Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Aplicación de notificación de malas prácticas de conducción

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Adrián Gilabert de Vargas

Tutor: Carlos David Martínez Hinarejos

2018/2019

Resumen

La aplicación de malas prácticas conduciendo está destinada a minimizar este tipo de prácticas. Para conseguir esto la aplicación permite a los usuarios crear una incidencia cada vez que observen a otro conductor cometiendo este tipo de prácticas. Las incidencias podrán ser creadas de dos formas distintas. La primera forma seria mediante el uso de la interfaz de la aplicación, es decir, rellenando un formulario. La segunda forma seria crear una incidencia mediante una orden de voz, de esta forma el usuario podría crear una incidencia mientras va conduciendo sin la necesidad de apartar la vista de la carretera. Para que estas incidencias tengan repercusión las autoridades dispondrán de otra aplicación donde se les mostrara un listado de los conductores que disponen de más incidencias asociadas, de esta forma las autoridades podrían tomar las medidas que fueran necesarias.

Palabras clave: IATROS, Android, FireBase, JavaFx, reconocimiento de voz, conducción.

Abstract

The application of bad practises driving is intended to minimize this type of practice. To achieve this, the application allows users to create an incident every time they observe another driver committing this type of practice. The incidents can be created in two different ways. The first way would be through the use of the application interface, that is, completing a form. The second way would be to create an incidence by means of a voice command, in this way the user could create an incident while driving without the need to take his eyes off the road. For these incidents to have an impact, the authorities will have another application where they will be shown a list of the drivers that have more associated incidents, in this way the authorities could take the necessary measures.

Keywords: IATROS, Android, FireBase, JavaFx, speech recogniser, drive.

Resum

L'aplicació de males pràctiques conduint està destinada a minimitzar aquest tipus de pràctiques. Per aconseguir aço l'aplicació permet als usuaris crear una incidència cada vegada que observen a un conductor cometent aquest tipus de pràctiques. Les incidències poden ser creades de dues maneres diferents. La primera manera seria mitjançant l'ús de la interfaç de l'aplicació, és a dir, omplint un formulari. La segona forma seria crear una incidència mitjançant una ordre de veu, d'aquesta manera l'usuari podria crear una incidència mentres va conduint sense la necessitat d'apartar la vista de la carretera. Perquè aquestes incidències tinguin repercussió les autoritats disposaran d'una altra aplicació on se'ls mostrés un llistat dels conductors que disposen de més incidències associades, d'aquesta manera les autoritats podrien prendre les mesures que fossin necessàries.

Paraules clau: IATROS, Android, FireBase, JavaFx, reconeixement de veu, conducció.

Tabla de contenidos

Contenido

1. Introducción

Las malas prácticas conduciendo son un problema que se lleva intentando solventar prácticamente desde los inicios de la automoción, pues ha provocado un gran número de accidentes, muertes y pérdidas económicas, además de llegar al punto de considerar la conducción como una actividad de alto riesgo.

El problema radica principalmente en la mala educación de algunos conductores, que no están capacitados para respetar las normas de circulación ni a los demás usuarios de la vía. Por lo tanto, lo que se pretende solventar en este proyecto es poder decrementar este tipo de conductores sin castigar a aquellos conductores que solo cometen alguna irregularidad de manera puntual.

1.1 Motivaciones

La motivación de este proyecto es poder disfrutar de la conducción sin la necesidad de sufrir a aquellos conductores que no conocen o simplemente no procuran respetar las normas de circulación.

Estos conductores entorpecen, enfadan y frustran a los demás usuarios, además de que pueden provocar accidentes. En este proyecto se intenta proporcionar una solución facilitando a todos los conductores una herramienta colaborativa que sea fácil y sencilla de utilizar para poder informar a las autoridades de manera anónima sobre aquellos usuarios que cometen infracciones de manera asidua.

1.2 Objetivos

Los objetivos a cubrir en el desarrollo de este trabajo son:

- Mejorar la Comunidad Vial.
- Proporcionar a los usuarios una herramienta para avisar de cualquier infracción.
- Mantener a las autoridades informadas de los infractores habituales.
- Disminuir el número de infractores.
- Disminuir el número de accidentes.

1.3 Impacto Esperado

Para el estudio del impacto esperado del proyecto primero se ha analizado los distintos tipos de usuarios que pueden verse relacionados con la aplicación.

Infractores: Se espera que los usuarios informen sobre aquellos usuarios que realizan infracciones habitualmente; estos podrán ser castigados por las autoridades. De esta forma, se pretende que estos conductores dejen de cometer infracciones.

Autoridades: En cuanto a las autoridades, se espera que se produzca una mejora en la productividad al poder tener constancia de los conductores más problemáticos de la vía; de esta forma, las autoridades podrán mantener vigilados directamente a estos conductores.

Conductores: Al disminuir el número de infractores se espera que se produzca una mejora en la experiencia de conducción, gracias a la disminución de situaciones de peligro y al tráfico que provocan estos conductores.

Otro punto importante que cabe destacar, aparte de los usuarios implicados, es el impacto de la aplicación en el número de accidentes anuales, ya que gracias a esta aplicación se espera una disminución de estos.

Tabla I: Accidentes producidos en 2017

	Motocicleta	Turismo	Furgoneta	Camión hasta 3.500 kg	Camión más 3.500 kg	Autobús
Infracciones de velocidad	**7.061**	**44.577**	**5.071**	**801**	**3.533**	**288**
No respetar señal de STOP	14	1.056	141	14	46	5
No respetar paso para peatones	6	143	16	2	1	0
No respetar otra regulación de prioridad	118	1.785	232	32	199	8
Circular en sentido contrario o por lugar prohibido	31	97	12	2	5	0
Invadir parcialmente el sentido contrario	147	1.330	140	29	68	10
Adelantar antirreglamentariamente	135	316	38	7	24	1
No mantener el intervalo de seguridad	510	4.433	612	86	306	38

Como se puede observar en la tabla I obtenida de la DGT[1], se puede observar que el número de accidentes producidos por infracciones a la hora de conducir es muy alto. Con la aplicación que se ha construido no se espera reducir el número de estos accidentes a cero, ya que sería imposible, pero sí que se pretende reducir al mínimo posible.

[1] Estadísticas de la DGT: http://www.dgt.es/es/seguridad-vial/estadisticas-e-indicadores/accidentes-30dias/tablas-estadisticas/

2. Tecnologías Usadas

En este capítulo de la memoria se tratará de listar y explicar lo más detallado posible todas las herramientas, servicios, lenguajes… empleados en la creación del trabajo.

2.1 Java

Java es un lenguaje de programación propietario de la empresa Oracle. Es un lenguaje orientado a objetos y su sintaxis deriva en gran medida de C y C++. Una de las principales cualidades por la que se ha elegido este lenguaje es que, gracias a su forma de compilación en ByteCode, puede ejecutarse en cualquier máquina virtual, en cualquier dispositivo sin importar la arquitectura.

Gracias a las ventajas de este lenguaje, mencionadas anteriormente, se ha podido usar para desarrollar tanto la aplicación móvil como la aplicación para el servidor. Esto ha sido de gran utilidad, ya que ha permitido que las dos aplicaciones compartan varias clases modelo.

2.2 Tecnologías empleadas en el cliente

2.2.1 Android

Android es un sistema operativo de código abierto basado en el *kernel* de Linux. Está diseñado principalmente para dispositivos con pantallas táctiles, como móviles o tabletas, pero también está presente en algunas televisiones con AndroidTV.

Una de las ventajas principales de este sistema respecto a otros de la competencia, como IOS, es que al ser de código abierto está presente en una gran variedad de dispositivos y de multitud de fabricantes posibles.

Se ha elegido este sistema para construir la aplicación móvil ya que, gracias a las ventajas mencionadas, actualmente es la plataforma mayoritaria en cuanto a dispositivos móviles se refiere [8] y también dispone de una gran comunidad de desarrollo.

2.2.2 Android Studio

Android Studio es el entorno de desarrollo oficial de Android. Está basado en el entorno de desarrollo IntelliJ IDEA y soporta dos lenguajes de programación: Java y Kotlin.

Dado que es el entorno de desarrollo oficial, es el que más herramientas y ayudas proporciona para la creación de aplicaciones tanto en Java como en Kotlin, que son los dos lenguajes soportados por este entorno. En este proyecto se ha escogido Java; esto ha permitido que la aplicación móvil y el servidor Linux compartan código al poder usar los dos Java.

2.2.3 FireBase

FireBase es una plataforma en la nube de Google que dispone de una gran cantidad de herramientas multiplataforma y servicios útiles para el desarrollo de aplicaciones.

Las herramientas de FireBase que se han usado en el cliente son:

Authentication

FireBaseAuth es un servicio que permite autentificar a los usuarios utilizando únicamente el código del lado del cliente. Además, proporciona una gran integración para autentificarse mediante otros servicios como Facebook, Google, Github, etc. Incluye un sistema de administración de usuarios mediante el cual el desarrollador puede gestionar mediante su interfaz web.

Este servicio resulta muy útil, ya que permite la creación de un sistema de autentificación de una forma bastante simple sin la necesidad de preocuparse por cualidades como la seguridad o la implementación de reglas que impidan usuarios repetidos o falsos, ya que proporciona estos métodos ya incorporados. También integra métodos para recuperar la cuenta mediante email o SMS en caso de olvidarse de la contraseña.

En el proyecto se ha usado este servicio para la autentificación y para poder gestionar los usuarios, debido a las buenas cualidades que se han comentado.

Cloud FireStore

Cloud FireStore es un servicio que proviene de *google cloud plataform* adaptado a FireBase. Es una base de datos NoSql ubicada en la nube a la que pueden acceder las aplicaciones que tengan integrado FireBase.

FireStore almacena los datos en documentos que contienen campos a los que se asignan valores; estos campos pueden ser de muchos tipos, desde simples cadenas de caracteres a objetos anidados complejos.

Los documentos se almacenan en colecciones, que son contenedores para los documentos. Se pueden realizar consultas personalizadas a las colecciones para buscar los documentos deseados como en cualquier base de datos.

Mediante este servicio se almacenan todas las incidencias provenientes de los usuarios y se almacenan todos los tipos de malas prácticas. Se ha usado este servicio y no cualquier otro ya que, al igual que con FireBaseAuth, proporciona una serie de mecanismos de seguridad muy superiores a los que se podrían crear usando un servidor de base de datos más rudimentario.

Además, proporciona una serie de funcionalidades muy interesantes para el proyecto, como la capacidad de poder trabajar *offline* usando la memoria cache hasta conectarse con el servidor y poder actualizar los datos. Otra funcionalidad es la gran flexibilidad a la hora de transformar las clases de Java en documentos.

Cloud Storage

Este es otro servicio de FireBase que permite el almacenamiento de todo tipo de archivos en la nube. La cualidad más importante de este servicio, y por la que se ha elegido para el proyecto, es que incorpora mecanismos para la carga y descarga de estos archivos de forma segura.

Cloud Storage se ha usado para el almacenamiento en la nube de los audios generados por los usuarios.

2.3 Tecnologías empleadas en el servidor

2.3.1 JavaFx

JavaFx es un conjunto de paquetes gráficos que permiten dotar de una interfaz gráfica a aplicaciones desarrolladas en Java que operan en una gran cantidad de plataformas.

JavaFx usa el modelo MVC, es decir, *model-view-controller,* ya que para la creación de la interfaz gráfica usa los documentos Fxml (que son creados y editados mediante 2.3.3 SceneBuilder). Cada documento Fxml necesita una clase Java para controlarlo, es decir, un controlador. Y, por último, la clase modelo, que es la que contiene la información a mostrar; por ejemplo, en este caso sería la clase Incidence (ver cap.3), que son las incidencias que los usuarios crean.

Se ha utilizado JavaFx para la creación de la aplicación servidor; mediante este entorno se ha podido dotar de una interfaz gráfica muy simple con la que se pueden observar todos los datos que recibe el servidor FireBase.

2.3.2 IntelliJ IDEA

IntelliJ IDEA es un entorno de desarrollo integrado (IDE) para el desarrollo de programas informáticos. Es desarrollado por JetBrains (anteriormente conocido como IntelliJ), y está disponible en dos ediciones: edición para la comunidad y edición comercial. Para la creación del proyecto se ha usado la versión de la comunidad. IntelliJ IDEA no está basada en Eclipse, como MyEclipse u Oracle Enterprise Pack para Eclipse.

Se ha utilizado este entorno de desarrollo ya que posee una gran integración con JavaFx sin la necesidad de tener que instalar *plug-ins* ni añadir ninguna configuración adicional, como sucede con Eclipse. Otro punto muy importante que motiva el por qué se ha elegido este entorno es que, al ser el entorno muy parecido al de Android Studio (Android Studio está basado en IntelliJ, como se ha comentado en el apartado de Android Studio), no ha hecho falta ningún periodo de adaptación.

2.3.3 SceneBuilder

SceneBuilder es un editor de interfaces de código libre que trabaja con JavaFx. Permite a los usuarios crear una interfaz para sus aplicaciones simplemente arrastrando los elementos que quieren poner, sin la necesidad de tener que programar nada. Como se ha comentado antes, usa los documentos Fxml.

2.3.4 Maven

Maven es una herramienta que permite gestionar y construir proyectos Java. Tiene un modelo de construcción simple basado en XML. Para describir el proyecto de software a construir, Maven utiliza un *Project Object Model*, donde se añaden todas las dependencias, módulos y componentes externos para la creación del proyecto.

Uno de los puntos fuertes de Maven es que tiene conexión a internet y dispone de una gran cantidad de repositorios, por lo que para añadir una dependencia basta con añadir la dirección de la versión del paquete, y él solo se encarga de descargar todo lo necesario y compilarlo.

Esta herramienta ha sido de gran utilidad para poder añadir las bibliotecas de FireBase sin mayor complicación.

2.3.5 FireBase Admin

FireBase Admin es un SDK disponible en multitud de plataformas que permite realizar operaciones e interactuar con todas las herramientas y servicios de FireBase en una gran cantidad de plataformas, desde lenguajes de páginas web (como Php o Javascript) a otros más convencionales como Java. En este caso se ha usado en el servidor para recuperar las incidencias de Cloud FireStore y procesar los audios de Cloud Storage.

2.3.6 iATROS

Para el reconocimiento de voz se ha utilizado el sistema de reconocimiento de voz iATROS. Este sistema se desarrolló en el centro de investigación Pattern Recognition and Human Language Technologies (PRHLT) de la Universidad Politécnica de Valencia. Este sistema de reconocimiento de voz está basado en la decodificación por el algoritmo de Viterbi [2] de una secuencia de vectores de características sobre una red de estados finitos formada por tres tipos de modelos:

- Modelo morfológico, también llamado modelo acústico que es el utilizado en esta aplicación; son modelos ocultos de Markov continuos (CD HMM) [6], es decir, que utilizan mixturas de distribuciones gaussianas para la probabilidad de emisión; permiten relacionar las unidades de sonido básicas (fonemas) con los vectores obtenidos del audio.
- Modelo léxico, son modelos de estados finitos, que nos dicen cómo se combinan los sonidos para formar palabras.

- Modelo de lenguaje: pueden ser modelos basados en frecuencias (n-gramas) [4] o modelos de estados finitos; en cualquier caso, modelan cómo se unen las palabras para formar frases del dominio del problema.

3. Análisis de requisitos

En este apartado de la memoria se tratará de explicar todo el proceso relacionado con la especificación de requisitos, desde identificar a los *stakeholders* (personas implicadas) a definir los límites del sistema.

3.1 Actores Implicados

En el análisis de impacto esperado ya se analizaron algunos tipos de usuarios, pero en este apartado se va a realizar un análisis más en profundidad de todos los actores que puedan haber implicados [9]. La tabla II resume los actores detectados.

