UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Máster en Ingeniería de Computadores y Redes

Trabajo Fin de Máster

Car-me Diagnosis:

Sistema de diagnóstico de vehículos mediante la integración de aplicaciones móviles y tecnología OBDII

Autor: *María Savall García*

Director(es): *Juan Carlos Cano, Carlos Tavares Calafate*

Julio 2018

Índice de Figuras

Índice de Tablas

Índice de Algoritmos

Índice General

Abstract

Currently, in the age of the Internet of Things, most manufactured vehicles allow us to use and install applications on their integrated multimedia screens. This characteristic, together with the OBDII interface that vehicles have been integrating for more than a decade, offers the user many new possibilities that were previously not contemplated.

This work takes advantage of the advances mentioned above to develop a software module for the SmartCar application that is responsible for the diagnosis of errors. Such application will be installed in the multimedia terminal of the vehicle, and through Bluetooth technology it will communicate with the ELM327 interpreter of the OBDII protocol. For this purpose, we surveyed the state of the art for similar diagnostic applications, which allowed us to have an idea of the functionalities to take into account when making the implementation. For the development of this project the Android language, based on Java, has been used. Then, for the validation of our software module, a test plan has been made that allowed us to carry out the tests proofs on the application in a cyclical way, arranging in each cycle the errors obtained in the tests of the previous cycle, until obtaining a fully functional block.

The error diagnosis developed obtains the errors or PIDs that the OBDII protocol has detected in the vehicle, showing the user all the information related to them. By obtaining these data it becomes easier to know the status of a vehicle, allowing us to manage their maintenance more effectively. Likewise, we can manage errors from the application by deleting them, both in the application and in the vehicle, when the user wishes.

Key words: Android, OBDII, ELM327, Diagnosis, Bluetooth, PIDs, SmartCar

Resumen

Actualmente, en la época del Internet de las cosas, la mayoría de los vehículos fabricados permiten utilizar e instalar aplicaciones en sus pantallas multimedia integradas. Esta característica, junto al interfaz OBDII que integran los vehículos desde hace más de una década, ofrece al usuario nuevas posibilidades que anteriormente ni se contemplaban.

Este trabajo aprovecha los avances mencionados anteriormente para desarrollar un módulo software para la aplicación SmartCar encargado del diagnóstico de errores. Esta aplicación se instalará en la terminal multimedia del vehículo y, mediante la tecnología Bluetooth, se comunicará con el intérprete ELM327 del protocolo OBDII. Para ello, se ha realizado el análisis del estado del arte de las aplicaciones de diagnóstico similares. Esto ha permitido tener una idea de las funcionalidades a tener en cuenta a la hora de realizar la implementación. Para su desarrollo se ha utilizado el lenguaje Android, basado en Java. Seguidamente para la validación de este bloque se ha realizado un plan de test que ha permitido realizar las pruebas de testeo sobre la aplicación de forma cíclica, solucionando en cada ciclo los errores obtenidos en las pruebas del ciclo anterior, hasta obtener un bloque completamente funcional.

El diagnóstico de errores desarrollado obtiene los errores o PIDs que el protocolo OBDII ha detectado en el vehículo, mostrando al usuario toda la información relacionada con ellos. La obtención de estos datos facilita poder conocer el estado de nuestro vehículo, permitiendo gestionar de forma más eficaz su mantenimiento. Así mismo, es posible gestionar desde la aplicación los errores pudiendo borrarlos, tanto de la aplicación como del vehículo, cuando el usuario desee.

Palabras clave: Android, OBDII, ELM327, Diagnóstico, Bluetooth, PIDs, SmartCar

1. Introducción

Mediante este trabajo, se pretende realizar un estudio y posterior implementación de un módulo de la aplicación Android SmartCarsNet que se encargue de diagnosticar el estado de un vehículo mediante la herramienta OBD II a través de una conexión Bluetooth.

Este tipo de módulos permiten al usuario, básicamente, llevar un registro en tiempo real de los diferentes parámetros medibles de un vehículo, como pueden ser los diferentes códigos de error experimentados por el automóvil, la aceleración y los litros de gasolina utilizados por kilómetro recorrido entre otros. Es decir, permiten obtener y gestionar valores de cualquiera de los sensores instalados en el vehículo para posteriormente mostrarlo a los usuarios de forma amigable en la aplicación.

Todos los datos que la aplicación muestra son recogidos accediendo a los sensores que se encuentran en el vehículo mediante la herramienta de lectura de sensores OBD II, y transferidos posteriormente a la aplicación mediante el sistema Bluetooth. La aplicación será la encargada de tratar y gestionar posteriormente los datos obtenidos.

La aplicación SmartCarsNet, en su conjunto, pretende ser una red social dirigida a conductores. El módulo desarrollado en este trabajo permite al usuario realizar un diagnóstico del estado del vehículo, mediante el análisis de los diferentes PID de error que proporciona la interfaz OBD II. Aunque incluye muchas otras funciones como la función GPS, encargada de guiar a los conductores por las diferentes rutas para llegar al destino, indicándo al mismo tiempo si se está utilizando un tipo de conducción eficiente o no. Igualmente incluye el registro del mantenimiento del vehículo, que indicará las revisiones que se le han pasado al automóvil, así como cuando es necesario realizar una nueva revisión. El registro de los repostajes realizados y su precio, lo que permite conocer las gasolineras en las que se ha repostado y cuáles tenían el carburante más barato, permitiendo también que se sepa dónde se puede repostar de forma más económica y, en el caso de tener que realizar un viaje, permite trazar una ruta de repostajes económicos basándose en los datos de otros usuarios que han realizado el trayecto anteriormente. Además, dispone de muchas otras funcionalidades que se explicarán en el capítulo correspondiente.

