UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Desarrollo e implementación de una aplicación móvil de comunicación empresarial por voz a través de un servidor autoadministrado

Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

AUTOR/A: Alventosa Corts, Carlos

Tutor/a: Belda Ortega, Román

CURSO ACADÉMICO: 2023/2024

Resumen

El trabajo consiste en el desarrollo y la implementación de una aplicación móvil para facilitar la colaboración y comunicación por voz en entornos empresariales. El proyecto abarca tanto el desarrollo de la aplicación en sí, como de los servicios de *backend* necesarios para permitir la interacción entre los dispositivos móviles.

El desarrollo de la aplicación móvil incluirá una interfaz gráfica intuitiva que permita a los usuarios comunicarse en tiempo real a través de redes IP. Las comunicaciones se realizarán a través de un servicio dedicado, y desarrollado también en este proyecto, para tal efecto.

La plataforma para la que se va a desarrollar la aplicación móvil será Android, haciendo uso del lenguaje de programación Kotlin, que también se aprovechará para el desarrollo del servicio. A lo largo del trabajo, se detallarán las tecnologías empleadas y las alternativas que existen.

Resum

El treball consisteix en el desenvolupament i la implementació d'una aplicació mòbil per a facilitar la col·laboració i comunicació per veu en entorns empresarials. El projecte comprèn tant el desenvolupament de l'aplicació en sí, com dels serveis de backend necessaris per a permetre la interacció entre els dispositius mòbils.

El desenvolupament de l'aplicació mòbil inclourà una interfície gràfica intuïtiva que permeta als usuaris comunicar-se en temps real a través de xarxes IP. Les comunicacions es realitzaran a través d'un servei dedicat, i desenvolupat també en este projecte, per a tal efecte.

La plataforma per a la que es va a desenvolupar l'aplicació mòbil serà Android, fent ús del llenguatge de programació Kotlin, que també s'aprofitarà per al desenvolupament del servei. Al llarg del treball, es detallaran les tecnologies empleades i les alternatives que existeixen.

Abstract

This work involves the development and implementation of a mobile application to facilitate collaboration and voice communication in business environments. The project encompasses both the development of the application itself, and the backend services necessary to enable interaction between mobile devices.

The development of the mobile application will include an intuitive graphical interface that allows users to communicate in real-time over IP networks. Communications will be conducted through a dedicated service, also developed in this project, for this purpose.

The platform for which the mobile application will be developed will be Android, making use of the Kotlin programming language, which will also be used for the development of the service. Throughout the work, the technologies used and the alternatives that exist will be detailed.

RESUMEN EJECUTIVO

La memoria del TFG del Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación debe desarrollar en el texto los siguientes conceptos, debidamente justificados y discutidos, centrados en el ámbito de la ingeniería de telecomunicación

CONCEPT (ABET)	CONCEPTO (traducción)	¿Cumple? (S/N)	¿Dónde? (páginas)
1. IDENTIFY:	1. IDENTIFICAR:		
1.1. Problem statement and opportunity	1.1. Planteamiento del problema y oportunidad	S	1, 2
1.2. Constraints (standards, codes, needs, requirements & specifications)	1.2. Toma en consideración de los condicionantes (normas técnicas y regulación, necesidades, requisitos y especificaciones)	S	3
1.3. Setting of goals	1.3. Establecimiento de objetivos	S	2 - 5
2. FORMULATE:	2. FORMULAR:		
2.1. Creative solution generation (analysis)	2.1. Generación de soluciones creativas (análisis)	S	6 - 22
2.2. Evaluation of multiple solutions and decision-making (synthesis)	2.2. Evaluación de múltiples soluciones y toma de decisiones (síntesis)	S	9 - 12, 23 - 58
3. SOLVE:	3. RESOLVER:		
3.1. Fulfilment of goals	3.1. Evaluación del cumplimiento de objetivos	S	59 - 63
3.2. Overall impact and significance (contributions and practical recommendations)	3.2. Evaluación del impacto global y alcance (contribuciones y recomendaciones prácticas)	S	64, 65

Índice

Capítulo 1. Introducción ... 1
1.1 Presentación de la idea y concepto a desarrollar ... 1
1.2 Relevancia del proyecto ... 1
1.2.1 Principales competidores ... 2
1.3 Objetivos ... 2
1.4 Requisitos y limitaciones técnicas ... 3
1.5 Metodología ... 3
Capítulo 2. Diseño general del proyecto ... 6
2.1 Plataforma de desarrollo ... 6
2.1.1 Cliente ... 6
2.1.2 Servidor de comunicación ... 8
2.2 Comunicación entre usuarios ... 9
2.2.1 Opciones técnicas ... 9
2.3 Pantallas empleadas ... 12
2.3.1 Flujo de navegación entre las pantallas ... 14
2.4 *Protocol Buffers* ... 15
2.4.1 Introducción ... 15
2.4.2 Mensajes empleados ... 16
2.5 Características de audio en la transmisión ... 19
2.5.1 Codificación ... 20
2.6 Gestión de los datos ... 20
Capítulo 3. Desarrollo Android ... 23
3.1 Utilidades de Kotlin ... 23
3.1.1 Corutinas ... 23
3.1.2 Flows y StateFlows ... 24
3.2 Arquitectura Model-View-Intent ... 25
3.3 Sistema de inyección de dependencias ... 28
3.3.1 Beneficios e inconvenientes. Alternativas ... 29
3.4 *NavigationScreen* ... 29
3.4.1 Sustitución de los Activity ... 29
3.4.2 Navegación entre pantallas ... 30

3.4.3 Animaciones 31
3.4.4 Intercambio de datos entre pantallas 31
3.5 Guardado y persistencia de los datos 32
3.6 Cliente WebSocket de *Ktor* 33
3.7 *Model* y funcionamiento general de la comunicación 34
3.7.1 Definición de variables y gestión de datos del Model 34
3.7.2 Esquema de funcionamiento del habla 35
3.7.3 Diagrama de funcionamiento de la escucha 36
3.8 Grabado y reproducción de audio 37
3.8.1 AudioRecorder 38
3.8.2 AudioPlayer 39
3.9 Cálculo de parámetros de la conexión 41
3.9.1 Modificación del búfer del AudioPlayer 42
3.10 Permisos en Android 43
3.11 Búsqueda *Bluetooth* y conexión con botones Bluetooth *Low Energy* 46
3.11.1 Descubrimiento de dispositivos Bluetooth push-to-talk 46
3.11.2 Conexión con el dispositivo encontrado 47
Capítulo 4. Desarrollo del servidor 48
4.1 Servidor *WebSocket* de *Ktor* 48
4.2 Gestión de conexiones y autenticación de usuarios 49
4.3 Recepción, tratamiento y envío de mensajes 51
4.3.1 Envío de mensajes cliente-servidor y difusión en un grupo 52
4.4 Gestión de desconexión de sesiones 53
4.5 Servicio Web de administración del servidor 54
4.5.1 Páginas empleadas 54
4.5.2 Creación, modificación y eliminación de elementos. Persistencia 57
4.5.3 Reconexión de clientes y actualización reactiva 58
Capítulo 5. Comprobación de resultados y despliegue 59
5.1 Aplicación resultante y recomendaciones seguidas 59
5.2 Retardo en el audio. Interrupciones. Estudio del *jitter* 60
5.3 Creación de la imagen de Docker 61
5.4 Nombre de dominio y certificado SSL/TLS 62
Capítulo 6. Conclusión y trabajo futuro 64
6.1 Conclusión 64
6.2 Alcance 64
6.3 Trabajo futuro 64

Índice de figuras

Figura 1. Ejemplo de teléfono Android con aspecto robusto. Fuente: UNIWA ... 6

Figura 2. Cuota mundial de mercado de iOS y Android. Fuente: Backlinko [6] ... 7

Figura 3. Logo de Kotlin ... 7

Figura 4. Cuota de mercado de las versiones de Android a fecha 1 de mayo de 2024. Fuente: Android Studio ... 8

Figura 5. Logo de Ktor ... 9

Figura 6. Diagrama de funcionamiento del servidor *Fast-ll*. Fuente: GitHub, *robelor/fast-ll* [17] ... 11

Figura 7. Esquema general de diseño propuesto para las pantallas. De izquierda a derecha y de arriba a abajo: pantalla de grupos, pantalla de participantes de un grupo, pantalla de ajustes y pantalla de conexión con dispositivos Bluetooth *Low Energy* ... 13

Figura 8. Flujo de navegación entre pantallas ... 14

Figura 9. Logo de *Protocol Buffers* ... 15

Figura 10. Código *gradle* para compilar *protobuf* automáticamente ... 15

Figura 11. Ejemplo de definición de un mensaje en *protobuf* ... 16

Figura 12. Ubicación de las clases Java generadas por el compilador de *protobuf* ... 16

Figura 13. Definición del mensaje principal del protocolo ... 17

Figura 14. Mensaje de petición y respuesta de autorización en el servidor ... 17

Figura 15. Mensajes de petición y respuesta de grupos y participantes (l. 1-9). Definición de participantes y grupos (l. 11-25) ... 18

Figura 16. Mensajes de petición y respuesta para la solicitud de permiso de habla (l. 1-9) y liberación del canal (l. 11-19) ... 19

Figura 17. Mensajes de inicio de audio (l. 1-9), muestra de audio (l. 11-16) y fin de audio (l. 18-21) ... 19

Figura 18. Posible diseño de base de datos relacional ... 21

Figura 19. Ejemplo de estructura de la base de datos JSON ... 22

Figura 20. Ejemplo de uso de corutinas para recibir mensajes desde el *WebSocket* ... 24

Figura 21. Ejemplo de uso de *MutableStateFlows* para controlar la pulsación del botón *push-to-talk* ... 25

Figura 22. Ciclo de datos de *Model-View-Intent* ... 26

Figura 23. Orbit Container que representa el estado de la pantalla de grupos (l. 1-11), e *intents* disponibles para realizar desde el *View* (l. 13-19) ... 27

Figura 24. ViewModel que gestiona interacciones y actualización de la pantalla de grupos ... 27

Figura 25. Ejemplo de declaración de módulos de *Koin* ... 28

Figura 26. Inicialización de Koin en la clase *Application* 29
Figura 27. Clase *MainActivity*, actividad principal y única de la aplicación 30
Figura 28. Estructura de la función *NavigationScreen()* 30
Figura 29. Transiciones empleadas en la pantalla de ajustes 31
Figura 30. Declaración del módulo de *Koin* con el parámetro *groupAlias* 32
Figura 31. Mensajes *protobuf* para almacenar datos localmente 32
Figura 32. Propiedades de extensión de *Context* para almacenar y leer contenido del almacenamiento interno 33
Figura 33. Serializador *ConfigStoreSerializer*, para la clase *RomeoConfig* 33
Figura 34.Diagrama de flujo de la clase *WebSocketKtorClient* 34
Figura 35. Diagrama de flujo del funcionamiento del habla en la aplicación 36
Figura 36. Diagrama de flujo del funcionamiento de la escucha en la aplicación 37
Figura 37. Utilización de la clase *AudioRecorder* 38
Figura 38. Función para grabar un paquete de audio 39
Figura 39. Funcionamiento del búfer de audio 40
Figura 40. Mensaje *RttTest* para conocer el RTT de los paquetes 41
Figura 41. Clase *FifoArray* para el almacenamiento de muestras de retardo y cálculo del *jitter* 42
Figura 42. Elección del tamaño del búfer al reproducir audio 42
Figura 43. Diagrama de flujo para declarar permisos en Android. Fuente: Android Developers [21] 43
Figura 44. Diagrama de flujo para solicitar permisos en Android 43
Figura 45. Permisos declarados en el *manifest* 44
Figura 46. Elementos necesarios para solicitar permisos en la pantalla de grupos 45
Figura 47. Comprobación de garantía de permisos (l. 2) y solicitud de permisos (l. 5) 45
Figura 48. UUIDs del servicio y característica que identifican la funcionalidad *push-to-talk* ... 46
Figura 49. Función para realizar la búsqueda de dispositivos Bluetooth LE *push-to-talk* 47
Figura 50. Inicialización del servidor de *Ktor* 48
Figura 51. Diagrama de flujo del funcionamiento del servidor *WebSocket* cuando recibe una nueva conexión 49
Figura 52. Función para establecer la conexión de un objeto *Participant*, de modo que pueda enviar y recibir mensajes desde la instancia de *WebSocket* 50
Figura 53. Función para autenticar a un usuario 50
Figura 54. Construcción del mensaje de respuesta para informar de los grupos a los que pertenece un usuario 51
Figura 55. Establecimiento del permiso para hablar de un participante en un grupo 52
Figura 56. Tratamiento del mensaje de audio 52
Figura 57. Función para difundir un mensaje en un grupo 52

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Figura 58. Función para difundir un mensaje de audio en un grupo 53

Figura 59. Gestión de desconexiones en el bloque *try-catch* 54

Figura 60. Página de *login* del portal web 55

Figura 61. Página principal del portal web de administración de usuarios y grupos 55

Figura 62. Página para modificar las credenciales del administrador en el portal web 56

Figura 63. Página para crear un nuevo grupo en el portal web 56

Figura 64. Gestión de la creación de un usuario desde el portal web 58

Figura 65. Logo de la aplicación 59

Figura 66. Interfaz gráfica en la versión final de la aplicación. De izquierda a derecha: pantalla de grupos, pantalla de participantes de un grupo, pantalla de ajustes y pantalla de conexión con dispositivos Bluetooth *Low Energy* 59

Figura 67. *Dockerfile* con la configuración para crear la imagen de *Docker* del servidor 61

Figura 68. Método para añadir un certificado SSL/TLS al servidor de *Ktor*. Fuente: *ktor.io* 63

Figura 69. Creación del *Visualizer* para obtener las muestras de FFT del audio reproducido en tiempo real 70

Capítulo 1. Introducción

1.1 Presentación de la idea y concepto a desarrollar

La idea principal y, con ella, la motivación de este trabajo, surge a partir de la necesidad básica de comunicar a las personas. La comunicación eficiente y rápida se ha vuelto indispensable tanto en el ámbito personal como profesional. Este hecho ha propiciado el crecimiento exponencial de las aplicaciones de mensajería y comunicación, especialmente en los últimos años, adaptándose a las demandas de los usuarios que, cada vez más, buscan inmediatez y simplicidad.

Desde sus inicios, la comunicación por voz ha evolucionado significativamente, comenzando con el uso de la radio y los *walkie-talkies*, que constituyeron las primeras formas de comunicación inalámbrica de corto alcance. Estos dispositivos se fueron popularizando en diversos sectores de la sociedad, conviviendo incluso con el teléfono convencional, dada su sencillez y la ausencia de necesidad de grandes infraestructuras que permitiesen establecer dicha comunicación. Hoy en día, gracias a la existencia de Internet, se han superado muchas de las limitaciones impuestas por estos dispositivos, siendo la más importante de ellas, la distancia máxima de funcionamiento de estos aparatos.

Dentro de este contexto, surge la idea de desarrollar una aplicación *push-to-talk* enfocada a emular el funcionamiento de un sistema *walkie-talkie*, incluyendo mejoras que permitan aprovechar las características propias de un teléfono móvil, como pueden ser la conectividad a Internet, la capacidad multimedia o la existencia de una interfaz táctil que permita gestionar la direccionalidad de la comunicación.

El concepto de *push-to-talk* previamente mencionado será el que permita poner en contacto a los usuarios de manera casi instantánea, tal y como su nombre indica, con solo presionar un botón, transformando así el teléfono móvil en un *walkie-talkie* digital con funcionalidad ampliada. Este tipo de comunicación es especialmente útil en situaciones donde priman la rapidez y la claridad, como en la coordinación de trabajos en equipo, emergencias o eventos en tiempo real.

1.2 Relevancia del proyecto

En la actualidad, existe una gran cantidad de implementaciones distintas cuya finalidad coincide con la descrita en el punto anterior. Cada una de ellas puede aportar su propia visión del problema y enfocar la solución de la forma que le resulte más conveniente. La gran mayoría de estas soluciones, sin embargo, no están enfocadas a proporcionar una comunicación por voz en un entorno empresarial, en el que la privacidad y la inmediatez son máximas que no deben ser menospreciadas.

Este proyecto tiene a estas dos máximas como los pilares fundamentales en los que se sustentará la comunicación de las empresas que adopten esta solución como su principal forma de comunicar a sus empleados. En primer lugar, la privacidad, se proporciona en el momento en que la empresa en cuestión utiliza su propio *hardware* para ejecutar el servidor de voz, haciendo que los paquetes que transportan las conversaciones de sus empleados se enruten directamente a través de servidores propios de la empresa, en lugar de viajar a servidores externos de compañías de terceros cuya seguridad o privacidad pueden no estar garantizadas. En segundo lugar, la inmediatez se proporciona en la aplicación móvil, cuya interfaz está diseñada específicamente para que el empleado que la utilice mantenga activo y abierto en todo momento el canal en el que va a hablar y los canales que está escuchando, y pueda enviar un mensaje de voz en tiempo real solamente con presionar un botón.

1.2.1 Principales competidores

A continuación, se detallan y prueban algunas de las alternativas actuales que podrían competir en el ámbito que engloba este proyecto.

1.2.1.1 Zello Walkie Talkie

Esta aplicación, desarrollada por la empresa *Zello Inc* se vende como la sustitución directa del *walkie-talkie* a través de Internet. Se basa en el *push-to-talk*, una técnica según la cual, cuando el usuario presiona el botón de habla, la aplicación transmitirá el audio grabado por el emisor al resto de personas que se encuentran en un mismo grupo.

Sin embargo, tiene un enfoque distinto a lo que se espera de este proyecto, pues no valora tanto la inmediatez de la reproducción de los mensajes, sino que, en muchas ocasiones, el mensaje se reproducirá en el receptor tras haber sido grabado por completo. Además, mantiene un histórico con mensajes de audio reproducibles a modo de "chat".

Permite el establecimiento de estados del hablante (disponible, ocupado, etc.), y configurar y crear grupos de habla distintos, que pueden ser públicos o privados, basados en roles.

El servicio proporcionado por *Zello Inc* cuenta con tres rangos de precio, empezando por 6.80 USD por usuario al mes en su versión más básica. Únicamente ofrece la posibilidad de instalar un servidor de voz local llegando a un acuerdo con la propia empresa, y cuyo precio puede ser variable [1].

Zello tiene clientes en las plataformas móviles Android y iOS, y cuenta también con programas para Mac y Windows.

1.2.1.2 TeamSpeak 3

TeamSpeak es un *software* orientado a proporcionar comunicación VoIP convencional haciendo uso de servidores locales.

Este servicio no está enfocado en el método *push-to-talk*, sino que se podría clasificar como un servicio de llamadas al uso, ofreciendo un retardo mínimo y encriptación de nivel militar [2].

En cuanto al servidor, su licencia gratuita permite la conexión simultánea de hasta 32 usuarios, con posibilidad de ampliar el número de *slots* en sus versiones de pago. Cuenta con clientes gratuitos para plataformas de PC (Windows, Mac y Linux), y clientes de pago para plataformas móviles (Android y iOS).

1.3 Objetivos

El objetivo principal de este trabajo Fin de Grado, definido a grandes rasgos, es el diseño y la creación de una aplicación móvil de voz sobre IP (VoIP) que permita a los usuarios comunicarse en tiempo real con una interfaz intuitiva y fácil de usar. El sistema debe garantizar que el audio se transmita al servidor conforme se graba y se reproduzca en el receptor de inmediato, tratando de obtener la menor latencia posible teniendo en cuenta las limitaciones que introduce la propia red.

Además de la funcionalidad *push-to-talk* implementada directamente sobre la interfaz gráfica de la aplicación, se proporcionarán diferentes métodos de habla y escucha que facilitarán el uso de la aplicación a aquellas personas que requieran comunicarse mientras tienen sus manos ocupadas, o no pueden alcanzar su teléfono móvil con facilidad. Esto significa que la aplicación deberá ser compatible con dispositivos externos como auriculares *Bluetooth* con micrófono integrado, o botones *push-to-talk* físicos.

Debido al claro enfoque empresarial y laboral, se debe contar con un sistema de inicio de sesión basado en servidor, en el que cada cliente utilizará su nombre de usuario y contraseña para

identificarse dentro de su empresa u organización, y posteriormente tener acceso a todas las funcionalidades que se le permita. Un administrador tendrá acceso a un portal web desde el que manejar la creación o modificación de nuevos usuarios o grupos de comunicación.

Por lo tanto, otro de los objetivos es el desarrollo de un servicio que las empresas puedan desplegar fácilmente en su infraestructura. Una vez los clientes hayan establecido conexión con el servidor y se hayan autenticado de manera exitosa, el servidor deberá responder a todas las peticiones de los usuarios, pues será este el que almacene permanentemente la configuración del servicio.

1.4 Requisitos y limitaciones técnicas

El principal requisito que se impone en este proyecto es que el retardo que existe entre el instante en que el emisor empieza a hablar y el instante en el que el receptor comienza a escuchar el mensaje debe ser mínimo. Este requisito conlleva la limitación impuesta por la propia red, puesto que introducirá un retardo inevitable, y que además será variable, con lo que el reproductor deberá adaptarse a todas estas variaciones durante la transmisión de los paquetes. A causa de esto, también se convierte en requisito añadido el hecho de no guardar los mensajes de audio en ninguna ubicación de forma permanente, garantizando así la privacidad y la inmediatez.

En cuanto al *hardware* necesario para ejecutar el servidor de comunicación, cabe destacar que este deberá ser accesible a través de Internet, además de contar con algún método que permita cifrar las conexiones con los clientes. En caso de ser desplegado en empresas con un gran número de empleados, se debe garantizar una conexión a Internet con ancho de banda suficiente.

Si se emplease el protocolo de transporte UDP para el envío de los mensajes de audio, y no fuese necesario servidor alguno para realizar el intercambio de paquetes entre clientes, sería requisito indispensable lidiar con los sistemas NAT y el *firewall* de modo que los clientes pudiesen hablarse de forma directa a través de Internet. Esto consiste en la incorporación de un servidor de STUN/TURN, que funcione de la siguiente manera [3]:

- Los clientes mantienen una conexión activa permanente con el servidor de STUN (*Session Traversal Utilities for NAT*), de modo que este conoce la dirección IP y puerto que tienen disponible cada uno de los clientes para establecer la comunicación.
- Cuando un cliente quiere comunicarse con otro, el servidor STUN le comunicará al primero la dirección IP y puerto de su interlocutor, con lo que, en caso de que el NAT lo permita (es decir, no le importe que los paquetes se reciban desde una dirección y puerto distintos a los del paquete original de conexión con el servidor), la comunicación directa entre ambos clientes quedará establecida.
- Si, de otro modo, el NAT es más restrictivo, se deberá emplear un servidor TURN (*Traversal Using Relays around NAT*), a través del cual pasarán todos los paquetes y se retransmitirán al cliente correspondiente, convirtiéndose a efectos prácticos en una conexión cliente-servidor al uso.

Todo esto requiere de una mayor infraestructura de red que, además, está sujeta a varios fallos y elementos propios de las redes privadas que no se pueden gestionar, dificultando el despliegue *on-premise*. Es por ello por lo que esta opción queda descartada, imponiéndose la limitación de necesitar un servidor TCP a través del cual se ejecuten todas las comunicaciones.

1.5 Metodología

A lo largo de este trabajo, se ha decidido emplear una metodología propia del desarrollo de *software* para tratar de conseguir los objetivos propuestos. Tanto la aplicación del cliente como el desarrollo del servidor han seguido un desarrollo incremental. Esta metodología de desarrollo de *software* consiste en la separación del conjunto del problema en bloques funcionales de menor

tamaño que, una vez desarrollados y comprobados, se unen con el resto de los bloques ya existentes.

Esto permite, entre otras cosas, el desarrollo en paralelo de las distintas funcionalidades que se quieren implementar, además de poder identificar fallos de manera más eficaz. Sin embargo, esto conlleva un aumento en la complejidad de control una vez el proyecto crece en tamaño y cantidad de código.

En el caso de la aplicación móvil, el primer paso será conseguir un proyecto base sin funcionalidades específicas, pero con la estructura necesaria para facilitar el desarrollo del resto de características. Una vez conseguido esto, los bloques funcionales a desarrollar serán los siguientes:

- **Gestión de los parámetros de uso de la aplicación:** Para poder funcionar, el cliente deberá introducir, en primer lugar, la dirección del servidor al que se quiere conectar. Posteriormente, deberá introducir el nombre de usuario y la contraseña que le sirvan para autenticarse en este servidor.
- **Gestión de grupos y participantes:** Una vez el cliente se ha autenticado en el servidor, realizará una petición para obtener todos los grupos en los que este participa, así como el resto de los participantes de cada grupo.
- **Grabación de audio:** Cada vez que el cliente quiera hablar en un grupo, deberá seleccionarlo y pulsar el botón *push-to-talk* para empezar a grabar el audio. Esta grabación resultará en la generación de pequeñas muestras de audio que se enviarán progresivamente al servidor.
- **Reproducción de audio:** Cuando el servidor reenvía al resto de participantes de un grupo las muestras de audio comentadas en el punto anterior, estos las deben reproducir. Con anterioridad a la reproducción, las muestras deberán ser almacenadas en un búfer que permita reducir la cantidad de interrupciones que se producen en el audio a causa de la baja calidad de la conexión de red.
- **Codificación y decodificación:** Con el fin de reducir el tamaño de los paquetes de audio que se envían por la red, estos deberán codificarse en el cliente emisor y decodificarse en el cliente receptor antes de ser reproducidos.
- **Gestión de dispositivos *Bluetooth Low Energy*:** La aplicación deberá ser capaz de descubrir dispositivos *Bluetooth Low Energy* con capacidad *push-to-talk*, conectarse a ellos y manejar reconexión y desconexiones.

Por la parte del servidor, los bloques funcionales a desarrollar son los siguientes:

- **Gestión de conexiones nuevas para autenticación**: Se debe establecer el formato de los mensajes que se aceptarán en el servidor, y permitir únicamente conexiones nuevas con paquetes de autenticación.
- **Almacenamiento, carga y gestión de datos**: El servidor será el que mantenga la información relativa a usuarios y grupos, por lo que se debe establecer una forma de guardar y cargar los datos. A partir de dicha información, el servidor creará los objetos de Participantes y Grupos pertinentes.
- **Contestación de peticiones**: Una vez un usuario se ha autenticado de forma exitosa en el servidor, podrá enviar mensajes con peticiones que deberán ser resueltas por el servidor.
- **Portal web**: Con el objetivo de facilitar la tarea del administrador, se desarrollará una página web ejecutada sobre el mismo servidor de voz con la que se proporcionará una interfaz con la que crear, modificar o eliminar usuarios de la aplicación, o grupos de comunicación.

Cada uno de estos bloques será correctamente validado e introducido en el modelo general a desarrollar, donde también se encontrarán otras funciones no comentadas, como la verificación de que dos participantes no hablen simultáneamente en un grupo, o la navegación entre pantallas.

Todos los cambios realizados a lo largo del desarrollo se han subido progresivamente a un repositorio privado de *Gitlab*, de modo que siempre se pudiese volver rápidamente a una versión anterior en caso de fallo, y observar qué cambios han resultado problemáticos.

Capítulo 2. Diseño general del proyecto

2.1 Plataforma de desarrollo

2.1.1 Cliente

Cuando se habla del "cliente", se debe aclarar a qué tipo de público irá dirigida la aplicación a desarrollar. Como ya se ha comentado, la aplicación tiene un claro enfoque empresarial, y esto da lugar a pensar en una gran cantidad de perfiles potenciales.

Entre todos ellos, se puede pensar en personas que estén en constante movimiento, como pueden ser transportistas, conductores de autobús o taxi, o en personas que trabajen en equipo, pero separadas por distancias razonables, como puede ser en el sector de la construcción o en el equipo de empleados de un supermercado.

En muchos de estos oficios mencionados suele ser habitual que la empresa provea al empleado con un dispositivo específico para realizar estas funciones. Este tipo de dispositivos suelen contar con un aspecto robusto (véase la Figura 1), o formar parte de otro tipo de dispositivos como datáfonos o lectores de códigos de barras. Usualmente, además, incluyen botones programables específicos que las aplicaciones pueden aprovechar para ejecutar distintas funciones, entre las que se podría encontrar la acción de *push-to-talk*.

Figura 1. Ejemplo de teléfono Android con aspecto robusto. Fuente: UNIWA

Todos estos dispositivos tienen en común que comparten el mismo sistema operativo: **Android**, por lo que será este el sistema en el que se va a enfocar el desarrollo del cliente.

2.1.1.1 Android

Según la revista *ADSLZone*: "Android es un sistema operativo móvil diseñado para dispositivos móviles con pantalla táctil como teléfonos inteligentes o tablets, pero que también lo encontramos en otros dispositivos como relojes inteligentes, televisores […]" [4].