Tabla II: Actores implicados

Nombre y/o Rol	Usuario Directo	Intereses
Usuario	Sí	Agilidad para informar sobre conductores.
Representante de alguna autoridad	Sí	Notificación adecuada de aquellos conductores que cometen infracciones

3.2 Requisitos Funcionales

Los requisitos funcionales son las características que permiten definir el sistema tanto por sus cualidades como por las funcionalidades que va a proporcionar. Los requisitos no funcionales estarán presentes en los casos de usos especificados más adelante, donde se analiza cómo los actores interactuarán con estos [10]. Estos requisitos se muestran a continuación en una seria de cuadros, con su descripción, entradas, salidas y proceso asociado.

3.2.1 Aplicación Móvil

Crear una cuenta de usuario	
Descripción	Permite a un usuario crearse una cuenta para empezar a usar la aplicación.
Entrada	• Correo electrónico • Contraseña
Proceso	El usuario, desde la ventana inicial, seleccionará “crear cuenta” e introducirá las entradas mencionadas.

Salida	• Un mensaje de error si el correo electrónico del usuario ya existe • Un mensaje de error si no se han introducido los datos • Un mensaje de error si el correo electrónico no es válido • Una nueva entrada en la base de datos si se ha registrado correctamente

Requisito 1

Iniciar sesión	
Descripción	Permite a un usuario registrado iniciar sesión.
Entrada	• Correo electrónico • Contraseña
Proceso	El usuario desde la ventana inicial introducirá las entradas mencionadas.
Salida	• Un mensaje de error si la cuenta no existe • Un mensaje de error si falta algún campo por rellenar • Un mensaje de error si la contraseña no es correcta • Se redigirá a la ventana principal de la aplicación

Requisito 2

Crear incidencia	
Descripción	Permite a un usuario crear una incidencia.
Entrada	• Mala Práctica • Matrícula • Descripción -Opcional-
Proceso	El usuario, desde la ventana principal, introducirá los datos necesarios.

Salida	• Un mensaje de error si no se han introducido todos los datos necesarios. • Un mensaje de error si los datos no son válidos. • Una nueva entrada en la base de datos si se ha registrado correctamente

Requisito 3

Crear incidencia con orden de voz	
Descripción	Permite a un usuario crear una incidencia mientras va conduciendo sin la necesidad de utilizar la interfaz.
Entrada	• Archivo de voz con la matrícula • Archivo de voz con la descripción
Proceso	El usuario, desde la ventana inicial, pulsará un botón para iniciar la grabación.
Salida	• Una nueva entrada en la base de datos si se ha registrado correctamente

Requisito 4

Listar incidencias	
Descripción	Permite a un usuario observar todas las incidencias creadas por él mismo.
Entrada	• Este requisito no necesita ninguna entrada
Proceso	El usuario, desde la ventana principal, pulsará un botón para mostrar todo el historial de incidencias.
Salida	• Una lista con todas las incidencias

Requisito 5

Modificar incidencia	
Descripción	El usuario podrá modificar una incidencia una vez ya creada en caso de querer corregir algún dato o añadir más información.
Entrada	• La incidencia que se desea modificar • Matrícula • Fecha • Mala Práctica • Descripción -Opcional-
Proceso	El usuario, desde la ventana de listar incidencias, seleccionará la incidencia que se desea modificar.
Salida	• Un mensaje de error si no se han introducido los datos • Un mensaje de error si los datos no son válidos • Una modificación en la base de datos de la entrada seleccionada si no sucede ningún error

Requisito 6

Cerrar sesión	
Descripción	El usuario podrá cerrar sesión en caso de querer cambiar de cuenta.
Entrada	• Este requisito no necesita ninguna entrada
Proceso	El usuario, desde la ventana inicial, seleccionará el botón de cerrar sesión.
Salida	• Se redigirá a la ventana inicial para crear una cuenta nueva o iniciar sesión

Requisito 7

3.2.2 Servidor

Listar incidentes agrupados por matrícula	
Descripción	Permite al representante de las autoridades observar en tiempo real las matrículas con más incidentes relacionados.
Entrada	• Este requisito no necesita ninguna entrada
Proceso *Requisito 8*	El usuario, desde la ventana principal, ya podrá observar la lista.
Salida	• Lista de matrículas con número de incidentes y el último incidente producido

Listar historial de incidentes de una matrícula	
Descripción	Permite al representante de las autoridades observar en tiempo real los incidentes producidos y que se van produciendo de una matrícula determinada.
Entrada	• Matrícula
Proceso	El usuario, desde la ventana principal, seleccionará una matrícula y pulsará sobre el botón de historial.
Salida	• Lista de incidentes relacionados con la matrícula seleccionada

Requisito 9

Listar todos los incidentes	
Descripción	Permite al representante de las autoridades observar en tiempo real todos los incidentes producidos y que se van produciendo.
Entrada	• Este requisito no necesita ninguna entrada

Proceso	El usuario, desde la ventana principal, pulsará sobre un botón para mostrar todos los incidentes.
Salida	• Lista de todos los incidentes en tiempo real

Requisito 10

Modificar incidencia	
Descripción	El usuario podrá modificar una incidencia una vez ya creada en caso de querer corregir algún dato o añadir más información.
Entrada	• La incidencia que se desea modificar • Matrícula • Fecha • Mala Práctica • Descripción -Opcional-
Proceso	El usuario, desde cualquier ventana con un listado de incidencias, podrá seleccionar una, y pulsar sobre el botón de editar.
Salida	• Un mensaje de error si no se han introducido los datos • Un mensaje de error si los datos no son válidos • Una modificación en la base de datos de la entrada seleccionada si no sucede ningún error

Requisito 11

3.3 Requisitos no funcionales

Los requisitos no funcionales, a diferencia de los funcionales, estos se centran más en la implementación y las cualidades que debe cumplir la solución, es decir, define las características de funcionamiento [10]. A continuación, se listan estos requisitos por cada componente del sistema.

Aplicación móvil

- Interfaz fácil y sencilla basada en Material Design.
- Tiempo de aprendizaje para el uso de la aplicación lo mínimo posible.
- Botón para iniciar la grabación accesible y reconocible.
- El usuario solo podrá acceder al historial de sus incidencias.
- El tiempo para realizar alguna operación sobre la base de datos no debe superar los 5 segundos.

Servidor

- El sistema debe proporcionar una interfaz bien formada.
- Debe poder procesar más de 10 audios por minuto.
- El sistema debe de estar disponible más del 99% del tiempo.
- El tiempo de aprendizaje debe de ser menor a 2 minutos.
- El tiempo para realizar alguna operación sobre la base de datos no debe superar los 5 segundos.

3.4 Casos de uso

Los casos de uso son muy importantes para el desarrollo de una nueva aplicación de software, ya que especifica cómo los actores van a interactuar con el sistema [9].

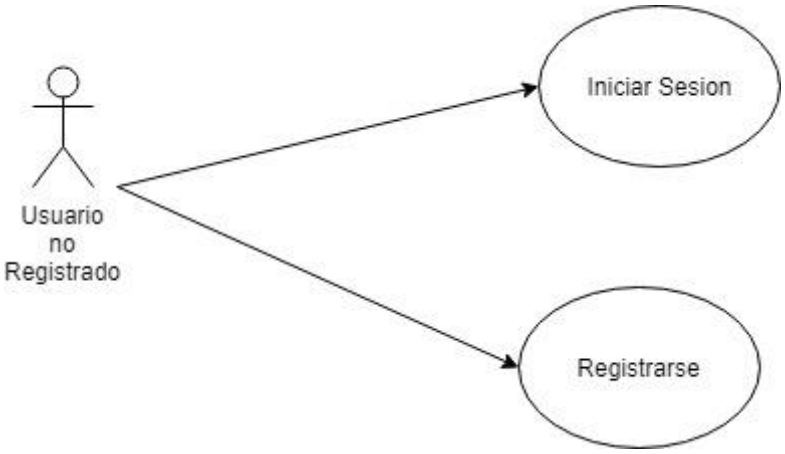

Figura I: Caso de uso de un usuario no registrado

El caso de uso presentado en la figura I muestra cómo un usuario no registrado puede realizar solo dos acciones con el sistema, “Registrarse” o “Iniciar Sesión”. Este caso de uso cubre los requisitos funcionales 1 y 2 de la aplicación móvil.

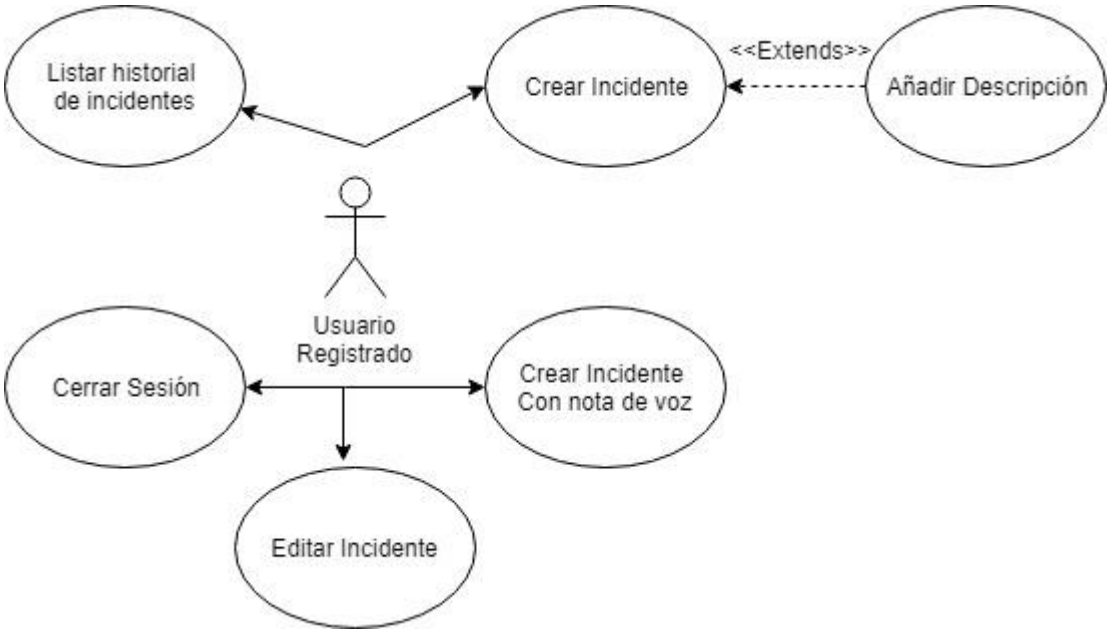

Figura II: Caso de uso de un usuario registrado

El caso de uso presentado en la figura II muestra todas las acciones que puede realizar un usuario de la aplicación con la precondición de que éste ya se haya registrado e

iniciado sesión. Este caso de uso cubre los requisititos funcionales 3, 4, 5, 6, 7 y 8 de la aplicación móvil.

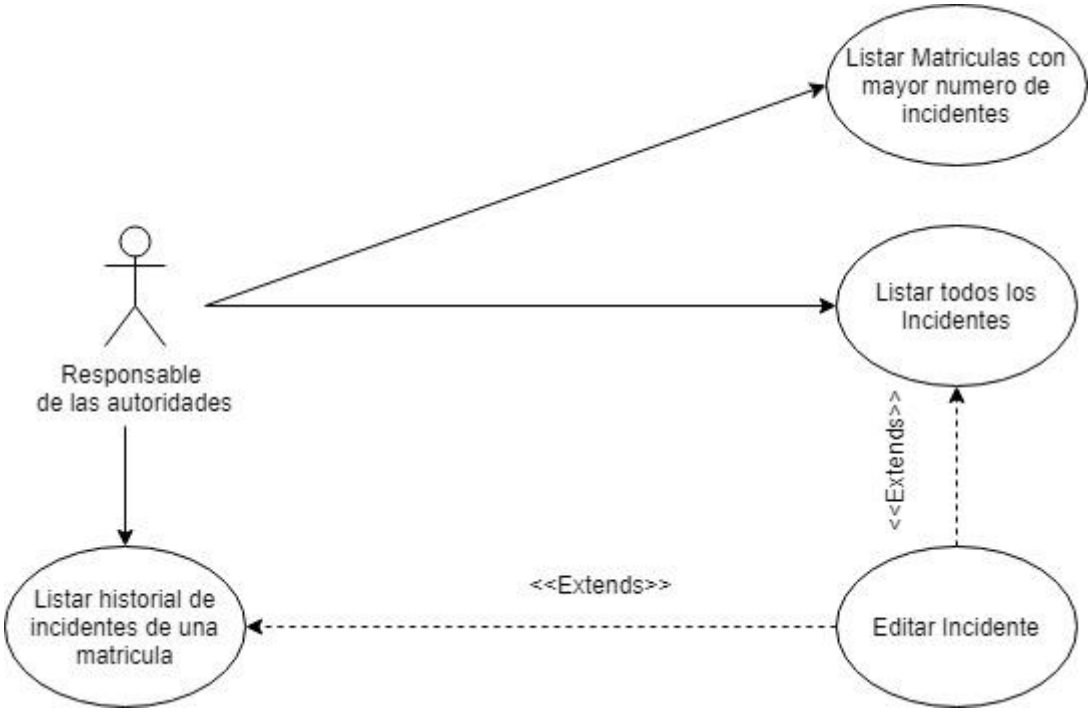

Figura III: Caso de uso del responsable de las autoridades

El caso de uso presentado en la figura III es el que corresponde a la aplicación del servidor; representa todos los requisitos funcionales de la aplicación del servidor.

3.5 Modelo de dominio

El modelo de dominio es un esquema creado mediante UML que se usa para clarificar la diferente terminología especifica del proyecto [9]. En este caso el modelo de domino es muy simple, ya que no dispone de un gran vocabulario especifico, y se presenta en la figura IV.

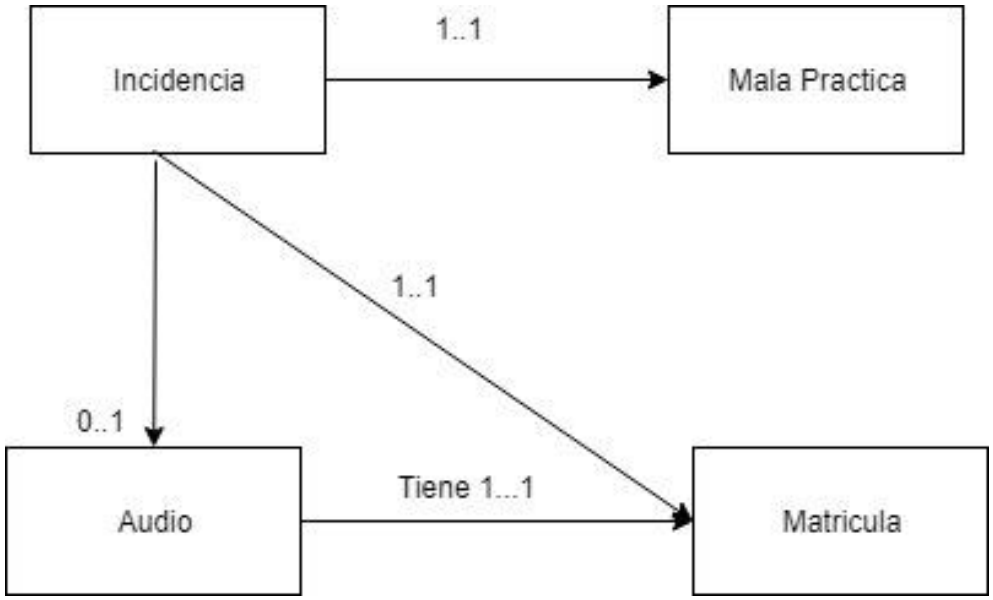

Figura IV: Modelo de dominio

Los elementos del modelo de dominio son:

Mala Práctica: Se refiere a la infracción que ha cometido el infractor; un ejemplo de mala práctica puede ser cambiar de carril sin el uso de los intermitentes.