1.1. Motivación

La tecnología Android es una de las tecnologías que actualmente se encuentran a la vanguardia del desarrollo en cuanto a sistemas operativos móviles se refiere. Además, la integración de este tipo de tecnología en los automóviles es una realidad que cada día está más presente. Aunque la mayor parte de las centralitas disponibles en los vehículos que se encuentran actualmente en el mercado suelen estar pre-programadas y cerradas a la inclusión de nuevo software por parte de los usuarios, ya se empiezan a encontrar modelos de automóviles con centralitas Android que permiten al usuario conectar sus dispositivos móviles y hacer uso de las aplicaciones allí descargadas.

Durante el desarrollo de este proyecto se espera poder ampliar el conocimiento sobre esta tecnología, así como también sobre el conocimiento de los fallos sufridos por los vehículos y la forma en la que la herramienta OBD II es capaz de detectarlos y transmitirlos mediante Bluetooth a la aplicación encargada de gestionarlos y mostrarlos, para así poder ponerlos en práctica, tanto durante la parte práctica de este proyecto cómo posteriormente en mi trabajo diario como desarrolladora de aplicaciones.

1.2. Objetivos

El principal objetivo que se quiere alcanzar en este trabajo es la creación de un módulo de diagnóstico para la aplicación SmartCarsNet que sea funcional y fácil de usar para el usuario, y que le permita, en una única aplicación, poder gestionar desde la propia terminal del automóvil o desde un dispositivo móvil o tablet todas las necesidades relacionadas con la gestión de los errores de su vehículo.

Los objetivos de este trabajo son los siguientes:

- La creación de un módulo amigable y sencillo con una interfaz cuidada.
- La presentación de los códigos de error que ha sufrido el vehículo de forma sencilla y clara a los usuarios.
- La presentación, de forma clara, de la gravedad que supone la avería sufrida por el vehículo.
- La explicación de las consecuencias que puede ocasionar la no reparación de dicha avería.
- La presentación de la información sobre el coste medio de la reparación.
- Mostrar dónde se encuentra ubicada la avería.

- Obtener la información de los talleres de reparación cercanos.
- Ofrecer la posibilidad de recordatorios de reparaciones vía whatsapp o SMS.
- Hacer un estudio sobre la posibilidad de resetear los indicadores del fallo en el salpicadero del vehículo.

1.3. Estructura de la memoria

El resto de la memoria está organizada como sigue:

El capítulo 2 presenta un estudio realizado a partir de otras aplicaciones similares, disponibles para dispositivos móviles Android, IOS o Windows Phone, que ha permitido realizar un estado del arte sobre lo que había ya en el mercado, así como, conocer las posibles deficiencias que hiciera falta suplir. Se ha tenido en cuenta estos antecedentes como punto de partida de la aplicación.

El capítulo 3 explica el contexto tecnológico utilizado para el desarrollo de la aplicación. Concretamente, el interfaz OBDII, que es la tecnología encargada de realizar la lectura de los diferentes sensores ubicados en el vehículo y, posteriormente, enviar los datos recogidos a la aplicación móvil. Estos datos serán enviados a la aplicación SmartCarsNet encargada del tratamiento y la gestión de los datos mediante la tecnología de comunicación conocida como Bluetooth. Finalmente se utilizará la herramienta de desarrollo de Android Studio, para el desarrollo funcional y de la interfaz de la aplicación.

El capítulo 4 muestra la visión general de la aplicación SmartCarsNet, ahondando en las diferentes funcionalidades de las que dispone la aplicación en cada bloque desarrollado, así como, la forma de cubrir las diferentes necesidades que actualmente tienen los usuarios y que no se encuentran cubiertas por otras aplicaciones similares que ya se encuentran disponibles en el mercado.

El capítulo 5 analiza las diferentes ampliaciones propuestas e implementadas para la aplicación SmartCarsNet, centrándose en el diagnóstico del estado del vehículo. Así mismo, muestra y explica diferentes ejemplos de cómo se ha llevado a cabo la implementación. Incluye, entre otros, un ejemplo de petición de conexión por Bluetooth y un ejemplo de petición de datos desde la aplicación al interfaz OBDII.

El capítulo 6 explica y valida las implementaciones propuestas y realizadas a lo largo de este trabajo. Para ello se diseñaron una serie de pruebas de test que se explicarán en este capítulo, comentando además los resultados obtenidos una vez realizadas dichas pruebas.

El capítulo 7 es un breve resumen del trabajo que se ha llevado a cabo, así como de las futuras ampliaciones que se le podrían realizar a la aplicación SmartCarsNet y de las impresiones y conclusiones a las que se ha llegado durante la realización de este trabajo.

El Apéndice A es un manual de usuario del módulo de diagnóstico implementado de la aplicación SmartCarsNet que explica de forma sencilla y muestra mediante capturas de pantalla, cómo hacer uso de la aplicación implementada de una forma amigable y fácil para el futuro usuario de la aplicación.