Se trata de un sistema operativo desarrollado en sus inicios por la compañía *Android Inc*, hasta que en el año 2005 fue adquirido por *Google LLC* y lanzado al mercado el 23 de septiembre de 2008 con su versión 1.0 [5]. En la actualidad, este sistema operativo sigue en desarrollo, con la segunda beta abierta de la versión 15 ya publicada, a la fecha de escritura del presente documento.

Este sistema operativo, junto con iOS (*Apple*) representan el 99.27% de la cuota mundial de mercado en materia de sistemas operativos móviles. Como se puede ver en la Figura 2, a fecha de febrero de 2024, Android posee un 70.69% del mercado, mientras que iOS representa un 28.58% mundialmente. No obstante, en cuanto al mercado de EE. UU., iOS lidera el primer puesto con un 60.77% frente al 38.81% de Android [6].

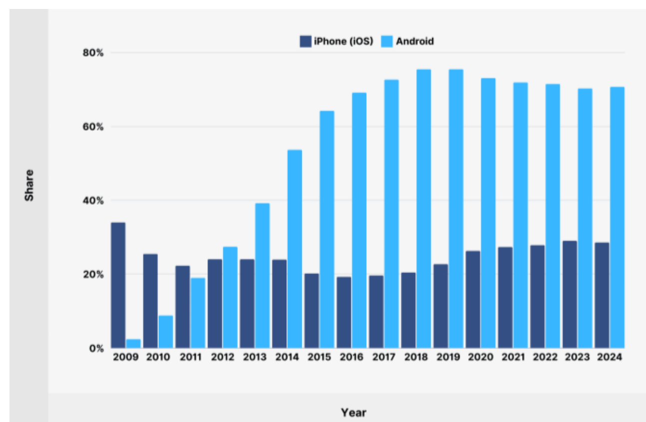

Figura 2. Cuota mundial de mercado de iOS y Android. Fuente: Backlinko [6]

Es por todos estos motivos por los que resulta interesante el desarrollo de la aplicación para esta plataforma, aunque no se descarta el desarrollo en la plataforma de *Apple* como trabajo futuro.

2.1.1.2 Desarrollo en Android

Existen múltiples lenguajes de programación que permiten crear y publicar aplicaciones en la plataforma Android. Sin embargo, es la propia *Google* la que establece unas ciertas pautas que los desarrolladores pueden tener en cuenta a la hora del desarrollo. Es lo que se conoce como *Modern Android Development*: herramientas, librerías y guías que recomienda el equipo de Android para acelerar y facilitar el desarrollo de aplicaciones de alta calidad [7].

Entre las recomendaciones, se incluye el uso de librerías como *Jetpack Compose* para simplificar el desarrollo de la interfaz de usuario, la utilización del IDE *Android Studio* y, sobre todo, la recomendación de emplear el lenguaje de programación **Kotlin**, cuyo logo se puede ver en la Figura 3.

Figura 3. Logo de Kotlin

Kotlin es un lenguaje de programación administrado por la *Kotlin Foundation*, una organización establecida por JetBrains y Google. En la conferencia *Google I/O* del año 2019, la empresa desarrolladora de Android promovió el uso de Kotlin como primera opción a la hora de escoger un lenguaje de programación (*Kotlin-first*) [8] [9].

Se trata de un lenguaje de alto nivel que soporta tanto programación orientada a objetos (OOP) como programación funcional, con inferencia de tipos. Está diseñado para ser totalmente compatible con Java, de modo que se pueda llamar a código Java desde Kotlin y viceversa. Tanto es así que el *bytecode* generado por el compilador de Kotlin se ejecutará en la JVM [10].

De este modo, Kotlin se convierte en el principal sucesor a Java, que era tradicionalmente el lenguaje principal para el desarrollo de Android, por ser un lenguaje de programación más expresivo y conciso, más seguro, interoperable y concurrentemente estructurado.

2.1.1.3 Versión de API mínima

Como sucede en cualquier desarrollo Android, se debe establecer previamente las versiones que soportará la aplicación a desarrollar. En el caso que se plantea, y especialmente debido a los dispositivos con los que se ha contado para realizar el testeo de la aplicación durante el desarrollo (anexo 1), el nivel de API mínimo escogido es el 28, que se corresponde con Android 9 “*Pie*”.

Por otro lado, es pertinente indicar la cuota de mercado que abarca cada una de las versiones de Android, pues normalmente los teléfonos suelen recibir entre 2 y 3 años de actualizaciones principales.

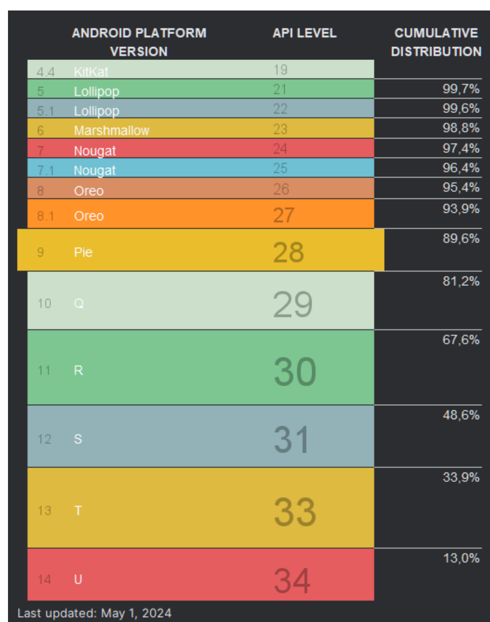

Figura 4. Cuota de mercado de las versiones de Android a fecha 1 de mayo de 2024. Fuente: Android Studio

Como se puede observar en la Figura 4, una aplicación que se desarrolle teniendo la API 28 como nivel mínimo podrá ser ejecutada en el 89.6% de dispositivos. El 11.4% de dispositivos siguen contando con una versión igual o inferior a Android 8, una versión que se presentó en el año 2017 (hace 7 años del planteamiento de este documento).

2.1.2 Servidor de comunicación

Habiendo ya elegido el lenguaje de programación Kotlin para la parte del cliente, al ser el lenguaje por defecto, según Google, para la programación en Android, resta escoger la plataforma en la que se va a desarrollar el servidor.

Existen muchas opciones a la hora de escoger un lenguaje para programar el servidor. Entre ellas, están JavaScript, con el servidor Node.js, Python a través de librerías como *Socket.IO*, Java con *Spring*, etc. Todos ellos cuentan con librerías específicas con las que poder iniciar un socket TCP y establecer una comunicación.

Sin embargo, se ha aprovechado el desarrollo del cliente, que se ha realizado en Kotlin, para desarrollar también el servidor.

No obstante, si en un futuro la aplicación se liberase con un despliegue *cloud* en lugar de *on-premise*, se podría pensar en cambiar el lenguaje de programación del servidor de comunicación a otro más eficiente como *Go* o *Rust*, para poder manejar un mayor volumen de peticiones. Esto se podría considerar trabajo futuro.

2.1.2.1 Ktor

Ktor es una librería propia de Kotlin, de código abierto, que permite crear aplicaciones web. Al emplear corutinas, permite aprovechar las bondades de la programación asíncrona en el desarrollo.

Además, Ktor tiene librerías para la parte del cliente, lo que convierte a esta librería en una opción perfecta para el caso de uso que se pretende desarrollar, al emplear la misma estructura en ambos lados de la comunicación [11]. Se puede observar su logo en la Figura 5.

Figura 5. Logo de Ktor

2.2 Comunicación entre usuarios

Como ya se ha comentado en la introducción de este documento, se busca una comunicación de voz exclusivamente unidireccional, de modo que se emule el funcionamiento de un dispositivo de radio o *walkie-talkie*. Este hecho también abarca el uso de la técnica *push-to-talk*, para iniciar y finalizar los mensajes de voz.

La comunicación se realizará entre los participantes de un mismo grupo. Es decir, desde la administración del servidor se crearán grupos y se asignarán participantes a dichos grupos, de modo que, cuando un cliente inicie sesión con su información, la aplicación le mostrará todos los grupos en los que está registrado, y podrá hablar y escuchar en ellos.

Al tratarse de una comunicación unidireccional, el cliente que desee empezar a hablar en un determinado grupo deberá primero cerciorarse de que nadie más está hablando en ese grupo. Será en ese instante cuando la aplicación dé permiso al usuario para hablar, lo que reservará el canal de ese grupo únicamente para ese usuario. De este modo, se consigue evitar las colisiones y permitir a un solo hablante simultáneo por grupo. De no hacer esto, podría resultar en una comunicación ininteligible y poco ordenada.

Además, se facilitará la comunicación manos libres, empleando botones Bluetooth *Low Energy* y haciendo compatibles los auriculares Bluetooth con micrófono incorporado.

En el planteamiento inicial de este proyecto se ha considerado que los mensajes no deben quedar registrados en ninguna ubicación. Esto aporta inmediatez y confidencialidad en la comunicación, pues cualquier mensaje que fue enviado en el pasado no podrá volver a ser reproducido. Sin embargo, permitiendo el almacenamiento o registro de mensajes se puede aportar una funcionalidad mayor. Por ejemplo, el servidor podría registrar los mensajes de audio en una cola a medida que se vayan recibiendo desde los clientes. Los mensajes en la cola se pasarían a través de un modelo de Inteligencia Artificial dedicado a la transcripción de audio, como por ejemplo *Whisper*, un modelo de código abierto de la empresa *OpenAI* [12].

La salida de este modelo, es decir, las transcripciones de cada uno de los mensajes de audio enviados por los clientes quedarían guardados en una lista de mensajes por grupo. De este modo, los clientes podrían seleccionar acceder a la vista de un grupo, y consultar la transcripción de todos los mensajes, junto con el participante del grupo que lo ha enviado.

Al requerir de *hardware* más potente, que permita la ejecución de un modelo de Inteligencia Artificial, se ha considerado esta implementación como trabajo futuro.

2.2.1 Opciones técnicas

En cuanto a la transmisión de los paquetes de datos entre el cliente y el servidor (y viceversa), hay varias opciones que podrían encajar con los requisitos y limitaciones que se presentan. Existen opciones de todo tipo; pueden emplearse protocolos especializados en la transmisión de contenidos multimedia, u otros más generales basados en HTTP.

2.2.1.1 Soluciones de streaming de flujos

- **RTP (*Real-Time Protocol*):** Se trata de un protocolo diseñado para manejar tráfico en tiempo real a través de Internet, que funciona sobre UDP. Principalmente se emplea en Voz sobre IP (VoIP), videollamadas por Internet y retransmisión de vídeo y audio en *streaming*. A pesar de ser un protocolo que minimiza la latencia, asegura la reproducción ordenada y síncrona de los medios gracias a protocolos como RTCP, y es compatible con formatos de audio como MPEG-2 y AAC [13], no garantiza la entrega de paquetes (por el hecho de emplear UDP), además de que no proporciona seguridad, con lo que sería necesario emplear otros protocolos como SRTP.
- **SRTP (*Secure Real-Time Protocol*):** En el RFC 3711 [14] se establecen las bases del protocolo SRTP, un perfil de RTP que proporciona confidencialidad, autenticación del mensaje y protección contra ataques de *replay*. Se sitúa en una capa intermedia entre RTP y la capa de transporte. La desventaja del uso de este protocolo es el aumento en la complejidad de configuración, pues se deben gestionar las claves que se emplean en la codificación de los flujos.
- **SRT (*Secure Reliable Transport*):** Se trata de un protocolo orientado a la conexión en la capa de aplicación, pero empleando UDP en lugar de TCP como protocolo de transporte. La parte *Secure* la obtiene pues soporta encriptación usando AES.
- **WebRTC (*Web Real-Time Communications*):** Es un estándar abierto que proporciona funcionalidad de comunicación en tiempo real a una aplicación, disponible en todos los navegadores modernos (Chrome, Edge, Safari, Firefox, etc.) [15]. A pesar de que establece conexiones de control empleando *WebSockets* seguros (WSS) a través del puerto 443, la transmisión de contenidos funciona principalmente sobre UDP, puesto que se basa en otros protocolos como SRTP, por lo que está más orientado a la transferencia de medios P2P, es decir, entre un cliente y otro, sin pasar por un servidor intermediario.

2.2.1.2 Soluciones de propósito general

- **HTTP:** Todas las opciones que se basan en HTTP tienen una clara ventaja, y es que se pueden emplear certificados SSL/TLS para conseguir comunicaciones cifradas desde la capa de aplicación mediante HTTPS. Además, todas siguen la estructura básica y ampliamente conocida de HTTP, por lo que se trata de protocolos sencillos de implementar. Hay que mencionar también que los puertos 80 y 443 suelen estar abiertos en la mayoría de las infraestructuras de red a nivel empresarial, a diferencia de otros protocolos como RTP, SRT, etc., por lo que sería un aliciente más a favor de emplear este protocolo. Aunque se puede transmitir contenido multimedia directamente sobre tramas HTTP, existen distintas implementaciones orientadas y pensadas para ello.
    - ***Dash* (*Dynamic Adaptive Streaming over HTTP*):** *Dash* es un protocolo que permite la reproducción en *streaming* de contenidos multimedia a través de HTTP empleando una tasa de bits adaptativa en función de la calidad de la conexión del cliente. El contenido que se va a reproducir se encuentra almacenado en el servidor dividido en segmentos, que se codifican a distintas calidades, de modo que, a petición del cliente, se le servirán unas calidades u otras. Funciona mediante un fichero en el servidor llamado MPD (*Media Presentation Description*) en el que se presentan numerados y clasificados todos los segmentos con sus calidades disponibles, de modo que el cliente pueda realizar la petición adecuada al servidor [16]. El problema que presenta emplear este protocolo es que se debería generar la MPD antes de recibir el contenido de audio completo, así como generar los segmentos a medida que se fuese recibiendo el audio en el servidor, lo que aumentaría la latencia.
    - ***Low-Latency Dash*:** El protocolo *Dash* no está optimizado para la transmisión de contenidos con requisitos de baja latencia, pues la latencia mínima es, de base, igual o superior al tamaño de los segmentos en los que se divide el contenido. Sin embargo, hay implementaciones que permiten reducir la latencia de forma

significativa. Un ejemplo es *Fast-ll*, un servidor *Dash* en el que los contenidos se codifican en segmentos subdivididos en fragmentos de forma interna; de este modo, cuando el cliente solicita la descarga de un segmento, el servidor retiene la petición HTTP hasta que cuenta con el primer fragmento completo del segmento solicitado. De este modo, se consigue empezar la reproducción en el cliente antes incluso de que el primer segmento se haya codificado y subido al servidor por completo [17]. En el diagrama de la Figura 6, se puede observar el funcionamiento de este servidor *Dash*.

- **TCP:** El protocolo TCP es un protocolo de transporte, orientado a la conexión, que permite establecer una comunicación bidireccional entre un cliente y un servidor. Es el que se emplea para transportar tramas HTTP. Sin embargo, su uso embarcaría el desarrollo en nuevos retos como, por ejemplo, tener que definir la estructura de la trama en capa de aplicación que se vaya a emplear, manejar el mantenimiento de la conexión activa con mensajes *KeepAlive*, entre otros. Se trataría de la opción más personalizable, pero también más costosa de desarrollar.
- ***WebSockets*:** Según *Mozilla*, un *WebSocket* es una tecnología que "permite abrir una sesión de comunicación interactiva entre el navegador del usuario y un servidor". Posibilita el envío de mensajes HTTP en ambas direcciones entre el cliente y el servidor, de modo que este primero pueda "recibir respuestas controladas por eventos sin tener que consultar al servidor para una respuesta" [18].

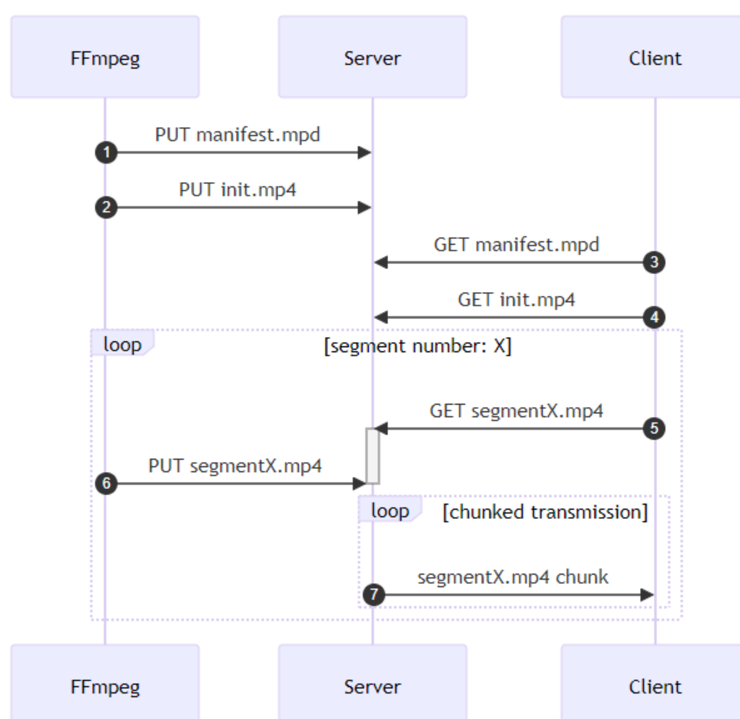

Figura 6. Diagrama de funcionamiento del servidor *Fast-ll*. Fuente: GitHub, *robelor/fast-ll* [17]

2.2.1.3 *Ventajas e inconvenientes de los* WebSockets

Una conexión *WebSocket* permite la comunicación bidireccional cliente-servidor a través de HTTP. Esto es algo que interesa a la hora de plantear este proyecto, pues se requiere que los clientes puedan recibir mensajes sin realizar ninguna consulta previa al servidor, por lo que necesitan mantener conexiones activas.

Al tratarse de una conexión TCP, los mensajes transmitidos no se pierden a menos que se interrumpa la conexión. Por ello, lo único que sucedería en caso de que hubiese algún error en la transmisión sería que aumentaría el retardo en la reproducción. Sin embargo, la naturaleza propia

de la comunicación en la aplicación hace que la experiencia de usuario no se vea degradada por este inconveniente; la comunicación no es continuada, sino unidireccional y, además es *push-to-talk*, por lo que no existe el requisito de inmediatez que sí está presente en una llamada de audio convencional, en la que todos los interlocutores deben tener el mínimo retardo posible sin excepción, para conseguir una conversación fluida.

Adicionalmente, el hecho de que se trate de una comunicación *one-to-many*, o "de uno a muchos", hace que sea necesaria la existencia de un servidor que retransmita los mensajes a todos los posibles receptores pues, de no existir este, el dispositivo móvil del emisor debería retransmitir tantos flujos como destinatarios hubiese en la comunicación. Dependiendo de cuántos receptores hubiese, la transmisión de un mensaje podría suponer una gran carga de computación al dispositivo móvil del emisor, además de un alto consumo de datos móviles, si fuese el caso. Por ello, se restringe la posibilidad de emplear cualquier protocolo no basado en servidor comentado en el punto 2.2.1.1.

Además, como la tecnología funciona sobre HTTP, como ya se ha comentado previamente, permite aprovechar certificados SSL/TLS para proporcionar encriptación a las conexiones. Es lo que se conoce como WSS (*WebSocket Secure*). En cuanto a los puertos, *WebSocket* trae la ventaja de poder emplear el mismo puerto tanto para la comunicación de voz como para el portal web, que se desarrollará más adelante.

Otra ventaja que se presenta con *WebSockets* es que, al mantener una conexión activa con el servidor en todo momento, además de poder recibir un mensaje desde el servidor sin necesidad de una petición previa del cliente, se reduce el *overhead* causado por la repetición de información en cabeceras HTTP, rebajando así el tamaño de cada uno de los paquetes enviados y, por tanto, la tasa binaria necesaria para funcionar.

Por otro lado, trae consigo la desventaja de no ser un protocolo dedicado exclusivamente a la transmisión de multimedia, por lo que puede no estar optimizado para ello, y no permitir la comunicación fluida. A partir de aquí, se formula la hipótesis de si se puede transmitir un flujo de audio en tiempo real a través de un *WebSocket*.

Para probar esta hipótesis, en primer lugar, se deberá definir la estructura de los paquetes que manejarán los *WebSockets*. Por suerte, según la especificación del protocolo, en el RFC 6455 [19], el mensaje *WebSocket* cuenta con un campo llamado "*Opcode*", de 4 bits, que permite escoger entre varios formatos de datos en su interior. Por ejemplo, un 0x1 indica que en el interior del mensaje se transporta un texto, mientras que un 0x2 denota una trama binaria, que es lo que interesa para poder transmitir audio codificado.

2.3 Pantallas empleadas

Durante el desarrollo de la aplicación Android, se ha planteado la estructuración de los elementos de la interfaz gráfica repartidos en cuatro pantallas:

- **Pantalla de grupos:** Se trata de la pantalla principal, y la primera en aparecer al iniciar la aplicación. Contará con una lista con los grupos en los que se encuentra el usuario que ha iniciado la sesión, con un botón con el que podrá controlar en qué grupo va a hablar. Además, en la parte inferior, se incorporará el botón con el que, además de actuar como *push-to-talk*, hará saber al usuario si puede o no hablar en el grupo seleccionado.
- **Pantalla de participantes de un grupo:** Se accede a esta pantalla pulsando en cualquier grupo de la pantalla de grupos. Contiene los participantes que componen el grupo en el que se ha pulsado, con una indicación del participante que está hablando en cada momento. Incorpora también el mismo botón *push-to-talk* que la pantalla de grupos. Cuando el usuario accede a esta pantalla, se seleccionará automáticamente a este grupo para hablar y escuchar.

- **Pantalla de ajustes:** Se accede pulsando sobre el icono situado en la parte derecha de la *app bar*, tanto desde la pantalla de grupos como desde la pantalla de participantes. En ella, se pueden configurar parámetros como la dirección del servidor al que se va a conectar el usuario, su propio nombre de usuario y contraseña o eliminar la aplicación del *background*.
- **Pantalla de conexión con dispositivos Bluetooth *Low Energy*:** Se puede acceder a esta pantalla desde la pantalla de ajustes. En ella, se puede observar el estado del dispositivo Bluetooth *Low Energy* registrado, es decir, si está conectado o desconectado, eliminar el botón registrado y buscar nuevos dispositivos.

Figura 7. Esquema general de diseño propuesto para las pantallas. De izquierda a derecha y de arriba a abajo: pantalla de grupos, pantalla de participantes de un grupo, pantalla de ajustes y pantalla de conexión con dispositivos Bluetooth *Low Energy*

Se puede ver en la Figura 7 un boceto del diseño que se plantea para la aplicación. Este diseño cuenta con una *app bar* en la parte superior de la pantalla, desde donde se podrá navegar a la pantalla de ajustes y retroceder a las pantallas anteriores.

2.3.1 Flujo de navegación entre las pantallas

A la hora de programar el funcionamiento de la navegación dentro de la aplicación, es necesario establecer previamente desde qué pantallas se debe poder acceder a otras, y cuál es el funcionamiento del botón de retroceso en cada una de ellas.

Se ha decidido establecer la pantalla de grupos como la pantalla principal, es decir, aquella que aparecerá en primera instancia en el momento de ejecutar la aplicación. A partir de esta pantalla, se ha planteado el siguiente esquema de flujo, que se puede observar en la Figura 8.

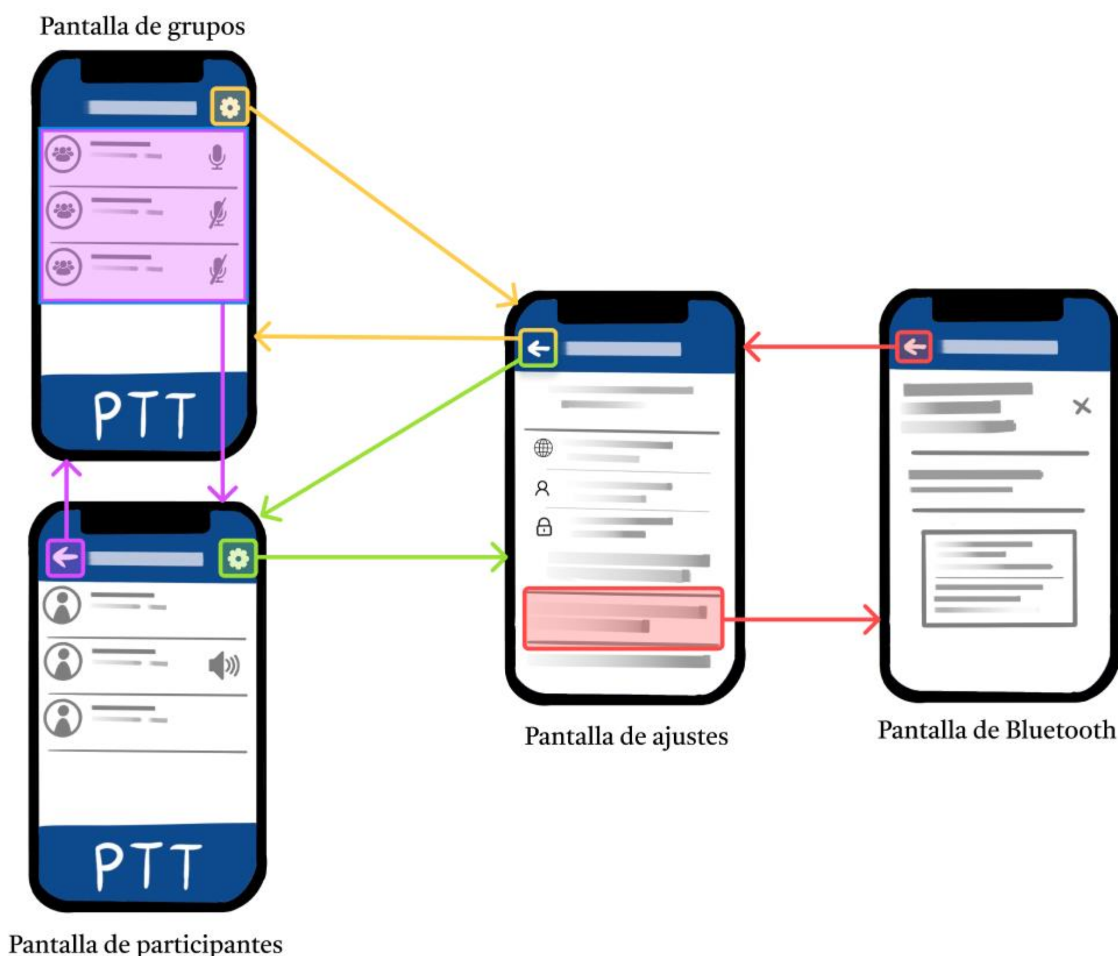

Figura 8. Flujo de navegación entre pantallas

Resulta más intuitivo observar el flujo de navegación entre las pantallas existentes de esta forma. La pantalla de los participantes de un grupo será accesible una vez seleccionado un grupo en la pantalla de grupos. La flecha de retroceso en la pantalla de participantes devolverá al usuario a la pantalla de grupos.

Desde ambas pantallas (grupos y participantes) se puede acceder a la pantalla de ajustes, cuya flecha de retroceso devolverá la pantalla desde la que se accedió a los ajustes inicialmente.

Por último, la pantalla de Bluetooth será accesible solamente desde la pantalla de ajustes, al presionar el botón "*Connect Bluetooth LE device*", y su flecha de retroceso llevará nuevamente a la pantalla de ajustes.

2.4 *Protocol Buffers*

2.4.1 *Introducción*

Protocol Buffers, también conocidos como *protobuf*, y cuyo logo se observa en la Figura 9, es un mecanismo de serialización de datos desarrollado por Google. Con este mecanismo, es posible definir eficazmente la estructura de los datos que se van a emplear en una aplicación, aumentando el rendimiento y la interoperabilidad de esta.

Soporta serialización y deserialización entre distintos lenguajes, como pueden ser Java, Python, Kotlin, Go, Swift, etc., por lo que se trata de un protocolo agnóstico de lenguaje.

Figura 9. Logo de *Protocol Buffers*

Protobuf permite, en este caso, serializar y estructurar los datos para ser intercambiados entre el cliente y el servidor, o almacenados de manera persistente en el cliente. Los datos se definen en archivos '*.proto*' utilizando una sintaxis específica, que posteriormente será compilada para generar código fuente en Java.

