «gestión de vuelos por computador» que tiene lugar en el segundo curso del máster en Ingeniería Aeronáutica impartido en la UPV. Se menciona pues a los autores Ángel Rodas Jordá y Joan Vila, si bien dichas clases fueron adaptadas a las solicitaciones de Android.

MainActivity

Es la actividad principal. Como todas, extiende a AppCompatActivity. Además, implementa las interfaces MarkerListener y TrafficMapListener, cuyos métodos implementa y que se definirán posteriormente.

En ella, se sobrescriben las funciones de llamada de diferentes tramos de su ciclo de vida, que se recogen en la figura 3: onCreate(), onResume(), onPause() y onStop(). En onPause(), si se estaba obteniendo la ubicación del dispositivo, se deshabilita su obtención. De la misma forma, en onResume se vuelve a habilitar. En la última función de llamada del ciclo de vida de la actividad, onStop, si había conexión con la antena, se desconecta. Al igual, se deja de grabar si se estaba grabando y se deselecciona el avión que estuviese seleccionado. Específicamente, En onCreate(), se definen los métodos de respuesta a distintos eventos sobre los elementos de la interfaz de usuario asociada. Dentro de cada función de escucha se especifica, además, la respuesta a cada acción. A continuación, se mencionan los más relevantes. Además, por estar definida como TrafficMapListener, sobrescribe a removeTraffic(), updateTraffic() y finished(). Por último, como también hereda de MarkerListener, tiene que implementar a showInfo() y a hideInfo().

btnAntenna.onClickListener

Responde a una pulsación sobre el botón que establece una conexión con la antena. Un valor booleano indica si había establecida conexión previa o no. Así, si no la había, la realiza. Se crea un nuevo TrafficMap al que se le pasa el servidor y el puerto de la antena y se comienza a recibir información a través de su método startit(). Si, por el contrario, la conexión estaba establecida, se desconecta y se deja de recibir información llamando a stopit(). También se distingue un periodo en el que aún no se está vinculado, sino que se está en proceso. En dicho caso, no se obtiene respuesta ante el evento de click. Además, cuando se comienzan a recibir datos, es necesario comprobar que no haya ningún marcador de la conexión anterior seleccionado, pues crearía una excepción.

btnRecord.onClickListener

En este caso, se emplea el mismo valor booleano que indicar si hay enlace con la antena. Así pues, si no lo hay o se está completando, no se obtiene ninguna respuesta. Por el contrario, si hay vinculación, se comienza o deja de grabar según corresponda, lo cual determina un nuevo booleano que indica si se está grabando. Cuando se da comienzo a la grabación, se crea una instancia de la clase AntennaRecorder. Al constructor de dicha clase se le pasa el TrafficMap a partir del cual se crearán las distintas capturas de los aviones y el intervalo de tiempo entre discretizaciones, que por defecto es un segundo. Llamando a startit(), se comienza a grabar. Para terminar se llama a stopit(). Además, para mejorar la experiencia de uso, hace falta que cuando comience la grabación se deseleccione cualquier avión que estuviese seleccionado y se dejen de mostrar sus datos.

btnLibrary.onClickListener

Siempre que se interactúe con este botón sin estar proceso de conexión, momento en el que estará bloqueado, se comienza la actividad de la librería por medio de un *intent*.

btnGetPosition.onClickListener

Cada vez que se pulse este botón, si es que se está mostrando, se solicita el permiso de ubicación. En la sección 6.3.1 se muestra más detalle sobre cómo funciona esta lógica

removeTraffic()

Este método se llama cuando se borra un avión del mapa. Lo único que se realiza dentro de él es comprobar si este avión era el que estaba destacado. En dicho caso, se deja de mostrar su información y ya no se especifica como la aeronave seleccionada

updateTraffic()

Se ejecuta cuando se actualizan los datos de algún avión, de nuevo, para comprobar si es del que se está mostrando información. En tal caso, sus datos, que se encuentran visibles, se actualizan.