Matricula: Se trata de la matrícula que corresponde al coche del conductor que ha cometido la infracción.

Audio: El audio se refiere a una nota de voz generada por un usuario donde indica la mala práctica, la matrícula y una descripción de lo sucedido.

Incidencia: Este es el termino más importante del proyecto, ya que es donde se recoge toda la información de lo sucedido cuando un usuario informa sobre otro conductor. Una incidencia se compone principalmente de la matrícula del infractor, la mala práctica cometida y puede contener, o no, un audio.

4. Presupuesto

En esta sección se va a desglosar todo el coste relacionado con el trabajo, desde el coste material, es decir, como son los equipos que se han necesitado, al coste personal en cuanto a las horas dedicadas al proyecto.

Material

El coste material de este proyecto no ha sido muy sustancial, ya que como se comentará en el capítulo 6 de implementación se ha usado una placa base Asrock J4205-ITX, la cual es casi un equipo completo, muy básico y de bajo consumo. A continuación, en la tabla 3 se mostrará el desglose completo del equipo.

Tabla III: Precio del material

Nombre	Cantidad	Precio	TOTAL
Placa base Asrock J4205-ITX	1	50€	50€
Fuente de alimentación Tacens Anima APII500	1	15€	15€
Kingston KVR13S9S8/4 – Memoria RAM de 4 GB	2	21€	42€
Disco duro interno Seagate BarraCuda de 500 GB	1	30€	30€
			137€

En cuanto al coste del equipo usado para el desarrollo de la aplicación móvil y servidor, no se ha tenido en cuenta, ya que se ha supuesto que es una herramienta de trabajo que todo el mundo posee. El dispositivo móvil tampoco se ha tenido en cuenta, ya que de la misma forma también se da por supuesto que todo el mundo posee uno.

Personal

Para la estimación del coste en personal se puede observar la tabla 4.

Tabla IV: Coste del personal

Nombre	Cantidad	Precio	TOTAL
Analista	20 horas	13€	260€
Desarrollador	300 horas	13€	3900€

En conclusión, se puede estimar el coste del proyecto en unos 4160€.

5. Diseño

5.1 Arquitectura del proyecto

La arquitectura de este proyecto está basada en la comúnmente conocida como Cliente-Servidor donde una aplicación (Cliente) solicita recursos a otra (Servidor), pero con una ligera variación. En esta variación hay dos tipos de servidores, el que almacena los datos para el cliente y el que procesa los datos para el cliente. Esta arquitectura se denomina arquitectura de tres capas [13] y se muestra en la figura V.

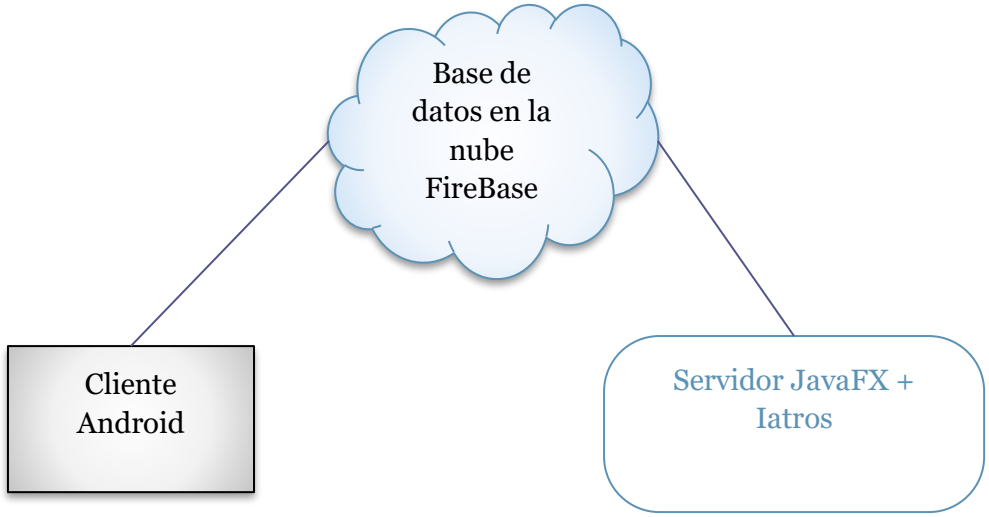

Figura V: Arquitectura del proyecto

Este diseño resulta muy útil para esta aplicación, ya que el servidor que recibe toda la carga y necesita procesar las solicitudes de forma rápida es el servidor de FireBase, el cual se escala automáticamente ante cualquier posible aglomeración de peticiones. De esta forma se garantiza que el servicio siga activo de forma ininterrumpida ante cualquier circunstancia.

Mientras el primer servidor se encarga de recibir todo el conjunto de peticiones, el segundo servidor se encarga de procesar los audios poco a poco, sin ningún requisito de tiempo; de esta forma, permite que el segundo servidor sea implementado en una máquina de pocos recursos.

5.2 Diagrama de Clases

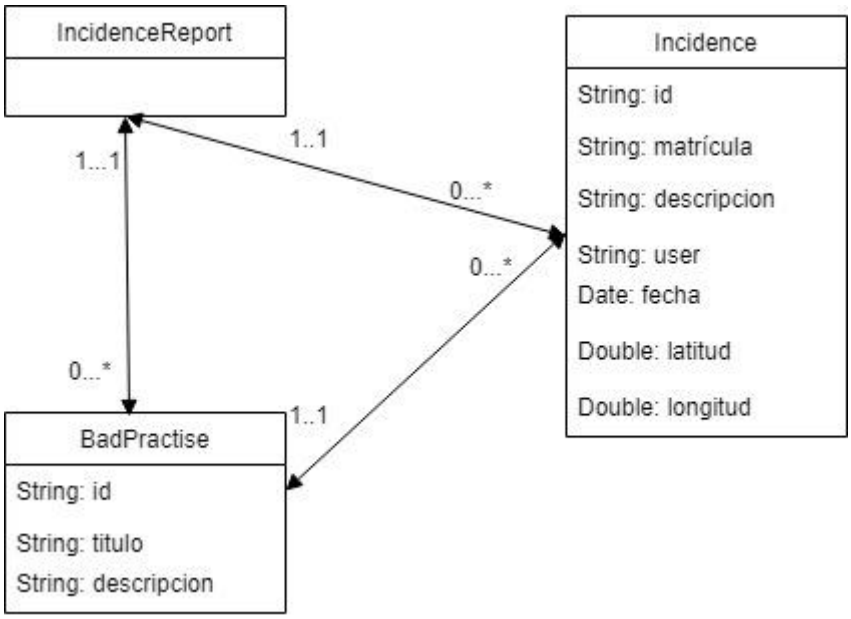

Figura VI: Diagrama de clases

El diagrama de clases [9], como se puede observar en la figura VI, es muy sencillo; tan solo se necesitan tres clases modelo. Este esquema se usa tanto en la aplicación móvil como en el servidor de FireBase.

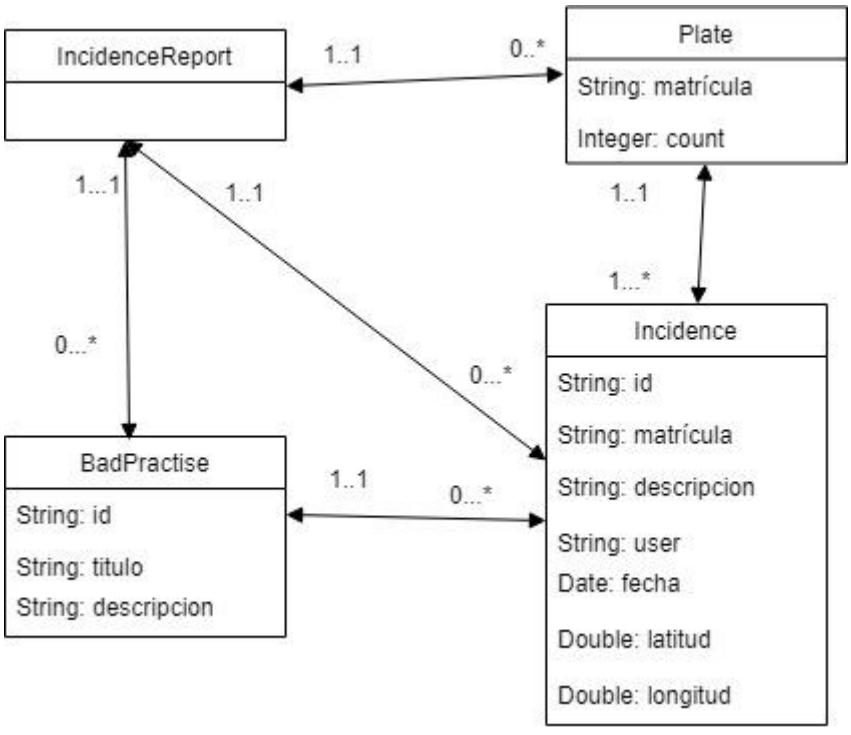

Figura VII: Diagrama de clases del servidor

En cuanto al diagrama de clases de aplicación del servidor de JavaFx, se ha usado el diagrama de la figura VII. Como se puede observar hay una clase adicional llamada *Plates*, esta se ha usado, para poder mostrar las incidencias agrupadas por matrículas.

5.3 Diagramas de secuencia

Mediante un diagrama de secuencia se muestra la interacción de los diferentes objetos a lo largo del tiempo y se modela para cada caso de uso; esto resulta útil para complementar el diagrama de clases, ya que mediante el diagrama de secuencia se muestra cómo los objetos del diagrama de clases se relacionan a lo largo del tiempo. Además, el diagrama de secuencia proporciona detalles sobre la implementación [14].

Para desarrollar un diagrama de secuencia se suele examinar el caso de uso que describe la acción que se quiere describir en el diagrama, y se determina qué objetos son necesarios. Un diagrama de secuencia muestra los objetos que intervienen en el escenario con líneas discontinuas verticales y los mensajes pasados entre los objetos como flechas horizontales [11].

5.3.1 Aplicación Android

En este apartado se mostrarán los diferentes diagramas de secuencia que reflejan cada caso de uso de la aplicación móvil.

5.3.1.1 Iniciar sesión

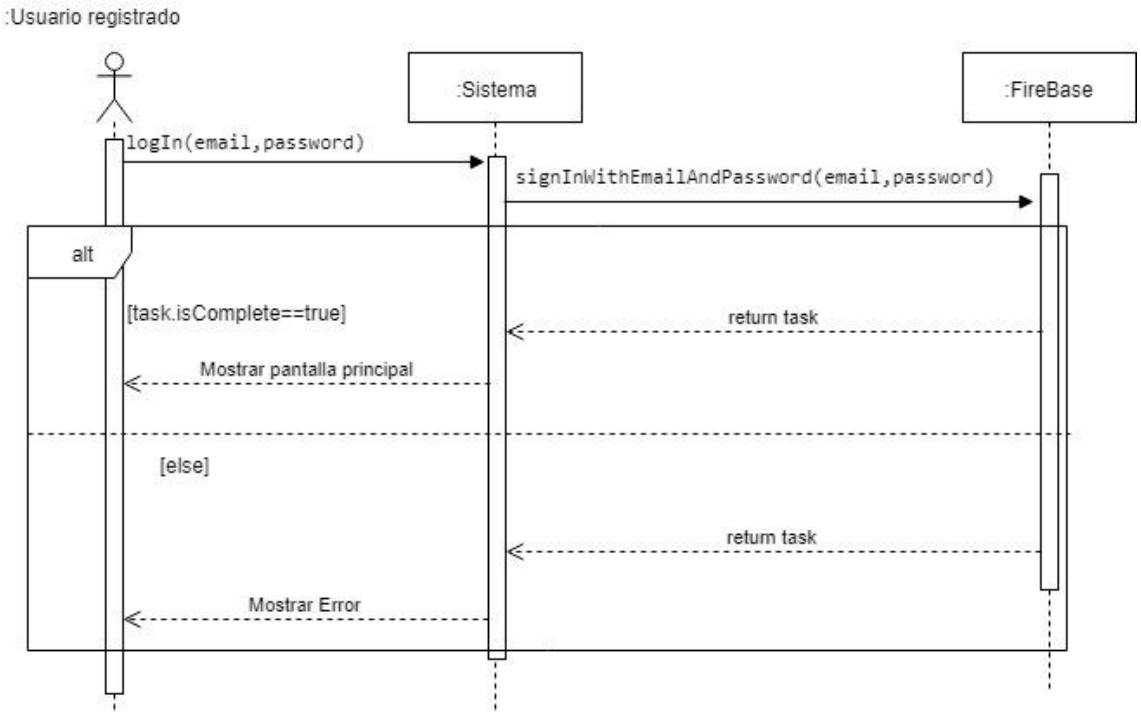

Figura VIII: Diagrama de secuencia de iniciar sesión

En la figura número 8 se puede observar el diagrama de secuencia del caso de uso de iniciar sesión; el funcionamiento de este sería el siguiente:

Primero, un usuario que ya dispone de una cuenta, introduce sus datos en el formulario y pulsa el botón de iniciar sesión; cuando el usuario realiza esta acción automáticamente se lanza el método `logIn(email,password);` dentro de este método se comprueban los datos y se llama a la clase FireBase mediante el método `signInWithEmailAndPassword(email,password)`. Una vez se ha hecho esto, FireBase comprueba en su base de datos si el usuario con ese correo y esa contraseña existe; si es correcto devuelve un objeto de la clase *Task* que describe que la tarea se ha realizado satisfactoriamente y el sistema mostrará la pantalla principal. En caso de que el usuario no exista o se haya producido algún error, devolverá una tarea con un error y se mostrará un mensaje de error por pantalla.

5.3.1.2 Registrar usuario

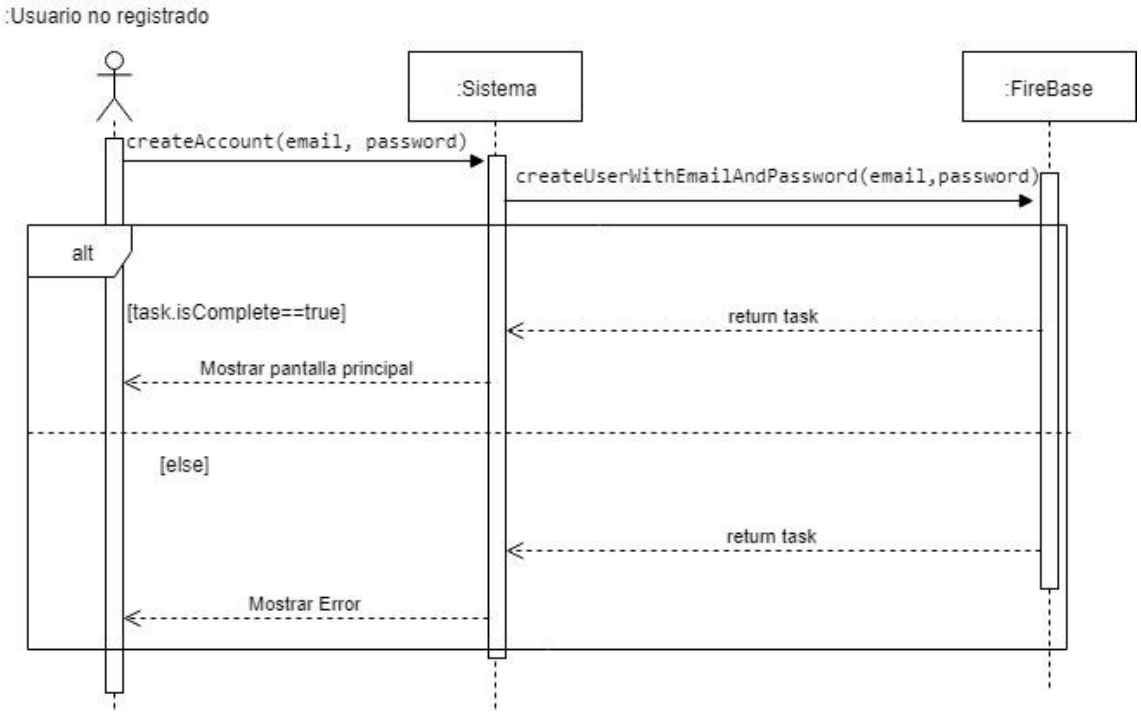

Figura IX: Diagrama de secuencia de registrar usuario

En la figura 9 se muestra el diagrama de secuencia correspondiente al caso de uso de registrar un usuario nuevo; cómo se puede observar, el funcionamiento es prácticamente idéntico al diagrama de secuencia de iniciar sesión.