Con el objetivo de conseguir que esta compilación suceda en el mismo instante que se presiona el botón '*run*' para compilar la aplicación completa en Android Studio, se añade el código de la Figura 10 en el archivo '*gradle*' del proyecto.

```
1. protobuf {
2.     protoc {
3.         artifact = "com.google.protobuf:protoc:3.22.4"
4.     }
5.     generateProtoTasks {
6.         all().forEach {
7.             it.builtins.create("java") {
8.                 option("lite")
9.             }
10.         }
11.     }
12. }
```

Figura 10. Código *gradle* para compilar *protobuf* automáticamente

La sintaxis empleada a la hora de desarrollar los archivos '*.proto*' empieza por definir la versión de *protobuf* que se está empleando, en este caso, '*proto3*'. Posteriormente, se indica el paquete donde se va a generar el código dentro del proyecto. Por último, se definen los mensajes, los cuales, en caso de haber marcado como '*true*' la opción '*java_multiple_files*', se convertirán a Java como clases separadas.

```
1. syntax = "proto3";
2.
3. option java_package = "xyz.romeo.store";
4. option java_multiple_files = true;
5.
6.
7. message RomeoConfig{
8.   string server = 1;
9.   string userAlias = 2;
```

```
10.   string secret = 3;
11. }
```

Figura 11. Ejemplo de definición de un mensaje en *protobuf*

En la Figura 11, se observa un ejemplo de definición de un mensaje en *protobuf*. En el caso presente en dicha figura, se muestra cómo se almacenará el servidor, el nombre de usuario y la contraseña del cliente para que persista en el dispositivo móvil una vez introducido por el usuario.

Con la palabra '*message*' se define el nombre del mensaje *protobuf* que se va a crear. Posteriormente, se enumera cada uno de los parámetros que van a componer dicho mensaje. Además, se debe establecer el tipo de dato del parámetro que se esté definiendo, por ejemplo, *string*, *int32*, *bool*, etc. Finalmente, se numeran dichos parámetros, de modo que queden identificados a la hora de transmitirlos, y que se puedan deserializar correctamente en el destino. Esto incrementa la eficacia del protocolo y reduce el tamaño de los mensajes al no estar el nombre del parámetro dentro del mensaje serializado.

El código Java generado como resultado de compilar los archivos se encontrará en la carpeta *'java (generated)*', con cada clase dentro del paquete correspondiente, especificado en cada uno de los archivos, tal y como se observa en la Figura 12.

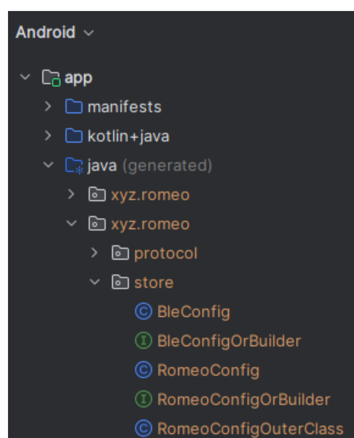

Figura 12. Ubicación de las clases Java generadas por el compilador de *protobuf*

Cada una de estas clases Java incluye los métodos *getters* y *setters* necesarios para poder serializar los datos instanciando la clase correspondiente, y deserializar los datos de un mensaje *protobuf* recibido.

2.4.2 Mensajes empleados

Los mensajes *protobuf* que se creen serán los que, posteriormente, se transmitan en formato binario a través de la conexión *WebSocket*. Para simplificar el proceso de envío y recepción de los mensajes a través de este canal, se ha decidido establecer un solo formato de mensaje que englobará a todo el resto de los mensajes, y será únicamente este el que finalmente se transmita.

Esto es posible gracias a la construcción '*oneof*' a la hora de declarar un mensaje. El nombre del mensaje principal es '*RomeoMessageMsg*'. En la Figura 13 se puede observar el nombre de cada uno de los mensajes que se han implementado para conseguir la funcionalidad deseada.

```
1. message RomeoMessageMsg{
2.   oneof message{
3.     KeepAliveRequestMsg keepAliveRequest = 1;            // Not used anymore
4.     KeepAliveResponseMsg keepAliveResponse = 2;          // Not used anymore
5.     AuthorizationRequestMsg authorizationRequest = 3;
6.     AuthorizationResponseMsg authorizationResponse = 4;
7.     GroupsRequestMsg groupsRequest = 5;
8.     ParticipantsRequestMsg participantsRequest = 6;
9.     GroupsResponseMsg groupsResponse = 7;
```

```
10.     ParticipantsResponseMsg participantsResponse = 8;
11.     TalkPermissionRequestMsg talkPermissionRequest = 9;
12.     TalkPermissionResponseMsg talkPermissionResponse = 10;
13.     ChannelReleaseRequestMsg channelReleaseRequest = 11;
14.     ChannelReleaseResponseMsg channelReleaseResponse = 12;
15.     AudioStartMsg audioStart = 15;
16.     AudioMsg audio = 16;
17.     AudioEndMsg audioEnd = 17;
18.     RttTest rttTest = 18;
19.     RestartMsg restartMsg = 19;
20.   }
21. }
```

Figura 13. Definición del mensaje principal del protocolo

A la hora de crear el mensaje, será necesario primero construir uno de los mensajes que se muestran, y posteriormente construir el *RomeoMessageMsg* con el método '*set*' y el mensaje anterior.

A continuación, se detallará el funcionamiento de los mensajes más importantes para la interoperabilidad de la aplicación, así como los campos que incorporan.

En la mayor parte de los mensajes se incluye un parámetro llamado '*requestId*'. Este parámetro, tal y como se desarrollará la aplicación, no se usará. Sin embargo, se ha decidido incluir de antemano para el caso en que se pueda emplear el protocolo UDP para transportar los mensajes. De este modo, ya quedarían identificados con un identificador único.

2.4.2.1 Mensajes de autorización

El mensaje '*AuthorizationRequestMsg*' realiza la función de transportar la información de inicio de sesión y autenticación del usuario en el servidor. Como se ve en la Figura 14, incluye una enumeración con el tipo de autenticación transportada (en este caso, solo se soporta la autenticación en plano), el nombre de usuario '*userAlias*' y su contraseña '*secret*'.

```
 1. message AuthorizationRequestMsg{
 2.   enum AuthorizationTypeMsg{
 3.     CLEAR = 0;
 4.   }
 5.   int32 requestId = 1;
 6.   string userAlias = 2;
 7.   AuthorizationTypeMsg type = 3;
 8.   string secret = 4;
 9. }
10.
11. message AuthorizationResponseMsg{
12.   enum AuthorizationResultMsg{
13.     SUCCESSFUL = 0;
14.     FAILURE = 1;
15.   }
16.   int32 requestId = 1;
17.   AuthorizationResultMsg result = 2;
18. }
```

Figura 14. Mensaje de petición y respuesta de autorización en el servidor

La contestación apropiada a este mensaje es un '*AuthorizationResponseMsg*', que incluye una enumeración con dos posibles valores: '*SUCCESSFUL*' y '*FAILURE*'.

2.4.2.2 Mensajes de petición y respuesta de grupos y participantes

Una vez el usuario se haya autenticado correctamente en el servidor, realizará peticiones para obtener la información de todos los grupos a los que pertenece, así como de todos los usuarios registrados en la aplicación. Para ello, se implementan mensajes como '*GroupsRequestMsg*' y

'*ParticipantsRequestMsg*', que no contienen ningún parámetro relevante, puesto que el hecho de tratarse de este tipo de mensajes ya indica al servidor lo que tiene que devolver.

Las respuestas a estos mensajes son como las que se indican en la Figura 15.

```
1. message GroupsResponseMsg{
2.   int32 requestId = 1;
3.   repeated GroupMsg groups = 2;
4. }
5.
6. message ParticipantsResponseMsg{
7.   int32 requestId = 1;
8.   repeated ParticipantMsg participants = 2;
9. }
10.
11. message ParticipantMsg{
12.   string uuid = 1;
13.   string userAlias = 2;
14.   string name = 3;
15.   string description = 4;
16.   string pictureUrl = 5;
17. }
18.
19. message GroupMsg{
20.   string groupAlias = 1;
21.   string name = 2;
22.   string description = 3;
23.   string pictureUrl = 4;
24.   repeated ParticipantMsg participants = 5;
25. }
```

Figura 15. Mensajes de petición y respuesta de grupos y participantes (l. 1-9). Definición de participantes y grupos (l. 11-25)

Cabe destacar el uso del término '*repeated*', con el que se puede construir el mensaje usando una lista del elemento deseado. Por ejemplo, un *GroupMsg* tiene una lista de *ParticipantMsg*. Tanto los participantes como los grupos se identifican unívocamente por su *alias*.

2.4.2.3 Mensajes de ocupación del canal y permiso de habla

Se han definido dos pares de mensajes, petición y respuesta, para:

- Solicitar la ocupación del canal de habla en un grupo determinado, cuando el usuario quiere empezar a hablar.
- Solicitar la liberación del canal una vez que dicho usuario ha terminado de utilizarlo.

En el caso del primer mensaje, llamado *TalkPermissionRequest*, se incluye el grupo en el que el usuario desea iniciar la comunicación. La respuesta al mismo incluirá un parámetro de tipo *bool* con la respuesta, que será *true* si no hay nadie hablando en el mismo instante, o *false* si ya había alguien hablando. De este modo, se evitan colisiones.

El segundo mensaje, *ChannelReleaseRequest*, se envía una vez se suelta el botón *push-to-talk*, para notificar al servidor que se va a terminar el mensaje de audio. En la Figura 16 se pueden observar estos cuatro mensajes.

```
1. message TalkPermissionRequestMsg{
2.   int32 requestId = 1;
3.   GroupMsg group = 2;
4. }
5.
6. message TalkPermissionResponseMsg{
7.   int32 requestId = 1;
8.   bool permission = 2;
9. }
```

```
10.
11. message ChannelReleaseRequestMsg{
12.   int32 requestId = 1;
13.   GroupMsg group = 2;
14. }
15.
16. message ChannelReleaseResponseMsg{
17.   int32 requestId = 1;
18.   bool response = 2;
19. }
```

Figura 16. Mensajes de petición y respuesta para la solicitud de permiso de habla (l. 1-9) y liberación del canal (l. 11-19)

2.4.2.4 Mensajes de audio

Con el objetivo de controlar la reproducción de los mensajes de audio en los dispositivos móviles, se ha decidido incorporar un par de mensajes para indicar el inicio y el final de la reproducción. Son '*AudioStartMsg*' y '*AudioEndMsg'*, respectivamente. En ambos mensajes, como sucede con el mensaje de audio *AudioMsg*, se indica el alias del usuario que está hablando y del grupo en el que está. Además, en el mensaje de inicio, se incluye un campo en el que se indica el *jitter* que sufre el hablante, de modo que se pueda adaptar el búfer del reproductor adecuadamente, cosa que se expondrá en un punto posterior de esta memoria. Finalmente, *AudioStartMsg* también incluye una enumeración con las posibles codificaciones que presenta el audio. En el caso que concierne a la aplicación, siempre será *Opus*.

En el mensaje de audio, además del alias del usuario y del grupo, aparecen el número de secuencia de la muestra de audio, obtenido por el bloque de grabación *AudioRecorder*, y un parámetro de tipo *bytes* con el audio codificado. En la Figura 17 se pueden ver los tres mensajes.

```
 1. message AudioStartMsg{
 2.   string groupAlias = 1;
 3.   string userAlias = 2;
 4.   double speakerJitter = 3;
 5.   enum AudioType{
 6.     CLEAR = 0;
 7.     OPUS = 1;
 8.   }
 9. }
10.
11. message AudioMsg{
12.   string groupAlias = 1;
13.   string userAlias = 2;
14.   int32 sequenceNumber = 3;
15.   bytes audioData = 4;
16. }
17.
18. message AudioEndMsg{
19.   string groupAlias = 1;
20.   string userAlias = 2;
21. }
```

Figura 17. Mensajes de inicio de audio (l. 1-9), muestra de audio (l. 11-16) y fin de audio (l. 18-21)

2.5 Características de audio en la transmisión

En puntos previos, se ha comentado que la transmisión de audio se realizará a través de la propia conexión *WebSocket* que se establezca con el servidor. Esto fuerza al desarrollo a determinar las características de los paquetes que se van a transmitir de modo que se logre un rendimiento adecuado junto con una latencia lo suficientemente baja. Todo ello, adaptando el audio a las capacidades de los micrófonos y altavoces con los que cuentan los dispositivos Android.

La documentación de Android recomienda escoger entre dos frecuencias de muestreo: 44.1 kHz y 48 kHz [20]. Estas dos frecuencias están soportadas por la gran mayoría de dispositivos Android, además que, en caso de grabar a una frecuencia no soportada, el *resampler* la adaptará a la frecuencia correcta. Como se verá en el punto 2.5.1, el codificador que se empleará será *Opus*. Este *codec* de audio admite 8, 12, 16, 24 o 48 kHz [21], por lo que se ha escogido esta última para la frecuencia de muestreo a la que grabará la aplicación.

En cuanto al paquete de audio como tal, el punto 2.4.2.4 ya ha ejemplificado el formato en el que se encapsulará para su transporte. Resta comentar el tamaño del campo '*bytes*' que lo forma. En este punto, se debe valorar la cantidad de tiempo que almacenará cada paquete, pues dicha cantidad de tiempo constituirá ya un *delay*, que se sumará al resto de retardos que se produzcan en el trayecto, hasta el punto de la reproducción.

Sin embargo, un número elevado de paquetes por segundo aumentará sustancialmente la tasa binaria necesaria en transmisión, debido a las cabeceras y otros datos que acompañan a cada una de las muestras de audio que se transmiten; es lo que se conoce como *overhead*.

Opus, en su documentación, también aclara el número de muestras de audio por paquete, para una frecuencia de muestreo de 48 kHz: 120, 240, 480, 960, 1920 o 2880 muestras. Dividiendo la tasa de muestreo entre el número de muestras por paquete se obtiene la cantidad de paquetes de audio por segundo que se transmitirán. Con el fin de minimizar el retardo, pero manteniendo en la medida de lo posible un número bajo de paquetes por segundo, se ha establecido la cantidad de 1920 muestras de audio en cada paquete, es decir, una tasa de 25 paquetes por segundo. Esto equivale a muestras de $^{1}/_{25}\,s = 40\,ms$ de duración cada una.

Además, la codificación será PCM a 16 bits por muestra. Teniendo en cuenta que la pista de audio será mono canal, la tasa de bits del flujo de audio plano se puede calcular siguiendo la ecuación (2.1).

$$48 \cdot 10^3 \frac{muestras}{segundo} \cdot 16 \frac{bits}{muestra} = 768 \cdot 10^3\, bps = 768\, kbps \quad (2.1)$$

2.5.1 Codificación

Dados los requisitos que se han establecido en el diseño de la aplicación, la transmisión de voz debe ser eficiente y tener una calidad suficiente para garantizar una comunicación clara y rápida. Para conseguirlo, es necesario contar con un codificador con el que comprimir el audio de tal forma que se reduzca el ancho de banda necesario para la transmisión, manteniendo una elevada calidad de audio, una latencia suficientemente baja, y aportando robustez ante posibles errores en la transmisión.

La plataforma Android soporta de manera nativa algunos codificadores de audio. Estos codificadores se actualizan conforme lo hace el sistema operativo, lo que dificulta mantener una uniformidad entre todos los usuarios que instalen la aplicación en sus dispositivos móviles.

Es por ello por lo que se ha decidido emplear *Opus*, un codificador de código abierto y disponible en todas las plataformas, puesto que se trata de una librería que se compila y se incluye automáticamente en la aplicación. De este modo, el codificador puede actualizarse de manera independiente al sistema operativo, y se consigue uniformidad entre todos los usuarios.

2.6 Gestión de los datos

En el desarrollo de aplicaciones, es fundamental aclarar desde un primer momento la forma en la que se van a almacenar los datos necesarios para su funcionamiento, pues supone un gran impacto tanto en el código que se vaya a escribir como en la funcionalidad que se consiga. Por otro lado, es importante tener claro el contexto en el que se va a desplegar esta aplicación, de modo que se pueda adaptar lo mejor posible a la mayoría de los entornos.

Los datos que van a ser manejados por la aplicación son, fundamentalmente, pares nombre-valor, listas de participantes y listas de grupos, que a su vez deben incluir una lista con referencias a los participantes que los conforman. Además, cabe destacar que la base de datos solo sería necesaria en la parte del servidor, ya que cada cliente mantendría su información en línea para poder iniciar sesión desde cualquier dispositivo.

Teniendo en cuenta todas estas condiciones, una base de datos relacional como *MySQL* puede parecer la mejor opción, y se podría integrar con Kotlin gracias al controlador JDBC. Si se emplease finalmente esta base de datos, es previsible que existiese una tabla para almacenar los datos de los usuarios, otra tabla para los grupos, y una última tabla que relacionase los participantes con cada uno de los grupos a los que pertenecen. Se puede observar el planteamiento en la Figura 18.

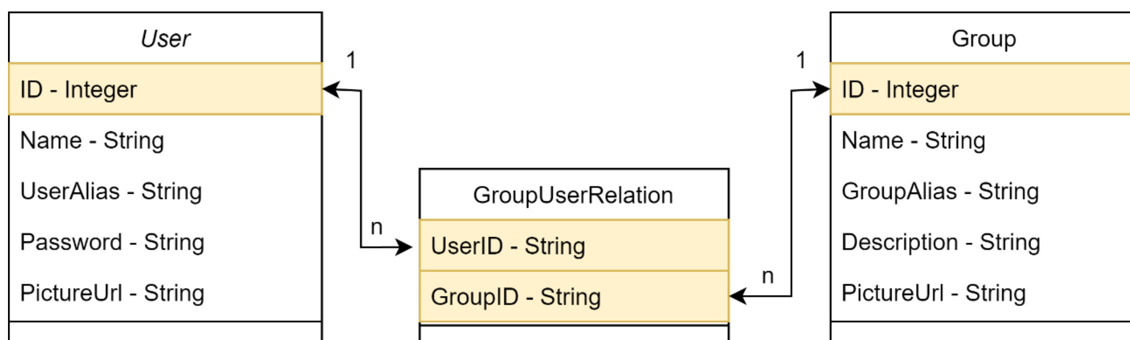

Figura 18. Posible diseño de base de datos relacional

El problema de esta elección es que la parte de servidor de la aplicación pasaría a necesitar de un programa externo (en este caso, el servidor de *MySQL*) para funcionar. Aunque se podrían valorar otras opciones, como bases de datos de persistencia de objetos embebidas en Kotlin, es ciertamente previsible que las empresas, entorno en el que se desplegará el servidor, ya cuenten con su propia infraestructura de almacenamiento de usuarios y grupos de trabajo, como pueden ser LDAP o *Active Directory*.

Además, es necesario tener en cuenta que, en el servicio que se ha desarrollado, solamente será necesario modificar la BBDD cuando se añadan, modifiquen o eliminen usuarios o grupos, por lo que es probable que no se puedan aprovechar todas las capacidades de una BBDD relacional más potente. Del mismo modo, al implementar una base de datos tan simple, se hace viable el hecho de mantenerla constantemente en la memoria del programa, simplificando el desarrollo y la distribución.

Todos estos motivos llevan a explorar otras posibilidades, entre las que se encuentra el uso de una base de datos *NoSQL*. Este tipo de bases de datos no relacionales puede ser de varios tipos, como las BBDD de documentos, de pares clave-valor, de columnas o de grafos. En este caso, por sencillez, se ha optado por emplear una base de datos de documentos en formato JSON. De este modo, cuando el servidor se inicialice, deserializará los datos almacenados en el documento JSON indicado para obtener todos los objetos de usuarios y grupos que se vayan a emplear.

En la Figura 19 se puede ver un ejemplo de la estructura que se plantea para este documento JSON.

```
1. {
2.   "app": "romeo",
3.   "version": "1.0.0",
4.   "adminUsername": "admin",
5.   "adminPassword": "admin",
6.   "participants": [
7.     {
```

```
 8.       "id": 1,
 9.       "userAlias": "user1",
10.       "hashpwd": "123",
11.       "name": "First User",
12.       "description": "This is the first user",
13.       "pictureUrl": "https://picsum.photos/200"
14.     },
15.     {
16.       "id": 2,
17.       "userAlias": "user2",
18.       "hashpwd": "123",
19.       "name": "Second User",
20.       "description": "This is the second user",
21.       "pictureUrl": "https://picsum.photos/200"
22.     }
23.   ],
24.   "groups": [
25.     {
26.       "id": 1,
27.       "groupAlias": "group1",
28.       "name": "First Group",
29.       "description": "This is the first group",
30.       "pictureUrl": "https://picsum.photos/200",
31.       "participants": [
32.         1,
33.         2
34.       ]
35.     }
36.   ]
37. }
```

Figura 19. Ejemplo de estructura de la base de datos JSON

Capítulo 3. Desarrollo Android

3.1 Utilidades de Kotlin

3.1.1 Corutinas

A lo largo del desarrollo de una aplicación móvil, suele ser habitual enfrentarse a la necesidad de realizar múltiples tareas de manera concurrente, es decir, simultáneamente, sin bloquear la ejecución del hilo principal. Por ejemplo, se puede pensar en algo tan simple como manejar un contador, hasta algo más complejo como la descarga de datos a través de una API, y que posteriormente deberán ser mostrados en la interfaz.

Tradicionalmente, esto se ha realizado empleando *threads*, esto es, distintos hilos de ejecución de la CPU realizando tareas diferentes. Android cuenta con APIs para manejar los *threads*. Sin embargo, a partir de la versión 1.3 de Kotlin, se introdujo el concepto de corutina, siendo recomendada por Google para su uso, pues, además de ser muy ligeras y poder lanzar una gran cantidad de ellas simultáneamente, evitan posibles errores que pudiesen ser causados por los *threads* [20].

Las corutinas son una característica del lenguaje Kotlin que facilita la escritura de código asíncrono, facilitando la gestión de tareas en segundo plano sin bloquear el hilo principal de ejecución. Para iniciar corutinas y ejecutar código en segundo plano, es necesario comprender algunos aspectos acerca de ellas:

- ***CoroutineScope***: Define el ámbito en el que se añadirán las nuevas corutinas, de modo que todas ellas hereden el mismo contexto, además de la cancelación. Un ejemplo de creación de un *CoroutineScope* es:
  `val scope = CoroutineScope(Job() + Dispatchers.Default)`
  Un *job* es una clase que representa el ciclo de vida de una corutina, que puede pasar por diferentes estados, entre los que se encuentran '*active*', '*completed*' y '*cancelled*'. Al definir `Job()` en la instancia del *CoroutineScope*, se define un *job* padre para todas las corutinas que se lancen en este *scope*, de modo que se puedan manejar estas corutinas de forma colectiva [10].
- ***Dispatchers***: Es un componente que establece en qué hilo o *pool* de hilos debe lanzarse una corutina. Hay varios tipos de *dispatchers* [10], siendo dos de ellos los más empleados:
  - ***Dispatchers.Default***: Utiliza un *pool* común de hilos en segundo plano. Según la documentación de Kotlin, es el más apropiado para corutinas computacionalmente costosas.
  - ***Dispatchers.IO***: Emplea hilos creados bajo demanda, especialmente diseñados para operaciones de Entrada/Salida, como manejo de archivos o *sockets* bloqueantes.
- **Función *launch***: Es la encargada de crear una corutina en el *scope* en el que se ejecuta la función, y la despacha al *Dispatcher* correspondiente.

En la Figura 20, se observa un ejemplo de uso de las corutinas en código. En este caso, se han empleado para manejar los mensajes que recibe el *WebSocket*.

```
1.     fun start() {
2.         scope = CoroutineScope(Job() + Dispatchers.IO)
3.         scope?.launch {
4.             while (isActive) {
5.                 try {
6.                     webSocketClient.webSocket("$server/romeo") {
7.                         // update current WebSocket
8.                         webSocketSession = this
9.
10.                        for (frame in incoming) {
```

```
11.                             frame as? Frame.Binary
12.                                 ?: continue
13.                             val romeoMessage: RomeoMessageMsg
14.                             // …
15.                         }
16.                     }
17.                 } catch (e: Exception) {
18.                     // …
19.                 }
20.                 delay(2000)
21.             }
22.         } ?: run {
23.             d { "Connection failed: scope is null" }
24.         }
25.     }
```

Figura 20. Ejemplo de uso de corutinas para recibir mensajes desde el *WebSocket*

Se puede apreciar en ese mismo ejemplo el uso de elementos propios de la corutina, como la función *isActive* propia del *Job*, o la posibilidad de parar la ejecución durante un periodo determinado de tiempo con la función *delay*.

3.1.2 Flows *y* StateFlows

Según la documentación de *Android Developers*, hablando de corutinas, un *Flow* es un tipo de dato con el que se pueden emitir múltiples valores de forma asíncrona, coleccionándose en otras partes del código [20]. Se establecen así las figuras del 'productor', que será quien emita los valores que se añaden al flujo, y del 'consumidor', que será quien los reciba y los maneje. Se dice que los flujos son '*cold*', puesto que las funciones que se implementan dentro del *Flow* no se ejecutan hasta que el *Flow* se coleccione.

A lo largo del desarrollo de la aplicación, se ha hecho uso de otro tipo de *Flow* conocido como *MutableStateFlow*. Este objeto hereda de otro conocido como *SharedFlow*, que se trata de un *Flow* '*hot*', es decir, un flujo que comparte los valores emitidos entre todos sus colectores y que se mantiene activo independientemente de estos últimos. La principal diferencia entre un *StateFlow* y un *SharedFlow* reside en que se puede obtener el valor actual del *StateFlow* con su propiedad '*value*' [10].

El uso de *MutableStateFlows* permite la definición de un *data-model* práctico y accesible desde cualquier parte del código, actuando como una '*single source of truth*', además de conformar un flujo de datos unidireccional, técnicas que se recomiendan desde la guía de *Android Developers* [20]. Cabe destacar también la utilidad de estos en el ámbito de la notificación de eventos pues, al tratarse de funciones concurrentes, permiten ejecutar código cada vez que se colecta un nuevo valor en el flujo.

Este tipo de flujos en Kotlin permiten realizar operaciones con los valores de varios flujos, por ejemplo:

- ***Zip*:** Crea un nuevo flujo con los pares de valores que se han ido recolectando en ambos flujos, de modo que, en caso de que se reciban más en un flujo que en el otro, la ejecución se bloquee hasta tener el par completo y poder emitirlo.
- ***Combine*:** Tiene un funcionamiento similar al *zip*, pero el par de valores en el nuevo flujo se emite cada vez que se colecciona alguno de los dos flujos, y lo hace con el último valor que se haya emitido en cada uno de ellos.

En este caso, se han empleado los flujos para notificar el cambio de valor en la gran mayoría de datos del *model*, y actuar en consecuencia con el valor modificado. Por ejemplo, en la Figura 21 se puede observar cómo se usa un *MutableStateFlow* para controlar la pulsación del botón *push-to-talk*.

```
1. // GroupInfoScreenViewModel
2. fun processUiAction(uiAction: UiAction) {
3.     when (uiAction) {
4.         UiAction.ButtonTalkPressed -> {
5.             viewModelScope.launch {
6.                 model.talkButtonPressed.emit(true)
7.             }
8.         }
9.         UiAction.ButtonTalkReleased -> {
10.            viewModelScope.launch {
11.                model.talkButtonPressed.emit(false)
12.            }
13.        }
14.    }
15. }
16.
17. // Model
18. scope.launch {
19.     _talkButtonPressed.collect { pressed ->
20.         if (pressed) {
21.             talkingGroup.value?.let {
22.                 group -> askPermissionToTalk(group.toProtoGroup())
23.             }
24.         } else {
25.             talkingGroup.value?.let {
26.                 group -> releaseChannel(group.toProtoGroup())
27.             }
28.         }
29.     }
30. }
```

Figura 21. Ejemplo de uso de *MutableStateFlows* para controlar la pulsación del botón *push-to-talk*

3.2 Arquitectura Model-View-Intent

En la programación Android, es fundamental decidir la arquitectura de la interfaz gráfica que se va a seguir, con el fin de establecer desde un primer momento dónde se van a guardar los datos, cómo se va a representar la interfaz, y cómo los eventos de la interfaz deben actualizar los datos de la aplicación y renderizar los cambios adecuadamente. En general, se requiere para definir un flujo de datos y una estructura homogénea dentro de la aplicación.

Existen múltiples arquitecturas, siendo las más comunes, por ejemplo:

- ***Model-View-Controller* (MVC):** El *model* maneja los datos, el *view* gestiona la interfaz de usuario, y un *controller* media entre las interacciones del *model* y el *view*. Por ejemplo, el *controller* puede mantener el código que se ejecutará al presionar el botón de la vista, mostrar otra vista, etc.
- ***Model-View-ViewModel* (MVVM):** Es similar a MVC, pero en este caso el *ViewModel* no interactúa con la vista, como sí sucede en MVC, sino que el *View* observa los cambios que se producen en él. La vista, por lo tanto, está más desacoplada del *ViewModel* en MVVM que del *Controller* en MVC, lo que facilita la reutilización de código y el testeo.

La arquitectura MVVM ha sido recomendada por Google para el desarrollo de interfaz gráfica y, de hecho, esta arquitectura origina el uso de *ViewModels* en este proyecto concreto. Aunque no existe ningún problema en emplearla, es una arquitectura más enfocada al uso de *Views* realizados mediante XML, por lo que se ha preferido optar por una opción distinta.