finished()

sirve para evitar problemas cuando se termine la recepción de datos por un error de conexión con la antena o alguna excepción en la transmisión. En él se instancia e invoca un fragmento de diálogo llamado ConnectionAlertDialogFragment (ver el capítulo 6.3.2) y se termina de grabar, se desconecta y se deja de mostrar información si se estaba realizando alguna de las cosas anteriores.

showInfo() y hideInfo()

Ambos son métodos de llamada de un evento de pulsación sobre alguno de los aviones que se dibujan en el mapa. El primero de ellos recibe el nombre del avión seleccionado, lo define como tal y muestra su información en el FragmentContanerView. El segundo se invoca para dejar de visualizar el fragmento y los datos de la aeronave. Por otra parte, en él se deselecciona dicho marcador.

EmulatorLibraryActivity

En este caso solo se sobrescribe la función de llamada onCreate(). En ella, se obtiene el nombre de todos los archivos «demo», que se encuentran en los assets del proyecto, y los creados por el usuario, que están en una carpeta del almacenamiento interno. Estos nombres se guardan en dos ArrayLists separados, cada uno de los cuales se carga en su vista de lista correspondiente. Para cada ListView, se define un onClickListener de forma que al clicar sobre un elemento de las listas se genere un Intent que dé comienzo a la actividad del emulador. Además, también se pasa el nombre del archivo seleccionado, para poder conocerlo en la siguiente actividad. Por último, el ListView se vincula a un menú contextual, que se muestra al usuario cuando se realiza una pulsación larga, mediante el método registerForContextMenu().

Existen un par de métodos más que se sobrescriben y que están relacionados con el menú contextual. El primero, onCreateContextMenu, vincula la interfaz del menú y obtiene el elemento de la lista sobre el cual se está mostrando. El segundo es onContextItemSelected. Aquí, primero se tiene que hallar la acción seleccionada. Una vez se conoce, se realizan las acciones correspondientes, como bien se detalla en la sección 6.3.3.

EmulatorActivity

El funcionamiento de esta actividad es muy similar al de la principal. Esta también se define como MarkerListener y TrafficMapListener. No obstante, no cuenta con botón de conexión, de grabación ni de biblioteca. Sí que incluye, no obstante, un botón que ocupa toda la pantalla y da comienzo a la reproducción. Así pues, en onCreate se crea un método de escucha para la pulsación sobre este botón, de forma que cuando se actúe sobre él se cree un nuevo TrafficMap. En esta tarea se emplea la propiedad de

polimorfismo, pues, en contraste con la forma en la que se instanciaba en la MainActivity, ahora se le pasan como argumento, entre otros, el nombre del fichero del que se leen los datos. Este se recibe como extra del Intent que da comienzo a esta actividad desde la EmulatorLibraryActivity. Entonces, una vez creada la base de datos TrafficMap, se comienza la reproducción mediante, como se llevaba a cabo en la actividad principal, el método startit().

Al igual que ocurría en la MainActivity, esta clase implementa las interfaces MarkerListener y TrafficMapListener. Por ello, tiene que implementar todos sus métodos, los cuales tienen el mismo papel que en la actividad principal. No obstante, finished() adquiere una nueva función. En este caso, este método sirve para cuando se termina de reproducir la grabación. Si había un avión seleccionado, se deselecciona y se deja de mostrar su información.

ReadSourceActivity

Como se introduce en la sección 5.2.4, esta actividad muestra el archivo de texto fuente de una grabación en pantalla y permite navegar por él mediante *scroll*. Para ello, la interfaz asociada tiene un ScrollView que acoge un TextView. En dicho TextView se carga el texto del archivo que se especifique. Al igual que en la actividad anterior, la del emulador, el nombre del archivo se obtiene como un extra mediante el Intent que abre esta nueva actividad. Conociendo este nombre, se puede abrir el archivo y generar un BufferedInputStream que permita ir leyéndolo y escribiendo los datos en un ByteArrayOutputStream. Una vez se ha atravesado todo el archivo, se muestra en el TextView.