En primer lugar, el usuario rellena sus datos en un formulario que se muestra en la interfaz; cuando el usuario lo ha completado y pulsa sobre el botón de crear cuenta se lanza automáticamente el método `createAccount(email,password);` este método comprueba los datos y lanza el método de FireBase `createUserwithEmailAndPassword( email,password).`

Finalmente, igual que en el caso anterior de iniciar sesión, FireBase comprueba que todo sea correcto, crea el usuario devolviendo un objeto de la clase *Task* que describa que todo ha sido correcto y el sistema mostrará la pantalla principal; en caso contrario se mostrará un mensaje de error.

5.3.1.3 Crear incidencia

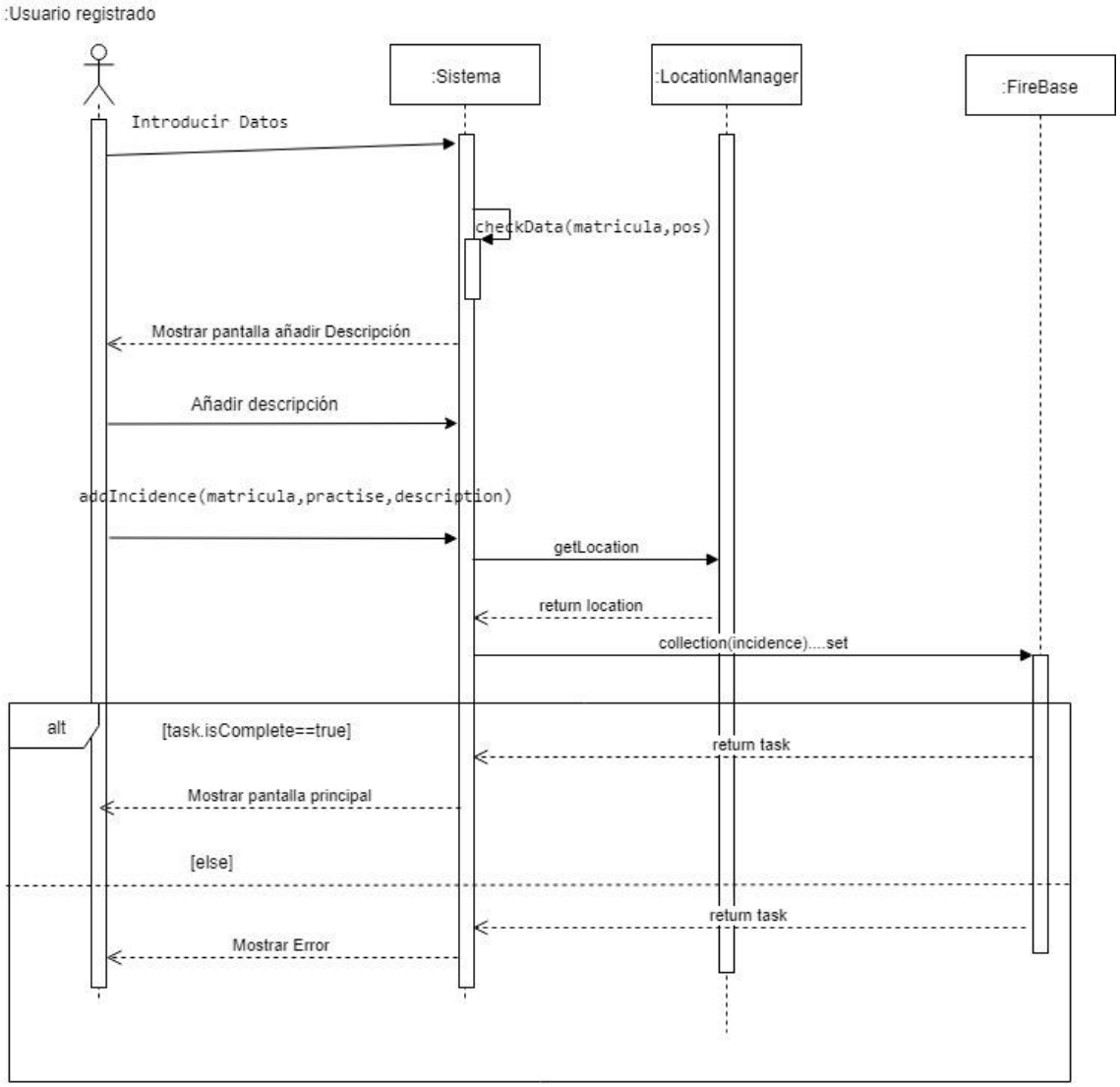

Figura X: Diagrama de secuencia de crear incidencia

En la figura 10 se puede observar la representación del caso de uso de crear incidencia en diagrama de secuencia. En este caso el procedimiento es similar a los anteriores, pero más complejo, ya que se ven más objetos involucrados.

En primer lugar, en este diagrama se da por precondición que el usuario esté en la ventana principal; en la ventana principal es donde se muestra el formulario para introducir los datos de una nueva incidencia, de forma que el primer paso que hará el usuario es añadir estos datos y pulsar el botón siguiente.

En el siguiente paso el sistema comprobara estos datos; si son correctos el sistema pasará a una nueva ventana donde el usuario podrá introducir la descripción o no, ya que es opcional; una vez pulse sobre el botón siguiente se lanzará automáticamente el método `addIncidence`. El primer paso que realiza este método es obtener la localización de *LocationManager* para almacenarlo en la incidencia. *LocationManager* es una clase que permite obtener la ubicación del dispositivo; cómo se puede observar ya está inicializada, ya que se inicializa automáticamente cuando se inicia la aplicación. En el penúltimo paso se crea la incidencia mediante el constructor de la clase *Incidence* y se le pasan todos los datos recolectados.

A continuación, se añade la incidencia a FireBase mediante la orden set (en el apartado de implementación se explica más detalladamente). El último paso es exactamente igual a los anteriores, FireBase devuelve un objeto *Task* y según sea el resultado se trata de una forma u otra.

5.3.1.4 Crear incidencia por orden de voz

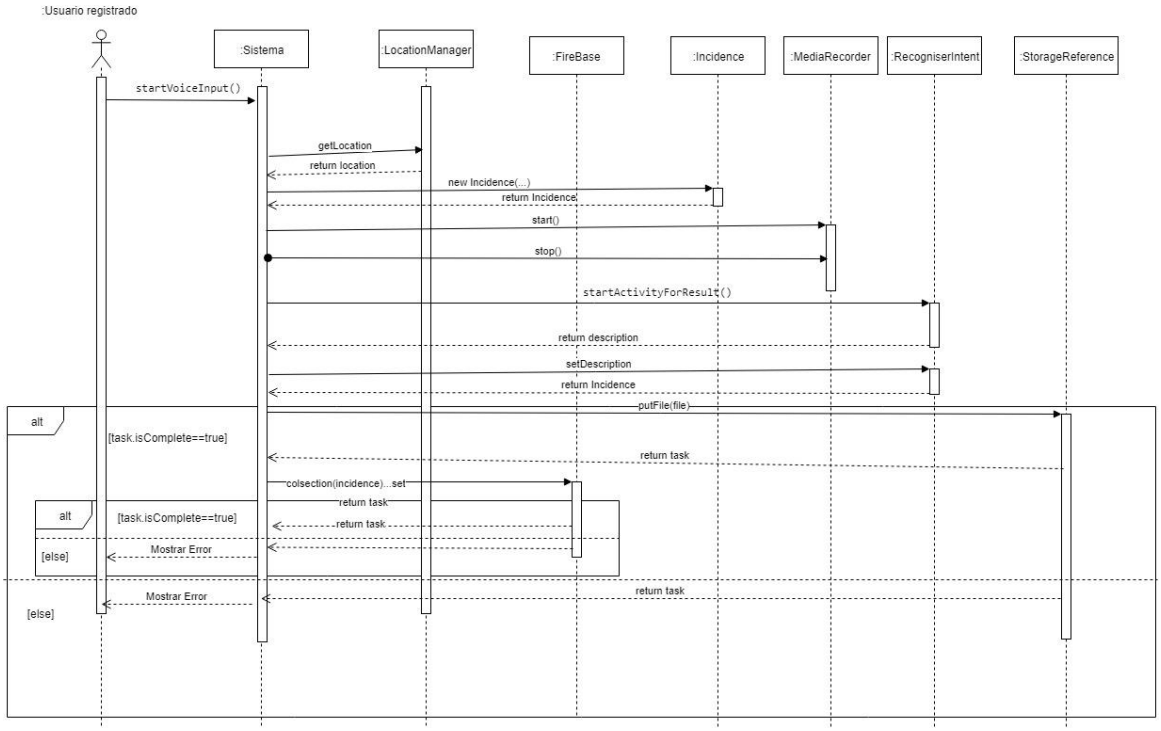

Figura XI: Diagrama de secuencia de crear incidencia por orden de voz

En la figura 11 se localiza el diagrama de secuencia relacionado con el caso de uso de crear incidencia por orden de voz.

La secuencia comienza cuando el usuario pulsa el botón para comenzar la grabación; cuando se realiza esta acción se lanza el método `startVoiceInput();` la primera acción que realiza es obtener la localización de *LocationManager* de forma muy similar a como se realizó en el diagrama de secuencia anterior de crear incidencia.

El siguiente paso que se realiza es llamar a la clase *MediaRecorder* para comenzar a grabar el audio; unos 5 segundos después se para el audio y se llama la clase *RecogniserIntent* con el método `startActivityForResult`. Esto lo que lanza es el sistema de reconocimiento de voz de Google para que el usuario pueda dictar la descripción; una vez el reconocimiento de Google finaliza, devuelve una cadena de caracteres con la descripción; por tanto, el siguiente paso es añadir toda esta información obtenida a la incidencia.

Una vez hecho esto se llama a la clase *StorageReference*; mediante esta clase se almacenan los archivos de audio en el servidor de FireBase, así que en esta llamada se almacena el archivo generado mediante la clase *MediaRecorder;* el archivo es subido mediante la orden `putFile`. Una vez hecho esto se devuelve un objeto de la clase *Task* y según sea el resultado se procederá al siguiente paso o se mostrará un mensaje de error.

En el último paso se añade la incidencia a la colección de FireBase mediante el comando *set*, este paso es exactamente igual que en el caso anterior de crear incidencia.

5.3.1.5 Listar historial de incidencias

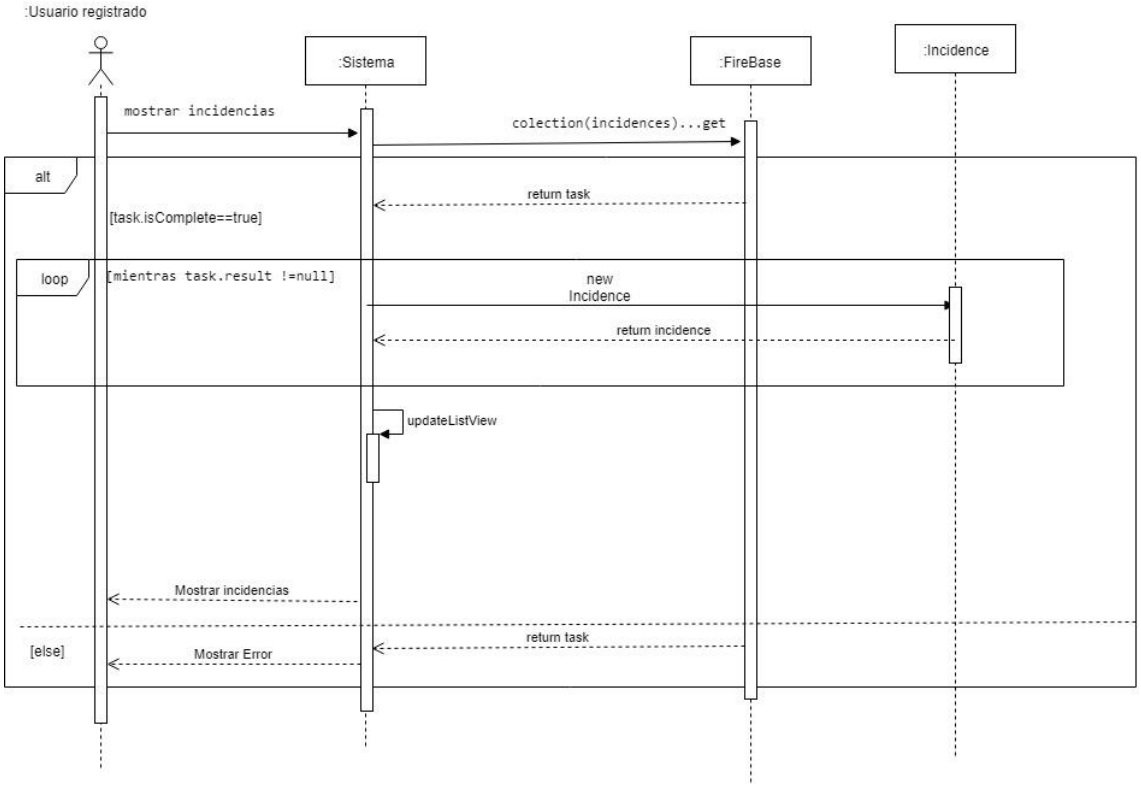

Figura XII: Diagrama de secuencia de listar incidencias

En la figura 12 se muestra el diagrama de secuencia correspondiente al caso de listar incidencias.

Para iniciar la secuencia primero el usuario, desde la ventana principal, pulsa el botón de mostrar el historial; cuando el usuario haya pulsado este botón el sistema automáticamente pedirá a FireBase la lista de todos incidentes de ese usuario (ver apartado de implementación 6.1.5 para más detalles). Al igual que en los casos anteriores FireBase devuelve un objeto de la clase *Task*, pero a diferencia de los casos anteriores, mediante la orden `task.getResult` se obtienen las incidencias que ha devuelto FireBase; a continuación se recorre esta respuesta como si fuera una lista mediante un bucle y se van convirtiendo estos "archivos" (estos "archivos son documentos y se describen mejor en el apartado 6.1.5) a la clase *Incidence*.

Por último, se llama al método `updateListView` y se muestra al usuario la lista de incidencias.