En este caso, se ha escogido una arquitectura llamada *Model-View-Intent* (MVI). Esta arquitectura está basada en la adopción de flujos de datos unidireccionales, cosa que aporta fuerza a la convención '*single source of truth*' de la que se ha hablado en el punto 3.1.2, puesto que los datos

únicamente se moverán desde el *Model* hasta el *View*, siendo actualizados gracias a *intents* que se ejecutan en el *ViewModel* a causa de la interacción del usuario con la interfaz gráfica.

Una de las implementaciones más empleadas para el desarrollo de esta arquitectura es *Orbit*. A grandes rasgos, en la arquitectura MVI, el *Model* almacenará todos los datos relativos al estado de la aplicación, así como los objetos y la lógica necesarios para el correcto funcionamiento de esta. Existirá también un objeto *ViewModel* para cada una de las pantallas, que almacenará en un *Orbit Container* el estado y la información necesaria para la pantalla que gestiona. Cuando los valores del *Model* al que se haya suscrito un contenedor de estado se modifiquen, se lanzará la función *intent*, de la librería de *Orbit*, para actualizar dicho contenedor. Al actualizarse este contenedor de estado, la interfaz gráfica se actualizará también automáticamente, consiguiendo así la unidireccionalidad de los datos. El funcionamiento de la arquitectura de la interfaz gráfica se muestra en la Figura 22.

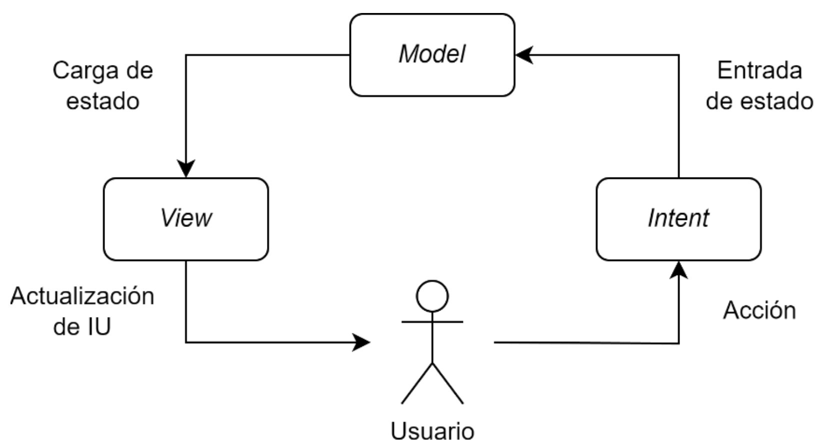

Figura 22. Ciclo de datos de *Model-View-Intent*

Un *intent* no es más que una intención de realizar una acción. Esta intención lanzará el código necesario para que el *Model* consiga nuevos datos, y estos se muestren en la interfaz, tal y como se ha comentado. Puede ser, por ejemplo, la acción de presionar un botón. Es necesario aclarar que un *intent* en la arquitectura MVI no tiene ninguna relación, más allá del nombre, con los *intents* que Android emplea para comunicarse con el sistema, y realizar acciones como iniciar otro *Activity*, seleccionar una fotografía de la galería, etc.

Para observar cómo se ha implementado esta funcionalidad programáticamente, se va a recurrir a un ejemplo en código. En este caso, el ejemplo trata sobre cómo se ha gestionado la pantalla de grupos.

```
 1. data class GroupsScreenState(
 2.     val status: ServerConnectionStatus = ServerConnectionStatus.DISCONNECTED,
 3.     val authStatus: ClientAuthStatus = ClientAuthStatus.UNAUTHENTICATED,
 4.     val groups: List<Group> = mutableListOf(),
 5.     val talkingGroup: Group?,
 6.     val listeningGroups: List<Group>,
 7.     val talking: Boolean,
 8.     val playingAudio: Boolean = false,
 9.     val currentlyTalkingGroup: Group? = null,
10.     val currentAudioSamples: FloatArray? = null
11. )
12.
13. sealed class UiAction {
14.     data class StartTalking(val group: Group) : UiAction()
15.     data class StartListening(val group: Group) : UiAction()
16.     data class FocusIn(val group: Group) : UiAction()
17.     data object ButtonTalkPressed : UiAction()
```

```
18.     data object ButtonTalkReleased : UiAction()
19. }
```

Figura 23. Orbit Container que representa el estado de la pantalla de grupos (l. 1-11), e *intents* disponibles para realizar desde el *View* (l. 13-19)

En la Figura 23, se han declarado, por un lado, todas las variables de estado que se necesitan en la vista y, por otro lado, todos los *intents* que podrá ejecutar el usuario al interactuar con la interfaz gráfica.

```
 1. class GroupsScreenViewModel(private val model: Model) : ViewModel(),
        ContainerHost<GroupsScreenState, Nothing> {
 2.
 3.
 4.     override val container = container<GroupsScreenState, Nothing>(
 5.         GroupsScreenState(
 6.             talkingGroup = model.talkingGroup.value,
 7.             listeningGroups = model.listeningGroups.value,
 8.             talking = model.talkingAllowed.value
 9.         )
10.     )
11.
12.
13.     fun processUiAction(uiAction: UiAction) {
14.         when (uiAction) {
15.             is UiAction.StartTalking -> {
16.                 viewModelScope.launch {
17.                     if (!model.talkingAllowed.value) {
18.                         model.talkingGroup.emit(uiAction.group)
19.                     }
20.                 }
21.             }
22.
23.             // …
24.
25.         }
26.     }
27.
28.     init {
29.
30.         viewModelScope.launch {
31.             model.serverConnectionStatus.collect {
32.                 intent { reduce { state.copy(status = it) } }
33.             }
34.         }
35.
36.         // …
37.     }
38. }
```

Figura 24. ViewModel que gestiona interacciones y actualización de la pantalla de grupos

La clase *GroupsScreenState* que se observa en la Figura 23 se crea como *Orbit Container* en el *ViewModel* de la forma que se muestra en las líneas 4 a 10 de la Figura 24. De esta forma, se podrá acceder al estado del *ViewModel* desde el *View*. Además, se implementan todos los métodos que conforman los *intents* en una función llamada *processUiAction*, que se pasará al *View* con `groupsScreenViewModel::processUiAction`. También se implementan en el constructor *init* de la clase *ViewModel* todos los *intents* necesarios para actualizar el estado y, con ello, también la interfaz gráfica.

Se puede observar además en la Figura 24 el uso tanto de corutinas como de *Flows* para manejar el flujo de datos. Con el método *collect*, cada vez que se emita un nuevo valor en el *StateFlow*, se ejecutará el código dentro de ese bloque, en este caso, para actualizar el estado.

3.3 Sistema de inyección de dependencias

A la hora de crear un objeto en Kotlin, es probable que dicho objeto requiera de otros objetos para funcionar. Siguiendo con el ejemplo de la Figura 24, la pantalla que tiene asociado a este *ViewModel* necesitará tener acceso a este último para emplear, como ya se ha visto, las funciones de los *intents* o el estado en sí mismo. Si se instanciase el *ViewModel* directamente dentro de la pantalla, cada vez que se crease este objeto se crearía un objeto distinto de *ViewModel*, haciendo que los datos o el estado que se guarda en él se deba regenerar en el mismo punto en el que se dejó cuando se eliminó anteriormente, y esto puede acarrear problemas.

Es por ello por lo que, en lugar de crear un nuevo objeto para suplir esa dependencia, el objeto, que existe en el contexto de la aplicación en general, se debería 'inyectar' directamente en aquellos sitios donde sea necesario.

De entre todas las librerías existentes capaces de realizar inyección de dependencias, que se comentarán próximamente, se ha elegido *Koin*. Se trata de una librería capaz de manejar inyección de dependencias, no sólo en Android, sino también en aplicaciones multiplataforma, servidores como *Ktor*, etc.

Consiste en declarar todos los módulos que tendrá *Koin* a su disposición. Se pueden declarar esos módulos como *singleton* para que siempre se obtenga la misma instancia de un determinado objeto, *factory* para obtener instancias distintas, o incluso como *ViewModel*, de modo que sea el propio sistema de inyección de dependencias el que se encargue de mantener la instancia de los *ViewModels* que se crean para cada pantalla. En la Figura 25 se puede observar cómo se definen dichos módulos.

```
 1. val appModule = module {
 2.     singleOf(::Model)
 3. }
 4.
 5. val settingsScreenViewModel = module {
 6.     viewModelOf(::SettingsScreenViewModel)
 7. }
 8.
 9. val groupsScreenViewModel = module {
10.     viewModelOf(::GroupsScreenViewModel)
11. }
12.
13. // …
```

Figura 25. Ejemplo de declaración de módulos de *Koin*

Una vez declarados todos los módulos, en la clase *Application* principal se debe iniciar *Koin*, proporcionándole además el contexto Android (l. 7), de la forma en la que se muestra en la Figura 26. Como se puede observar, se deben instanciar todos los módulos previamente declarados (l. 8-17) para que sean accesibles desde cualquier lugar del contexto.

```
 1. class App : Application() {
 2.     override fun onCreate() {
 3.         super.onCreate()
 4.         // …
 5.         startKoin {
 6.             androidLogger()
 7.             androidContext(this@App)
 8.             modules(
 9.
10.                 appModule,
11.                 settingsScreenViewModel,
12.                 groupsScreenViewModel,
13.                 groupInfoScreenViewModel,
14.                 bluetoothScreenViewModel,
15.                 permissionsViewModel
```

```
16.
17.             )
18.         }
19.         // …
20.     }
21. }
```

Figura 26. Inicialización de Koin en la clase *Application*

3.3.1 Beneficios e inconvenientes. Alternativas

Se presentan en este caso dos alternativas principales a *Koin*: *Dagger* y *Hilt*. *Dagger* es la librería de inyección de dependencias más antigua, mientras que *Hilt* es otra librería que emplea *Dagger* internamente para adaptarse a su uso en Android.

El principal beneficio que presenta *Koin* es la simplicidad. A diferencia del resto de sistemas de inyección de dependencias comentados, no funciona con anotaciones, ni genera código en tiempo de compilación, por lo que el resultado es un código más legible y sencillo de entender. El punto negativo que esto conlleva es que, en caso de haber configurado mal una dependencia, *Dagger* y *Hilt* darían un fallo en tiempo de compilación y no compilarían, mientras que *Koin*, al funcionar en tiempo de ejecución, sí compilaría, e incluso podría no dar ningún error al iniciar la aplicación, lo que podría llevar a confusión.

Por supuesto, como era de esperar, *Dagger* y *Hilt* tienen un cierto impacto en tiempo de compilación puesto que, como ya se ha comentado, deben generar el código necesario para implementar la inyección de dependencias en cada caso concreto, pero esto hace que aumente la eficiencia del código en tiempo de ejecución.

3.4 *NavigationScreen*

En este punto de la memoria, se va a tratar la gestión de las múltiples pantallas con las que cuenta la aplicación, y cómo es posible pasar de una pantalla a otra y retroceder animando la transición e incluso transfiriendo información entre pantallas.

3.4.1 *Sustitución de los* Activity

A lo largo del desarrollo de la plataforma Android, y especialmente con la introducción de tecnologías y herramientas como *Jetpack Compose*, ha cambiado la forma en la que se programan las diferentes pantallas de una aplicación. Esto sucede debido a que, sin el uso de este *toolkit*, cada pantalla se debía asignar en un *Activity* independiente. Ahora, sin embargo, se puede alojar toda la interfaz gráfica de la aplicación en un solo *Activity*, y manejar las pantallas en un nivel lógico superior.

Este *Activity* tiene la estructura que se muestra en la Figura 27. Se sobrescribe la función *onCreate()* de modo que, cuando se deba crear la actividad, el único contenido que deba mostrar sea el *NavigationScreen()*.

```
 1. class MainActivity : ComponentActivity() {
 2.     override fun onCreate(savedInstanceState: Bundle?) {
 3.         super.onCreate(savedInstanceState)
 4.         setContent {
 5.             RomeoTheme(false) {
 6.                 Surface(
 7.                     modifier = Modifier.fillMaxSize(),
 8.                     color = MaterialTheme.colorScheme.background
 9.                 ) {
10.                     NavigationScreen()
11.                 }
12.             }
13.         }
14.     }
```

```
15. }
```

Figura 27. Clase *MainActivity*, actividad principal y única de la aplicación

La función *NavigationScreen()* es una función de *Compose* que tiene como elemento principal un *NavHost*. Esta otra función será la que, gracias a un *NavHostController*, maneje la navegación entre pantallas dentro de la aplicación, así como las animaciones y el traspase de datos entre ellas.

Para conseguirlo, primero se debe obtener este *NavHostController* con la función *rememberNavController()*. Posteriormente, se inicia el *NavHost* y se le indica una pantalla inicial que, en este caso, será la pantalla de grupos. Dentro del *NavHost* se declaran todas las pantallas, cada una como una función *composable*, a la que se le deberá indicar tanto el nombre de ruta para identificar a la pantalla como las diferentes animaciones para entrar y salir de ella. En la Figura 28 aparece la estructura básica de la función comentada en este punto, con la pantalla de ajustes instanciada.

```
 1. @Composable
 2. private fun NavigationScreen() {
 3.     val navController = rememberNavController()
 4.     Scaffold(
 5.         content = { innerPadding ->
 6.             NavHost(navController = navController,
                        startDestination = NAV_DESTINY_GROUPS) {
 7.                 composable(
 8.                     route = NAV_DESTINY_SETTINGS,
 9.                     enterTransition = {
10.                         // …
11.                     },
12.                     popExitTransition = {
13.                         // …
14.                     },
15.                     exitTransition = {
16.                         // …
17.                     },
18.                     popEnterTransition = {
19.                         // …
20.                     },
21.                 ) {
22.                     SettingsScreen(innerPadding = innerPadding,
                                       navController = navController)
23.                 }
24.                 // …
25.             }
26.         }
27.     )
28. }
```

Figura 28. Estructura de la función *NavigationScreen()*

3.4.2 Navegación entre pantallas

Una vez la pantalla de navegación está correctamente definida, resta ver cómo se realiza la transición de una pantalla a otra. Para poder hacer la transición, es necesario que todas las pantallas cuenten con el *NavHostController* que se ha creado previamente en la *NavigationScreen*.

Por ejemplo, para acceder a la pantalla de ajustes desde la pantalla de grupos al presionar el botón correspondiente, la función que reciba el parámetro *onClick* del *IconButton* debe ser `navController.navigate(NAV_DESTINY_SETTINGS)`, siendo `NAV_DESTINY_SETTINGS` un alias para poder identificar a la pantalla.

Para, posteriormente, regresar a la pantalla anterior con la flecha de retroceso de la pantalla de ajustes, se deberá ejecutar la función `navController.popBackStack()`. Se debe imaginar la navegación como un sistema de capas (llamado *stack*), en el que cada vez que se accede a una pantalla nueva, esta se añade en lo alto del *stack*, de modo que siempre se visualice la pantalla

situada en esta posición. Con la función `popBackStack()` se retira la pantalla superior del *stack*, haciendo que, de esa forma, se muestre la pantalla que había previamente.

3.4.3 Animaciones

Las animaciones en las transiciones entre pantallas consiguen una experiencia de usuario más fluida y parecida a los gestos que el usuario podría realizar de forma natural al interactuar con la aplicación. Hay numerosas animaciones ya integradas en *Compose*; además, es posible crear animaciones personalizadas a partir de las ya existentes.

Si a los campos *enterTransition*, *exitTransition*, *popEnterTransition* y *popExitTransition* (Figura 28) no se les proporciona un valor, se ejecuta la transición por defecto de *Compose*, que se trata de un *fade* o fundido entre las dos pantallas. En lugar de esto, la animación que se desea incorporar en la aplicación es un desplazamiento horizontal, de modo que, al entrar en una pantalla, la nueva pantalla aparezca desde la derecha desplazándose hacia el centro. En cuanto a la animación de retroceder a una pantalla anterior, deberá ser a la inversa: la pantalla se desplazará hacia la derecha, y la pantalla a la que se vuelve aparece por detrás con un fundido.

Para comprender qué función de animación se ejecutará cada vez, se requiere un ejemplo. Suponiendo que se pasa de una pantalla A, a una pantalla B: la pantalla A ejecutará *exitTransition*, mientras que la pantalla B ejecutará *enterTransition*. Sin embargo, si en la pantalla B se presiona el botón *back* para volver a la pantalla A, entonces la pantalla B ejecutará *popExitTransition*, mientras que la pantalla A ejecutará *popEnterTransition*.

Es necesario observar el flujo de navegación entre pantallas de la Figura 8 y escoger las animaciones respectivamente. Por ejemplo, las animaciones para la pantalla de ajustes serán las que se muestran en la Figura 29. El argumento *exitTransition* es nulo ya que para pasar a la pantalla de Bluetooth, la pantalla de ajustes se mantendrá como está; lo mismo sucede con el argumento *popEnterTransition*, pues al volver de dicha pantalla de Bluetooth, la pantalla de ajustes no realizará ninguna animación. No obstante, al entrar en la pantalla, se ha asignado la animación de desplazamiento hacia la izquierda (l. 3), mientras que para salir de ella se empleará el desplazamiento a la derecha (l. 9).

```
 1. enterTransition = {
 2.     slideIntoContainer(
 3.         AnimatedContentTransitionScope.SlideDirection.Left,
 4.         animationSpec = tween(200)
 5.     )
 6. },
 7. popExitTransition = {
 8.     slideOutOfContainer(
 9.         AnimatedContentTransitionScope.SlideDirection.Right,
10.         animationSpec = tween(200)
11.     )
12. },
13. exitTransition = {
14.     null
15. },
16. popEnterTransition = {
17.     null
18. },
```

Figura 29. Transiciones empleadas en la pantalla de ajustes

3.4.4 Intercambio de datos entre pantallas

En una aplicación Android, repetidamente surge la necesidad de pasar de una pantalla a otra guardando información relativa a la pantalla anterior. En este caso, es necesario durante la transición desde la pantalla de grupos hasta la pantalla de información de un grupo, pues se debe recordar qué grupo se ha pulsado para mostrar la información relativa a dicho grupo en la siguiente pantalla. Por lo tanto, la información que se traspasará será el *alias* del grupo seleccionado.

Para ello, en primer lugar, se ha añadido un nuevo campo *String* al constructor de la clase *GroupInfoScreenViewModel*, donde se deberá especificar el *groupAlias*. Además, como esta clase conforma un módulo de *Koin*, se ha añadido también, como se ve en la Figura 30.

```
1. val groupInfoScreenViewModel = module {
2.     viewModel {parameters ->
3.         GroupInfoScreenViewModel(get(), parameters.get())
4.     }
5. }
```

Figura 30. Declaración del módulo de *Koin* con el parámetro *groupAlias*

Posteriormente, en la pantalla de navegación, se debe especificar la estructura de la ruta para incluir los parámetros necesarios. En este caso, se ha estructurado así: `"groupInfo/{groupAlias}"`. Además, en el *composable*, se deben declarar los tipos de los argumentos, con:

```
arguments = listOf(navArgument("groupAlias") { type = NavType.StringType })
```

Finalmente, se instancia la función de *Compose* con el argumento correspondiente:

```
navBackStackEntry.arguments?.getString("groupAlias")
```

De este modo, tanto la función de *Compose* con la interfaz gráfica como el *ViewModel* asociado conocen el grupo al que se ha accedido, de modo que se puede representar correctamente.

3.5 Guardado y persistencia de los datos

La gran mayoría de los datos empleados por la aplicación y visualizados en la interfaz gráfica se obtienen desde el servidor con cada nueva conexión. Sin embargo, hay ciertos datos que son independientes para cada usuario, por lo que se deben almacenar de forma local en los dispositivos móviles. Son, en este caso, el nombre de usuario y contraseña, la dirección del servidor de voz y el último dispositivo Bluetooth *Low Energy* al que se habían conectado.

Para almacenar todos estos datos, previamente se serializan a mensajes *Protocol Buffers*. La estructura de los mensajes es la que se puede ver en la Figura 31.

```
1. syntax = "proto3";
2.
3. option java_package = "xyz.romeo.store";
4. option java_multiple_files = true;
5.
6.
7. message RomeoConfig{
8.   string server = 1;
9.   string userAlias = 2;
10.   string secret = 3;
11. }
12.
13. message BleConfig{
14.   string name = 1;
15.   string address = 2;
16. }
```

Figura 31. Mensajes *protobuf* para almacenar datos localmente

Se obtienen, por lo tanto, dos mensajes *protobuf* distintos que deben persistir en el almacenamiento del dispositivo y cada uno se leerá cuando sea necesario. Para conseguir esto, se han creado dos propiedades de extensión del objeto *Context*, que son las que se muestran en la Figura 32.

```
1. val Context.configDataStore: DataStore<RomeoConfig> by dataStore(
2.     fileName = "romeoConfig.pb",
3.     serializer = ConfigStoreSerializer
4. )
5.
6. val Context.bleConfigDataStore: DataStore<BleConfig> by dataStore(
7.     fileName = "bleConfig.pb",
8.     serializer = BleConfigSerializer
9. )
```

Figura 32. Propiedades de extensión de *Context* para almacenar y leer contenido del almacenamiento interno

En ellas, se especifica el archivo *Protocol Buffers* en el que se guardarán los mensajes, junto con el serializador de cada uno de ellos. Estos serializadores (objetos que implementan la interfaz *Serializer<T>*) implementan las funciones *readFrom()* y *writeTo()*; funciones que empleará el objeto para almacenar y leer los datos. En la Figura 33 se observa el objeto *ConfigStoreSerializer*, serializador para los datos del servidor, nombre de usuario y contraseña. La forma es análoga para la clase *BleConfig*.

```
 1. object ConfigStoreSerializer : Serializer<RomeoConfig> {
 2.
 3.     override val defaultValue: RomeoConfig = RomeoConfig.getDefaultInstance()
 4.
 5.     override suspend fun readFrom(input: InputStream): RomeoConfig {
 6.         try {
 7.             return RomeoConfig.parseFrom(input)
 8.         } catch (exception: InvalidProtocolBufferException) {
 9.             throw CorruptionException("Cannot read proto.", exception)
10.         }
11.     }
12.
13.     override suspend fun writeTo(t: RomeoConfig,
                                     output: OutputStream) = t.writeTo(output)
14. }
```

Figura 33. Serializador *ConfigStoreSerializer*, para la clase *RomeoConfig*

Finalmente, para poder emplear esto en el código, se recurre a las funciones *updateData()*, para guardar nuevos datos, y *data.collect()*, de modo que el cambio en los datos almacenados pueda ser detectado, como sucede con un *StateFlow* (punto 3.1.2), y aprovechar su nuevo valor como se requiera.

3.6 Cliente WebSocket de *Ktor*

Para manejar la conexión *WebSocket* se ha creado la clase '*WebSocketKtorClient*'. El constructor de esta clase requiere un *String* con la dirección del servidor, así como funciones para actualizar el estado de la conexión y retransmitir los mensajes dentro del *Model*. Un cliente *WebSocket* no es más que un cliente HTTP al que se le instala el *plugin* de *WebSockets* para *Ktor*. Además, se definen tanto un *scope* (punto 3.1.1) como una *DefaultClientWebSocketSession* para mantener identificada la sesión, y poder enviar mensajes en ella.

Con la función *start*, se inicializa el *scope* con el *Dispatcher* 'IO', pues se van a manejar datos, y se lanza una corutina. Esta corutina ejecuta un bucle, que continuará ejecutándose hasta que se cancele el *scope*. En cada iteración del bucle, se intentará una nueva conexión *WebSocket* con la ruta especificada para ello en el servidor ("{dirección_servidor}/romeo"), y con un nuevo bucle *for* se recorrerán todas las tramas que se van recibiendo en el canal *incoming*.

Este canal *incoming* es un *ReceiveChannel* que almacenará a modo de cola los mensajes recibidos desde el servidor. Las tramas recibidas por el *WebSocket* pueden ser tanto de texto como binarias; sin embargo, en el caso que se presenta, y debido al uso de *Protocol Buffers* para transmitir los paquetes de datos, se filtrarán para descartar todas las tramas que no sean binarias. Las tramas

binarias resultantes se deserializarán para obtener el mensaje *protobuf* esperado, y se emitirán en un *Channel* presente en el *Model* para su posterior tratamiento.

En caso de producirse alguna excepción, se cancelará la *DefaultClientWebSocketSession* y, al permanecer en el bucle activo, se volverá a crear. De este modo, si se pierde la conexión, el terminal móvil hará lo posible por reconectarse automáticamente sin intervención del usuario.

El código de la función *WebSocketKtorClient.start()* es el que se muestra en la Figura 20. Además, la Figura 34 aporta un diagrama de flujo para comprender el funcionamiento de la misma.

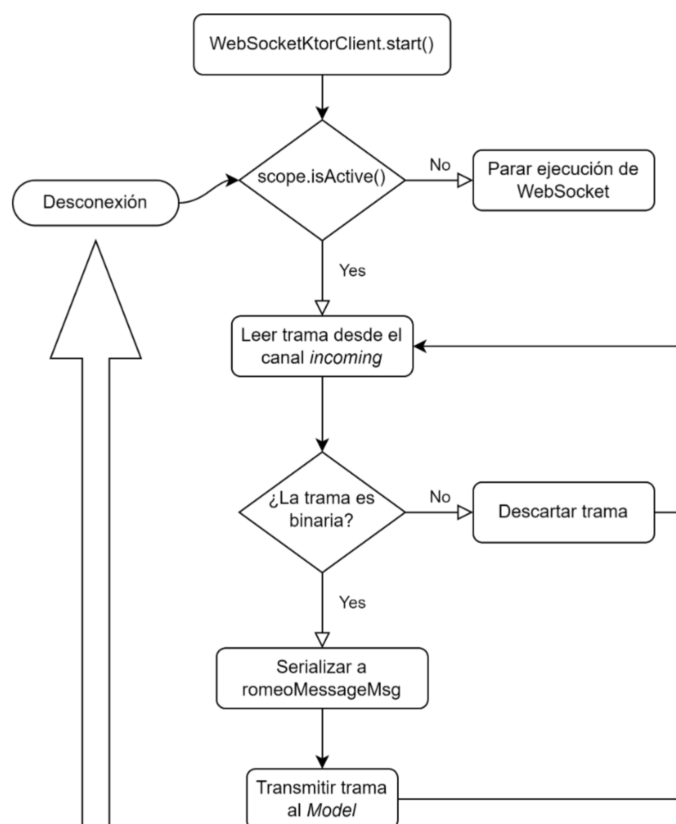

Figura 34.Diagrama de flujo de la clase *WebSocketKtorClient*

3.7 *Model* y funcionamiento general de la comunicación

Tal y como se ha introducido en el punto 3.2, la arquitectura *Model-View-Intent* requiere de la existencia de una clase *Model* que actúe como contenedor de todos los datos y la lógica necesarios para hacer funcionar la aplicación. En los siguientes puntos, se estudiará qué es lo que sucede dentro de esta clase desde que un usuario selecciona un grupo y presiona el botón de hablar, hasta que el resto de los usuarios del grupo reproducen el mensaje de audio.

3.7.1 *Definición de variables y gestión de datos del* Model

El *Model* contiene los datos fundamentales que necesita la aplicación para su correcto funcionamiento. Como se ha comentado ya, estos datos están guardados, en su mayoría, en forma de *MutableStateFlows*. A grandes rasgos, los flujos que se han incluido en el *Model* son:

- **Estado de conexión y estado de autenticación del usuario:** *serverConnectionStatus* y *clientAuthStatus*. Estos flujos informan al usuario del estado de la conexión con el

servidor, que puede pasar por los estados *CONNECTED, CONNECTING* y *DISCONNECTED*, además del estado de la autenticación, que puede ser *AUTHENTICATED*, en caso de haber iniciado la sesión con un usuario y contraseña correctos y almacenados en el servidor, o *UNAUTHENTICATED*, si el usuario o la contraseña no existen o no coinciden con los almacenados.