Antenna

Esta interfaz es implementada por AntennaEmulator, AntennaReceiver y AntennaRecorder y define tres métodos:

- startit() que, como ya se ha mencionado, se utiliza para comenzar una transmisión de datos por medio de una iteración.
- stopit(), el cual termina dicha iteración y transmisión.
- isAlive(). Esta última función devuelve un valor booleano, que será positivo si la antena está funcionando y negativo en caso contrario.

TrafficGenerator

La única clase del proyecto que implementa esta interfaz es TrafficMap. En ella se entiende mejor el papel de cada uno de estos dos métodos:

- getTraffics() que devuelve una lista con objetos de la clase Traffic[].

- getMarkers(). Esta vez se devuelve una lista de objetos de la clase PlaneMarker[].

TrafficMapListener

La función de los métodos de esta interfaz ya ha sido explicada con anterioridad, pues las clases que la implementan son MainActivity y EmulatorActivity. Estas, se convierten en oyentes ante las acciones de quitar o añadir un objeto Traffic al TrafficMap o terminar de transmitir datos, de acuerdo con los métodos:

- removeTraffic()
- updateTraffic
- finished()

MarkerListener

La función principal de esta interfaz es la misma que la anterior, responder ante eventos. Así, las clases que la implementan, que de nuevo son MainActivity y EmulatorActivity, que se convierten en oyentes que responden a llamadas por medio de estos métodos:

- showInfo()
- hideInfo()

Traffic

Sus objetos representan los aviones que la antena detecta y que posteriormente son dibujados. Por tanto, sus variables son las distintas propiedades de la aeronave que la antena transmite. Estos campos pueden verse reunidos en la tabla 3.

El constructor de esta clase toma como argumento un vector de String llamado *values*, en cada posición del cual hay una característica del avión y, en consecuencia, de Traffic. Así pues, el constructor recibe este vector e invoca un método público de esta misma clase: update(). De esta forma, se actualizan las variables de la clase en función de los valores que el *values* ofrece, lo cual depende del tipo de mensaje. También existe el método updateAll(), al que se le introduce como argumento, de nuevo, un vector de String y que actualiza todos los valores de Traffic sin tener en cuenta el tipo de mensaje recibido.

Otros métodos públicos que son de importancia y se definen en esta clase son los siguientes:

- toString(), devuelve el código hexadecimal de la aeronave.
- getInfo(), da como resultado un único String donde las propiedades de la aeronave se encuentran ordenadas según el estándar de los documentos de texto empleados para el emulador. En el capítulo 6.2.1 se entra en más detalle en esto.

PlaneMarker

Representa cada uno de los marcadores de avión que se dibujan sobre el mapa. Lo más interesante de esta clase es que extiende a OverlayItem y puede emplearse para crear una ItemizedIconOverlay. El constructor toma un objeto Traffic como argumento y crea un OverlayItem a partir de él. Mediante el método getTraffic() se devuelve el objeto. Una última función es toString(), cuyo resultado es el mismo que en la clase Traffic.

Cabe comentar que esta clase, a nivel operacional, resulta innecesaria pues podría simplemente Traffic extender a OverlayItem. No obstante, es de gran ayuda al comprender y emplear de una mejor forma el código.

TrafficMap

Una vez definido Traffic se puede entender el significado de la clase TrafficMap y su importancia. Esta es una clase que extiende a ConcurrentHashMap y mapea claves String con valores PlaneMarker. Actúa como base de datos en la que se almacenan todos los marcadores que representan los aviones dibujados en el mapa y permite acceder a ellos por medio de su clave. Esta clave, como ya se ha indicado, es un String, en concreto es el código hexadecimal de dicho avión.

La clase cuenta con dos constructores para dos necesidades completamente distintas. El primero de ellos es el que se emplea en la actividad principal. Toma como argumentos, entre otros, el servidor y el puerto, para la conexión con la antena, y el TrafficMapListener. En este caso, dentro de esta función se genera una instancia de AntennaReceiver y otra de TrafficCleaner. El segundo, se emplea en la EmulatorActivity. El constructor difiere del otro en el hecho de que ahora no se introducen como argumentos el servidor y el puerto, sino el nombre del archivo que se quiere leer. Además, en este caso se instancia a AntennaEmulator en vez de a AntennaReceiver.