5.3.1.6 Editar incidencia

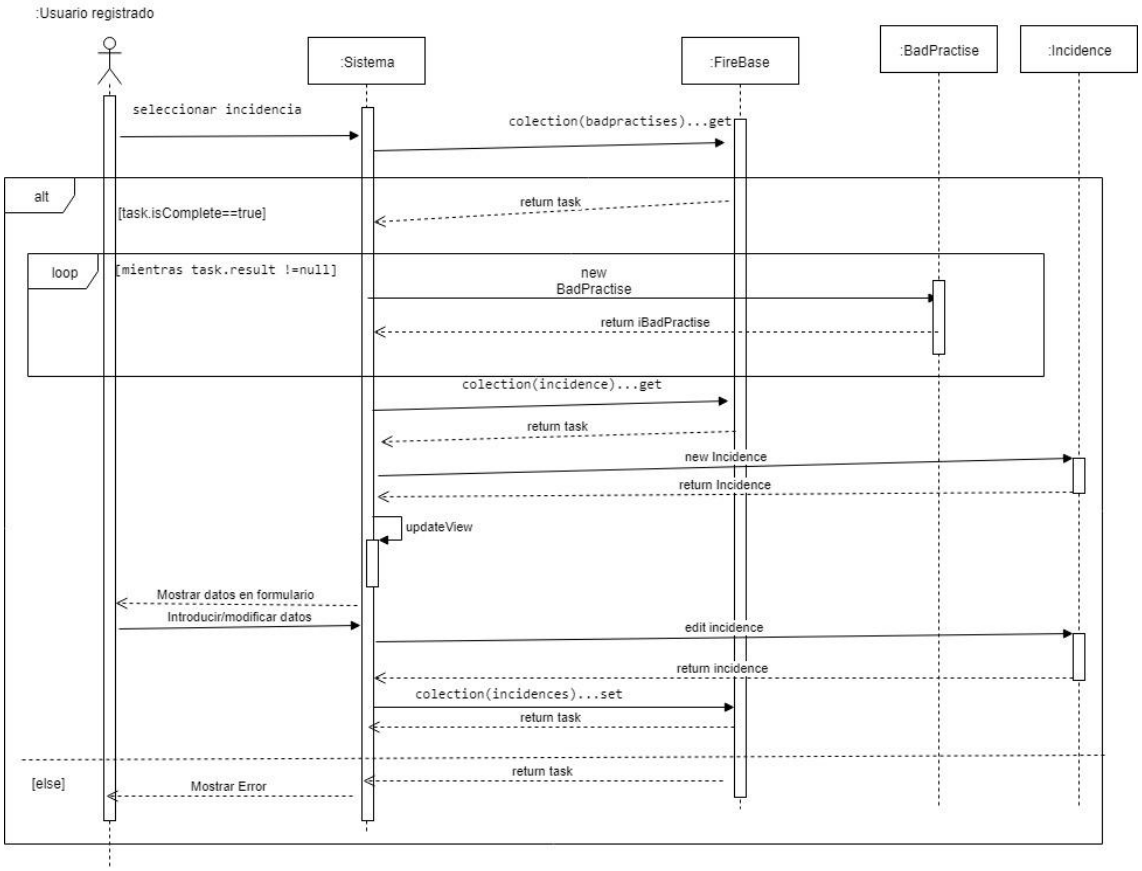

Figura XIII: Diagrama de secuencia de editar incidencia

El diagrama de la figura 13 es el correspondiente al caso de uso de editar incidencia. En este diagrama se da la precondición de que el usuario se encuentre en la pantalla de listar de incidencias, de forma que cuando el usuario seleccione una incidencia se realizarán las acciones que se muestran en el diagrama.

Como se observa en el diagrama, el primer paso que se realizará es descargar todas las malas prácticas disponibles; esto se debe a que estas malas prácticas se añaden en una lista en la interfaz.

El siguiente paso consiste en obtener la incidencia de FireBase que se ha seleccionado de la misma forma que en el caso anterior de listar historial de incidencias, pero sin recorrer la respuesta. Una vez se ha obtenido la incidencia se llama al método `updateView` que muestra un formulario ya rellenado con la información actual de la incidencia. A continuación, el usuario introducirá o modificará los datos de la incidencia.

Por último, cuando el usuario confirme los datos pulsando el botón de siguiente, esta incidencia se actualizará en FireBase mediante la orden `set` y se comprobará que se haya actualizado satisfactoriamente de la misma forma que en el caso anterior de crear incidencia.

5.3.1.3 Cerrar sesión

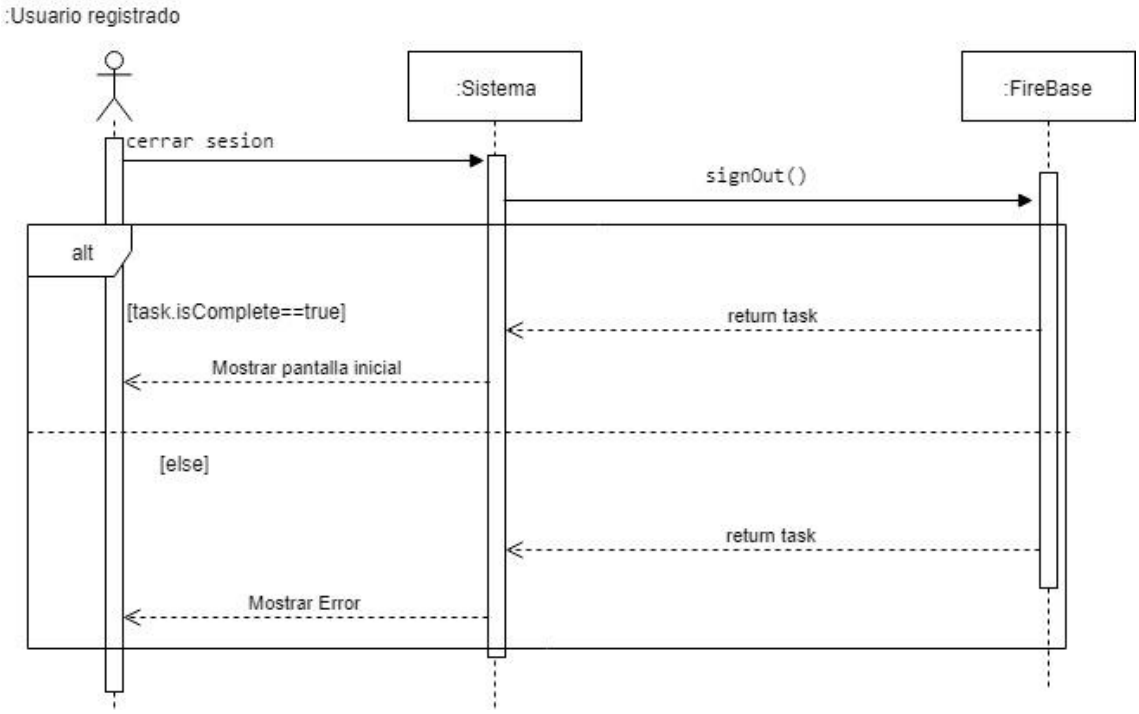

Figura XIV: Diagrama de secuencia de cerrar sesión

En la figura 14 se puede observar el diagrama de secuencia relacionado con el caso de uso de cerrar sesión; cómo se puede observar es muy similar al diagrama de iniciar sesión de la figura 8 y al de registrar usuario de la figura 9. El procedimiento es el siguiente.

El usuario, desde la ventana principal, pulsa sobre el botón de cerrar sesión; una vez el usuario ha realizado esta acción, el sistema ejecutará el método `signOut()` sobre la instancia de FireBase.

Por último, igual que en los casos anteriores, FireBase devolverá un objeto de tipo *Task* y, según sea su descripción, se volverá a la pantalla inicial o se mostrará un error.

5.3.2 Servidor Linux

En este apartado se mostrarán los diferentes diagramas de secuencia que reflejan cada requisito funcional del servidor Linux.

5.3.2.1 Listar incidentes agrupados por matrícula

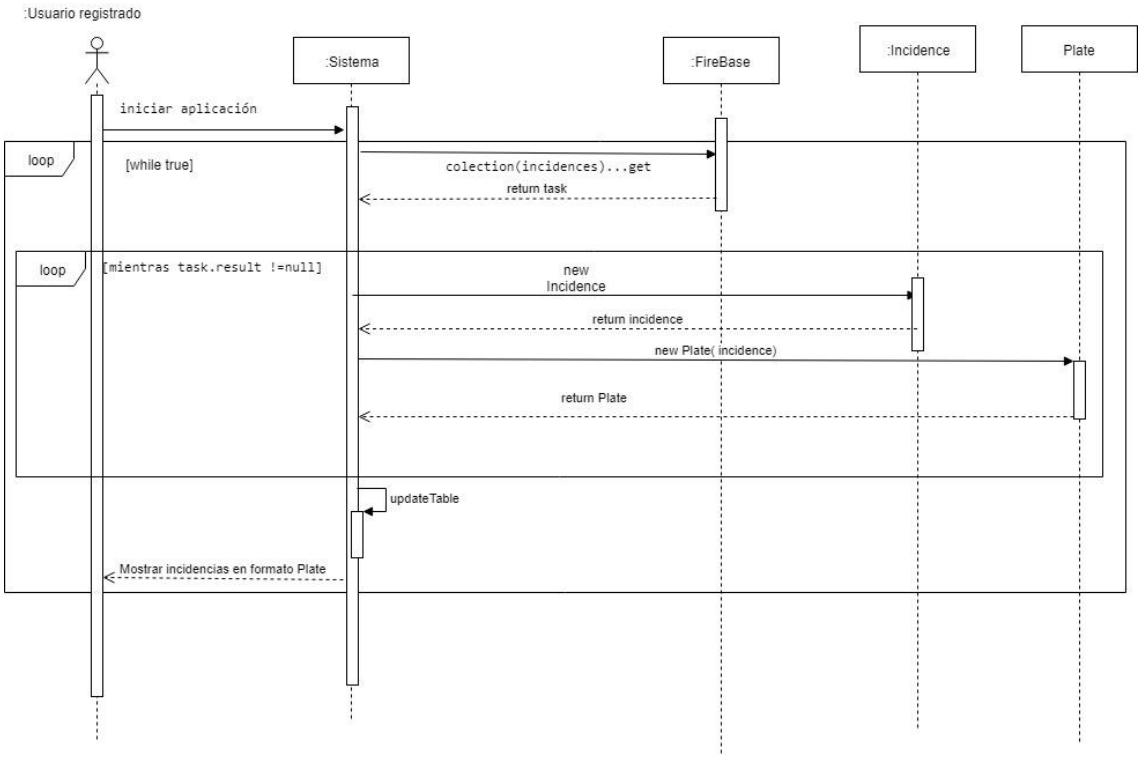

Figura XV: Diagrama de secuencia de listar incidentes agrupados por matrícula

La figura 15 muestra el diagrama de secuencia sobre el caso de uso de listar incidentes agrupados por matrícula. Como se puede observar en el diagrama esta lista será inicializada nada más inicializar la aplicación.

El primer paso que realiza el sistema es obtener todas las incidencias disponibles de la misma forma que se hizo en el apartado 5.3.1.5; una vez obtenidas estas incidencias se agrupan en objetos de tipo *Plate*. A continuación, se lanza el método `updateTable`, que se encarga de mostrar las incidencias al usuario.

Como se puede apreciar, toda esta secuencia está dentro de un bucle infinito; esto realmente se ha implementado mediante un *listener;* éste se analizará más adelante en el apartado 6.2.1 de la fase de implementación.

5.3.2.2 Listar historial de incidentes de una matrícula

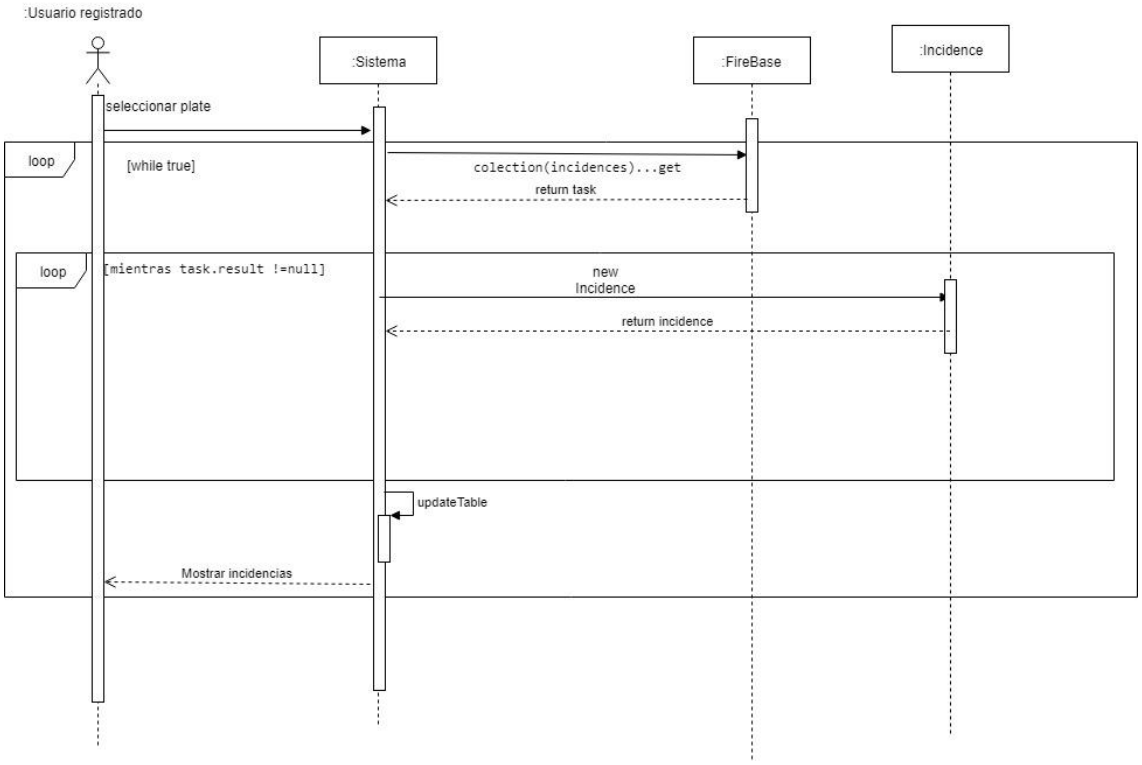

Figura XVI: Diagrama de secuencia de listar historial de una matricula

Como se puede observar, en la figura 16 se encuentra el diagrama de secuencia de listar incidentes de una matrícula. Este diagrama refleja lo que reproduce el sistema cuando el usuario pulsa un incidente de la lista de incidentes agrupados por matrícula y selecciona el botón de historial.

Lo primero que realiza el sistema es obtener todas las incidencias que llevan asociada esta matrícula (en el apartado de implementación 6.2.2 se especifica mejor) mediante la orden `get` sobre la instancia de FireBase.

A continuación, al igual que en los casos anteriores, como el de listar historial de incidencias, se recorre el resultado del objeto *Task* y se van creando lo objetos de tipo *Incidence*; una vez finalizado se ejecuta el método `updateTable` y se muestra la tabla al usuario.

Al igual que sucedía en el caso anterior todo el diagrama está envuelto en un bucle infinito; esto realmente se implementa mediante un *listener* que se verá más detalladamente en el apartado de implementación.