- **Lista de grupos y lista de participantes:** *groupsList* y *participantsList*. Los flujos que se actualizan en el momento en que se recibe un *GroupsResponseMsg* o un *ParticipantsResponseMsg* (mirar punto 2.4.2.2).
- **Configuración de la aplicación:** *server*, *userAlias* y *secret*. Se trata de tres flujos de *String* que almacenan la dirección del servidor de voz, el nombre de usuario y la contraseña introducidos.
- **Grupos de habla y escucha:** *talkingGroup*, *listeningGroups* y *currentlyListeningGroup*. Existe un flujo que almacena el grupo seleccionado para hablar, así como otro flujo con una lista de los grupos marcados como escucha. Adicionalmente, existe un flujo para conocer el grupo del que se está reproduciendo audio en cada momento, pues, aunque se pueden seleccionar múltiples grupos para escuchar, solamente se reproducirá un mensaje de voz simultáneo, para así evitar colisiones.
- **Grupos previamente marcados como habla y escucha:** *rememberTalkingGroup* y *rememberListeningGroups*. En el momento en que el usuario accede a la pantalla de un grupo en concreto, solamente podrá hablar y escuchar en ese grupo. Para que la aplicación recuerde los grupos que el usuario había marcado como habla y escucha previamente, se emplean otros dos flujos.
- **Gestión del habla:** *talkButtonPressed* y *talkingAllowed*. La aplicación cuenta con un flujo *Boolean* con el estado del botón *push-to-talk* (presionado o soltado), además de otro flujo, también *Boolean*, que indica si el usuario tiene el permiso del servidor para hablar, o no.
- **Gestión de escucha:** *playingAudio*. Con este flujo, diferentes componentes de la aplicación pueden conocer si se está reproduciendo audio en un momento determinado.
- **Canal de difusión de mensajes en el *Model*:** *broadcastReceivedMessage*. Como se ha comentado ya en el punto 3.6, existe un *Channel* en el que se emiten los mensajes que se reciben por el *WebSocket*.
- **Muestras de RTT y de *Jitter*, y muestras de FFT:** *rttFlow*, *jitterFlow* y *fftDataFlow*. Empleando dos flujos, uno de *Integer* y otro de *Double*, se emiten las muestras de RTT y *Jitter* calculadas, aspecto que será comentado posteriormente. De la misma forma, con un flujo de *FloatArray*, se emiten las muestras de FFT calculadas (mirar anexo 2).
- **Dispositivo *Bluetooth LE* conectado:** *currentBLEDevice*. Se emplea también otro flujo en el que se emitirá el dispositivo *Bluetooth LE* que se conecte al terminal móvil del usuario.

3.7.2 Esquema de funcionamiento del habla

Se va a definir exactamente cuál es el funcionamiento de la aplicación cuando el usuario quiere enviar un mensaje de voz. En la Figura 35 se puede observar un diagrama de funcionamiento más detallado.

1. El usuario selecciona un grupo para hablar.
2. El usuario presiona el botón *push-to-talk*.
3. El servidor decide si el usuario puede hablar o no. Si ya hay alguien hablando en el grupo, el usuario verá que no puede hablar mediante la ausencia de cambio de color en el botón pulsado.
4. El usuario tiene permiso para hablar. Se le reserva el canal del grupo.
5. El usuario habla y se envían las muestras de audio.
6. El usuario suelta el botón *push-to-talk* y el canal del grupo se libera.

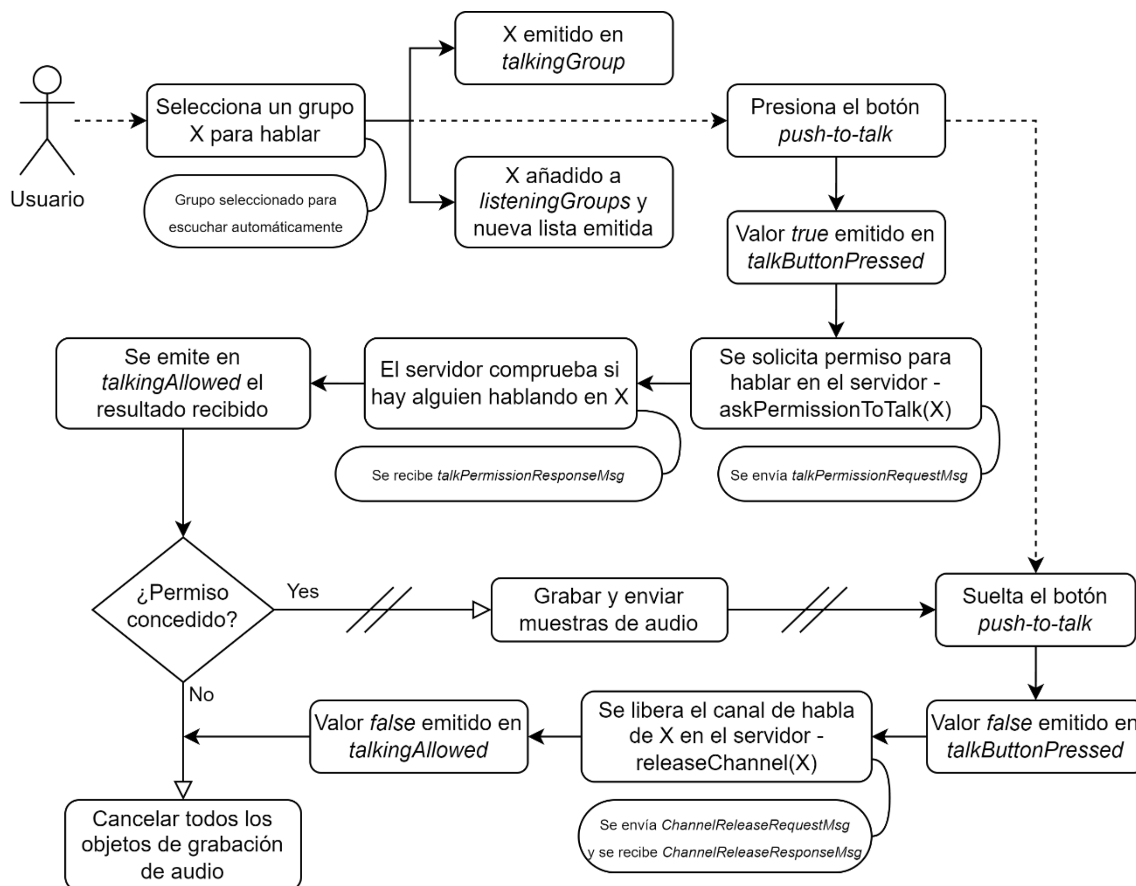

Figura 35. Diagrama de flujo del funcionamiento del habla en la aplicación

3.7.3 Diagrama de funcionamiento de la escucha

De la misma forma que en el punto anterior, se puede definir el flujo que sigue la aplicación para reproducir un mensaje de audio una vez se empieza a recibir. Se puede observar en la Figura 36.

1. El usuario selecciona uno o varios grupos para escuchar.
2. El usuario recibe un paquete inicial con las características del audio que va a reproducir, así como el *jitter* del emisor.
3. El usuario adapta el búfer del reproductor de audio en función de su *jitter* y el del emisor.
4. El usuario reproduce las muestras de audio hasta recibir un paquete final, que cierra el reproductor.

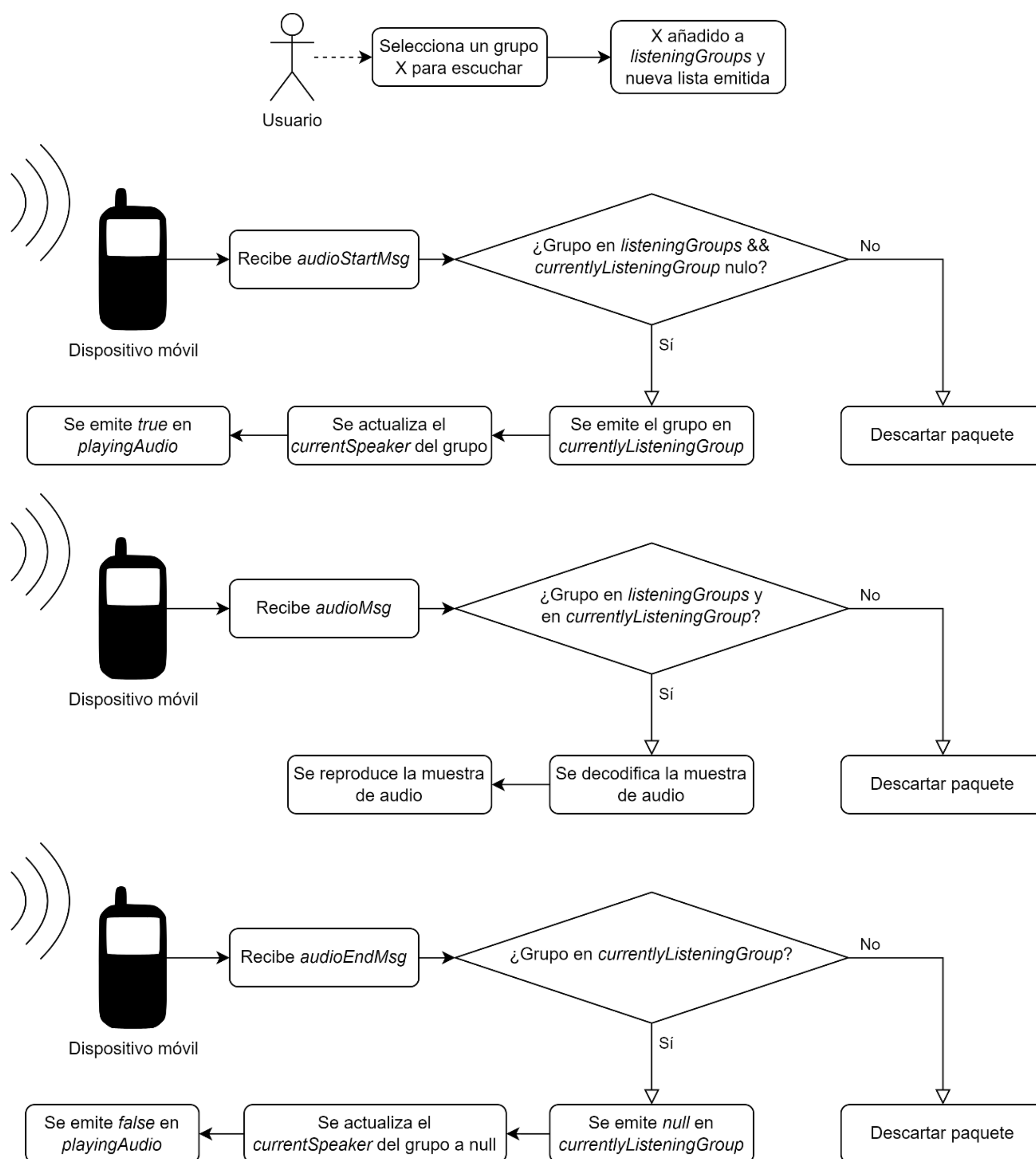

Figura 36. Diagrama de flujo del funcionamiento de la escucha en la aplicación

3.8 Grabado y reproducción de audio

En el proceso de grabado y reproducción de audio se han empleado las siguientes clases:

- ***AudioRecord***: Es una clase de la librería *android.media*, dedicada a la grabación de audio desde el dispositivo *hardware* de entrada de audio de la plataforma. Para ello, se deben leer los datos del objeto, con el método *read()*.
- ***AudioTrack***: Se trata de otra clase de la librería *android.media*, que gestiona la reproducción de flujos de audio en Android. Para conseguirlo, se escriben las muestras de audio en el objeto, con el método *write()*.
- ***OpusEncoder* y *OpusDecoder***: Son dos clases de la librería de *Opus* programada en C para codificar y decodificar las muestras de audio.

Los objetos *AudioRecord* y *AudioTrack* han sido manejados mediante dos clases: *AudioRecorder* y *AudioPlayer*. Cabe destacar que, tanto la clase *AudioRecorder* como los objetos *OpusEncoder* y *OpusDecoder* no han tenido que ser desarrollados desde cero para este trabajo, sino que han sido proporcionados previamente por el tutor.

3.8.1 *AudioRecorder*

Esta es la clase que emplea un objeto *AudioRecord* para capturar muestras de audio desde el micrófono del dispositivo móvil, o desde alguno de los dispositivos de audio conectados a él, como auriculares con micrófono integrado. Se crea un objeto de esta clase en el *Model*, cuando se obtiene un valor *true* en el flujo *talkingAllowed*.

Previamente, del objeto *audioManager* también creado en el *Model*, se extraen los dispositivos disponibles, tanto Bluetooth como alámbricos. En caso de ser un dispositivo Bluetooth, se activa el *Headset Profile* (HSP) del dispositivo, con el que se puede grabar audio para llamadas, mediante la función *startBluetoothSco()*.

Para extraer el audio, se emplea un *SharedFlow* en el objeto *AudioRecorder*, en el que se emiten las muestras de audio capturadas. Posteriormente, se codifican, se encapsulan en el mensaje *AudioMsg* de *Protocol Buffers* y se envían a través de la conexión *WebSocket*.

El empleo de la clase *AudioRecorder* se puede observar en la Figura 37.

```
1. if (_talkingAllowed.value) {
2.     // …
3.     val ad = audioManager.getDevices(AudioManager.GET_DEVICES_INPUTS)
4.     audioDevices = ad
5.
6.     val bluetoothDevice = audioDevices.find { device ->
7.         device.type == AudioDeviceInfo.TYPE_BLUETOOTH_SCO ||
8.         device.type == AudioDeviceInfo.TYPE_BLUETOOTH_A2DP
9.     }
10.    val wiredDevice = audioDevices.find { device ->
11.        device.type == AudioDeviceInfo.TYPE_WIRED_HEADPHONES ||
12.        device.type == AudioDeviceInfo.TYPE_WIRED_HEADSET ||
13.        device.type == AudioDeviceInfo.TYPE_LINE_ANALOG
14.    }
15.
16.    d { "AudioDevices: $audioDevices, BluetoothDevices: $bluetoothDevice" }
17.
18.    if (bluetoothDevice != null) {
19.        audioManager.setMode(MODE_IN_CALL)
20.        audioManager.setBluetoothScoOn(true)
21.        audioManager.startBluetoothSco()
22.    }
23.
24.    val ar = AudioRecorder(preferredDevice = bluetoothDevice ?: wiredDevice)
29.
30.    ar.init()
31.    ar.start()
32.
33.    // … (envío del mensaje AudioStartMsg)
34.    webSocketKtorClient?.send(audioStart)
35.
36.    recordingJob = scope.launch {
37.        ar.audioFlow.collect { audioSample ->
38.            // … (envío del mensaje AudioMsg)
39.            webSocketKtorClient?.send(audioMessage)
40.        }
41.    }
42. }
```

Figura 37. Utilización de la clase *AudioRecorder*

En su interior, la clase *AudioRecorder* funciona de la siguiente forma. En primer lugar, para crear un objeto de esta clase, se le puede indicar la frecuencia de muestreo del audio y el tamaño del paquete de audio deseado. En este caso, ambos aspectos se han comentado en el apartado 2.5, concluyendo que se les debe dar por defecto una frecuencia de muestreo de 48 kHz, y un tamaño de paquete de 1920 muestras, para conseguir una tasa de 25 paquetes por segundo. Adicionalmente, se puede indicar el dispositivo preferido de captura de audio, que se determina de la forma que se ha mostrado en la Figura 37, de modo que, si es nulo, se empleará el dispositivo de captura de audio predeterminado.

A continuación, se definen las variables que se van a emplear. Por ejemplo, un número de secuencia, el *SharedFlow* en el que se emitirán los paquetes de audio, o el *scope* que se empleará para grabar.

Cuando se ejecuta la función *start()*, se crea el objeto *AudioRecord*, previamente habiendo calculado el tamaño mínimo del búfer para que se puedan recoger las muestras de audio antes de que sean sobrescritas por el grabador. Después, se establece el dispositivo preferido para realizar la grabación, se inicializan ciertos efectos de audio que mejoran su calidad (AGC, cancelación de eco, reducción de ruido, etc.), y finalmente se inicia un bucle *while(isActive)*, en el que se grabarán y emitirán los paquetes de audio de 1920 muestras.

La grabación se realiza siguiendo este procedimiento. En primer lugar, se crea un *ShortArray* de tamaño igual al número de muestras deseadas por paquete. Seguidamente, con un contador, se recorre el búfer del objeto *AudioRecord* hasta que se extraen todas las muestras deseadas. El resultado se almacena en el *ShortArray* creado previamente. En la Figura 38 se puede ver cómo se ha implementado la función para grabar un paquete de audio, especificando el AudioRecord empleado en grabación y el número de muestras deseadas en el paquete.

```
1.     private fun recordAudioSample(audioRecord: AudioRecord, sampleSize: Int)
                   : ShortArray {
2.         val buff = ShortArray(sampleSize)
3.         var count = 0
4.         while (count < sampleSize) {
5.             val read = audioRecord.read(buff, count, sampleSize - count)
6.             count += read
7.         }
8.         return buff
9.     }
```

Figura 38. Función para grabar un paquete de audio

3.8.2 AudioPlayer

La clase *AudioPlayer* es la encargada de alojar y gestionar el *AudioTrack* en el que se van a reproducir las muestras de audio recibidas a través del *WebSocket*. Como ya se ha comentado anteriormente, previo al envío de dichas muestras, se envía un paquete *AudioStartMsg*, en el que se incluye información sobre el audio a reproducir, así como el *jitter* del hablante. Cuando dicho mensaje es recibido, se emite el valor *true* en el flujo *playingAudio*. Es en ese momento cuando se crea el objeto *AudioPlayer*. De ese modo, cuando lleguen las muestras de audio, el objeto ya existirá en el *Model* y podrá reproducir las muestras recibidas.

En lo que se refiere a la reproducción del audio, se deben distinguir dos búfers: uno propio del *AudioTrack*, y otro externo, más manejable. El búfer del *AudioTrack* asegura que las muestras puedan reproducirse sin pequeños cortes entre ellas, mientras que el búfer externo permite controlar la cantidad de muestras que se almacenan antes de empezar la reproducción. Este segundo búfer es una lista, *MutableList*, en principio, de capacidad infinita, a la que se le indica la cantidad de paquetes de audio que debe haber en la lista para poder empezar a reproducir.

El búfer funciona de la forma indicada en la Figura 39. Se define la variable `buffering` como la que indica si se puede extraer un paquete de la lista o no. Cuando se añade un paquete de audio

con *addSample()*, si previamente `buffering` tenía el valor *true* y el tamaño de la lista supera el tamaño de búfer especificado en el constructor, `buffering` pasa a *false*. En cambio, si al extraer una muestra la lista se vacía por completo, `buffering` debe regresar a *true* para alargar el tiempo en el que la reproducción está parada y acumulando nuevos paquetes. De cualquier manera, al solicitar un paquete de audio con *getFirstSample()*, solamente se proporcionará si `buffering` está a *false* y hay paquetes en la lista.

```
1. class AudioBuffer(private val bufferSize: Int) {
2.     private var buffering = true
3.     private val audioSamples: MutableList<ShortArray> = mutableListOf()
4.
5.     @Synchronized
6.     fun addSample(sample: AudioSample) {
7.         audioSamples.add(sample.audioSample)
8.         if (buffering && audioSamples.size >= bufferSize) buffering = false
9.     }
10.
11.    @Synchronized
12.    fun getFistSample(): ShortArray? {
13.        if (!buffering && audioSamples.size > 0) {
14.            val firstSample = audioSamples.removeAt(0)
15.            if (audioSamples.size == 0) buffering = true
16.            return firstSample
17.        }
18.        return null
19.    }
20. }
```

Figura 39. Funcionamiento del búfer de audio

En su interior, la clase *AudioPlayer* funciona de manera análoga a la clase *AudioRecorder*. En primer lugar, se declaran las variables, entre las que se encuentran el *scope* que se va a emplear, el búfer de audio comentado anteriormente y el propio *AudioTrack*. En el constructor *init*, se lanza una corutina en el *scope* especificado, y se construye el *AudioTrack*, indicando parámetros como la frecuencia de muestreo, la codificación PCM a 16 bits, el número de canales, y el búfer interno del *AudioTrack*, cuyo valor mínimo aceptable se obtiene automáticamente con la función *getMinBufferSize()*. Es importante establecer el *transferMode* a *MODE_STREAM*, pues no se cuenta con la totalidad del audio en el momento de iniciar la reproducción.

Con *audioTrack.play()* se inicia la reproducción. Mientras el *scope* está activo, se ejecuta un bucle en el que se van extrayendo todos los paquetes de audio recibidos. Para cada paquete, un contador llamado `currentPosition` indica, a modo de puntero de escritura, la cantidad de muestras de audio que se han escrito en el *AudioTrack*, y es necesario para la función *write()*, con la que se escribe el audio en el dispositivo de salida.

Por tanto, en cada paquete de audio se ejecuta un nuevo bucle que recorre todas sus muestras, y las escribe con la función *write()* con los siguientes parámetros:

- Paquete de audio.
- Puntero de escritura.
- Número de muestras a escribir (la resta entre el tamaño del paquete y `currentPosition`)
- Tipo de escritura. Puede ser *WRITE_BLOCKING* o *WRITE_NON_BLOCKING*. En este caso, se ha escogido la segunda opción, pues existe el búfer externo que ya limitará de por sí la escritura.

La función *write()* devuelve la cantidad de muestras que se han escrito. Si devuelve un valor negativo, indica un código de error y se pasa al siguiente paquete. Si la escritura ha sido satisfactoria, el número de muestras escritas se añade a `currentPosition`, y continúa el recorrido del paquete de audio, hasta que `currentPosition` sea igual al tamaño del paquete.

3.9 Cálculo de parámetros de la conexión

La red de Internet es *best-effort*. Esto quiere decir que no hay ninguna garantía de que los paquetes que se envían a través de Internet lleguen a su destinatario, ni tampoco que lo hagan con un cierto retardo máximo estipulado. En función de la carga la red en cada momento, los paquetes se entregarán con un retardo mayor o menor, que además no será constante.

El retardo de un paquete es el tiempo que pasa desde que se envía hasta que se recibe. En el caso de la transmisión de audio, un retardo mayor supondrá una mayor diferencia temporal entre las palabras pronunciadas por el emisor y las escuchadas por el receptor. Para el caso de uso que se espera de la aplicación, en el que los interlocutores no están manteniendo una conversación bidireccional simultánea, el retardo no es tan perjudicial para la calidad de experiencia.

Sin embargo, es la variación del retardo la que sí puede afectar a la QoE significativamente. Las muestras de audio deben reproducirse de forma continuada en el receptor, de modo que, antes de finalizar la reproducción de una determinada muestra de audio, el reproductor ya debe contar con la siguiente. Si esto no ocurre, se producirá un parón en la reproducción que afectará a la inteligibilidad del mensaje de audio en su conjunto.

Para evitar esto, es necesario introducir un búfer que almacene las muestras de audio hasta que se tengan suficientes para evitar que la reproducción se pare en algún momento. Además, se debe dimensionar la cantidad de muestras en búfer hasta que empiece la reproducción, y hacerlo acorde al estado de la red en el momento en que se transmite el mensaje de audio. Por lo tanto, es necesario estimar el estado de la red.

Mediante el envío de mensajes periódicos como los que se ven en la Figura 40, se puede estimar el RTT (*Round-Trip Time*), que no es más que el tiempo de ida y vuelta de un paquete al servidor.

```
1. message RttTest{
2.   int64 sendTime = 1;
3. }
```

Figura 40. Mensaje *RttTest* para conocer el RTT de los paquetes

Cuando el estado de la conexión con el servidor (*serverConnectionStatus*) pasa a *CONNECTED*, cada 2 segundos se registra la hora del dispositivo, medida en milisegundos, se añade al mensaje y se transmite al servidor. Una vez el servidor lo recibe, lo reenvía a su emisor automáticamente. Cuando el terminal lo vuelve a recibir, registra de nuevo la hora del dispositivo, y calcula el RTT mediante la diferencia entre el valor apuntado en el paquete y el valor registrado en el instante de la recepción. Una vez calculada la diferencia, se emite en el flujo *rttFlow*.

Posteriormente, cada vez que se colecciona un nuevo valor de RTT, este se añade a un objeto creado específicamente para este caso: *FifoArray*. Se trata de una lista de *Integer* a la que se le indica el tamaño que debe tomar. Cada vez que se ejecuta la función *add*, se desplazan todas las muestras una posición, descartando la muestra más antigua e incluyendo la muestra nueva en primera posición. De esta manera, se consigue un funcionamiento FIFO (*First in, first out*). En este caso, se ha tomado un tamaño de la *FifoArray* de 20 elementos, aunque este valor se puede modificar para probar otros resultados.

Ahora que ya están localizadas todas las muestras de *delay*, resta aclarar cómo se calcula el *jitter*. El *jitter* es la variación del retardo, por lo que se puede calcular de manera estadística como la desviación estándar del conjunto de las muestras del *delay*, como se puede ver en la ecuación (3.1).

$$J = \sqrt{\frac{1}{N}\sum_{1}^{N}(D_i - \overline{D})^2} \quad \textbf{(3.1)}$$

De este modo, cada vez que se colecta una nueva muestra de *delay* se calcula el *jitter* y el nuevo valor se emite en el flujo *jitterFlow*. Se puede observar el código necesario para almacenar las muestras de *delay* y calcular el *jitter* en la Figura 41.

```
 1. class FifoArray(val size: Int) {
 2.     private val fifo = LinkedList<Int>()
 3.
 4.     fun add(item: Int) {
 5.         if (fifo.size == size) {
 6.             fifo.removeLast()
 7.         }
 8.         fifo.addFirst(item)
 9.
10.     }
11.
12.     fun getAvg(): Double {
13.         val sum = fifo.sum()
14.         return sum.toDouble() / fifo.size
15.     }
16.
17.     fun getStd(): Double {
18.         val avg = getAvg()
19.         val sqDiffSum = fifo.sumOf { (it - avg) * (it - avg) }
20.         val variance = sqDiffSum / fifo.size
21.         return sqrt(variance)
22.     }
23. }
```

Figura 41. Clase *FifoArray* para el almacenamiento de muestras de retardo y cálculo del *jitter*

3.9.1 *Modificación del búfer del* AudioPlayer

El motivo por el que se ha calculado el *jitter* no es otro que para poder dimensionar el búfer del reproductor de audio de forma adecuada. Se puede suponer que, teniendo muestras de audio de 40 ms (mirar punto 2.5), de tamaño fijo, el búfer deberá tener las muestras suficientes como para poder suplir la cantidad de tiempo medida en el *jitter*. De este modo, por ejemplo, si el *jitter* fuese de 90 ms en un instante, sería necesario que el búfer no empezase la reproducción hasta acumular 3 muestras, que equivaldrían a 120 ms de audio. En cambio, cualquier valor de *jitter* por debajo de 40 ms, no requerirá ninguna muestra en el búfer para poder empezar a reproducir el mensaje. Sin embargo, para evitar riesgos, se ha decidido incluir una muestra de búfer más que la calculada. Esto aumenta el retardo en 40 ms fijos, pero evita las pausas en la reproducción si se produce un cambio brusco.

No obstante, no se puede tener en cuenta únicamente el *jitter* del receptor del mensaje de audio, puesto que el emisor también está sujeto a las mismas condiciones, y los paquetes de audio que transmite podrían retrasarse de la misma forma. Es por ello por lo que, en el mensaje *AudioStartMsg*, se ha incluido un campo específico en el que el hablante indica su *jitter* en el momento de empezar la grabación. De este modo, el receptor escogerá entre el valor de su *jitter* y el *jitter* del emisor, empleando el valor más alto para el búfer del reproductor.

En la Figura 42 se puede ver cómo se escoge el tamaño del búfer de forma programática cada vez que se recibe *true* en el flujo *playingAudio*.

```
1. val jitter = max(_jitterFlow.value, speakerJitter)
2. val bufferSize = ceil(jitter / (1000 / WORKING_SAMPLES_PER_SECOND)).toInt()
3. val ap = AudioPlayer(bufferSize = bufferSize, model = this@Model)
```

Figura 42. Elección del tamaño del búfer al reproducir audio

3.10 Permisos en Android

El objetivo de los permisos en Android es que el usuario pueda controlar y perfilar todos los aspectos de su terminal móvil a los que tiene acceso una determinada aplicación. Por ejemplo, las aplicaciones pueden pedir permiso para acceder al micrófono o a la cámara, a la ubicación, acceso a Internet, etc.

La documentación que proporciona Google en *Android Developers* indica la forma correcta de proceder para que la aplicación pueda solicitar permisos, y que estos puedan ser aceptados o rechazados por el usuario.

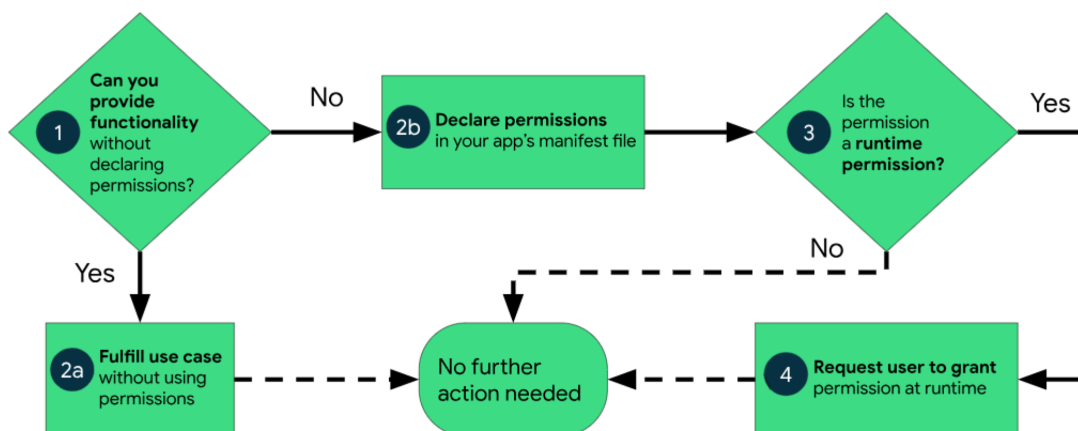

Figura 43. Diagrama de flujo para declarar permisos en Android. Fuente: Android Developers [20]

Según el diagrama de la Figura 43, en caso de realmente necesitar la declaración de un permiso, se debe hacer en el fichero *manifest* de la aplicación. En caso de que el permiso sea además un permiso de tiempo de ejecución (como el acceso al micrófono o la cámara), se deberá solicitar al usuario que acepte dicho permiso [20].