El Apéndice B es una lista de los comandos AT reconocidos por el OBDII para su configuración. Estos están divididos en seis grupos dependiendo de sus características.

El Apéndice C contiene a lista de todos los modos en los que el OBDII puede trabajar, así mismo se explican de forma detallada los PIDs que forman parte de cada modo, centrándose en los PIDs utilizados en el desarrollo de la aplicación.

Finalmente, el Apéndice D contiene la implementación de todos los algoritmos utilizados para la implementación del módulo de diagnóstico de la aplicación.

2. Antecedentes

En este capítulo se presentan trabajos relacionados con este proyecto. Concretamente, se centra en aplicaciones Android similares a la que se quiere implementar, ya que, mediante su estudio, se pueden obtener ideas de cómo organizar e implementar ciertas funcionalidades, y se puede observar si hay alguna funcionalidad en la que no se haya pensado y que se pueda incorporar al módulo de la aplicación a implementar. Además, permite identificar funcionalidades que sería interesante incluir y que actualmente no se encuentra disponible en el mercado.

Para facilitar el estudio y la toma de decisiones sobre lo que se quiere incluir en la aplicación, se ha reducido la lista de aplicaciones similares a una lista de las seis aplicaciones Android de este tipo mejor valoradas por los usuarios. Estas aplicaciones, que se explicaran en detalle más adelante, son las siguientes [1]:

- Torque Lite
- OBD Escáner de Auto – OBD2 ELM327 coche diagnóstico
- DashCommand
- Manual del coche
- Midas Connect
- iOBD2

La finalidad de este estudio es tener una idea clara de lo que actualmente se encuentra disponible en el mercado, y conocer si todas las necesidades de los usuarios se encuentran cubiertas, o si por el contrario, hay alguna funcionalidad que se pudiera añadir a la aplicación desarrollada como parte de este trabajo, para así hacer que esta destaque de entre todas las aplicaciones similares ya disponibles.

Así mismo, se puede aprovechar este estudio para poder observar las diferentes interfaces de las que disponen las aplicaciones, tales como la organización de la información y la forma de presentársela a los usuarios, pudiendo valorar también aquellas características que más se acercan al diseño de la aplicación que se tiene en mente desarrollar.

2.1. Estudio estado del arte

A continuación, se explica, de forma detallada, las diferentes características y funcionalidades de las aplicaciones Android estudiadas:

2.1.1. Torque lite

La aplicación Torque es una aplicación para sistemas ANDROID que permite, mediante un lector ECU Bluetooth conectado al dispositivo OBDII, la monitorización de parámetros internos del motor, además de proporcionar gráficas y tiempos usando el GPS del móvil [2].

Esta aplicación dispone de dos versiones una gratuita conocida como Torque (Lite) y otra versión de pago conocida como Torque PRO [3]. Las principales características que añade la versión PRO son las siguientes:

- Se elimina la publicidad.
- Tiene soporte para plugins.
- Manejo de perfiles.
- Se pueden compartir los resultados en redes sociales.

Para hacer uso de la aplicación primeramente se debe sincronizar el teléfono mediante Bluetooth al dispositivo OBDII. Una vez sincronizados se debe configurar la aplicación con los parámetros relativos al vehículo, tales como, peso, cilindrada y rpms. Finalmente, la aplicación dispone de un “dashboard” que, previa selección de los parámetros que se quieren visualizar va llenándose con las diferentes gráficas de los parámetros seleccionados.

Así mismo, la aplicación Lite también permite ver los códigos de error de las averías sufridas por el vehículo, sin mostrar la descripción de estos; no obstante, facilita un link que permite ir a su descripción. También permite reiniciar los códigos de error sufridos [4].

También se observa en la Figura 1 que, aunque la interfaz es bastante sencilla, la elección de las gráficas puede ser un poco compleja, y los usuarios deben saber cómo leer las gráficas que les aportan los datos.

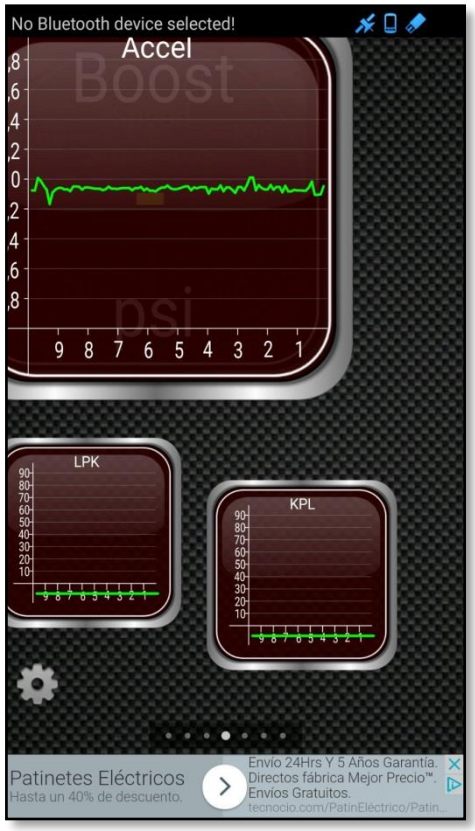

Figura 1: Torque Lite

2.1.2. OBD escáner de auto – OBD2 ELM327 car diagnostic

La aplicación OBD Escáner de Auto, cuya interfaz se puede ver en la Figura 2, es una aplicación para sistemas ANDROID que permite, mediante un lector ECU bluetooth conectado a un dispositivo OBDII, la monitorización de parámetros internos del motor, además de poder obtener datos en tiempo real sobre el vehículo.