5.3.2.3 Listar todas las incidencias

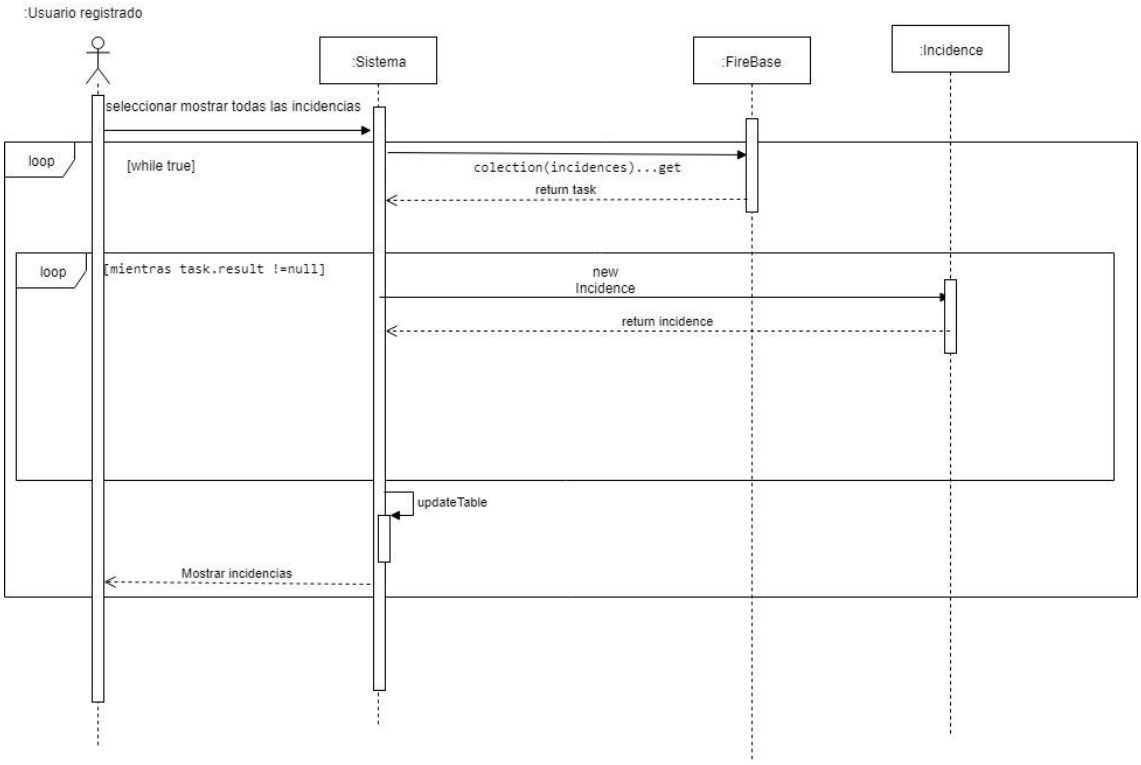

Figura XVII: Diagrama de secuencia de listar todas las incidencias

Como se puede observar, en la figura 17 se encuentra el diagrama de secuencia de listar todos los incidentes generados por los usuarios. Este diagrama se diferencia únicamente del diagrama anterior de la figura 16 en que se inicia cuando el usuario pulsa el botón de mostrar todas las incidencias desde la ventana principal.

5.3.2.4 Editar incidencia

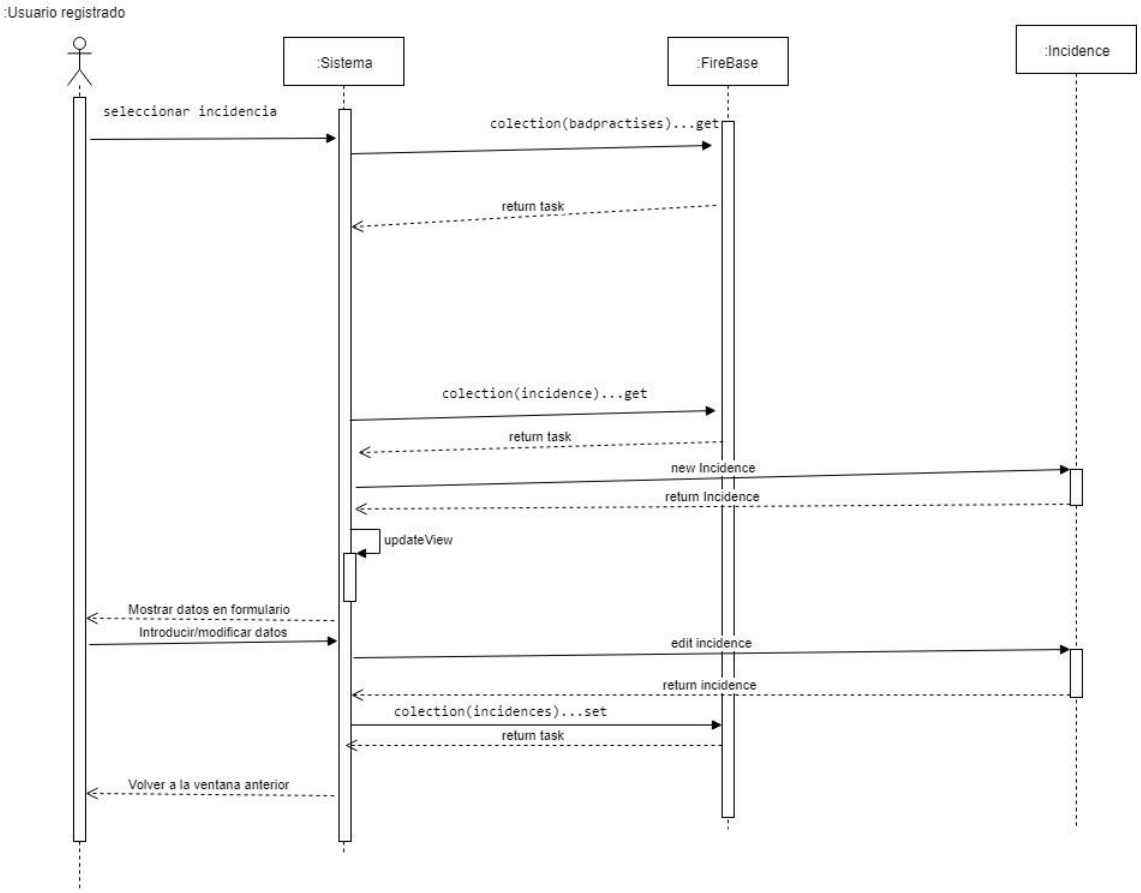

Figura XVIII: Diagrama de secuencia de editar incidencia

En la figura 18 se encuentra el último diagrama de secuencia correspondiente al caso de uso de editar incidencia.

Como se puede observar, es muy similar al diagrama de la figura 13 de editar incidencia en la aplicación Android. El procedimiento para editar la incidencia es el siguiente.

En primer lugar, el usuario selecciona cualquier incidencia; pulsará sobre el botón de editar, a continuación, el sistema obtendrá todas las malas prácticas disponibles. Lo siguiente es obtener de la misma forma la incidencia que ha seleccionado el usuario; la incidencia se obtiene de FireBase de la misma forma que en los casos anteriores, como el caso de editar incidencia del apartado 5.3.1.3. Una vez se ha obtenido, se convierte a objeto de tipo *Incidence*. Después de que se haya obtenido toda la información, se ejecuta el método `updateView`, que mostrará al usuario un formulario ya rellenado con la información de la incidencia.

Por último, el usuario rellenará/modificará el formulario con la información que desee introducir en la incidencia; al pulsar sobre el botón de confirmar el sistema se encargará de actualizar la incidencia en FireBase mediante la instrucción `set` de la misma forma que en los casos anteriores, como el caso de editar incidencia del apartado 5.3.1.3.

6. Implementación

En este apartado se va a analizar cómo se ha realizado la implementación del software y cómo se han integrado todas las tecnologías comentadas anteriormente en el proyecto.

6.1 Cliente Android

Como se ha comentado en el apartado de tecnologías usadas, para el desarrollo de la aplicación se ha usado Android Studio, en concreto la versión 3.3.1 ya que era la última versión disponible en ese momento.

La versión de Android para la que se ha desarrollado la aplicación y desde la cual se han hecho todas las pruebas es Android 9.0 (API 29).

El dispositivo que se ha usado para el desarrollo de la aplicación es un Xiaomi mi note 2, un dispositivo de 2016. Se trata de un dispositivo con un procesador SnapDragon 821 de cuatro núcleos y 8 gigabytes de memoria RAM, unas características más que suficientes para el desarrollo de la aplicación. Todo el código que se va a mostrar a continuación se encuentra en GitHub con el nombre de `IncidenceReportClient`[2].

6.1.1 Log-In

Para implementación del sistema de autentificación se ha usado FireBase Auth.

En la figura 19 se puede observar el código que se ha implementado al pulsar el botón de iniciar sesión en la pantalla inicial de la aplicación.

```
auth.signInWithEmailAndPassword(email,password).addOnCompleteListener( activity: this,new OnCompleteListener<AuthResult>(){
    @Override
    public void onComplete(@NonNull Task<AuthResult> task) {
        if (task.isSuccessful()) {
            progressBar.setVisibility(View.INVISIBLE);

            // Sign in success, update UI with the signed-in user's information
            FirebaseUser user = auth.getCurrentUser();

            Intent intent = new Intent( packageContext: LoginActivity.this, MainActivity.class);

            goToMainActivity(intent);

            //startActivity(intent);
            // updateUI(user);
        } else {
            progressBar.setVisibility(View.INVISIBLE);
            // If sign in fails, display a message to the user.
            Toast.makeText( context: LoginActivity.this,  text: "Authentication failed.",
                    Toast.LENGTH_SHORT).show();
        }
    }
});
```

Figura XIX: Código log-in

Como se puede observar, el código para iniciar sesión es muy simple; tan solo se necesita crear una instancia de FireBase (en este caso es `auth`). Mediante esta

[2] GitHub: https://github.com/AdrianGilabert/IncidenceReportClient (contraseña para descomprimir los archivos mencionados en el archivo Readme “tfg-curso_2018/2019”).

instancia se llama al método `signInWithEmailAndPassword("email del usuario", "contraseña del usuario")`.

Para comprobar que la tarea se ha finalizado correctamente, o ha surgido algún error, se le ha asignado un *listener*; mediante este *listener* se consigue que cuando se haya comprobado correctamente que el usuario existe se lance el menú principal; si se diera el caso de que el usuario no existe o la contraseña es incorrecta, aparecerá un mensaje de error con el mensaje "Authentication failed".

```
@Override
public void onStart() {
    super.onStart();
    if (auth.getCurrentUser() != null) {
        Intent intent = new Intent( packageContext: LoginActivity.this, MainActivity.class);
        goToMainActivity(intent);
    }
}
```

Figura XX: Código Log-In comprobador de usuario

Por último, en la figura 20 se puede observar un código muy sencillo que se ha usado para que si el usuario ya iniciado sesión, después de cerrar la aplicación y volverla a abrir, no tenga que volver a introducir sus datos e iniciar sesión. Esto es fundamental para mejorar la agilidad de la aplicación.

La explicación de la figura 20 es muy sencilla; mediante la instancia de FireBase mencionada anteriormente(`auth`) se ejecuta el método `getCurrentUser()`, que como su nombre indica devuelve el usuario actual; si se da que el caso que devuelva `null` significa que el usuario no ha iniciado sesión, así que no hace nada y se procede al inicio de sesión; si se da el caso contrario, la aplicación salta a la ventana principal.

6.1.2 Registrar

Para la implementación del método de registro se ha utilizado un método muy similar al anterior de iniciar sesión. Este código visible en la figura 21 se ha implementado para que se ejecute una vez que el usuario haya presionado el botón de crear cuenta.

```
mAuth.createUserWithEmailAndPassword(email,password)
        .addOnCompleteListener( activity: this, (task) → {
                if (task.isSuccessful()) {
                    // Sign in success, update UI with the signed-in user's information
                    FirebaseUser user = mAuth.getCurrentUser();
                    Intent intent = new Intent( packageContext: SignupActivity.this, MainActivity.class);

                    goToMainActivity(intent);
                } else {
                    // If sign in fails, display a message to the user.
                    Toast.makeText(getApplicationContext(), text: "Authentication failed.",
                            Toast.LENGTH_SHORT).show();

                }
                // [START_EXCLUDE]
                removeProgressBar();
                // [END_EXCLUDE]
        });
```

Figura XXI: Código registrar

En este caso, al igual en el código anterior, se crea una instancia de FireBase, pero en vez de llamar al método de iniciar sesión, se llama al método `createUserWithEmailAndPassword("email del usuario", "contraseña del usuario").` Finalmente, para comprobar que se haya creado la cuenta perfectamente se usa el mismo método que en el caso anterior.

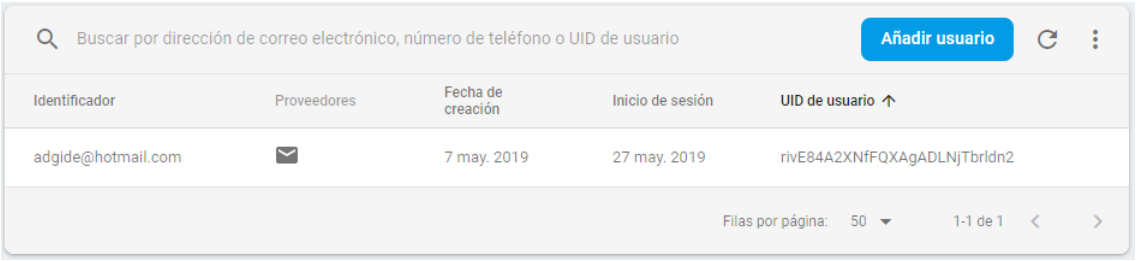

Figura XXII: Muestra de usuarios en FireBase

En la figura 22 se puede observar el resultado de la entrada creada en FireBase al crear un usuario.

6.1.3 Crear incidencia

Para la creación de una incidencia se ha creado una clase modelo con nombre "Incidence"; en esta clase se han definido los distintos atributos necesarios, como el número de la matrícula, o la mala práctica cometida.

Una vez creado el objeto "Incidence", se ha implementado el código que se muestra en la figura 22 para subir la incidencia a la base de datos FireBase Cloud Firestore.

```
firestoreDatabase.collection( collectionPath: "incidences").document(n.getId()).set(n).addOnSuccessListener(new OnSuccessListener<Void>() {
    @Override
    public void onSuccess(Void aVoid) {
        goToHome();

    }
})
        .addOnFailureListener(new OnFailureListener() {
            @Override
            public void onFailure(@NonNull Exception e) {
                Toast.makeText(getApplicationContext(),
                        text: "Se ha producido un error, por favor vuelva a intentarlo.", Toast.LENGTH_SHORT).show();
            }
        });
```

Figura XXIII: Código de crear incidencia

En el apartado 2.2.3 de tecnologías usadas se comentó que la base de datos de FireBase guardaba la información en documentos y estos documentos se almacenan en colecciones. El código de la figura 23 refleja cómo es la creación de una colección y de un documento dentro de esta colección (la primera vez que se sube una incidencia se crea la colección automáticamente).

Al igual que en las implementaciones anteriores, se necesita una instancia de FireBase; en este caso es `firestoreDatabase`. Posteriormente se llama al método `collection` para acceder a la colección deseada; en este caso, el nombre de la colección que se ha usado es "incidences".

Finalmente se llama al método `document("nombre del documento")` y, mediante la orden `set`, se sube el documento; también se puede usar la orden `add` que, a diferencia del anterior, no remplaza el documento, pero para evitar errores se ha optado por usar la orden `set`.

Por último, como se puede observar en la figura 23, el código para manejar la respuesta del servidor es muy similar al de log-in (figura 19) y al de iniciar sesión (figura 21); si no sucede ningún error vuelve a la ventana inicial y si sucede el caso contrario aparece un mensaje de error.