Además, a la hora de solicitar un permiso al usuario, Google también especifica una ruta a seguir con el objetivo de unificar la funcionalidad en todas las aplicaciones, también para que el usuario pueda saber por qué se le solicita dicho permiso.

En este caso, se va a seguir el flujo mostrado en el diagrama de la Figura 44.

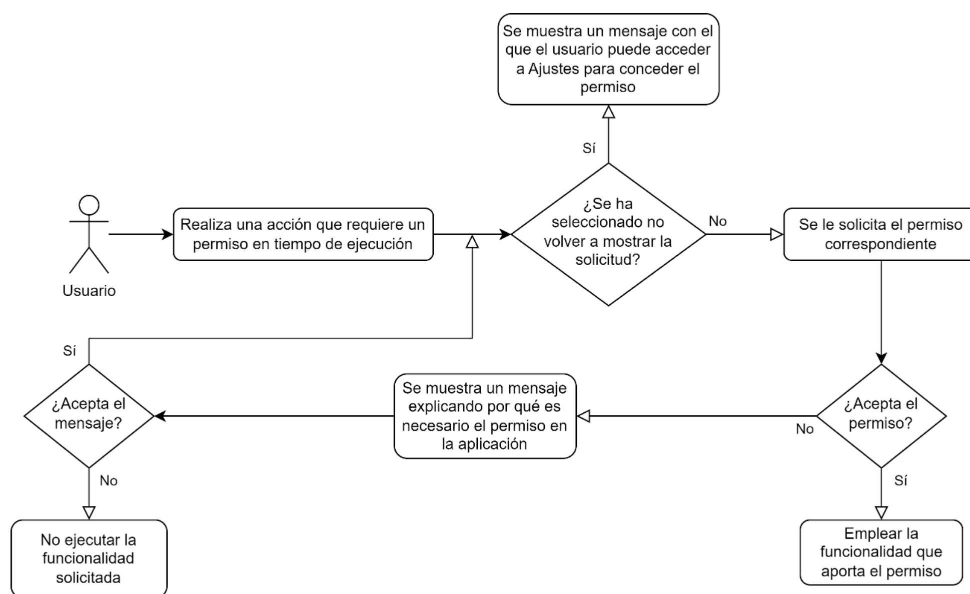

Figura 44. Diagrama de flujo para solicitar permisos en Android

A continuación, se mostrarán los cambios que se han llevado a cabo en el código para declarar los permisos necesarios y solicitarlos al usuario siguiendo el diagrama de flujo expuesto en la Figura 44. En primer lugar, se declaran los permisos en el documento *AndroidManifest.xml*, tal y como se muestra en la Figura 45.

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3.     xmlns:tools="http://schemas.android.com/tools">
4.
5.     <!--network-->
6.     <uses-permission android:name="android.permission.INTERNET" />
7.     <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
8.     <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
9.     <uses-permission android:name="android.permission.CHANGE_NETWORK_STATE" />
10.    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
11.    <uses-permission android:name="android.permission.CHANGE_WIFI_MULTICAST_STATE" />
12.
13.    <!--foreground service-->
14.    <uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
15.    <uses-permission android:name=
16.                     "android.permission.FOREGROUND_SERVICE_MICROPHONE" />
17.
18.    <!--audio-->
19.    <uses-permission android:name="android.permission.RECORD_AUDIO" />
20.    <uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS" />
21.
22.    <!--battery optimizations-->
23.    <uses-permission android:name=
                        "android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS" />
24.    <uses-permission android:name="android.permission.WAKE_LOCK" />
25.
26.    <!--bluetooth-->
27.    <uses-permission android:name="android.permission.BLUETOOTH"
28.        android:maxSdkVersion="30" />
29.    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"
30.        android:maxSdkVersion="30" />
31.    <uses-permission android:name="android.permission.BLUETOOTH_SCAN"
32.        android:usesPermissionFlags="neverForLocation"
33.        tools:targetApi="s" />
34.    <uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE" />
35.    <uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
36.
37.    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
38.    <!--…-->
39.    </application>
40. </manifest>
```

Figura 45. Permisos declarados en el *manifest*

La mayor parte de estos permisos solo requieren estar declarados en el *manifest* para que la aplicación pueda emplear su funcionalidad. Sin embargo, hay permisos que sí requieren ser solicitados al usuario en tiempo de ejecución, y son:

- **`android.permission.RECORD_AUDIO`:** Con este permiso, la aplicación puede grabar audio del micrófono. Se debe solicitar en todas las versiones de Android, y se hará cuando el usuario presione el botón *push-to-talk* sin el permiso concedido.
- **`android.permission.BLUETOOTH_SCAN`:** Este es el permiso con el que el dispositivo móvil puede buscar dispositivos Bluetooth. En este caso, se empleará para encontrar botones Bluetooth *Low Energy*.
- **`android.permission.ACCESS_FINE_LOCATION`:** Con este permiso, se accede a la ubicación exacta del dispositivo móvil. Se debe solicitar junto con el permiso *BLUETOOTH_SCAN* en versiones de Android 12 y posteriores.

Una vez declarados en el *manifest*, resta solicitarlos al usuario. Se ha creado una clase enumerada llamada *RomeoPermissions* para hacer referencia a todos los permisos que se deben solicitar, junto con los textos de los mensajes explicativos. En una nueva clase *PermissionsViewModel* se aloja una lista mutable que almacenará todos los diálogos de permisos que se deben mostrar, llamada *visiblePermissionDialogQueue*; cuando se descarte (acepte o rechace) uno de ellos en la interfaz gráfica, se eliminará de la lista. Además, en este *ViewModel*, se manejará el resultado de la solicitud de permisos de modo que, si el usuario ha denegado el permiso, se actualizará el estado de dicho permiso a *NOT_GRANTED* y se mostrará el mensaje correspondiente. Este *PermissionsViewModel* se deberá instanciar en todas las pantallas que requieran solicitar algún permiso.

Posteriormente, se pasa a solicitar por primera vez los permisos al usuario. Se va a emplear como ejemplo el permiso *RECORD_AUDIO*. Dentro de la función *Compose GroupsScreen* se incluyen los objetos que se observan en la Figura 46.

```
 1. val dialogQueue = permissionsViewModel.visiblePermissionDialogQueue
 2.
 3. val audioRecordPermissionResultLauncher = rememberLauncherForActivityResult(
 4.     contract = ActivityResultContracts.RequestPermission(),
 5.     onResult = { isGranted ->
 6.         permissionsViewModel.onPermissionResult(
 7.             permission = RomeoPermission.RECORD.permission.permissionString,
 8.             isGranted = isGranted
 9.         )
10.         if (isGranted) {
11.             val serviceIntent = Intent(activity, RomeoService::class.java)
12.             activity.startService(serviceIntent)
13.         }
14.     }
15. )
```

Figura 46. Elementos necesarios para solicitar permisos en la pantalla de grupos

En primer lugar, se encuentra el *dialogQueue* comentado anteriormente (l. 1) y, en segundo lugar, aparece un lanzador de un *Activity* que espera un resultado (l. 3-15). En este caso, se establece un contrato para solicitar un permiso en la actividad lanzada, y dicho contrato tiene como resultado un *Boolean* que indicará si se ha concedido el permiso o no.

En el momento en que se desee solicitar al usuario el permiso de grabación, que, en esta circunstancia, sucederá cuando el usuario presione el botón *push-to-talk*, se comprobará si dicho permiso no ha sido concedido todavía, y si ese es el caso, se lanzará la actividad para solicitarlo, como se ve en la Figura 47.

```
1. onPress = {
2.     if (activity.checkSelfPermission(Manifest.permission.RECORD_AUDIO)
           == PackageManager.PERMISSION_GRANTED) {
3.         // Lógica del botón push-to-talk
4.     } else {
5.         audioRecordPermissionResultLauncher.launch(Manifest.permission.RECORD_AUDIO)
6.     }
7. }
```

Figura 47. Comprobación de garantía de permisos (l. 2) y solicitud de permisos (l. 5)

El resultado de esta solicitud será recogido y tratado en la función *lambda* llamada *onResult* en el lanzador, ejecutando la función *onPermissionResult()* del *PermissionsViewModel*, en la que se manejará el posible mensaje de explicación del permiso a mostrar. Cabe destacar que dicho mensaje se debe incluir en la función de *Compose* que contiene la interfaz gráfica de la página en la que se soliciten los permisos, de modo que se puedan mostrar adecuadamente.

En caso de que el usuario deniegue definitivamente la concesión del permiso solicitado, se mostrará un cuadro de diálogo distinto en el que se instará al usuario a ir a la página de ajustes

del teléfono y conceder los permisos manualmente, para lo que se añade un botón que, si se pulsa, lanzará un *Activity* que empleará un *Intent* para redirigir al usuario automáticamente.

3.11 Búsqueda *Bluetooth* y conexión con botones Bluetooth *Low Energy*

A lo largo de este punto, se va a suponer que ya se han solicitado al usuario los permisos necesarios para poder realizar una búsqueda de dispositivos Bluetooth, tal y como se ha comentado en el punto 3.10.

Android soporta de manera nativa la conexión y el control de dispositivos Bluetooth *Low Energy*, con APIs que permiten descubrir dispositivos, realizar peticiones a servicios o transmitir información. En este caso, se deberá leer el valor de estado del botón *push-to-talk*: presionado o soltado. Todos los dispositivos Bluetooth *Low Energy* que transmiten pocos datos trabajan en el perfil GATT (*Generic Attribute Profile*); cada atributo se identifica de forma unívoca por un UUID estándar de 128 bits, y representa una característica o un servicio.

Una característica es un tipo de dato, con un solo valor y varios descriptores para que el valor pueda ser comprensible por el lector. Un servicio es un conjunto de características. Tanto el servicio como la característica de botón asociada están identificados por los UUID, de modo que han sido declarados como constantes, como se ve en la Figura 48.

```
1. object BleConstants {
2.     val BLE_PTT_SERVICE: ParcelUuid = ParcelUuid.fromString
           ("0000FFE0-0000-1000-8000-00805F9B34FB")
3.     val BLE_PTT_CHARACTERISTIC: ParcelUuid = ParcelUuid.fromString
           ("0000FFE1-0000-1000-8000-00805F9B34FB")
4. }
```

Figura 48. UUIDs del servicio y característica que identifican la funcionalidad *push-to-talk*

3.11.1 *Descubrimiento de dispositivos Bluetooth* push-to-talk

Para descubrir dispositivos Bluetooth en Android, se debe crear un escáner de Bluetooth *Low Energy* a partir de un objeto *BluetoothAdapter*, el cual se obtiene desde el servicio *BluetoothManager* del sistema. Una vez obtenido el escáner, se emplea la función *startScan()*, función a la que se le debe indicar un *ScanCallback* que definirá el procedimiento a seguir con todos los dispositivos encontrados. Opcionalmente, se le puede añadir ajustes en el escaneo (con el que se indica el modo *SCAN_MODE_LOW_LATENCY* para aumentar la velocidad) y filtros que, en este caso, se han configurado para capturar únicamente dispositivos que contengan el servicio indicado en la línea 2 de la Figura 48.

El *ScanCallback* devuelve un objeto *ScanResult* del que se saca el dispositivo encontrado como un *BluetoothDevice*. Si la dirección MAC del dispositivo encontrado no coincide con la dirección de ninguno de los dispositivos encontrados hasta el momento, se añade al *container* de estado del *BluetoothScreenViewModel* para su visualización en la interfaz gráfica.

En la Figura 49 se puede observar cómo se realiza la búsqueda.

```
 1. private fun startBleScan() {
 2.     if (appContext.hasRequiredBluetoothPermissions()) {
 3.         d { "Bluetooth permission requirements met" }
 4.
 5.         if (bluetoothAdapter == null || !bluetoothAdapter.isEnabled) {
 6.             d { "Bluetooth is not enabled" }
 7.             Toast.makeText(appContext,
                               "Bluetooth is not enabled", Toast.LENGTH_SHORT).show()
 8.         } else {
 9.             val scanFilter = ScanFilter.Builder()
10.                 .setServiceUuid(ParcelUuid(BLE_PTT_SERVICE.uuid))
```

```
11.                 .build()
12.
13.             val filters = listOf(scanFilter)
14.             val scanSettings = ScanSettings.Builder()
15.                 .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
16.                 .build()
17.
18.             scanner?.startScan(filters, scanSettings, leScanCallback)
19.         }
20.     }
21. }
```

Figura 49. Función para realizar la búsqueda de dispositivos Bluetooth LE *push-to-talk*

3.11.2 Conexión con el dispositivo encontrado

Una vez se haya encontrado el dispositivo deseado y se haya mostrado en la pantalla de Bluetooth, el usuario podrá presionarlo y conectarse al dispositivo. Cuando esto sucede, se actualiza la configuración *bleConfigDataStore* (punto 3.5), se desconecta de cualquier dispositivo que pudiese tener conectado previamente, se obtiene de nuevo el objeto *BluetoothDevice* a partir de su dirección MAC, se emite en el flujo *currentBLEDevice* y se trata de conectar a él.

Para conectarse, se emplea la función *connectGatt()*, que recibe como argumentos:

- Contexto de la aplicación.
- *Boolean* para determinar si el dispositivo se debe reconectar automáticamente en el futuro, variable que se establece a *true*.
- Un objeto *BluetoothGattCallback*, de forma similar al descubrimiento de los dispositivos.

En la implementación del *BluetoothGattCallback* se decide qué se debe hacer en caso de que cambie el estado de la conexión con el dispositivo. Si el estado cambia a *STATE_CONNECTED*, se actualiza el estado del dispositivo en el flujo *currentBLEDevice* a *CONNECTED*, y se descubren los servicios del que dispone con la función *discoverServices()*. En cambio, en cualquier otro estado, se actualizará el estado del dispositivo en el flujo a *DISCONNECTED*.

Además, en la misma implementación, se puede gestionar lo que pasa cuando se descubren los servicios. En este caso, se busca el servicio identificado con el UUID *BLE_PTT_SERVICE* (Figura 48) y, si existe, se buscará la característica que contiene dicho servicio, identificada con el otro UUID, *BLE_PTT_CHARACTERISTIC*. Si, de nuevo, esa característica existe dentro del servicio, el dispositivo se suscribirá a la recepción de valores emitidos por ella, empleando la función *setCharacteristicNotification()*.

Ahora que el terminal se ha suscrito a la característica que indica el estado del botón *push-to-talk*, resta definir el comportamiento que debe tener un cambio en el valor suscrito. Esto se puede conseguir en el mismo *callback*, sobrescribiendo la función *onCharacteristicChanged()*. Como debe actuar de botón *push-to-talk*, el valor obtenido (*true* o *false*) se emitirá en el flujo *talkButtonPressed*, el mismo flujo que maneja el botón de la interfaz gráfica, pues se desea la misma funcionalidad.

Capítulo 4. Desarrollo del servidor

4.1 Servidor *WebSocket* de *Ktor*

El programa que ejecuta el servicio de voz necesario para el funcionamiento de la aplicación se ha desarrollado en su totalidad en el IDE *IntelliJ IDEA*, que soporta Kotlin y, en su versión *Ultimate*, permite inicializar un proyecto en el que se empleen *WebSockets* de *Ktor*.

Al iniciar el programa, se debe aportar el argumento '*-config*' con el que indicar la ruta al archivo JSON que almacena la configuración de usuarios y grupos, entre otras cosas, como ya se ha comentado en el punto 2.6. Si el archivo se puede leer correctamente, se inicializará el servidor deserializando la información incluida en el JSON a una clase *RomeoConfig*, que mantendrá los objetos de participantes y grupos en un formato temporal, y no como objetos propios del *model* directamente. Esto sucede así, pues el JSON almacena la lista de participantes de cada grupo como una lista de enteros, donde cada uno de ellos es la referencia a un participante. Cuando los objetos se incluyan en el *model*, se crearán los participantes en primer lugar, y después se crearán los grupos buscando en la lista de usuarios las referencias a cada participante. La información del *model* se guardará en una clase llamada *RomeoServer*.

Cuando se obtiene la clase *RomeoConfig* con la información del JSON de configuración, se inicia el sistema de inyección de dependencias *Koin*, instanciando los módulos como ya se ha visto en el punto 3.3. En este caso, los módulos son *RomeoConfig* y *RomeoServer*. Este último, como requiere para su creación de un *RomeoConfig*, se le asigna automáticamente usando la inyección de dependencias con *get()*.

Una vez se tiene el objeto *RomeoServer*, se inicia el servicio. Tal y como se ve en la Figura 50, se le debe indicar el *framework* que se empleará internamente, en este caso *Netty*, el puerto en el que va a escuchar el servidor, el host y la función de módulo de aplicación en la que se desarrollará la lógica del servicio.

```
1. server = embeddedServer(
2.     Netty,
3.     port = 8080,
4.     host = "0.0.0.0",
5.     module = Application::module
6. )
```

Figura 50. Inicialización del servidor de *Ktor*

Dentro de esta función *Application.module()*, se instala el *add-on* de *WebSockets*, donde se indican diferentes parámetros relativos a la conexión:

- ***pingPeriod*:** Intervalo de tiempo entre mensajes *ping* para comprobar el estado de la conexión del dispositivo con el servidor.
- ***timeout*:** Tiempo que debe pasar el cliente sin responder a ningún *ping* para considerarse conexión fallida.
- ***maxFrameSize*:** Se trata del tamaño máximo que puede tomar una trama *WebSocket*. En este caso, se establece al valor máximo permitido, que corresponde con la cantidad máxima representable por una variable en formato *Long*.
- ***masking*:** Se enmascara el contenido de la trama con una secuencia de 32 bits. Esto es útil para evitar que atacantes puedan insertar código malicioso en la infraestructura que emplea *WebSockets*. En este caso, se establece a *false*, pues no se considera necesario.

Una vez instalado el *add-on* para *WebSockets*, se pueden emplear tantos *routing* como se desee para crear puntos del servidor a los que el usuario se podría conectar. En este caso, para todo lo relacionado con el cliente móvil que va a transferir audio, se ha empleado el *routing* con `webSocket("/romeo")`. En cambio, para el manejo, creación, eliminación y modificación de

usuarios y grupos por parte del administrador de la aplicación, se ha empleado una interfaz web accesible a través de otro *routing* distinto al empleado para el *WebSocket*, en el que existirán diversos `get()` y `post()` con los que poder servir las páginas web al usuario modificar los datos de forma acorde.

4.2 Gestión de conexiones y autenticación de usuarios

Cuando se establece una nueva conexión *WebSocket* al *endpoint* '/romeo', se ejecuta una nueva instancia del código que contiene el apartado *routing* para esa nueva conexión en particular. El funcionamiento se puede observar en la Figura 51.

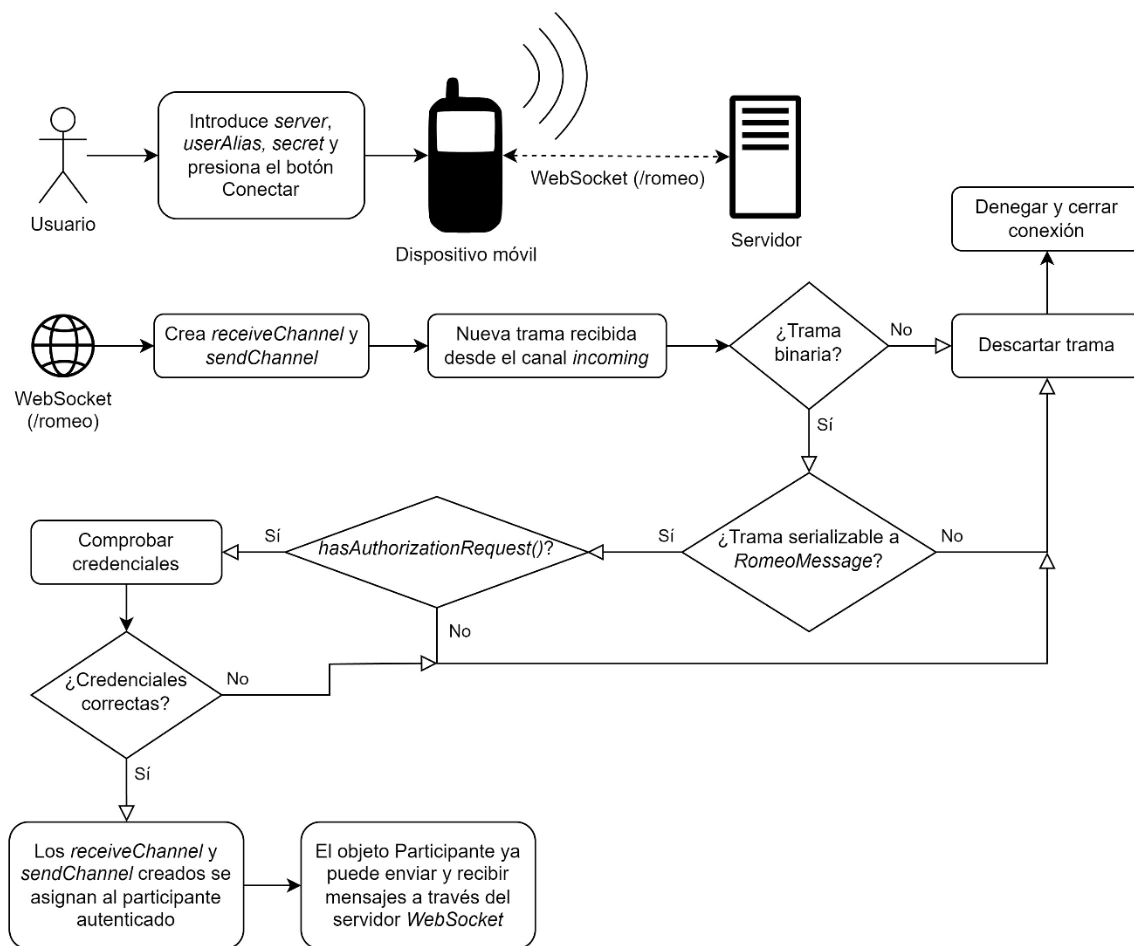

Figura 51. Diagrama de flujo del funcionamiento del servidor *WebSocket* cuando recibe una nueva conexión

De este diagrama, cabe destacar el uso de los *Channels* para manejar las diferentes conexiones y mensajes de los usuarios. Cuando se crea cada objeto *Participant* al iniciar el servidor de comunicación, se crean las variables *receiveChannel* y *sendChannel*, aunque se inicializan con valor *null*. Una vez el usuario introduce sus credenciales e intenta autenticarse en el servidor, si la autenticación ha sido satisfactoria, se crean los canales desde la instancia del *WebSocket* y se le asignan al usuario.

La función del objeto *Participant* que se ejecuta desde la instancia del *WebSocket* recibe el nombre de *setConnection()*. Como se puede ver en la Figura 52, los canales de envío y recepción son proporcionados por la instancia del *WebSocket*, y asignados a las variables *receiveChannel* y *sendChannel* del *Participant*. Posteriormente, se inicia una corutina en la que el *Participant* puede recibir los nuevos mensajes que recibe el *WebSocket*, y tratarlos en la función *newMsg()*.

```
1. fun setConnection(receiveChannel: Channel<RomeoMessageMsg>,
                     sendChannel: Channel<RomeoMessageMsg>) {
2.     this.receiveChannel = receiveChannel
3.     this.sendChannel = sendChannel
4.
5.     val auxScope = CoroutineScope(Job() + Dispatchers.IO)
6.     scope = auxScope
7.
8.     auxScope.launch {
9.         try {
10.            // connected
11.            while (isActive) {
12.                newMsg(receiveChannel.receive())
13.            }
14.            auxScope.cancel()
15.        } catch (e: Exception) {
16.            logger.info { "Exception in Participant: ${e.message}" }
17.            this@Participant.sendChannel = null
18.            this@Participant.receiveChannel = null
19.        }
20.    }
21. }
```

Figura 52. Función para establecer la conexión de un objeto *Participant*, de modo que pueda enviar y recibir mensajes desde la instancia de *WebSocket*

A continuación, una vez el objeto *Participant* tiene asignados los canales para enviar y recibir mensajes, cada vez que dicha sesión de *WebSocket* recibe un nuevo mensaje, se envía a través del *receiveChannel* con `receiveChannel.send(romeoMessage)` y se recibe en el *Participant* con `receiveChannel.receive()`, como se observa en la línea 12 de la Figura 52. Asimismo, en la instancia del *WebSocket*, se inicia una corutina en la que, con un bucle *while(isActive)*, se transmiten los mensajes que se reciben en el *sendChannel* a los dispositivos móviles de los usuarios con `send(binaryFrame(sendChannel.receive()))`; dichos mensajes previamente han sido enviados desde un *Participant* con `sendChannel?.send(romeoMessage)`. De este modo, se puede realizar un tratamiento individualizado de cada paquete en función del usuario que lo haya enviado, para tener acceso directo a la información de dicho usuario a la hora de plantear una respuesta a cada uno de los mensajes.

Adicionalmente, cabe mencionar la función que se emplea para autenticar al usuario. Se trata de la función *validateAuthenticationRequestMsg()*. A esta función se le informa del mensaje de autenticación recibido y devuelve un par de datos con un *Boolean* y un objeto *Participant*, que será la instancia del usuario autenticado, en caso de que la autenticación haya sido exitosa. El código de dicha función se puede observar en la Figura 53. La función *isAlreadyAuthenticated()*, de la línea 4, únicamente comprueba si los canales del *Participant* son nulos, o no.

```
1. fun validateAuthenticationRequestMsg(authorizationRequest: AuthorizationRequestMsg):
                                         Pair<Boolean, Participant?> {
2.     participant(authorizationRequest.userAlias)?.let {
3.         if (it.hashpwd == authorizationRequest.secret) {
4.             if (!it.isAlreadyAuthenticated()) {
5.                 logger.info { "Participant ${it.userAlias} authenticated" }
6.                 return Pair(true, it)
7.             }
8.         }
9.     }
10.    logger.info { "Participant ${authorizationRequest.userAlias} not authenticated" }
11.    return Pair(false, null)
12. }
```

Figura 53. Función para autenticar a un usuario

4.3 Recepción, tratamiento y envío de mensajes

Una vez se ha establecido correctamente la sesión *WebSocket*, el usuario se ha autenticado correctamente y se han asignado los *Channels* de envío y recepción de mensajes al participante en cuestión, resta contemplar cómo se tratan los mensajes que se reciben y cómo se actúa en cada caso. Como ya se ha comentado, esto se realiza mediante la función *newMsg(romeoMessage: RomeoMessageMsg)*.

Se van a mostrar en las figuras siguientes la forma en la que se trata la recepción de algunos de los mensajes más significativos para el funcionamiento de la aplicación. En primer lugar, en la Figura 54, aparece la generación del mensaje *GroupsResponseMsg* a partir de la petición *GroupsRequestMsg*.

```
1. RomeoMessageMsg.MessageCase.GROUPSREQUEST -> {
2.     val groupsResponseMsg = protoGroupsResponseMsg(groups)
3.     sendMsg(groupsResponseMsg)
4. }
5. // …
6. fun protoGroupsResponseMsg(groups: List<Group>): RomeoMessageMsg {
7.     val protoGroups = protoGroups(groups)
8.     val groupsResponseMsg = GroupsResponseMsg.newBuilder()
9.         .setRequestId(2)
10.        .addAllGroups(protoGroups)
11.        .build()
12.
13.    val romeoMessageMsg = RomeoMessageMsg.newBuilder()
14.        .setGroupsResponse(groupsResponseMsg)
15.        .build()
16.
17.    return romeoMessageMsg
18. }
```

Figura 54. Construcción del mensaje de respuesta para informar de los grupos a los que pertenece un usuario

Se puede observar que, para generar el mensaje de respuesta con los grupos, existe la función *protoGroupsResponseMsg()*, que toma como argumento una lista de grupos. Como el objeto *Participant* mantiene dicha lista con los grupos a los que pertenece, es directo hacer `protoGroupsResponseMsg(groups)`.

En la Figura 55, en cambio, aparece la forma en la que se decide si un participante tiene permiso, o no, para hablar en un determinado grupo. El permiso, como ya se ha comentado, se establecerá en función de si hay otro participante ya hablando en el grupo solicitado, de modo que se eviten las colisiones entre participantes. Como se ve, cada objeto de *Group* tiene una referencia al hablante actual en dicho grupo, por lo que es directo determinar si hay alguien hablando, o no, solo comprobando que la referencia *currentSpeaker* es nula, o no.

En caso de que la referencia sea nula, se establece al participante que ha solicitado hablar como *currentSpeaker* de ese grupo y se le devuelve el mensaje *TalkPermissionResponseMsg* con la respuesta.