Esta aplicación OBD escáner de Auto también dispone de dos versiones diferentes:

La versión gratuita de la aplicación permite ver el número de errores de un coche, los códigos de fallo y elegir entre más de 70 parámetros dinámicos del vehículo, tales como la velocidad, revoluciones, temperatura del refrigerante, presión del combustible… hasta dos parámetros para ser mostrados en pantalla.

La versión completa permite disponer de todas sus características. Permite ver cualquier código de error con la descripción del problema, seleccionar hasta 5 parámetros dinámicos para mostrar en pantalla, y leer los datos de Freeze Frame

(lectura del sensor en el momento en que se da el error) para ver más información sobre el vehículo en tiempo real.

Además, esta aplicación permite reiniciar los sensores para borrar los códigos de error sufridos por el vehículo. Para la utilización de esta aplicación es necesaria la sincronización de los dispositivos mediante Bluetooth, y la inclusión de los datos del vehículo a analizar.

En cuanto a la interfaz que se observa en la Figura 2, se ven de forma sencilla las diferentes opciones elegibles en blanco sobre un fondo más oscuro. Finalmente, destacada sobre un fondo rojo se encuentra la opción que permite comprar la versión completa de la aplicación.

Figura 2: OBD Escáner de Auto

2.1.3. DashCommand

DashCommand es una aplicación Android dirigida a ser utilizada en el coche junto con un dispositivo de lectura de sensores OBDII, y mediante el cual se comunica a través del Bluetooth. Esta aplicación convierte un teléfono o tableta Android en una pantalla avanzada para poder visualizar los datos del motor [5].

Entre las diferentes funcionalidades de DashCommand se encuentran las siguientes:

- Avisa de los errores sufridos por el vehículo.
- Permite conocer la potencia, torque o aceleración, entre otros, en tiempo real.
- Monitoriza y mejora el consumo de combustible mientras se conduce.
- Permite personalizar los diferentes parámetros a visualizar.
- Contiene un ordenador de viaje sofisticado que mantiene las estadísticas de hasta 5 viajes.

No obstante, hay que saber que la versión gratuita de la aplicación es sólo de evaluación, por lo que sus funcionalidades se verán reducidas, y una vez terminado el periodo de prueba se deberá comprar la aplicación completa para poder usarla.

Para la utilización de esta aplicación es necesaria la sincronización de los dispositivos mediante Bluetooth, y la inclusión de los datos del vehículo a analizar.

En cuanto a la interfaz de la aplicación que se puede ver en la Figura 3, se observan las diferentes opciones a seleccionar en blanco sobre un fondo negro que aporta claridad y facilidad de comprensión al usuario y, al mismo tiempo, le da un toque moderno. También se puede ver que la forma de advertir al usuario de si está conectado al OBDII o no, es una esfera en la parte baja de la aplicación con un código de colores, anaranjado/verde, y letras centrales muy fácil de comprender [6].

Figura 3: DashCommand

2.1.4. Manual del coche

Manual del coche es una aplicación muy útil y práctica, que le permite tanto encontrar una solución a un problema concreto del vehículo, como realizar un diagnóstico completo y también mantenerse al día con el mantenimiento del vehículo. Esta aplicación Android, no utiliza el dispositivo OBDII para funcionar, sino que, como su nombre indica es una guía completa sobre el vehículo que permite de forma manual llevar un recuento de la gasolina utilizada y los gastos que ha supuesto el mantenimiento del vehículo.

Las principales funcionalidades de esta aplicación son:

- Guardar información útil sobre el mantenimiento del automóvil.
- Aprender con la ayuda de las guías como poder solucionar ciertos problemas sin necesidad de llevar el coche al taller.

- Establecer alarmas para que avise sobre cuando es necesario hacer el mantenimiento del vehículo.
- Permite guardar los elementos específicos de su vehículo, para así, crear un manual personalizado por cada usuario.

En cuanto a la interfaz se observa en la Figura 4 es una aplicación ligera, rápida, intuitiva, con un diseño muy funcional que sigue los patrones de diseño de Android.

Figura 4: Manual del coche

2.1.5. MidasConnect

Midas, la empresa de mantenimiento de automóviles, ha creado MidasConnect. Disponible para iOS y Android, esta aplicación pretende ser la herramienta perfecta para la monitorización del vehículo.

Entre las diferentes funcionalidades de Midas Connect se encuentran [7] [8]:

- La localización en tiempo real del vehículo (Geolocalizador).
- El análisis con precisión de los trayectos.
- Procesamiento de información de las etapas clave del mantenimiento de tu vehículo.
- Controla el estado general de tu vehículo.
- Si eres cliente MIDAS permite acceder a los documentos de tu taller Midas y a los documentos de tu taller favorito.
- Recibe consejos útiles en función del uso.