Como ya se ha indicado, la única clase que implementa la interfaz TrafficGenerator es esta. Como tal, tiene que sobrescribir todos sus métodos, que son, getTraffics() y getMarkers(). Por medio del primero de ellos es posible obtener una lista con todos los objetos de Traffic que hay en la base de datos. El segundo, por otra parte, devuelve una lista con los marcadores.

Además, existen otros métodos que resultan de gran utilidad:

- startit() da comienzo al trabajo de la antena y el limpiador. Para realizar esto, llama al método startit() de cada uno de ellos.
- stopit() realiza la misma tarea pero para que antena y limpiador dejen de funcionar.
- messageTCP() toma como argumento un String que representa el mensaje enviado por el servidor sin procesar. Es en esta función en la que se procesa, es decir, se separa en campos y convierte en un vector, se comprueba si ya se había recibido información acerca de esta aeronave y se crea o actualiza dicha información según convenga. Además, después de todo esto, se llama al método drawPlanes() de esta misma clase y, en el caso de haberse actualizado los datos de alguna aeronave, se le comunica al TrafficMapListener.
- messageFILE() actúa de la misma manera que messageTCP(), solo que en este caso se procesan los mensajes recibidos por medio de un archivo de texto. Esta función se emplea para reproducir grabaciones y la gran diferencia con respecto a la otra es que, en esta, no se dibujan los aviones.
- drawPlanes(). Como ya había ocurrido en otra ocasión, a nivel funcional, este método no es necesario. En él, simplemente se llama al método de PlanesMap drawPlanes() al que se le introduce una lista con todos los valores del HashMap. No obstante, aumenta la limpieza del código.
- cleanLostFlights(), esta es el método que tiene el objetivo de eliminar de la base de datos aviones de los que no se obtienen actualizaciones. TrafficCleaner llama a esta función cada cierto tiempo. En ella se genera una enumeración con todas las claves que se encuentran presentes en la base de datos. Se recorre esta enumeración y se comprueba el tiempo que ha transcurrido desde la última actualización de cada marcador. Si es superior a un minuto, se elimina del HashMap. Cuando esto ocurre, este método también se encarga de avisar al oyente.

PlanesMap

Aquí se realizan todas las tareas sobre el MapView mediante sus métodos. Entre sus variables destacan tres estáticas que representan los iconos de los aviones: por defecto, seleccionado y cuando se está grabando. Otra variable que es de gran relevancia es un String público en el que se guarda el nombre del avión seleccionado. Los argumentos de su constructor son el MapView, el contexto de la aplicación y el MarkerListener. Además, en dicho constructor se realizan automáticamente ciertas acciones sobre el mapa. De este modo, se obtiene su controlador, se define la fuente de los mosaicos (*tile source*), se habilitan las cualidades *multi-touch* y se definen los límites del mapa en cuanto a zoom y navegación.

Todos los métodos públicos de esta clase añaden alguna capa al mapa, salvo dos. El primero de ellos, setMarkerOverlayIndex(), define el nivel en el que debe estar la capa en la que se dibujan los aviones. Se creó este método para que siempre que se dibujasen

los iconos estuviesen por encima de todas las capas. El segundo, setDefaulViewPoint() define el punto inicial en el que está centrado el mapa cuando se carga.