```
1. RomeoMessageMsg.MessageCase.TALKPERMISSIONREQUEST -> {
2.     val permission: Boolean
3.     val groupAliasRequested = romeoMessage.talkPermissionRequest.group.groupAlias
4.     val groupRequested = romeoServer.groupMap?.get(groupAliasRequested)
5.     if (groupRequested?.currentSpeaker == null) {
6.         if (groupRequested != null) {
7.             groupRequested.currentSpeaker = this
8.             permission = true
9.
10.        } else {
11.            permission = false
12.        }
```

```
13.     } else {
14.         permission = false
15.     }
16.     logger.info { "Permission for ${this.name}: $permission" }
17.     val talkPermissionResponseMsg = protoTalkPermissionResponse(permission)
18.     sendMsg(talkPermissionResponseMsg)
19. }
```

Figura 55. Establecimiento del permiso para hablar de un participante en un grupo

El mensaje *ChannelReleaseRequestMsg* tiene el comportamiento contrario a este último. En el caso de que el *currentSpeaker* del grupo solicitado sea el participante que ha enviado el mensaje, se liberará el canal, y se establecerá a *null* la referencia *currentSpeaker*.

La recepción de un mensaje que contiene una muestra de audio (*AudioMsg*) se trata de la manera expuesta en la Figura 56. En este caso, el reenvío del mensaje en el grupo es directo. Únicamente se comprueba que el grupo indicado por el campo *groupAlias* en el mensaje de audio no sea nulo. En siguientes puntos se explicará el funcionamiento de *sendAudioRelatedMessage()*.

```
1. RomeoMessageMsg.MessageCase.AUDIO -> {
2.     val groupAliasInAudio = romeoMessage.audio.groupAlias
3.     val groupInAudio = romeoServer.groupMap?.get(groupAliasInAudio)
4.     if (groupInAudio != null) {
5.         sendAudioRelatedMessage(romeoMessage, groupInAudio, this)
6.     }
7. }
```

Figura 56. Tratamiento del mensaje de audio

Otros mensajes, como el mensaje *RttTest*, simplemente se devuelven al emisor sin realizar ningún cómputo intermedio. Empleando todas las técnicas observadas a lo largo de este punto, se ha conseguido la creación de un protocolo de comunicación personalizado y adaptado a todas las funcionalidades deseadas.

4.3.1 Envío de mensajes cliente-servidor y difusión en un grupo

De todos los mensajes que envía el *Participant* por el *WebSocket* vistos anteriormente, hay algunos que solo deben llegar al usuario que mandó la petición inicial, otros que deben llegar al resto de participantes del grupo en cuestión, y otros que deben recibir todos los participantes, incluyendo el que provocó el envío de dicho mensaje. Por ello, se emplean tres funciones distintas en la clase *Participant*.

En primer lugar, la función *sendMsg(romeoMessage: RomeoMessageMsg)* envía un mensaje al propio usuario representado por el *Participant* desde el que se envía. Se emplea `sendChannel?.send(romeoMessage)` para entregar el mensaje al *WebSocket* y que sea enviado al usuario.

En segundo lugar, se utiliza la función *sendMsgInGroup(romeoMessage: RomeoMessageMsg, group: Group)* para difundir un mensaje a todos los integrantes de un grupo determinado. Para ello, se recorre la lista de participantes del grupo indicado, de la forma que se observa en la Figura 57.

```
1. private fun sendMsgInGroup(romeoMessage: RomeoMessageMsg, group: Group) {
2.     scope?.launch {
3.         for (auxParticipant in group.participants) {
4.             logger.info { "SENDING $romeoMessage TO ${auxParticipant.userAlias}" }
5.             auxParticipant.sendChannel?.send(romeoMessage)
6.         }
7.     }
8. }
```

Figura 57. Función para difundir un mensaje en un grupo

Por último, para enviar un determinado tipo de mensajes, como los mensajes que transportan muestras de audio, que deben ser recibidos por todos los integrantes de un grupo a excepción del participante que los envió, se emplea la función *sendAudioRelatedMessage(romeoMessage: RomeoMessageMsg, group: Group, participant: Participant)*. A esta función se le proporcionan como argumentos el mensaje a enviar, el grupo en el que debe ser enviado, y además el participante de dicho grupo al que el mensaje no debe ser transmitido. El código que consigue esto se puede ver en la .

```
 1. private fun sendAudioRelatedMessage(romeoMessage: RomeoMessageMsg,
                                        group: Group,
                                        participant: Participant) {
 2.     scope?.launch {
 3.
 4.         val auxParticipantList = group.participants.toMutableList()
 5.
 6.         auxParticipantList -= participant
 7.         for (auxParticipant in auxParticipantList) {
 8.             if (auxParticipant.sendChannel != null) {
 9.                 auxParticipant.sendChannel?.send(romeoMessage)
10.             }
11.         }
12.     }
13. }
```

Figura 58. Función para difundir un mensaje de audio en un grupo

4.4 Gestión de desconexión de sesiones

Dada la naturaleza de la aplicación, esta va a ser empleada en multitud de situaciones en las que, probablemente, la conexión no sea estable y sufra desconexiones constantes. Por ejemplo, un conductor en una carretera con baja señal de red móvil, o un trabajador en un almacén en el que la señal Wi-Fi no cubre toda el área de trabajo. Es por ello por lo que se debe tratar la pérdida de conexión de forma que, en el momento que el dispositivo se vuelva a conectar a la red, la aplicación vuelva a iniciar una conexión *WebSocket* con el servidor.

Una sesión de *WebSocket* se cierra en el momento en que se pierde la conexión en alguno de los extremos debido a la expiración de un temporizador de *timeout* TCP, o por un error en la interfaz de red del dispositivo. En cualquier caso, la capa TCP elevará un error a las capas superiores, lo que causará una excepción en bucle de recepción de la instancia del *WebSocket*. En ese momento, se cancelarán los canales *receiveChannel* y *sendChannel*, y se lanza una excepción personalizada llamada *ConnectionException(message: String)*. De este modo, al haberse lanzado una excepción que no será tratada, se cancela el *scope* en el que se ejecutaba la recepción de mensajes, cancelando así el *WebSocket* en la parte del servidor. Esto se puede observar en la Figura 59.

```
 1. coroutineScope {
 2.
 3.     // coroutine to handle the incoming messages
 4.     launch {
 5.         while (isActive) {  // this loop will be executed as long as
                                   the scope lives through the execution
 6.
 7.             try {
 8.                 val frame = incoming.receive()
 9.                 // TRATAMIENTO DE LAS TRAMAS ENTRANTES
10.             } catch (e: Exception) {
11.                 receiveChannel.cancel()
12.                 sendChannel.cancel()
13.                 throw ConnectionException("Connection Closed")
14.             }
```

```
15.         }
16.     }
17.     // …
18. }
```

Figura 59. Gestión de desconexiones en el bloque *try-catch*

Por el lado de los *Participant*, como los canales de recepción y envío se han cerrado desde el extremo del *WebSocket*, al ejecutar nuevamente *receiveChannel.receive()* en el bucle de la Figura 52, se produciría una *CancellationException*, que se trata en el bloque *catch* eliminando las referencias a los canales en el objeto *Participant* y, por lo tanto, desconectando al *Participant*.

Por otro lado, cuando la conexión se pierda por el lado del cliente, este se volverá a conectar automáticamente al servidor con las mismas credenciales tan pronto como recupere la conexión a la red.

4.5 Servicio Web de administración del servidor

Tal y como ya se ha visto en el punto 2.6, el servidor requiere de un fichero JSON para cargar los datos de todos grupos y los usuarios que los forman, pues se ha considerado, por diversos motivos, que esta era la mejor forma de manejar dichos datos en una implementación *on-premise*, modificable en un futuro.

Sin embargo, con el funcionamiento que se ha descrito hasta ahora, cada vez que se quiera modificar, añadir o eliminar un usuario o un grupo sería necesario parar la ejecución del servidor de comunicación, modificar los datos sobre el archivo JSON directamente, y volver a iniciar el servidor con el nuevo archivo de configuración. Por ello, con el objetivo de facilitar todo este proceso al administrador, y además hacer que no sea necesario reiniciar el servidor con cada modificación, se ha desarrollado un portal web de gestión desde el que el administrador puede iniciar sesión con sus credenciales y realizar todos los cambios que sean necesarios.

Para llevarlo a cabo, se ha recurrido a un nuevo apartado *routing* en la función de extensión *Application.module()*. Dentro de este nuevo apartado, se han desarrollado cada una de las páginas que se deberían mostrar en formato HTML, pero programado con objetos Kotlin empleando la librería *kotlinx.html*, además de tratar todos los datos de los formularios que se le presentan al administrador para realizar los cambios oportunos, ya sea con peticiones GET o POST.

4.5.1 Páginas empleadas

En primer lugar, cuando el administrador introduzca la dirección del servidor en el navegador, será redirigido automáticamente a la página "*/login*" (Figura 60), donde podrá introducir sus credenciales de administrador. Dichas credenciales se guardan en campos del mismo archivo JSON, aunque también se pueden modificar desde el portal web, como se verá posteriormente.

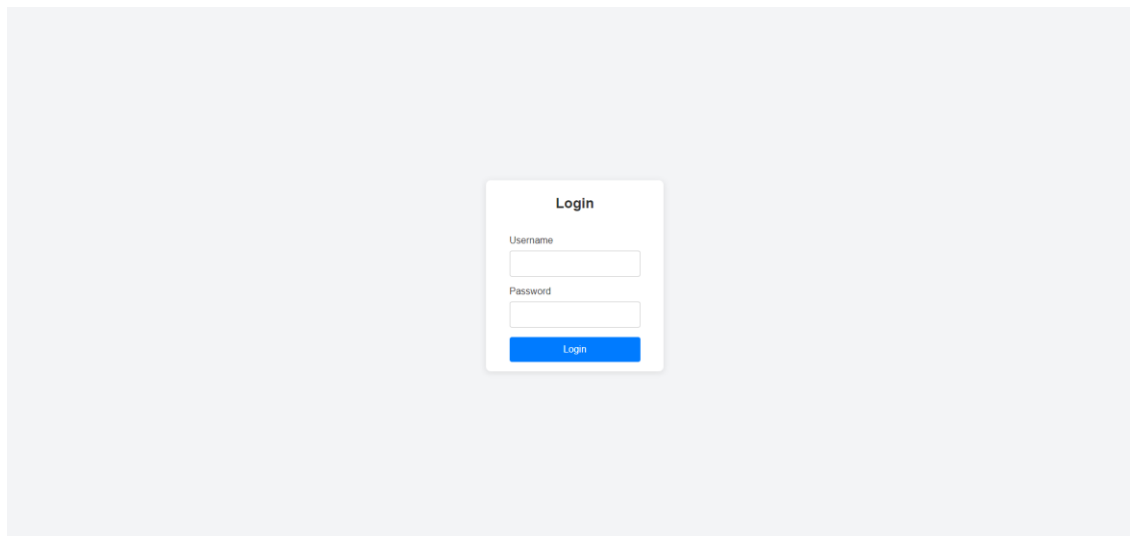

Figura 60. Página de *login* del portal web

Una vez introducidas las credenciales del administrador, se verificarán en la ruta "*/checklogin*", donde, en caso de haber introducido las credenciales correctas, se le proporcionará una *cookie* de sesión con la que podrá acceder al resto de páginas del portal. Posteriormente, se le mostrará la página "*/mainpage*". En caso de haber introducido datos incorrectos, se le redirigirá a la página de *login* de nuevo. La página principal tiene una estructura como la que se observa en la Figura 61.

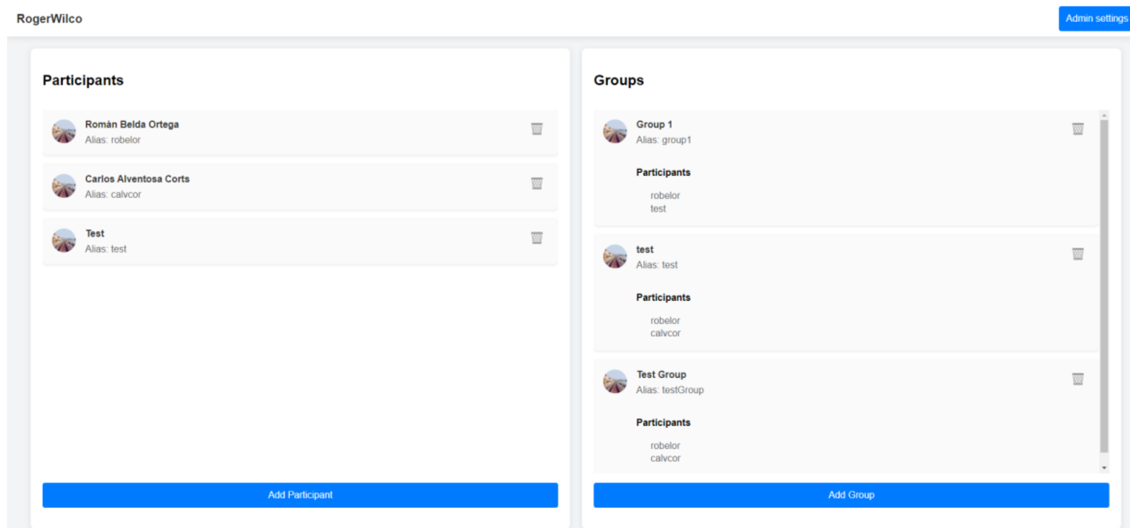

Figura 61. Página principal del portal web de administración de usuarios y grupos

En esta página, aparece la lista con todos los usuarios y todos los grupos. Haciendo *click* sobre cualquier elemento de ambas listas llevará al administrador a una página en la que podrá editar los parámetros de ese usuario o grupo en cuestión. Además, todos los elementos, ya sean usuarios o grupos, pueden ser eliminados con el icono de la papelera que aparece a la derecha de cada nombre. Los botones de la parte inferior permiten añadir participantes o grupos, por lo que llevarán al administrador a una página con un formulario, en el que podrá aportar los datos del usuario o grupo que esté creando.

Adicionalmente, en la parte superior derecha hay un botón llamado "*Admin settings*" que, al ser pulsado, llevará a la página "*/modifyadmindata*" en la que el administrador podrá cambiar su nombre de usuario o su contraseña, gestionando el resultado en "*/changecredentials*" con el método POST; así como cerrar la sesión en "*/logout*", donde se eliminará la *cookie* de sesión que

se le había otorgado al administrador al iniciar la sesión. Esta página tiene la estructura que se muestra en la Figura 62.

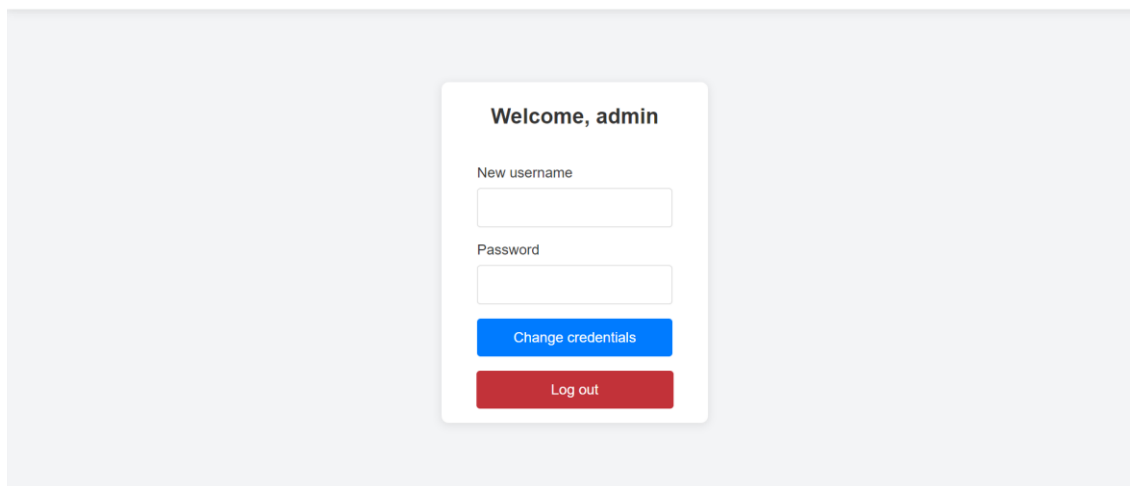

Figura 62. Página para modificar las credenciales del administrador en el portal web

La página en la que se puede añadir un nuevo usuario se encuentra en la ruta "*/addparticipant*" y, de forma análoga, en "*/addgroup*" se encuentra el formulario con el que añadir un grupo. Para guardar el nuevo usuario o grupo, la gestión de los formularios se realiza en "*/submitparticipant*" y "*/submitgroup*", respectivamente, mediante el método POST.

Del mismo modo, se puede editar un usuario o grupo en "*/editparticipant*" y "*/editgroup*", tratando sus respectivos formularios en "*/submiteditparticipant*" y "*/submiteditgroup*", también con el método POST. Los usuarios se pueden eliminar en "*/removeparticipant*", y los grupos, en "*/removegroup*", aportando el alias del usuario o el grupo que se desee eliminar en la petición GET.

Un ejemplo de página para modificar o introducir datos se encuentra en la Figura 63. En ese caso, se trata de la página para crear un grupo. Se puede observar cómo los participantes del grupo se eligen en una lista con *checkbox* en la que se selecciona a los participantes de entre todos los usuarios de la organización.

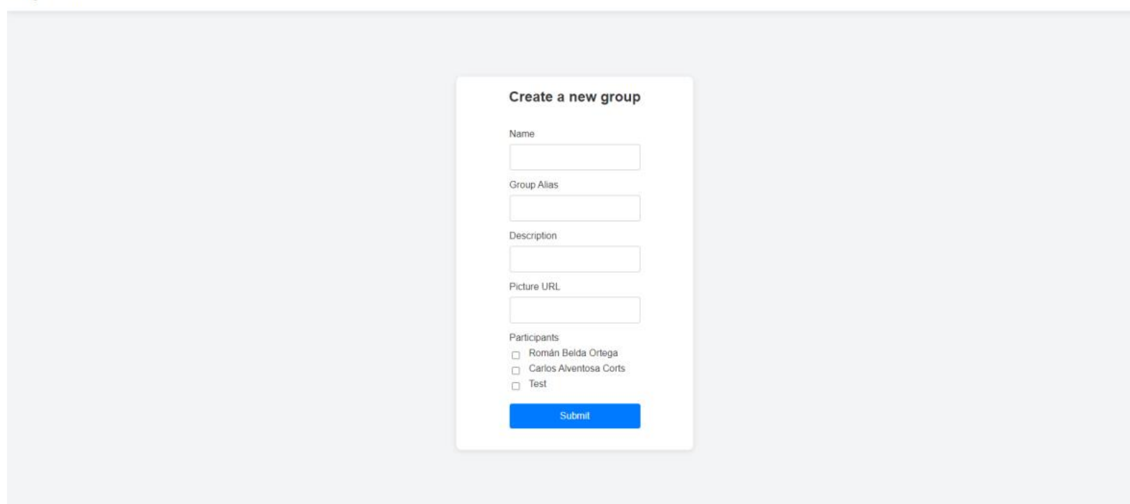

Figura 63. Página para crear un nuevo grupo en el portal web

Además, es necesario mencionar que la hoja de estilos "*styles.css*" se sirve con la ruta "*/static/styles*", empleando la función *staticFiles()*, que recibe como argumento la ruta, el fichero

y un índice. En cada una de las pantallas se añadirá el enlace para que el navegador se descargue la hoja de estilos, en caso de necesitarla, con:

```
link(rel = "stylesheet", href = "/static/styles")
```

4.5.2 Creación, modificación y eliminación de elementos. Persistencia

Para comentar la forma en que se crean, modifican y eliminan usuarios y grupos empleando el portal web, se va a recurrir a un ejemplo. Concretamente, se hablará de los usuarios, aunque para los grupos el procedimiento es análogo.

La creación de un usuario ocurre en la ruta "*/submitparticipant*", tras haber recibido los datos mediante el método POST. Lo primero que se hace es extraer cada uno de los parámetros de la petición, y asignarlos a variables que se puedan emplear en el código. Además, se obtiene un nuevo objeto *RomeoConfig* a partir del fichero JSON. Se comprueba en la lista de *ParticipantConfig* de este objeto que no exista ningún usuario con el mismo *userAlias* que el introducido en el formulario, pues deben ser identificadores únicos. Además, se obtiene el ID (*Integer*) más alto de entre todos los usuarios existentes, necesario para poder identificar al nuevo usuario en la lista de participantes de cada grupo (se recuerda que los grupos almacenan la lista de participantes como una lista de enteros, Figura 19). El nuevo usuario tendrá un ID superior en una unidad al ID más alto obtenido anteriormente.

Una vez hecho esto, se procede a la creación de un nuevo objeto *ParticipantConfig* con los datos recibidos del formulario y el identificador calculado, y se añade a la lista *participants* del objeto *RomeoConfig*. Si no ha habido ningún error, se serializa el objeto de nuevo a un documento JSON y se guarda en la estructura de archivos del programa, sobrescribiendo el documento anterior. Se puede ver cómo se gestiona la creación de un usuario en el código de la Figura 64.

```
 1. post("/submitparticipant") {
 2.     val userSession: UserSession? = call.sessions.get<UserSession>()
 3.     if (userSession == null) {
 4.         call.respondRedirect("/login")
 5.     } else {
 6.         val parameters = call.receiveParameters()
 7.         val name = parameters["name"]
 8.         val userAlias = parameters["userAlias"]
 9.         val pictureUrl = parameters["pictureUrl"]
10.         val secret = parameters["secret"]
11.
12.         val jsonString = File("config.json").readText()
13.         val config = jsonAdapter.fromJson(jsonString)
14.
15.         if (name != null && userAlias != null &&
                pictureUrl != null && secret != null && config != null) {
16.             if (config.participants.firstOrNull
                                      { it.userAlias == userAlias } == null) {
17.
18.                 // choose an id 1 higher than the highest current id
19.                 val idList = config.participants.maxOfOrNull { it.id } ?: 0
20.
21.                 config.participants += ParticipantConfig(
22.                     idList + 1,
23.                     name = name,
24.                     userAlias = userAlias,
25.                     hashpwd = secret,
26.                     description = "",
27.                     pictureUrl = pictureUrl
28.                 )
29.
30.                 val updatedJsonString = jsonAdapter.toJson(config)
31.                 File("config.json").writeText(updatedJsonString)
32.
33.                 romeoServer.updateConfig(config)
34.
35.                 call.respondRedirect("/mainpage")
```