Los puntos diferenciales de esta aplicación respecto de los estudiados al son los siguientes:

- Es imprescindible el registro del usuario.
- La aplicación es completamente gratuita.
- No es una aplicación únicamente de diagnóstico y obtención de parámetros en tiempo real, si eres cliente MIDAS te permite gestionar diferentes aspectos del mantenimiento de tu vehículo a través de la aplicación.
- Lleva un registro que facilita el control del mantenimiento del vehículo.

Aunque la aplicación es gratuita hay que tener en cuenta que esta aplicación funciona mediante un dispositivo que Midas instala en tu vehículo debiendo pagar por ello 59,95€.

Para poder hacer uso de la aplicación los pasos a seguir son los siguientes [9] [10]:

- Ir a un centro Midas.
- En el centro se instalará el dispositivo en el interior del vehículo.
- Descargar la aplicación Midas Connect.
- En el centro sincronizan la aplicación con el dispositivo del vehículo y te enseñan a usarlo.

En cuanto a la interfaz, se observa en la Figura 5 que está dividida en diferentes secciones de forma clara, amigable y fácil de utilizar para el usuario.

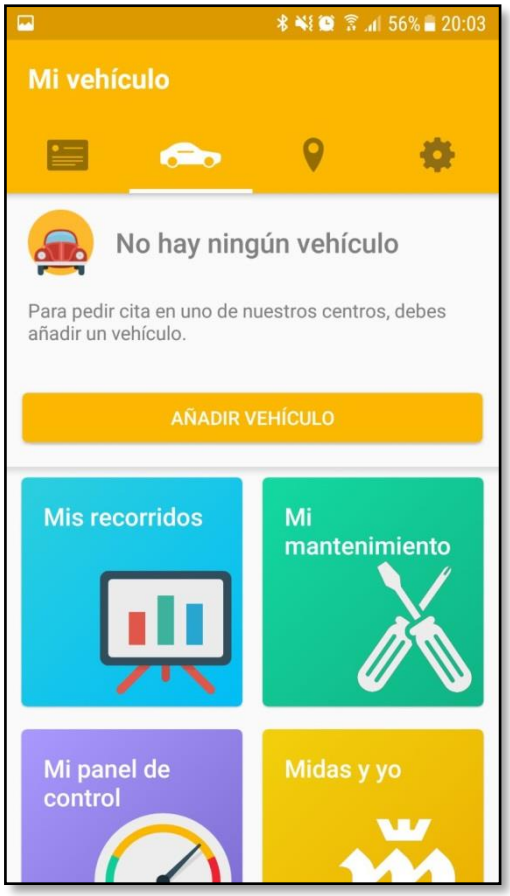

Figura 5: MidasConnect

2.1.6. iOBD2

iOBD2 es una aplicación Android y iOS dirigida a ser utilizada en el coche junto con un dispositivo de lectura de sensores OBDII y mediante el cual se comunica a través del Bluetooth. Esta aplicación convierte un teléfono o tableta Android/iOS en una pantalla avanzada para poder visualizar los datos del motor.

Esta herramienta que no es de pago y no incluye publicidad tiene entre sus principales funcionalidades las siguientes:

- Permite hacer un diagnóstico de los problemas que sufre el automóvil y luego pedir ayuda por Twitter o Facebook a otros usuarios.
- Se puede obtener información en tiempo real, como el consumo de combustible, par, o aceleración… en el teléfono.

- La aplicación permite personalizar sus medidores y después de seleccionar los datos que interesan, sólo se mostrará aquellos datos en los que se esté interesado.
- Mantiene un histórico con los diferentes diagnósticos y monitorizaciones en tiempo real.
- Permite hacerle pruebas de rendimiento al vehículo [11].

En cuanto a la interfaz, Figura 6, en este caso se observa que se trata de una aplicación ligera, simple, amigable y fácil de utilizar por el usuario.

Figura 6: iOBD2

2.2. Tabla Comparativa

En este apartado se puede consultar una tabla comparativa, Tabla 1, dónde se relacionan las diferentes características que presentan las aplicaciones estudiadas y las características que se desea que tenga la aplicación, pudiendo ver así de una forma clara las funcionalidades que comparten todas las aplicaciones, así como aquellas que solo se encuentran disponibles en unas pocas, o las que no están disponibles en ninguna de las estudiadas.

A continuación, se asigna una letra a cada aplicación que será la utilizada en la tabla comparativa, parte superior, para referirse a dicha aplicación:

A. Torque Lite
B. OBD Escáner de Auto – OBD2 ELM327 car diagnosis
C. DashCommand
D. Manual del coche
E. MidasConnect
F. iOBD2

En la parte izquierda de la tabla hay una lista de todas las características que se desea que tenga la aplicación a implementar. A la derecha, marcado mediante una X se indica en que aplicaciones de las estudiadas se encuentran disponibles esas características:

Características	*A*	*B*	*C*	*D*	*E*	*F*
Incluye Publicidad	X			X		
App exclusiva de diagnóstico	X	X				
Interfaz sencilla, cuidada y fácil de utilizar		X	X	X	X	X
Información sobre los errores clara	X	X			X	X
Información clara sobre la gravedad de la avería sufrida				X		
Información de las consecuencias de no reparar la avería				X		
Información sobre el coste medio de la reparación						
Información de la ubicación de la avería				X		
Información sobre los talleres de reparación cercanos					X	

Recordatorios de reparaciones/mantenimiento vía Whatsapp o SMS						X
Posibilidad de resetear los indicadores de fallo desde la aplicación	X	X	X			X

Tabla 1: Tabla comparativa de la revisión del estado del arte

2.3. Conclusiones

Lo primero que se observa es que la mayoría de las aplicaciones estudiadas sí que cumplen el requisito de tener una interfaz ligera, clara, amigable y fácil de utilizar por el usuario.