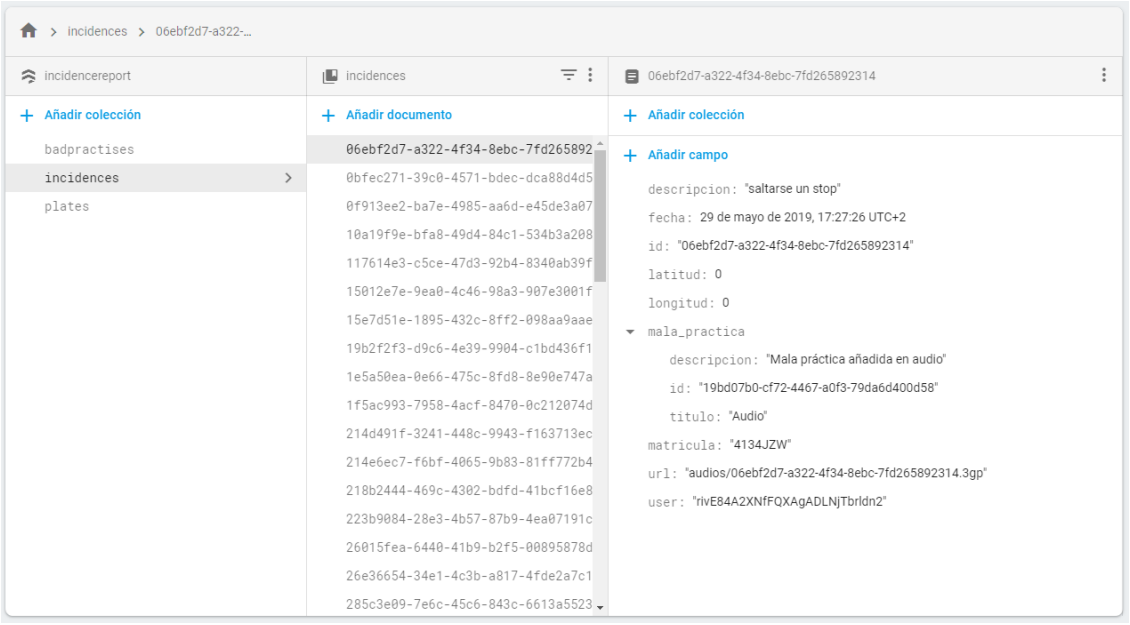

Figura XXIV: FireBase colección de documentos

Para la comprender mejor el sistema de almacenamiento de información de FireBase Cloud Storage basado en documentos y colecciones se puede observar la figura 24.

6.1.4 Crear incidencia por orden de voz

Para la implementación de la creación de una incidencia por orden de voz se ha utilizado un proceso muy similar al de crear una incidencia mediante la interfaz, pero con la diferencia de que a ésta se la añadido un enlace que hace referencia al archivo de audio.

```
fileName = getExternalCacheDir().getAbsolutePath();
fileName += "/" + audioIncidence.getId() + ".3gp";
recorder = new MediaRecorder();
recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
recorder.setAudioEncodingBitRate(16*44100);
recorder.setAudioSamplingRate(16000);
recorder.setOutputFile(fileName);
recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
try {
    recorder.prepare();
} catch (IOException e) {
    e.printStackTrace();
}
```

Figura XXV: Código de MediaRecorder

Como se puede observar en la figura 25, se ha usado la clase que proporciona Android Studio llamada *MediaRecorder* para grabar el audio. En esta figura se muestra toda la configuración posible, como el nombre del audio; en este caso se ha usado el identificador de la incidencia y la localización de éste, que en este caso se ha usado el directorio por defecto destinado para el *cache*. El último paso para finalizar la

configuración es ejecutar la orden `prepare` para que esta configuración sea validada y guardada.

```
recorder.start();
Handler handler = new Handler();
handler.postDelayed(() -> {
        recorder.stop();
        Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL, RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE, Locale.getDefault());
        intent.putExtra(RecognizerIntent.EXTRA_PROMPT,  value: "Introduce una Descripción");
        try {
            startActivityForResult(intent, REQ_CODE_SPEECH_INPUT);
        } catch (ActivityNotFoundException a) {

        }
},  delayMillis: 5000);
```

Figura XXVI: Código de crear grabación

Una vez se ha configurado el *MediaRecorder* (ver figura 25), en la figura 26 se observa que ya se puede comenzar a grabar mediante la orden `start`.

El último paso para finalizar la grabación es ejecutar la orden stop, pero si se hubiera puesto en la línea siguiente de la orden `start` no daría tiempo para dictar la matricula, así que como se puede observar en la figura 26 se ha lanzado un *Handler* (este objeto permite lanzar un hilo en segundo plano que ejecute una orden al cabo de un tiempo determinado) para que la orden se ejecute al cabo de 5 segundos, ya que mediante las pruebas que se han hecho este es el tiempo idóneo para dictar una matrícula.

Ahora que ya se ha finalizado la grabación, el siguiente paso es añadir una descripción a la incidencia, pero no se puede añadir una descripción mediante una entrada de texto convencional, ya que si no todo el esfuerzo de haber implementado un reconocimiento de voz para las matrículas no tendría ninguna ventaja. Así que se ha implementado el reconocimiento de voz de Google para que transcriba la entrada de voz a texto.

Esto se ha implementado, como se puede ver en la figura 26, de forma que cuando se pare la grabación del *MediaRecorder*, el paso siguiente será crear y preparar el Intent (este objeto permite lanzar una nueva actividad) y finalmente lanzarlo mediante la orden `startActivityForResult`.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    switch (requestCode) {
        case REQ_CODE_SPEECH_INPUT: {
            if (resultCode == RESULT_OK && null != data) {
                ArrayList<String> result = data.
                        getStringArrayListExtra(RecognizerIntent.EXTRA_RESULTS);
                audioIncidence.setDescripcion(result.get(0));
                Uri file = Uri.fromFile(new File(fileName));
                StorageReference storageRef = storage.getReference();
                StorageReference riversRef = storageRef.child(audioIncidence.getUrl());
                UploadTask uploadTask = riversRef.putFile(file);

                // Register observers to listen for when the download is done or if it fails
                uploadTask.addOnFailureListener((exception) → {
                        // Handle unsuccessful uploads
                }).addOnSuccessListener((OnSuccessListener) (taskSnapshot) → {
                        firestoreDatabase.collection( collectionPath: "incidences")
                                .document(audioIncidence.getId())
                                .set(audioIncidence);
                });

            }
            break;
        }

    }
}
```

Figura XXVII: Código para el reconocimiento de voz de Google

Cuando el *Intent* generado en la figura 26 finaliza, es decir, cuando la actividad del reconocimiento de Google detecta que el usuario ya ha acabado de dictar, se ejecuta el método `OnActivityResult` que se encuentra en la figura 27.

El método `OnActivityResult` se encarga de añadir la descripción a la incidencia mediante el método `setDescription`; también se encarga de subir el audio a la nube de FireBase *CloudStorage*. Para esto primero se crea una referencia de *FireStorage* y se le añade una cadena de caracteres con la ruta donde se va a localizar el audio mediante la orden `child("ruta donde se va a localizar el archivo").`

Finalmente, se crea una tarea de tipo UploadTask para subir el archivo; a esta tarea se le añade un *listener* para que una vez el archivo se haya subido correctamente se suba la incidencia creada a la base de datos mediante la instrucción `set`, de la misma forma que se hizo en el apartado anterior 4.1.3.

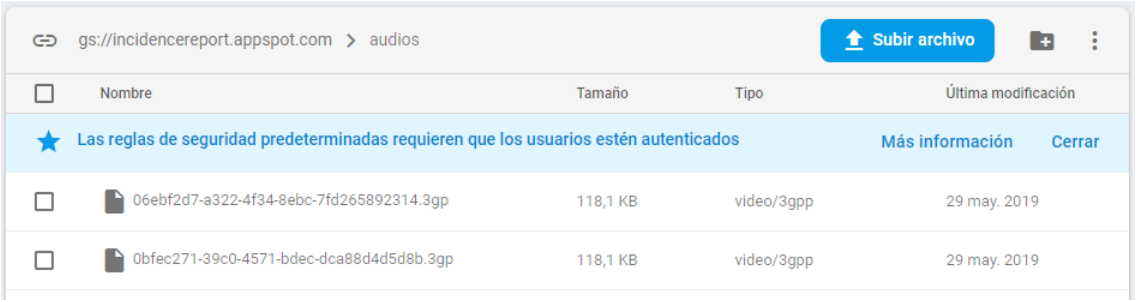

Figura XXVIII: Resultado en FireBase de subir archivo

En la figura 28 se muestra el resultado de FireBase Cloud Storage al subir un archivo; cómo se puede observar, los audios se nombran correctamente con el identificador de la incidencia y se localizan en la carpeta llamada “audios”.

6.1.5 Listar incidencias

Para el proceso de descarga de las incidencias de los usuarios de FireBase Cloud *FireStorage* se ha implementado el código que aparece en la figura 29.

```
firestoreDatabase.collection( collectionPath: "incidences") CollectionReference
        .whereEqualTo( field: "user", FirebaseAuth.getInstance()
                .getCurrentUser().getUid()) Query
        .get() Task<QuerySnapshot>
        .addOnCompleteListener(new OnCompleteListener<QuerySnapshot>() {
            @Override
            public void onComplete(@NonNull Task<QuerySnapshot> task) {
                if (task.isSuccessful()) {
                    for (QueryDocumentSnapshot document : task.getResult()) {
                        incidences.add(document.toObject(Incidence.class));

                    }
                    updateListView();
                }
            }
        });
```

Figura XXIX: Código de obtener historial de incidencias

Como se puede observar, el código es muy similar al de la figura 23, pero en este caso, en vez de usar la orden `set` se usa la instrucción `get.` Para limitar la búsqueda a las incidencias que se desean buscar se usa el método `whereEqualTo("atributo",valor).`

Una vez se ha completado la descarga, se obtiene una lista de *QueryDocumentSnapshot*, que es la clase que usa FireBase para definir sus documentos; así, el siguiente paso es recorrer esta lista y convertir cada documento a la clase *Incidence* mediante el método `toObject(class).` De esta forma se ha obtenido una lista de incidencias y ya solo queda actualizar la interfaz.

6.1.6 Cerrar sesión

Por último, para la implementación del requisito de cerrar sesión tan solo basta con crear una instancia de FireBase y mediante el la orden `sign-out` ya se cierra la sesión.

En la figura 30 se puede observar cómo se ha implementado.

```
if (id == R.id.exitbutton) {
    auth.signOut();
    Intent intent = new Intent( packageContext: this, LoginActivity.class);
    this.startActivity(intent);
}
```

Figura XXX: Código de cerrar sesión

6.2 Servidor Linux

El desarrollo para la aplicación del servidor Linux se ha llevado a cabo usando el programa IntelliJ IDEA mencionado en el apartado 2.3.2 mediante el lenguaje de programación JavaFx (ver el apartado 2.3.1).

En cuanto al equipo en el que se ha alojado el programa, se ha optado por una placa base Asrock J4205-ITX con un procesador integrado Intel j4205 de cuatro núcleos de bajo coste con 5 gigabytes de memoria RAM instalado. Este equipo tiene un sistema operativo Linux basado en Debian llamado Parrot Security OS Home Edition en la versión 4.6.

Como se puede observar, es un equipo bastante modesto, ya que una de las principales cualidades que se tenían en cuenta en este proyecto era que se pudiera ejecutar en un ordenador de bajo consumo. La razón es que este equipo debe de estar funcionando prácticamente durante todos los días del año.

Todo el código que se va a mostrar a continuación se encuentra en GitHub con el nombre de `IncidenceReportServer`[3].

[3] GitHub: https://github.com/AdrianGilabert/IncidenceReportServer (contraseña para descomprimir los archivos mencionados en el archivo Readme “tfg-curso_2018/2019”).

6.2.1 Listar incidentes agrupados por matrícula

Para la descarga y el acceso de las incidencias a la base de datos se ha usado FireBase Admin (ver apartado 2.3.1).

```
db.collection( path: "incidences")
        .addSnapshotListener(new EventListener<QuerySnapshot>() {

            @Override
            public void onEvent(QuerySnapshot snapshots, FirestoreException error) {
                if (error != null) {
                    System.err.println("Listen failed:" + error);
                    return;
                }
                misdatos.clear();
                plateMap.clear();
                for (DocumentSnapshot doc : snapshots) {
                    Plate aux;
                    Incidence in = doc.toObject(Incidence.class);
                    if (plateMap.containsKey(in.getMatricula())) {
                        aux = plateMap.get(in.getMatricula());
                        aux.setCount();
                        System.out.println(aux.getCount());
                        if (aux.getIncidence().getFecha().getTime() <= in.getFecha().getTime()) {
                            aux.setIncidence(in);
                        }
                    } else {
                        aux = new Plate(in);
                    }
                    plateMap.put(in.getMatricula(), aux);

                }
                updateTable();

            }

        });
```

Figura XXXI:Codigo de listar incidentes por matrícula

En el código de la figura 31 se ha implementado el código para poder listar las matrículas con más incidentes en directo, de forma que conforme los usuarios van añadiendo incidencias esta lista se va actualizando de forma inmediata.

Como se puede observar en la figura 31, la estructura del código una vez creada la instancia de FireBase es muy similar a los códigos mostrados en el apartado del cliente Android.

Para obtener la lista se obtiene añadiendo un *listener* directamente a la colección, sin necesidad de usar la orden `get` como en el apartado 4.1.5. El método `onEvent` se ejecuta cada vez que la colección es actualizada, así que en este método es donde se pasan las incidencias a una lista mediante el bucle `for`, al igual que se hizo en el código de la figura 29.

Para poder almacenar las matrículas con sus correspondientes incidentes se ha usado la clase modelo *Plate,* que consta de una matrícula, el último incidente producido por esta matricula y un contador con todos los incidentes. Estos objetos de la clase *Plate* se han almacenado en un *HasMap* llamado `plateMap`, ya que éste, a diferencia de una

colección de tipo *List*, permite acceder de forma fácil al objeto introduciendo únicamente la matrícula, sin tener que recorrer todos los objetos buscando una matrícula concreta.

Por último, una vez se ha obtenido el *HashMap* de *Plates*, se llama al método `updateTable` para actualizar la interfaz.

6.2.2 Listar historial de incidentes de una matricula

```
db.collection( path: "incidences").whereEqualTo( field: "matricula", matricula)
        .addSnapshotListener((snapshots, error) → {
                if (error != null) {
                    System.err.println("Listen failed:" + error);
                    return;
                }
                misdatos.clear();
                for (DocumentSnapshot doc : snapshots) {
                    Incidence in = doc.toObject(Incidence.class);
                    misdatos.add(in);
                }
                updateTable();

        });
```

Figura XXXII:Codigo de obtener historial de una matrícula en el servidor

El código que se observa en la figura 32 es el que se ha implementado para mostrar en la parte del servidor el historial de incidencias de una matrícula. Como se puede observar es muy similar al de la figura 31, pero más simple; se obtienen las incidencias deseadas de la colección “incidences” mediante el método `collection`; en adelante se selecciona la matrícula mediante la orden `whereEqualTo(“matricula”,matricula)` y se le añade un *listener*.

De la misma forma que en el caso anterior, en el método `onEvent` se convierten todos los documentos obtenidos a objetos de tipo *Incidence*; por último, se pasan a una lista para posteriormente actualizar la interfaz mediante el método `updateTable().`

6.2.3 Procesar audios

Este es el apartado más importante de la aplicación; ya que se ha supuesto que la mayoría de los usuarios preferirían crear las incidencias mediante audios en vez de mediante la interfaz. Esto es puesto que es la única forma de poder usar la aplicación sin quitar la vista de la carretera mientras se conduce, así que tendría que ser lo más eficiente posible.

El primer paso para procesar los audios es encontrar aquellas incidencias que disponen de audios; para esto se ha especificado que en la creación de incidencias mediante una orden de voz, el nombre de la matrícula de estas sea “PROVISIONAL”; de esta forma se pueden obtener estas incidencias de manera fácil, tal y como aparece en la figura 33.