```
36.             } else {
37.                 call.respondRedirect("/addparticipant")
38.             }
39.         } else {
40.             call.respondRedirect("/addparticipant")
41.         }
42.
43.     }
44. }
```

Figura 64. Gestión de la creación de un usuario desde el portal web

La modificación de un usuario se realiza de forma similar, en la ruta "*/submiteditparticipant*". Cuando se accede a esta ruta, se obtienen de nuevo todos los parámetros de la petición POST, y se obtiene de nuevo el objeto *RomeoConfig* a partir del JSON existente. Una vez obtenido, se busca en la lista *participants* de *RomeoConfig* mediante el parámetro *userAlias* el usuario modificado. Se actualizan todos los campos de este *ParticipantConfig* con los recibidos desde el formulario, y se vuelve a sobrescribir el documento.

Por último, la eliminación de un usuario sucede en la ruta "*/removeparticipant*" a la que, con el método GET, se le proporciona como parámetro el ID del usuario a eliminar. Se vuelve a obtener de nuevo el objeto *RomeoConfig* pero, antes de eliminar el usuario directamente de la lista *participants*, es necesario comprobar y eliminar dicho usuario de las listas de todos los grupos en los que se encuentre dicho usuario. Finalmente, se elimina el usuario de la lista, y se sobrescribe el documento JSON anterior.

Cabe destacar que la modificación del archivo JSON no implica directamente la modificación de los datos del *model* de la aplicación. Es por ello por lo que, después de cada modificación, se llama a la función *updateConfig(config: RomeoConfig)* del *model* (*RomeoServer*) para actualizar sus datos.

4.5.3 Reconexión de clientes y actualización reactiva

En el punto 4.5.2, cuando se crea, modifica o elimina un elemento de la lista de usuarios o grupos, se sobrescribe el documento JSON en el que se almacena la información y se actualiza el *model* de modo que el programa del servidor funcione con los datos más actualizados. Sin embargo, los usuarios solamente obtendrán dichos datos actualizados cuando se desconecten y se vuelvan a conectar al servidor, pues es en ese momento en el que se envía la lista de grupos en los que participan, y los usuarios que pertenecen a dichos grupos.

Por lo tanto, es necesario que exista una forma de desconectar a un usuario desde el servidor. Para ello, se ha planteado un nuevo mensaje de *ProtocolBuffers* llamado *RestartMsg*. Cuando el servidor envía dicho mensaje a los dispositivos móviles de los usuarios, estos eliminan las conexiones *WebSocket* que tenían activas, forzando nuevas conexiones. Una vez se vuelven a conectar, el servidor les proporciona la nueva información actualizada, y esto permite que los cambios que se realicen en el portal web actualicen la interfaz gráfica de los clientes de forma inmediata.

Capítulo 5. Comprobación de resultados y despliegue

5.1 Aplicación resultante y recomendaciones seguidas

A la aplicación, una vez terminada, se le ha puesto el nombre de *Roger*. Se trata de una palabra clave muy empleada en aviación, como abreviatura de "*I have received all of your transmission*". Es, por lo tanto, una respuesta a una transmisión para informar al emisor de que se ha recibido su transmisión de forma correcta. Se emplea frecuentemente junto con la palabra *Wilco*, que abrevia a la expresión "*Will comply*", que significa que el receptor ha recibido las indicaciones del emisor, y actuará en consecuencia. El logo diseñado es el que se puede ver en la Figura 65, realizado a partir de una tipografía *open-source* llamada "*good timing*".

Figura 65. Logo de la aplicación

En la Figura 66, se puede observar cómo ha resultado la interfaz gráfica de las cuatro pantallas con las que cuenta la aplicación, aproximándose al diseño planteado en la Figura 7.

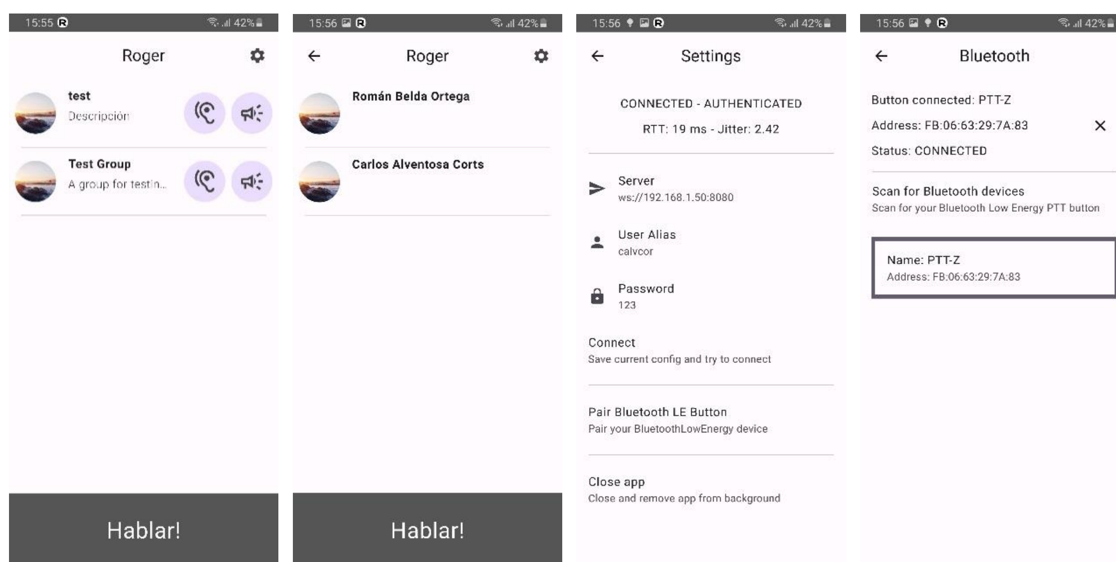

Figura 66. Interfaz gráfica en la versión final de la aplicación. De izquierda a derecha: pantalla de grupos, pantalla de participantes de un grupo, pantalla de ajustes y pantalla de conexión con dispositivos Bluetooth *Low Energy*

En la página *Android Developers* [20] se exponen ciertas prácticas que Google recomienda seguir para desarrollar una aplicación consistente y empleando las mejores prácticas. Algunas de las recomendaciones indicadas y seguidas en este proyecto son las siguientes.

- **Utilizar una capa de datos claramente definida:** En este proyecto, se ha desarrollado una clase *Model* que alberga todos los datos de la aplicación.

- **Utilizar una capa de interfaz de usuario claramente definida:** En este caso, se trata del *NavigationScreen* con todas sus pantallas, que representan los datos del *Model* a través del estado definido en el *ViewModel*.
- **Utilizar flujos y corutinas:** Se ha expuesto repetidamente a lo largo de la memoria todos los casos en los que se han empleado corutinas y flujos para manejar los datos, lo que lleva a otro punto:
    - **Seguir un flujo de datos unidireccional:** Este apartado se ha comentado en el punto 3.1.2, donde se ha hablado del concepto "*single source of truth*", y en el punto 3.2, en el que se ha expuesto cómo la arquitectura de interfaz gráfica MVI consigue esta unidireccionalidad de los datos.
- **Emplear una sola actividad en la aplicación:** Con el uso de herramientas como la navegación de *Compose* se ha logrado establecer distintas pantallas accesibles desde la misma actividad.
- **Emplear *Jetpack Compose*:** Esta ha sido la librería empleada para crear funciones representables en la interfaz gráfica.
- **Utilizar inyección de dependencias:** En el punto 3.3 se ha explicado cómo se ha empleado el sistema de inyección de dependencias *Koin* en la aplicación.

5.2 Retardo en el audio. Interrupciones. Estudio del *jitter*

En este momento, únicamente falta verificar que tanto la aplicación móvil como el servidor funcionan como se ha planteado en los objetivos de esta memoria. Para ello, se van a realizar diferentes pruebas. La primera de ellas consiste en medir el retardo que existe desde que el emisor empieza a hablar hasta que el receptor empieza a escuchar. Para ello, se ha grabado la prueba con un micrófono para, posteriormente, medir el retardo con exactitud.

La prueba ha consistido en situar dos teléfonos juntos, uno transmitiendo y el otro recibiendo y reproduciendo el audio. Durante la grabación, se ha provocado un sonido corto y fuerte, como un golpe o una palmada, de modo que pueda ser captado tanto el sonido original como el reproducido a través de la aplicación. El retardo se obtendrá visualizando la forma de onda de la grabación, y seleccionando el espacio entre dos picos de audio consecutivos: el primero será el sonido original, y el segundo, el que ha viajado hasta el servidor.

Cabe mencionar que esta prueba se ha hecho a una distancia de, aproximadamente, 33 km del servidor (Valencia - Algemesí). El retardo podría variar ligeramente en función de dicha distancia. En la Tabla 1 se encuentran los resultados, medidos de 3 formas distintas: con el emisor y el receptor conectados a una red Wi-Fi, con uno de los dispositivos conectados a la red móvil y el otro a una red Wi-Fi y, finalmente, con ambos dispositivos funcionando a través de la red móvil. La red móvil empleada ha sido la red de 4G de Orange, y la red Wi-Fi accede a Internet a través de una conexión de fibra óptica de la compañía Movistar.

	1ª prueba	2ª prueba	3ª prueba	4ª prueba
Wi-Fi - Wi-Fi	331 ms	313 ms	302 ms	310 ms
4G - Wi-Fi	453 ms	386 ms	432 ms	465 ms
4G - 4G	472 ms	423 ms	372 ms	537 ms

Tabla 1. Retardo en el audio en función de la red de acceso a Internet del dispositivo móvil (Wi-Fi vs 4G)

Se puede apreciar que, con la conexión Wi-Fi probada, los resultados de retardo son ligeramente mejores que los de las pruebas realizadas involucrando a la red móvil.

Otra de las pruebas que se puede realizar es el estudio del *jitter*. El parámetro, calculado anteriormente en el punto 3.9, se muestra constantemente, junto con el valor de RTT, en la pantalla

de ajustes de la aplicación. El valor dado depende de una gran cantidad de factores, como la red de acceso que se emplee, el estado de congestión de los *routers* que atraviesan la ruta al servidor, etc. Los datos medidos con las condiciones Wi-Fi expuestas previamente, son de entre 1.8 ms y 3 ms de *jitter*. En cambio, empleando la conexión 4G anterior, el *jitter* aumenta significativamente hasta valores comprendidos entre los 25 y 35 ms. Esto sucede por la naturaleza de la conexión 4G, en la que se compite por transmitir con numerosos dispositivos conectados a la misma estación base.

La siguiente prueba consiste en la medición de la tasa binaria empleada para transmitir o recibir un mensaje de audio. Con la configuración por defecto del codificador *Opus*, con la que se obtiene un flujo de audio de 64 kbps [21], teniendo en cuenta el *overhead* causado por las diferentes cabeceras, el flujo total asciende hasta los 80 kbps, aproximadamente. Este resultado es muy positivo, pues existe la posibilidad de incrementar el *bitrate* del audio con el objetivo de obtener una mejor calidad, pero sin suponer una carga para la red móvil que, en caso de 3G, alcanza unas tasas máximas de transmisión de entre 3 y 5 Mbps.

La última prueba que se ha realizado para comprobar la calidad de la conexión de la aplicación con el servidor ha sido el envío de mensajes de audio en diferentes redes de acceso, para comprobar la calidad del audio en recepción, si hay cortes de audio, etc. Se ha comprobado que, tanto en la red Wi-Fi anterior, como con conexiones 4G y 5G, la aplicación se comporta bien. Con una cobertura razonable, no se producen cortes en el audio, ni una degradación en la inteligibilidad del mensaje recibido. No obstante, es cierto que, con una buena cobertura 3G, se producen ciertos cortes en el audio, de pocos milisegundos que, de todos modos, no suponen un gran incremento en el retardo total ni una falta de comprensión del mensaje.

Todas las pruebas de funcionalidad realizadas, como la recepción de mensajes de múltiples emisores en distintos grupos simultáneamente o el envío de un mensaje durante la recepción de otro han sido satisfactorias.

5.3 Creación de la imagen de Docker

Para facilitar el despliegue del servidor de comunicación en cualquier máquina de forma sencilla, se ha recurrido a la utilización de contenedores *Docker*. Se trata de una forma de empaquetar todo lo necesario para ejecutar la aplicación (el *bytecode*, la máquina virtual de Java, las dependencias necesarias, etc.) en un formato de archivo estándar desplegable fácilmente con un comando [22].

Antes de crear la imagen, se debe compilar el programa para obtener el archivo JAR del programa del servidor de comunicación. Para conseguirlo, se ejecuta el comando `./gradlew build`, con el que se obtendrá el archivo JAR en la ruta `/build/libs` del proyecto.

Seguidamente, para crear la imagen, primero es necesario especificar los contenidos que esta debe tener, las rutas a los archivos que va a necesitar para ejecutarse (en este caso, el archivo JSON con la configuración de usuarios y grupos) y los puertos de red que se deben permitir. Todo esto se configura en un archivo llamado *Dockerfile*, que tiene la forma que se muestra en la Figura 67.

```
1. FROM openjdk:21
2. WORKDIR /app
3. COPY build/libs/romeo-server-all.jar /app/romeo-server-all.jar
4. COPY config.json /app/config/config.json
5. EXPOSE 8080
6. CMD ["java", "-jar", "/app/romeo-server-all.jar", "-conf", "/app/config/config.json"]
```

Figura 67. *Dockerfile* con la configuración para crear la imagen de *Docker* del servidor

En la primera línea, se define la imagen base de *Docker*. En este caso, la imagen *openjdk* en su versión 21 del *Java Development Kit* (JDK), pues incluye todas las herramientas necesarias para

la ejecución de aplicaciones Java (o cualquier lenguaje que se ejecute sobre la JVM; Kotlin, en este caso).

A continuación, en la línea 2, se define cuál será el directorio raíz en el interior del contenedor. Será en esa ruta, */app*, donde se tendrán que situar tanto el JAR obtenido anteriormente con el programa, como cualquier otro archivo necesario, como el archivo JSON de configuración. Estos dos archivos mencionados se copian con el comando COPY a dicha ruta (líneas 3 y 4), y en la línea 5 se informa de que el contenedor debe escuchar en el puerto 8080 para conexiones entrantes.

Finalmente, la línea 6 indica el comando que se debe ejecutar en la línea de comandos del contenedor para poder lanzar la aplicación.

Una vez hecho esto, se abre una línea de comandos en la ruta en la que se encuentre el *Dockerfile*, y se ejecuta el comando "`docker build -t roger_server .`". De esta forma, se crea la imagen que podrá ser posteriormente distribuida en un repositorio o compartida por cualquier otro medio.

Para lanzar la imagen creada, basta con introducir el siguiente comando en una máquina con *Docker* instalado: "`docker run -v ./config:/app/config -p 8080:8080 roger_server`". El argumento *-p 8080:8080* mapea el puerto 8080 del contenedor al puerto 8080 de la máquina, permitiendo así la comunicación con el servidor.

5.4 Nombre de dominio y certificado SSL/TLS

En este punto, el servidor ya está desplegado y funcionando. No obstante, al no haberse configurado ningún certificado SSL/TLS, todas las comunicaciones se están realizando en plano y sin encriptación. Aprovechando que el protocolo *WebSocket* se basa en HTTP, se pueden aprovechar estos certificados para obtener el cifrado de todas las comunicaciones con el servidor, de la misma forma que sucede con HTTPS.

Hay un gran abanico de opciones para conseguir un certificado y añadirlo al servidor para que lo emplee, y la elección de una de estas opciones u otra dependerá en gran medida de cómo la empresa que adopte el servicio tenga planteada su infraestructura de red.

Para conseguir un certificado, previamente se debe adquirir un nombre de dominio. En este caso, se ha escogido el dominio `rogerwilco.xyz`. Una vez adquirido, se debe establecer la dirección IP asociada a ese nombre de dominio para que apunte a la dirección de la máquina que ejecuta el servidor. Una vez hecho esto, se puede obtener un certificado de varias formas. Una de ellas, es comprándolo en una Autoridad de Certificación (CA), aunque esto conlleva un coste monetario y se debe renovar cada cierto tiempo. Otra forma es emplear CAs como *Let'sEncrypt*, que ofrecen certificados SSL/TLS gratuitos con una validez de tres meses.

Al servidor de *Ktor* se le puede ofrecer el certificado en formato JKS, de la forma que se muestra en la Figura 68.

```
 1. ktor {
 2.     security {
 3.         ssl {
 4.             keyStore = keystore.jks
 5.             keyAlias = sampleAlias
 6.             keyStorePassword = foobar
 7.             privateKeyPassword = foobar
 8.         }
 9.     }
10. }
11.
12. sslConnector(
13.     keyStore = keyStore,
14.     keyAlias = "sampleAlias",
```

```
15.     keyStorePassword = { "123456".toCharArray() },
16.     privateKeyPassword = { "foobar".toCharArray() }) {
17.     port = 8443
18.     keyStorePath = keyStoreFile
19. }
```

Figura 68. Método para añadir un certificado SSL/TLS al servidor de *Ktor*. Fuente: *ktor.io*

Para realizar las pruebas, sin embargo, se ha optado por emplear un *proxy* inverso llamado *nginx*. Este *proxy* permite redirigir las peticiones a un servidor u otro dentro de una misma máquina en función del nombre de *host*. Por ejemplo, en este caso, la dirección para el servidor de comunicación desarrollado podría ser *app.rogerwilco.xyz*. El uso de este *proxy*, además, permite ejecutar un *script* con el que revalidar el certificado automáticamente con *Let'sEncrypt*, con lo que se trata de un método gratuito y automático en lo que respecta a la renovación de los certificados empleados.

Capítulo 6. Conclusión y trabajo futuro

6.1 Conclusión

En este Trabajo Fin de Grado se han adquirido las aptitudes y el conocimiento necesario para realizar no solamente una aplicación para el Sistema Operativo móvil Android, sino que se ha logrado integrar su funcionamiento con una plataforma desarrollada específicamente para este propósito.

Antes de comenzar el desarrollo, se plantearon todas las posibles soluciones para cada apartado de la aplicación, proporcionando argumentos a favor y en contra de cada una. Una vez establecidos los requisitos y las tecnologías para emplear, se llevó a cabo un proceso de lectura de la documentación ofrecida por Google en *Android Development*, así como *Kotlin* y *Ktor* para el servicio de comunicación, hasta dar con la forma de manejar todos los sistemas satisfactoriamente.

La principal cuestión que se planteaba una vez establecidas todas estas tecnologías fue si el protocolo *WebSocket* sería capaz de transportar flujos de audio grabados en tiempo real y reproducirlos inmediatamente en su llegada al receptor. Los resultados obtenidos han confirmado que la tecnología *WebSocket* no solamente es capaz de transportar flujos de audio, sino que además constituye una herramienta muy conveniente para trabajar en entornos cliente-servidor como el que se ha planteado en este proyecto.

En lo personal, el desarrollo de esta aplicación ha supuesto un reto muy interesante, tanto por haber aprendido un nuevo lenguaje de programación, como por el hecho de haber llevado a cabo un proyecto que, en un futuro, podrá ser utilizado por cualquier persona que lo considere oportuno o útil en su día a día.

6.2 Alcance

La aplicación planteada y desarrollada en este proyecto puede tener una utilidad real en empresas y entornos en los que se prioriza la privacidad y la inmediatez en la comunicación. Es una buena forma de conectar a los usuarios con sus compañeros de trabajo, con la seguridad que aporta el hecho de que el audio se transmita mediante un servidor situado en la propia infraestructura de red de la empresa.

Se pueden plantear múltiples casos de uso, como se ha ido haciendo a lo largo de esta memoria. Pueden ser, por ejemplo, la coordinación de flotas de conductores, la gestión del inventario en almacenes, los servicios de respuesta a emergencias, la organización de eventos, entre muchos otros.

Además, se ha iniciado una relación con la empresa valenciana *Esveu Media Systems Sl.* Esta empresa se dedica a realizar grabaciones de plenos de ayuntamientos, incorporar sistemas de conferencias, etc. Han comunicado que estarían interesados en realizar una prueba desplegando la aplicación en un entorno real dentro de un Ayuntamiento de la Comunidad Valenciana.

6.3 Trabajo futuro

El desarrollo de una aplicación móvil es un proceso que podría no tener fin. Un desarrollador trabaja constantemente para adaptar su aplicación tanto a los cambios que se efectúan en la plataforma de desarrollo (Android, en este caso) como a la forma en la que los usuarios interaccionan con su aplicación, a medida que reciben reseñas con propuestas de mejora o posibles errores que sucedan con el uso frecuente de la misma.

Se pueden apreciar dos líneas clave a desarrollar en un futuro a partir del trabajo realizado y expuesto en la presente memoria.

En primer lugar, para conseguir abarcar a la práctica totalidad del mercado de telefonía móvil, se debería desarrollar la aplicación para el sistema operativo iOS. En caso de querer realizar un desarrollo nativo, como se ha hecho para Android, esto conllevaría el aprendizaje del lenguaje de programación *Swift* y del *framework* de desarrollo de interfaz gráfica *SwiftUI*.

Swift es un lenguaje de programación creado por la empresa Apple, para el desarrollo de aplicaciones en todos sus sistemas operativos: macOS, iOS, iPadOS, watchOS, Apple CarPlay, etc. El IDE que soporta este lenguaje es *XCode*, un programa que solamente está disponible en el sistema operativo macOS, lo que supone un requisito adicional.

En segundo lugar, se debería desarrollar una modalidad de la aplicación pensada para el público general. Un modo en el que los usuarios no tuviesen que desplegar un servidor privado, pero aun así pudiesen comunicarse entre sí de forma segura.

Para conseguirlo, se podrían plantear múltiples soluciones. Por ejemplo, una solución podría enfocarse en crear un sistema de usuarios globales e independientes, con un sistema de 'amigos' a los que poder hablar directamente, añadir a grupos, etc. Esto conllevaría el hecho de tener que desarrollar un método para que los usuarios pudiesen, directamente desde la aplicación, encontrar a otros usuarios.

Otra solución más simple podría encararse a una comunicación más pública, en la que cada usuario pudiese insertar un código o 'clave de habla', de modo que todos los usuarios que compartiesen el mismo código podrían hablar entre sí, con el riesgo de que un usuario desconocido pudiese obtener dicho código y escuchar o participar en las conversaciones ajenas.

En ambos casos, siempre que el servicio sea gratuito, debería, hallarse una forma de reducir al máximo la carga del servidor global y el ancho de banda requerido, que además debería dimensionarse en función del número de usuarios simultáneos de la aplicación. Para conseguirlo, se podría hacer uso de un codificador más eficiente: Codec 2.

Codec 2 es un codificador de audio de código abierto que consigue unas tasas binarias extraordinariamente bajas, con modos de funcionamiento que rondan desde los 3200 hasta los 450 bits por segundo [23]. Con una conexión de 10 Gbps, y asumiendo que el servidor pudiese mantener dicha carga, se podría hablar de hasta un millón de transmisiones de audio simultáneas, convirtiendo esta en una gran opción para un posible servicio de voz gratuito y global.

Aparte de estas dos grandes líneas de desarrollo expuestas, es evidente la necesidad de dar a conocer esta aplicación a potenciales usuarios a través de publicidad en Internet y contactos con empresas, además de plantear un sistema de precios para conseguir que el desarrollo de esta aplicación sea económicamente viable y sostenible a largo plazo.

Bibliografía

[1] Zello Inc., «Zello | Tarifas,» [En línea]. Available: https://zello.com/es-es/pricing/. [Último acceso: 21 Junio 2024].

[2] TeamSpeak, «Licencias de MyTeamSpeak 3,» [En línea]. Available: https://www.teamspeak.com/es/features/licensing/. [Último acceso: 21 Junio 2024].

[3] D. Anderson, «How NAT traversal works,» Tailscale, 21 Agosto 2020. [En línea]. Available: https://tailscale.com/blog/how-nat-traversal-works. [Último acceso: 21 Junio 2024].

[4] R. Adeva, «¿Qué es Android?,» Adslzone, 2 Febrero 2023. [En línea]. Available: https://www.adslzone.net/reportajes/software/que-es-android/. [Último acceso: 20 Junio 2024].

[5] J. Callaham, «Google made its best acquisition nearly 17 years ago: Can you guess what it was?,» AndroidAuthority, 13 Mayo 2022. [En línea]. Available: https://www.androidauthority.com/google-android-acquisition-884194/. [Último acceso: 21 Junio 2024].

[6] Backlinko Team, «iPhone vs Android statistics,» Backlinko, 13 Marzo 2024. [En línea]. Available: https://backlinko.com/iphone-vs-android-statistics. [Último acceso: 20 Junio 2024].

[7] Google, «Modern Android Development,» Google, [En línea]. Available: https://developer.android.com/modern-android-development. [Último acceso: 20 Junio 2024].

[8] F. Lardinois, «Kotlin is now Google's preferred language for Android app development,» TechCrunch, 7 Mayo 2019. [En línea]. Available: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development. [Último acceso: 21 Junio 2024].

[9] Google, «Kotlin first,» Google, [En línea]. Available: https://developer.android.com/kotlin/first. [Último acceso: 20 Junio 2024].

[10] Kotlin, «Kotlin,» JetBrains, [En línea]. Available: https://kotlinlang.org. [Último acceso: 20 Junio 2024].

[11] Ktor, «Ktor,» JetBrains, [En línea]. Available: https://ktor.io/. [Último acceso: 20 Junio 2024].

[12] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey y I. Sutskever, «Robust Speech Recognition via Large-Scale Weak Supervision,» OpenAI, 21 Septiembre 2022. [En línea]. Available: https://cdn.openai.com/papers/whisper.pdf. [Último acceso: 21 Junio 2024].

[13] M. Schmidt, F. de Bont, S. Doehla y J. Kim, «RTP Payload Format for MPEG-4 Audio/Visual Streams,» IETF, Octubre 2011. [En línea]. Available: https://datatracker.ietf.org/doc/html/rfc6416. [Último acceso: 20 Junio 2024].

[14] M. Baugher, E. Carrara, D. A. McGrew, M. Naslund y K. Norrman, «The Secure Real-time Transport Protocol (SRTP),» IETF, Marzo 2004. [En línea]. Available: https://datatracker.ietf.org/doc/html/rfc3711. [Último acceso: 20 Junio 2024].

[15] WebRTC, «Comunicación en tiempo real para la Web,» Google, [En línea]. Available: https://webrtc.org/. [Último acceso: 20 Junio 2024].

[16] Cloudflare, «¿Qué es MPEG-DASH? | HLS vs. DASH,» Cloudflare, [En línea]. Available: https://www.cloudflare.com/es-es/learning/video/what-is-mpeg-dash/. [Último acceso: 20 Junio 2024].

[17] R. Belda, «fast-ll,» 23 Febrero 2024. [En línea]. Available: https://github.com/robelor/fast-ll. [Último acceso: 20 Junio 2024].

[18] Mozilla, «WebSockets,» Mozilla, [En línea]. Available: https://developer.mozilla.org/es/docs/Web/API/WebSockets_API. [Último acceso: 21 Junio 2024].

[19] I. Fette y A. Melnikov, «The WebSocket Protocol,» IETF, Diciembre 2011. [En línea]. Available: https://datatracker.ietf.org/doc/html/rfc6455. [Último acceso: 20 Junio 2024].

[20] Google, «Android Developers,» Google, [En línea]. Available: https://developer.android.com/. [Último acceso: 20 Junio 2024].

[21] Opus, «Opus Encoder,» [En línea]. Available: https://opus-codec.org/docs/opus_api-1.3.1/group__opus__encoder.html. [Último acceso: 20 Junio 2024].

[22] Oracle, «¿Qué es Docker?,» [En línea]. Available: https://www.oracle.com/mx/cloud/cloud-native/container-registry/what-is-docker. [Último acceso: 20 Junio 2024].

[23] R. David, «Codec 2 - Open source speech coding at 2400 bits/s and below,» [En línea]. Available: https://www.tapr.org/pdf/DCC2011-Codec2-VK5DGR.pdf. [Último acceso: 20 Junio 2024].

Glosario

ataques de *replay*
: Se trata de un ataque de reinyección, en el que un atacante intercepta un mensaje válido y lo retransmite para efectuar un ataque enmascarado., 10

callback
: Se trata de un concepto de programación que consiste en informar a una clase de que la ejecución de un bloque de código en otra clase ha terminado., 47, 70

cookie
: Son pequeños datos guardados en el navegador del cliente con el objetivo de superar la limitación impuesta por HTTP al no guardar información de la sesión del cliente., 55

IDE
: Entorno de desarrollo integrado. Es el software que asiste a los programadores con el desarrollo de código de forma eficiente mediante la recopilación de todas las herramientas necesarias para poder desarrollar, compilar si es necesario, y testear código en un lenguaje determinado., 7, 48, 65

JVM
: Son las siglas de Java Virtual Machine o, en español, “máquina virtual de Java”. Como su nombre indica, se trata de una máquina virtual que permite ejecutar instrucciones codificadas en el bytecode de Java. Esto permite que los programas compilados en Java se puedan ejecutar en cualquier máquina que tenga instalada la JVM., 7, 62

NAT
: Network Address Translation. Se trata de un método mediante el cual un router es capaz de mapear un espacio de direcciones privado a una única dirección IP pública., 3

serialización
: Consiste en el proceso mediante el que los datos que forman un objeto cambian a un formato distinto, con el objetivo de guardar ese objeto en un fichero o transmitirlo por la red., 15

TCP
: Transport Control Protocol. Se trata de un protocolo de la capa de transporte según el modelo de referencia OSI, orientado a conexión y en el que se realiza control de errores y de flujo mediante métodos de reconocimiento., 3, 8, 10, 11, 53

UDP
: User Datagram Protocol. Es otro protocolo situado en la capa de transporte. Al contrario que TCP, no es orientado a conexión, sino que los paquetes se envían al destinatario indicado, sin realizar ningún tipo de control o reconocimiento. Puede perder paquetes y no maneja errores en la transmisión., 3, 10, 17

Anexo 1. *Hardware* empleado en el desarrollo

A lo largo del desarrollo de la aplicación, se han empleado los siguientes dispositivos para desarrollo y test:

- Portátil Windows: Intel Core i7-12700H @2.30GHz + RTX 3050-Ti (4GB VRAM) + 16GB RAM con Android Studio e IntelliJ IDEA.
- PC Windows: Intel Core i7-9700K @3.60GHz + RTX 2070 (8GB VRAM) + 32 GB RAM con Android Studio e IntelliJ IDEA.
- 2x OnePlus 3T, 6 GB RAM
- Google Pixel 5, 6 GB RAM
- Samsung Galaxy S9, 4 GB RAM

Anexo 2. Diseño de un visualizador de FFT en tiempo real

Con el fin de representar el audio que se está reproduciendo en un terminal, se ha diseñado un visualizador de audio con la librería '*android.media.audiofx.visualizer*'. Empleando esta librería, se puede calcular la FFT en tiempo real de una determinada sesión de audio del *audioTrack* empleado en la reproducción.

Esto se puede conseguir con el *callback* '*setDataCaptureListener*', implementando la función '*onFftDataCapture*'. Esta función recibe un *ByteArray* de longitud igual a la mitad de la muestra de audio más uno, y representa, en muestras intercaladas, la parte real y la parte imaginaria de todos los valores de FFT en frecuencias desde 0 Hz hasta la mitad de la frecuencia de muestreo (en el caso del audio empleado, a 48 kHz, se representarán hasta los 24 kHz).

Cada vez que se reciben nuevas muestras de FFT, se realizará el módulo de estas y se emitirán en un *MutableStateFlow* del *Model*, al que se suscribirán los *ViewModels* que vayan a emplearlas en la representación.

En la Figura 69 se puede observar el código del *AudioPlayer* para visualizar las muestras.

```
 1. visualizer = audioTrack?.audioSessionId?.let {
 2.     Visualizer(it).apply {
 3.         captureSize = Visualizer.getCaptureSizeRange()[1]
 4.         setDataCaptureListener(object : Visualizer.OnDataCaptureListener {
 5.             override fun onFftDataCapture(
 6.                 visualizer: Visualizer,
 7.                 fft: ByteArray,
 8.                 samplingRate: Int
 9.             ) {
10.                 d { "FFT data captured" }
11.                 val fftValues = FloatArray(fft.size / 2)
12.                 for (i in fft.indices step 2) {
13.                     val real = fft[i]
14.                     val imaginary = fft[i + 1]
15.                     fftValues[i / 2] = hypot(real.toDouble(),
                                                 imaginary.toDouble()).toFloat()
16.                 }
17. 
18.                 model.sendFftData(fftValues)
19. 
20.             }
21.        }, Visualizer.getMaxCaptureRate() / 2, false, true)  // enable FFT data capture
22. 
23.         // start the visualizer
24.         enabled = true
25.     }
26. }
```

Figura 69. Creación del *Visualizer* para obtener las muestras de FFT del audio reproducido en tiempo real

Ahora que el *ViewModel* tiene acceso a un *Flow* con las muestras de la FFT, se pueden representar en la interfaz gráfica. Para realizarlo, en primera instancia, se recorta la cantidad de muestras para representar únicamente las frecuencias más bajas, que se son más frecuentes en el habla humana. Una vez recortado el espacio muestral, las magnitudes se normalizan con respecto al valor mayor y al menor. Posteriormente, se dividen las muestras en tantos grupos como barras se desea en la representación gráfica, y se obtiene un valor para cada barra realizando la media de los valores de las muestras de cada uno de los grupos.

Con la función *Composable* '*animateIntByState*' se puede animar el cambio de altura en un *box*, que será lo que represente el valor de las muestras de la FFT en la interfaz gráfica de forma animada y fluida.

Anexo 3. Relación del trabajo con los objetivos de desarrollo sostenible de la Agenda 2030

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No procede
ODS 1. Fin de la pobreza.				X
ODS 2. Hambre cero.				X
ODS 3. Salud y bienestar.				X
ODS 4. Educación de calidad.				X
ODS 5. Igualdad de género.				X
ODS 6. Agua limpia y saneamiento.				X
ODS 7. Energía asequible y no contaminante.				X
ODS 8. Trabajo decente y crecimiento económico.	X			
ODS 9. Industria, innovación e infraestructuras.	X			
ODS 10. Reducción de las desigualdades.				X
ODS 11. Ciudades y comunidades sostenibles.				X
ODS 12. Producción y consumo responsables.				X
ODS 13. Acción por el clima.				X
ODS 14. Vida submarina.				X
ODS 15. Vida de ecosistemas terrestres				X
ODS 16. Paz, justicia e instituciones sólidas.				X
ODS 17. Alianzas para lograr objetivos.				X

Descripción de la alineación del TFG con los ODS con un grado de relación más alto:

- **ODS 8. Trabajo decente y crecimiento económico:** La aplicación, con su bajo coste de mantenimiento y despliegue, puede ayudar al crecimiento económico de las empresas que la adopten, además de posibilitar la creación de una empresa que comercialice y continúe con el desarrollo de la idea.
- **ODS 9. Industria, innovación e infraestructura:** El proyecto desarrollado constituye una idea innovadora que podrá ser desplegada en una infraestructura empresarial para aportar comunicación en muchos sectores, entre ellos el de la industria.