También hay que destacar que ninguna de las aplicaciones da información sobre el coste medio de la reparación a efectuar, por lo que se puede considerar que esta funcionalidad puede aportar un valor añadido a nuestra aplicación.

Finalmente, destacar que, aunque entre las seis aplicaciones en conjunto se cubren la mayoría de las funcionalidades deseadas, ninguna de ellas cubre los requisitos en su totalidad. Por ello, se necesita tener un mínimo de dos aplicaciones instaladas en nuestro dispositivo Android para cubrir todas las necesidades requeridas.

En este caso, se intenta implementar una aplicación que permita incluir todas estas funcionalidades, junto a una interfaz ligera, clara, amigable y fácil de utilizar que permita al usuario tener todas sus necesidades cubiertas en una sola aplicación, ahorrando así espacio de almacenamiento en el teléfono y tiempo a la hora de consultar los datos que se necesiten, debido a que todo lo necesario se encuentra disponible en una única aplicación, y no se debe estar cambiando constantemente entre dos aplicaciones para poder hacer las consultas.

3. Contexto tecnológico

En este capítulo se habla del contexto tecnológico necesario para llevar a cabo el diseño e implementación de la aplicación propuesta, así como, la tecnología de comunicación empleada por las diferentes herramientas utilizadas.

Para el desarrollo e implementación de la aplicación, se ha hecho uso del programa de desarrollo Android Studio, el cual utiliza Java como lenguaje nativo. Igualmente, se ha utilizado el estándar para diagnóstico del motor basado en OBDII, y la tecnología inalámbrica de comunicación Bluetooth.

A lo largo de este capítulo, se explican las diferentes características y usos de cada uno de los elementos utilizados para el desarrollo de la aplicación. Para ello, primero se explican la herramienta utilizada para el desarrollo Android Studio, así como, el lenguaje utilizado para la implementación Java. Posteriormente, se explica la herramienta de diagnóstico OBDII, y finalmente se detalla el funcionamiento del Bluetooth, la tecnología inalámbrica de comunicación que permite conectar ambos dispositivos: la aplicación Android ubicada en un terminal móvil y el OBDII.

3.1. Sistema operativo Android

Android es una plataforma creada por la empresa Google y basada en el sistema operativo Linux, por tanto de código abierto, lo que hace que este software sea libre y gratuito. Aunque en un inicio se pensó para teléfonos móviles, actualmente ofrece soporte multiplataforma.

Las aplicaciones destinadas a ser instaladas en Android pueden ser desarrolladas en una variante de Java conocida como Dalvik, lo que hace que el desarrollo de éstas sea sencillo. Esto, unido al hecho de que las herramientas de desarrollo son gratuitas, repercute en el número de las aplicaciones desarrolladas, que cada día es mayor.

Android fue implementado por los desarrolladores Rich Miner, Nick Sears, Chris White y Andy Rubin en el año 2003. Durante los dos años posteriores fue un sistema operativo casi desconocido, hasta que en 2005; la compañía Google compró el sistema operativo, pasando los desarrolladores iniciales a formar parte de esta empresa. Sin embargo, no fue hasta 2007 que Android se dio a conocer en el consorcio tecnológico Open Handset Alliance, formado por empresas como Samsung, Qualcomm, Google y HTC, entre otras. Finalmente, en 2008 se lanzó al

mercado el primer teléfono con sistema operativo Android, el teléfono HTC Dreams. En el último trimestre de 2010, Android se posicionó como el sistema operativo de móviles más vendido del mundo, y desde entonces el sistema ha llegado a todo tipo de plataformas y se ha ido actualizando, tomando como referencia para el nombre de cada actualización un postre o dulce reconocible [12].

Actualmente, la calidad ofrecida por Android presenta niveles muy altos, debido a las diferentes versiones y mejoras hechas a lo largo de los años. A continuación se ve brevemente las últimas versiones disponibles del sistema operativo Android [13] [14]:

- Android Ice Cream Sandwich: Es la versión 4.0 de Android. Se modifica el sistema de corrección del teclado virtual, permite crear carpetas en la pantalla de inicio con solo arrastrar un icono a otro, se añade el soporte para tecnología NFC y el desbloqueo facial.
- Android Jelly Bean: Es la versión 4.1.2 de Android. Aporta mejora de errores y estabilidad.
- Android Kit Kat: Es la versión 4.4 de Android. Mejora la conectividad con soporte NFC y facilita el acceso a las aplicaciones en la nube.
- Android Lollipop: Es la versión 5.0 de Android. Permite el uso de Android en pantallas grandes y pequeñas, y mejora la respuesta táctil de la pantalla.
- Android Marshmallow: Es la versión 6.0 de Android. Disminuye el consumo de batería por parte de las aplicaciones aumentando así su vida útil, e introduce además una nueva forma de gestionar los permisos dando al usuario mayor control sobre su dispositivo móvil.
- Android Nougat: Es la versión 7.0 de Android, y la más actual. Añade nuevos emojis y la posibilidad de configurar varios idiomas para el teclado. Además, se pueden utilizar aplicaciones en paralelo.