El resto de funciones son tres, de las cuales la más relevante es, sin duda, drawPlanes(). En lo relativo a las otras dos, drawScale() añade la capa en la que se observa la escala, mientras que drawZoomButtons(), dibuja los botones de zoom. La función que dibuja la escala además define su posición en la esquina superior izquierda. Por otra parte, los botones de lupa no son los habituales de la clase MapView. Se definieron unos con un icono propio, que resultaba más estético y que se dibujan en la parte derecha del mapa. En el mismo método, se explicita la respuesta a los eventos de pulsación sobre cada uno de ellos, la cual es, respectivamente, aumentar o disminuir la escala. Por último, drawPlanes(), que se menciona en gran detalle en la sección 6.1.3, es el método encargado de dibujar la capa con los iconos de los distintos aviones. Para ello, toma la información presente en TrafficMap y crea un ItemizedIconOverlay. Se define la respuesta a un click sobre alguno de los iconos. Por último, una vez creada la capa, se recorre cada uno de los marcadores para diferenciar los iconos. Estos se rotan para estar alineados con la derrota o se cambian de color según corresponda. Una vez ya se ha realizado todo lo anterior, se añade esta capa mediante el OverlayManager del MapView y se actualiza el mapa con el método invalidate().

Un último método presente en esta clase es rotateDrawable. Especificado como privado, no se puede acceder a él desde fuera de PlanesMap. Su objetivo es rotar los iconos, conforme ya se ha indicado. Emplea como recurso la clase de Java RotateDrawable.

AntennaReceiver

Esta clase, que extiende a Thread e implementa a Antenna, se emplea para comunicarse con la antena asíncronamente y recibir de ella los datos de las diferentes aeronaves que se encuentran en el espacio aéreo. Su constructor recibe el nombre del servidor y el puerto, para poder crear un socket. También recibe el TrafficMap, al cual llama para que analice los datos recibidos, y el TrafficMapListener, que escucha cuándo se interrumpe la comunicación.

Por ser Antenna, implementa los métodos startit() y stopit(), cuya función ya han sido explicada repetidamente con anterioridad. Encima, como extiende a Thread, se tiene que sobrescribir el método run(). En él, se definen las acciones que se realizan en segundo plano y que se comentan en detalle en el capítulo 6.1.1. En resumidas cuentas, se genera un lector capaz de recibir los bytes procedentes de la antena y se comienza un bucle en el que se van leyendo y analizando líneas. Cuando se sale del bucle, se cierran las comunicaciones.

AntennaEmulator

Como en el caso anterior, ahora también se extiende a Thread e implementa a Antenna. Esto implica que existan funciones startit(), stopit() y run() con el mismo objetivo que antes. Esta clase, no obstante, no se instancia para establecer una conexión TCP, sino para vincularse a un archivo de texto y leerlo. Debido a esto, el constructor recibe como datos el nombre del archivo, los segundos de discretización entre lecturas y, como en AntennaReceiver, el TrafficMap y TrafficMapListener.

AntennaRecorder

De nuevo, se extiende a Thread e implementa a Antenna. En este caso run() define que en un nuevo hilo se ejecute una lógica capaz de crear documentos de texto con los datos de las aeronaves conforme se especifica en la sección 6.2.1. En esta ocasión, el constructor recibe a TrafficMap para obtener datos de él en vez de para modificar o introducirle datos. La recepción de estos datos se emplea para escribir el archivo que describe la grabación, escritura la cual tiene se realiza mediante un bucle dentro del nuevo hilo que define esta clase. Cuando se sale de este bucle, quiere decir que ya se ha terminado con la grabación del archivo y se genera un GuardarGrabaciónDialog.

TrafficCleaner

Define el limpiador de la base de datos. Por medio de esta clase se eliminan del almacenamiento los aviones de los que ya no se reciben datos. Esta limpieza tiene lugar en un bucle en segundo plano como se explica a continuación, por lo que es necesario que se extienda a Thread.

El constructor recoge el objeto TrafficMap cuyos datos se van a analizar cada cierto tiempo. Por definición, este periodo se ha definido de medio segundo. Así pues, cada medio segundo el bucle llama a la función cleanLostFlights() del HashMap de los marcadores que actúa de la forma que se indicaba cuando se hablaba de esta clase.