```
final Firestore db = FirestoreClient.getFirestore();
db.collection( path: "incidences").whereEqualTo( field: "matricula",  value: "PROVISIONAL")
        .addSnapshotListener(new EventListener<QuerySnapshot>() {

            @Override
            public void onEvent(QuerySnapshot snapshots, FirestoreException error) {
                if (error != null) {
                    System.err.println("Listen failed:" + error);
                    return;
                }
                misaudios.clear();
                for (DocumentSnapshot doc : snapshots) {
                    Incidence in = doc.toObject(Incidence.class);
```

Figura XXXIII: Código de obtener incidencias con audio relacionado

Como se puede observar en la figura 33, el procedimiento para descargar las incidencias con audio es exactamente el mismo que en los casos anteriores.

El siguiente paso ya varía de los demás, pues consiste en descargar los audios de FireBase Cloud Storage. Para esto tan solo es necesario obtener una instancia del depósito de Cloud Storage, como se puede observar en la figura 34.

```
final Bucket bucket = StorageClient.getInstance().bucket();
```

Figura XXXIV: Código instancia de Storage

El siguiente paso es obtener el enlace donde está localizado el audio; esto se realiza mediante el método `getUrl` que proporciona la clase *Incidence*. Finalmente se descarga el audio usando el método `downloadTo`, como se puede ver en la figura 35.

```
Blob blob = bucket.get(in.getUrl());
Path path = Paths.get( first: "/home/adrian/iatros_matriculas/audios/" + in.getId() + ".3gp");

blob.downloadTo(path);
```

Figura XXXV: Código de descargar audio

Ahora ya está lista toda la información necesaria para proceder a convertir el audio a texto; para esto se ha creado una clase en especial llamada AudioRecogniser que, mediante la introducción del identificador del incidente, se encarga de devolver la matrícula generada. A continuación, en la figura 36 se muestra el uso de la clase AudioRecogniser.

```
                AudioRecogniser recogniser = new AudioRecogniser(in.getId());
                in.setMatricula(recogniser.recogniseAudio());
                System.out.println("matricula " + in.getMatricula());
                db.collection( path: "incidences").document(in.getId()).set(in);

            }
        }
});
```

Figura XXXVI: Código de usa de la clase AudioRecogniser

Este sería el proceso para buscar, descargar, procesar y actualizar las incidencias, pero para la mejora de la comprensión del procedimiento, a continuación se va a analizar el funcionamiento de la clase *AudioRecogniser*.

Clase AudioRecogniser

La clase *AudioRecogniser* procesa el audio en cuatro pasos. El primer paso consiste en convertir el audio de formato 3gp a un formato de audio universal como es el mp3; esto se realiza mediante la herramienta Ffmpeg. Para la comprensión de la implementación de la llamada a esta herramienta se puede observar la figura 37.

```
// Convert 3gp to mp3
Thread tl = new Thread(new Speech());
tl.start();
try {
    System.out.println("Convert to mp3");

    Process convtomp3 = Runtime.getRuntime()
            .exec( command: "ffmpeg -y -i " +
                    AUDIO_PATH + file + ".3gp -vn " + AUDIO_PATH + file + ".mp3");
    convtomp3.waitFor();
} catch (Exception e) {
    System.exit( status: -1);
}
```

Figura XXXVII: Código de llamar a la herramienta Ffmpeg

El siguiente paso es convertir el audio de mp3 a un formato de audio llamado raw, pero con unas cualidades más específicas para que el reconocimiento de voz lo pueda entender y procesar. En este caso el audio debe tener una frecuencia de 16Khz y un BitRate de 2 bytes. Para la conversión a raw se ha usado la herramienta Sox. En la figura 38 se puede observar cómo se ha implementado la conversión.

```
// Convert mp3 to raw
try {
    System.out.println("Convert to raw");
    Process convtoraw = Runtime.getRuntime()
            .exec( command: "sox " + AUDIO_PATH + file
                    + ".mp3 -t raw -r 16000 -b 16 -e signed-integer "
                    + AUDIO_PATH  + "matricula.raw");
    convtoraw.waitFor();
} catch (Exception e) {
    System.exit( status: -1);
}
```

Figura XXXVIII: Código de la llamada a la herramienta Sox

El tercer paso se trata de otra conversión, pero en este caso ya sería el formato final en el cual el reconocedor ya puede entender; la conversión sería del raw que se ha generado antes al formato CC. En la figura 39 se puede observar como el código que se ha implementado lo que hace es lanzar un *script* llamado `iatros_launche-modr.sh`; este *script* es el que se encarga de convertir el archivo raw a CC.

```
String[] cmd = {"sh", "/home/adrian/iatros_matriculas/iatros_launche-modr.sh"};
String result="";
try {
    System.out.println("Processing audio");

    Process p = Runtime.getRuntime().exec(cmd);
    p.waitFor();
```

Figura XXXIX: Código de la llamada al reconocimiento de voz

El último paso es observar qué ha devuelto el sistema por la consola y devolverlo en formato de cadena de caracteres; para esto se puede observar la figura 40.

```
        BufferedReader reader = new BufferedReader(new InputStreamReader(
                p.getInputStream()));
        String line = "";
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
            result+=line;
        }

    } catch (Exception e) {
        System.out.println(e.toString());
    }
    return result.trim();
}
```

Figura XL: Código para devolver la matrícula del reconocimiento de voz

6.2.4 Reconocimiento de habla

Para el reconocimiento del habla se ha utilizado, como se describe en la sección 2.3.6, el reconocedor IATROS.

El modelo acústico combina un conjunto de 23 fonemas y dos tipos de silencio, y son HMM con una mixtura de 64 gaussianas entrenados usando la herramienta HTK [7] sobre la base de datos Albayzin[5]. El modelo léxico incluye como vocabulario las posibilidades de deletreo de las matrículas, es decir, los diez dígitos y las 26 letras del alfabeto español, a excepción de la Ñ.

El modelo de lenguaje es un modelo probabilístico de estados finitos donde están incluidas las posibles combinaciones que se dan en las matrículas del parque automovilístico español:

- Identificador de provincia (una o dos letras), seis dígitos
- Identificador de provincia (una o dos letras), cuatro dígitos, una letra
- Identificador de provincia (una o dos letras), cuatro dígitos, dos letras
- Cuatro dígitos, tres letras

7. Conclusiones

El resultado del desarrollo del proyecto no cubre todos los objetivos planteados ya que para cubrirlos todos se necesitaría que la aplicación hubiera llevado ya un tiempo en la tienda de aplicaciones y haber sido usada por una cantidad significativa de usuarios para poder hacer un estudio del antes y el después en cuanto a número de accidentes e infracciones cometidas. Pero el objetivo principal, que era proporcionar una herramienta a los usuarios que les permitiera informar sobre las malas prácticas de otros conductores, y otra herramienta a las autoridades que permita estar al tanto de estas incidencias sí que se ha cumplido, por lo que se está satisfecho con el resultado. También cabe mencionar que todos los requisitos funcionales y no funcionales han sido implementados, por lo que la primera versión del producto ha sido concluida satisfactoriamente.

Este trabajo, a pesar de ser un proyecto relativamente pequeño, ha conllevado muchas horas de trabajo debido a la dificultad de integrar y coordinar todas las piezas y módulos de este proyecto.

Una de estas dificultades que cabe mencionar ha sido la implementación de FireBase tanto en la implementación de la parte del servidor como la del cliente. Como se ha podido observar a lo largo de la memoria, se ha hecho mucho hincapié en el funcionamiento y en la implementación de esta tecnología ya que, aunque se han tenido muchas dificultades con FireBase, el resultado que se ha obtenido es un gran conocimiento sobre esta herramienta, lo cual es muy útil, pues se ha descubierto el gran potencial de la misma.

Otro contratiempo a destacar ha sido la dificultad de tener que desarrollar la aplicación del servidor en un entorno totalmente distinto al que se estaba acostumbrado, como es Linux. Los problemas que han surgido fueron básicamente aquellos relacionados con la instalación de las herramientas y bibliotecas necesarias para el correcto funcionamiento de JavaFx con el SDK IntelliJ IDEA; el sistema operativo Linux es muy distinto a Windows en cuanto a la instalación de aplicaciones se refiere, ya que según el programa se instala de una manera u otra.

Por otro lado, ya que se han comentado todas las dificultades, también se ve conveniente comentar que, una vez solucionados estos inconvenientes; con Linux el proceso de desarrollo del servidor ha requerido menos esfuerzo del que se había estimado.

Otro aspecto del proyecto a comentar son los errores producidos durante el trabajo, como es el no haber realizado un proceso de análisis de requisitos más minucioso desde el principio, ya que esto ha conllevado retrasos durante el proceso de desarrollo. Otro error cometido ha sido el no haber realizado un estudio más minucioso de todas las tecnologías disponibles que puedan encajar con el proyecto, pues aunque el resultado de JavaFx se puede considerar satisfactorio, sí que habría sido mejor poder usar una tecnología más moderna, con más funciones.

En conclusión, pese a los inconvenientes, el resultado ha sido satisfactorio, ya que ha cumplido con todos los requisitos especificados además de haber servido para obtener

un conocimiento mayor sobre una nueva tecnología como es FireBase y haber servido para familiarizarse a consultar documentación especifica tanto de Android FireBase, Java, etc. También ha sido especialmente útil para poder planificar mejor el tiempo y realizar un mejor análisis de requisitos en trabajos futuros.

8. Relación del trabajo desarrollado con los estudios cursados

Para la realización de este proyecto ha servido de gran ayuda la realización de la rama de Ingeniería de Software ya que, aunque como se ha comentado en el apartado de conclusiones habría sido conveniente un análisis de requisitos más minucioso, el hecho de haber cursado la asignatura de Análisis de Requisitos ha servido para poder extraerlos sin mayor complicación. Otra asignatura que ha sido de gran utilidad ha sido IPC (Interfaces Persona Computador), ya que ha permitido crear una interfaz amigable y consistente en el apartado del servidor usando la tecnología de JavaFx, que es la que se estudió.

Para la integración de todos los módulos cabe mencionar la asignatura de IEI (Integración e interoperabilidad), que ha servido de gran utilidad, tanto para poder añadir las bibliotecas necesarias a los proyectos como para familiarizarse en buscar información en la documentación. Para la gestión del tiempo, haber cursado las asignaturas de gestión de proyectos y proyectos de ingeniería del software han sido de gran ayuda.

Finalmente, la asignatura que más ha servido para la aplicación del cliente ha sido DADM (desarrollo de aplicaciones móviles), ya que para desarrollarla se ha usado Android y si se hubiera empezado el proyecto sin ningún conocimiento previo de Android habría sido muy complicado y se habrían cometido muchos más errores.

En conclusión, si no se hubiera cursado el grado de Ingeniería informática no habría sido posible el desarrollo de este proyecto; aunque solo se han mencionado algunas asignaturas éstas no son las únicas que han servido para la creación del proyecto; de todas las asignaturas se ha podido extraer una porción de información o bien para aplicarlo al trabajo o bien para la mejora de comprensión de alguna herramienta.

Las competencias transversales que se han requerido en este proyecto son:

- Conocimiento de problemas contemporáneos: Esta ha sido la competencia principal de este trabajo, ya que esta destinado a disminuir un problema actual como son las malas prácticas conduciendo.
- Comprensión e integración: Para la integración de todas las herramientas y servicios, como FireBase, se ha requerido de esta competencia.
- Análisis y resolución de problemas: Esta competencia también ha sido fundamental debido a los problemas que han ido surgiendo a lo largo del proyecto.
- Comunicación efectiva: Esta competencia se ha requerido, ya que esta muy relacionada con la interacción entre el tutor y el desarrollador, y para la creación de la memoria, ya que esta debe de ser clara y concisa.
- Aprendizaje permanente: Por último, se ha requerido de esta competencia transversal, para poder estar informado de todas las últimas tecnologías, como FireBase.

9. Trabajos Futuros

En esta sección se analizarán todas las posibles mejoras o aspectos a mejorar que se han pensado implementar, pero no han sido posibles debida a la complejidad, falta de conocimiento o falta de tiempo.

Una posible mejora del proyecto sería poder hacer la aplicación compatible con Android Auto, el cual es un sistema operativo diseñado para automóviles; mediante esta plataforma la aplicación sería mucho más cómoda de usar, ya que se tendría una interfaz más amplia gracias a una pantalla más grande, tampoco sería necesario tener que encender la aplicación y poner el dispositivo en un soporte cada vez que se usa el coche.

Uno de los principales aspectos a mejorar sería la aplicación del servidor, ya que pudiendo acceder a todas las incidencias podría ser un centro de análisis de datos muy potente y útil para alguna entidad responsable de la vía, como por ejemplo la DGT. Se podrían extraer estadísticas de los lugares donde más se cometen infracciones o las horas donde se producen más malas prácticas.

También, para que esta aplicación tenga sentido, un futuro trabajo sería acordar con alguna entidad para que empiece a usar la aplicación del servidor; de esta forma, este proyecto ya se vería envuelto en un entorno real no hipotético.

Otra funcionalidad que se desearía incorporar en el futuro sería un sistema de visión artificial para el reconocimiento de matrículas en directo; mediante este sistema se permitiría que los dispositivos de los usuarios que están con este sistema activado y enfocando a la carretera puedan hacer una pequeña grabación cuando detecten automáticamente un coche con una matrícula que tenga asociada muchos incidentes. De esta forma se podría llegar a obtener una prueba real de la infracción que pueda servir de gran utilidad para las autoridades.

Además, a este sistema se le podrían ir incorporando más funcionalidades útiles como un algoritmo que detecte las malas prácticas más comunes, como son el exceso de velocidad o saltarse un ceda al paso o un stop.

10. Bibliografía

[1] ETSI. Etsi es 201 108 (v1.1.3), speech processing, transmission and quality aspects (stq); distributed speech recognition; front-end feature extraction algorithm; compression algorithms. Technical report, ETSI, 2003.

[2] G. D. Forney. The viterbi algorithm. *Proc. of the IEEE*, 61:268 – 278, March 1973.

[3] Míriam Luján-Mares, Vicent Tamarit, Vicent Alabau, Carlos D. Martínez Hinarejos, Moisés Pastor i Gadea, Alberto Sanchis, and Alejandro H. Toselli. iatros: A speech and handwritting recognition system. *In V Jornadas en Tecnologías del Habla* (VJTH'2008), pages 75–78, 2008.

[4] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[5] Asunción Moreno, Dolors Poch, Antonio Bonafonte, Eduardo Lleida, Joaquim Llisterri, José B. Mariño, and Climent Nadeu. Albayzin speech database: design of the phonetic corpus. In *EUROSPEECH*, pages 175–178. ISCA, 1993.

[6] Lawrence Rabiner and Biing-Hwang Juang. *Fundamentals of Speech Recognition.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[7] Steve J. Young, D. Kershaw, J. Odell, D. Ollason, V. Valtchev, and P. Woodland. *The HTK Book Version 3.4*. Cambridge University Press, 2006.

[8] Android vs IOS 2019 https://deviceatlas.com/blog/android-v-ios-market-share

[9] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley & Sons, 2009.

[10] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*, John Wiley & Sons, 2000.

[11] Reglas de UML https://holub.com/uml/

[12] Documentación de Android https://developer.android.com/

[13] L. Barnes and D. Shimberg, *Client/Server & Beyond: Strategies for the 21st Century*, Prentice Hall, 1997.

[14] Pressman, R., *Ingeniería del Software: Un enfoque práctico*, 6ª edición, McGraw-Hill, 2006.