Una vez vistas las diferentes versiones por las que Android ha ido pasando, vamos a ver las características actuales que esta herramienta ofrece. Para empezar, se parte de la base de que este sistema operativo proporciona una interfaz amigable y fácil de utilizar para los usuarios, aunque, debido a que es una plataforma abierta, los fabricantes y las operadoras pueden modificar esta base añadiendo sus propios diseños. Además, al ser código abierto, no hay coste de desarrollo e instalación en un dispositivo, es gratuito. También se ha incluido el Cloud Computing en la plataforma, y es compatible con muchas de las políticas de seguridad aplicadas en el desarrollo de aplicaciones. Por lo que, se habla de una tecnología a la vanguardia del desarrollo y líder en su sector [14] [15].

3.2. Android Studio

El programa Android Studio es el entorno de desarrollo que se utiliza para la implementación de nuestra aplicación.

Android Studio es un entorno oficial de desarrollo integrado (IDE) para la plataforma Android que utiliza la licencia de software libre Apache 2.0. Este IDE, programado en Java y multiplataforma, está basado en otro entorno de desarrollo conocido como IntelliJIDEA, creado por la compañía JetBrains. Debido a esto, el IDE proporciona mejoras respecto al plugin ADT (Android Developer Tools) diseñado para ser utilizado en el entorno de desarrollo Eclipse [16].

En otras palabras, Android Studio es el escritorio de trabajo del desarrollador Android, ya que ahí es donde se encuentran los proyectos en curso de desarrollo, las carpetas y archivos que lo componen, y todo lo necesario para terminar la implementación de la aplicación.

El IDE Android Studio fue creado por la empresa Google y presentado el 16 de mayo de 2013 en el congreso de desarrolladores Google I/O, con el objetivo de reemplazar a Eclipse, la plataforma utilizada en aquel momento para el desarrollo de aplicaciones Android, y que continua siendo utilizada por muchos desarrolladores actualmente, por otro entorno dedicado en exclusiva a la programación de aplicaciones para dispositivos Android, consiguiendo así Google un IDE propio que le permitiera un mayor control sobre el proceso de producción.

Aunque Android Studio estuvo durante casi un año en versión beta, el 8 de diciembre de 2014 se liberó la versión estable de Android Studio 1.0. Desde entonces, Google ha decidido recomendar este IDE como IDE principal para el desarrollo de aplicaciones Android, dejando de estar en desarrollo activo el plugin ADT para Eclipse.

En la actualidad, Android Studio es la plataforma que se postula como el más completo IDE para desarrollar aplicaciones Android con muchas características que destacan de los otros programas usados para este trabajo.

Las características son las siguientes [17]:

- Permite programar aplicaciones para Android Wear (Sistema operativo utilizado en los dispositivos corporales, como por ejemplo los relojes)
- Utiliza ProGuard para optimizar y reducir el código del proyecto al exportar a APK.
- Permite la edición de temas.

- Interfaz dirigida al desarrollo Android.
- Permite la importación de proyectos realizados en el entorno Eclipse.
- Permite realizar control de versiones.
- Alerta en tiempo real de errores sintácticos, compatibilidad o rendimiento antes de compilar la aplicación.
- Se puede ver la vista previa de como quedarán las aplicaciones en diferentes dispositivos y resoluciones.
- Se integra con Google Cloud Platform, para el acceso a los diferentes servicios que proporciona Google en la nube.
- El editor de diseño muestra una vista previa de los cambios realizados directamente en el archivo xml.
- Se renderiza en tiempo real.
- Proporciona una consola de desarrollador con consejos de optimización, ayuda para la traducción y estadísticas de uso.

Las funcionalidades que Android Studio ofrece para aumentar su productividad durante la compilación de sus aplicaciones son las siguientes [18]:

- Integra el sistema de compilación Gradle flexible para gestionar y automatizar la creación de proyectos.
- Facilita Herramientas Lint (detectan código no compatible entre arquitecturas diferentes, o código confuso que no es capaz de controlar el compilador) para detectar problemas de rendimiento, usabilidad y compatibilidad de versiones.
- Dispone de un emulador rápido con varias funciones.
- En el entorno se pueden realizar desarrollos para todos los dispositivos Android.
- Proporciona la herramienta Instant Run que permite aplicar cambios mientras la aplicación se ejecuta sin la necesidad de compilar un nuevo APK.
- Dispone de gran cantidad de herramientas y frameworks de prueba.
- Compatibilidad con C++ y NDK.

Además, cuenta con una estructura simple que facilita a los usuarios la organización de los proyectos, facilitando su ubicación y publicación, y les proporciona un entorno de desarrollo potente, fácil e intuitivo.