TrafficInfoFragment

En esta clase se define el fragmento del FragmentContainerView que se muestra en la pantalla principal cuando se selecciona un avión. Como ya se indica a lo largo del trabajo, se muestra o deja de mostrar como respuesta al evento de pulsación sobre algún icono del mapa, el cual llama a la función showInfo() de la interfaz MarkerListener. Un objeto de esta clase se crea por medio de un constructor e introduciendo el Traffic para el que se quiere generar.

Debido a su naturaleza, se extiende a Frament, lo que posibilita que pueda sobrescribir los métodos onCreateView() y onViewCreated(). En el primero de ellos, se vincula con su interfaz. En el segundo, se inicializan los elementos de la interfaz, los distintos TextView sobre los que se escribirá la información de interés.

Por otra parte se crea el método público showInfo(). En él se escribe en el lugar correspondiente la información que se muestra sobre el avión seleccionado. Este método se llama cada vez que se actualizan estos datos o cuando se cambia la aeronave sobre la que se están mostrando. No obstante, en él solamente se obtienen los datos de la variable Traffic de esta misma clase y se plasman en los TextViews. Es necesario modificar esta variable externamente antes de ejecutar showInfo(), de forma que aparezcan los datos actualizados. Es por ello que la variable en cuestión se define como pública, para que se pueda manipular su valor directamente y aumentar la limpieza del código.

ConnectionAlertDialog

Esta nueva clase define un mensaje de diálogo que se le muestra al usuario y le transmite una alerta de que no ha sido posible realizar una vinculación con la antena, dado que esta no se encuentra disponible. La cualidad de mensaje de diálogo se la otorga el hecho de extender a DialogFragment. Así, el método que sobrescribe es onCreateDialog(). En él, como se menciona en la sección 6.3.2, se crea un constructor como instancia de AlertDialog.Builder. Este constructor tiene unas funciones definidas que sirven para personalizar el mensaje que se muestra por defecto.

DeleteDialog

Al tratarse de una ventana de diálogo, también tiene que extender a DialogFragment. El objetivo de esta clase es que el cliente confirme que quiere borrar el archivo seleccionado, el cual se introduce en el código por medio del constructor.

En este caso, la interfaz de usuario asignada no es la automática, por lo que se especifica cuál es dentro de onCreateDialog(). En esta misma función también se inicializan los dos botones que se muestran y se define el método de respuesta de llamada a una pulsación sobre alguno de ellos. Dado el objetivo de esta interacción con el usuario, si se pulsa btnAccept, se borra el fichero llamando al método delete() de la clase File y se actualizan las entradas de la lista de la biblioteca mediante showFiles() de EmulatorLibraryActivity. Si en cambio, se actúa en btnCancel, se deja de mostrar la ventana sin hacer ningún proceso a través de dismiss().

RenameDialog

Representa un mensaje que salta cuando se solicita cambiar un archivo de nombre. Tiene la misma naturaleza que la clase anterior y, como ella, en este caso la interfaz del mensaje también es una propia. Esta tiene los mismos elementos que DeleteDialog e incluye un EditText en el que poder introducir el nuevo nombre que se desea para el archivo. Dentro de los métodos de llamada a los *clicks* sobre los botones,

se confirma el renombramiento si se pulsa a aceptar, mientras que actúa sobre cancelar, se aborta y se deja de mostrar el mensaje mediante dismiss().

En el caso de que se acepte el nuevo nombre para el archivo, este es se obtiene desde el EditText a través del método de esta clase getText(). El archivo sobre el que se ha solicitado la acción, que se introduce en RenameDialog por medio de su constructor, se renombra llamando a renameTo(). En último, se actualiza la librería ejecutando showFiles().

GuardarGrabacionDialog

Esta última clase muestra otro mensaje al usuario, esta vez cuando se terminan de grabar los aviones del espacio aéreo. En ese caso se le da la opción al usuario de elegir el nombre del archivo que se acaba de generar o de desecharlo en caso de que se haya considerado que no es de interés. Por lo tanto, es una combinación de las dos últimas que se han expuesto. En su interfaz de usuario se muestra un TextView donde introducir el nombre con el que se quiere guardar el archivo y dos botones, uno para guardarlo y otro para eliminarlo.