Figura 7: Android Studio

Para ser conscientes realmente de lo que significa esta nueva herramienta, a continuación, se muestra una tabla comparativa, Tabla 2, del IDE Android Studio y el ADT para Android del IDE Eclipse [19]:

Características	*Android Studio*	*ADT*
Sistema de Compilación	Gradle	ANT
Utilización Maven	Sí	No (plugin auxiliar)
Construcción variantes y generación de múltiples APK	Sí	No
Refactorización	Sí	No
Diseño del editor gráfico	Sí	Sí
Firma APK y gestión de almacén de claves	Sí	Sí
Soporte para NDK (Native Development Kit: herramientas para implementar código nativo escrito en C y C++)	Sí	Sí
Soporte para Google Cloud Platform	Sí	No
Vista en tiempo real de renderizado	Sí	No
Nuevos módulos en proyecto	Sí	No
Editor de navegación	Sí	No
Generador de assets	Sí	No
Datos de ejemplo en diseño de layout	Sí	No
Visualización de recursos desde editor de código	Sí	No

Tabla 2: Diferencias Android y ADT

Aunque actualmente muchos desarrolladores continúen prefiriendo Eclipse para el desarrollo de aplicaciones Android, la nueva opción que brinda Google cada día va ganando más adeptos debido a la estabilidad que aporta, y a las nuevas funcionalidades que facilitan en gran medida el trabajo al desarrollador.

Las ventajas que aporta este IDE de la compañía Google respecto a otros son las siguientes [18]:

- Facilita la creación de nuevos módulos dentro de un mismo proyecto, sin tener que cambiar para ello de espacio de trabajo.
- Un solo IDE aporta todas las herramientas necesarias para la creación de nuevas aplicaciones Android.
- El uso del Gradle en la construcción del apk aporta las siguientes ventajas más acordes a un proyecto Java:
    - Facilita el trabajo en equipo, facilitando para ello la distribución del código.
    - Se puede compilar desde línea de comandos.
    - Facilita la creación de versiones diferenciadas.
    - Permite reutilizar el código y los recursos.

Los requerimientos mínimos que necesita un equipo para la correcta instalación y utilización de la herramienta Android Studio son los siguientes [17] [19]:

Windows	MAC OS	Linux
Microsoft Windows 8/7/Vista/2003 (32 o 64 bit)	Mac OS X 10.8.5 o superior, hasta la 10.9	GNOME o entorno de escritorio KDE GNU Library C 2 15 o superior
Mínimo de 2 GB de RAM, recomendado 4 GB de RAM		
400 MB de espacio en disco		
Necesita de al menos 1 GB para Android SDK, emulador de imágenes del sistema, y cachés		
Resolución mínima de pantalla de 1280 x 800		

Java Development Kit (JDK) 7 o superior

Tabla 3: Requisitos instalación

Para instalar la aplicación lo primero hay que hacer es descargarla. Para ello se utiliza el siguiente URL:

https://developer.android.com/studio/index.html?hl=es-419

Para la instalación en Windows, una vez descargado se deberá hacer doble click sobre el ejecutable y seguir los pasos del instalador.

En sistemas con Windows 7 o Windows 8, puede haber problemas durante la instalación o al iniciar el programa debido a que este no encuentra la ruta en la que se encuentra Java instalado. Para solucionarlo se deben seguir unos sencillos pasos:

1. Posicionarse en la ruta menú Inicio → Equipo → Propiedades del Sistema → Configuración Avanzada del Sistema → Opciones Avanzadas → Variables de Entorno.
2. En el cuadro de Variables del Sistema, pulsar sobre Nueva y Añadir:
    a. JDK_HOME en la casilla de "Nombre de la variable"
    b. El directorio donde se encuentre Java instalado

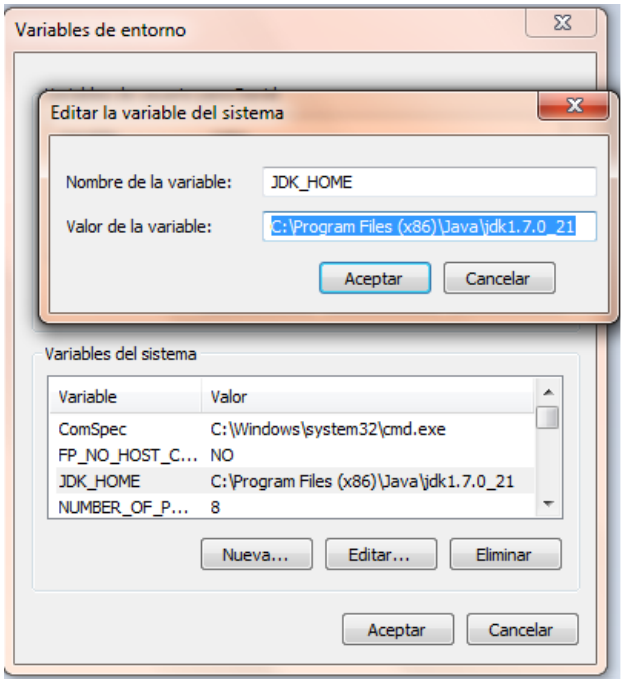

Figura 8: Variables de entorno

Para la instalación en MAC, una vez descargado bastará con abrir el archivo con extensión DMG y mover la carpeta creada a la carpeta de aplicaciones.

Para la instalación en Linux, una vez descargado se descomprimirá el archivo con extensión TGZ en el directorio deseado y se ejecutará mediante el fichero *studio.sh* que se encuentra en la ruta "android-studio/bin/studio.sh".

Una vez la instalación está hecha se puede empezar a crear los proyectos. Para ello se debe conocer la estructura que un proyecto Android posee. Se puede ver en la Figura 9 [16]:

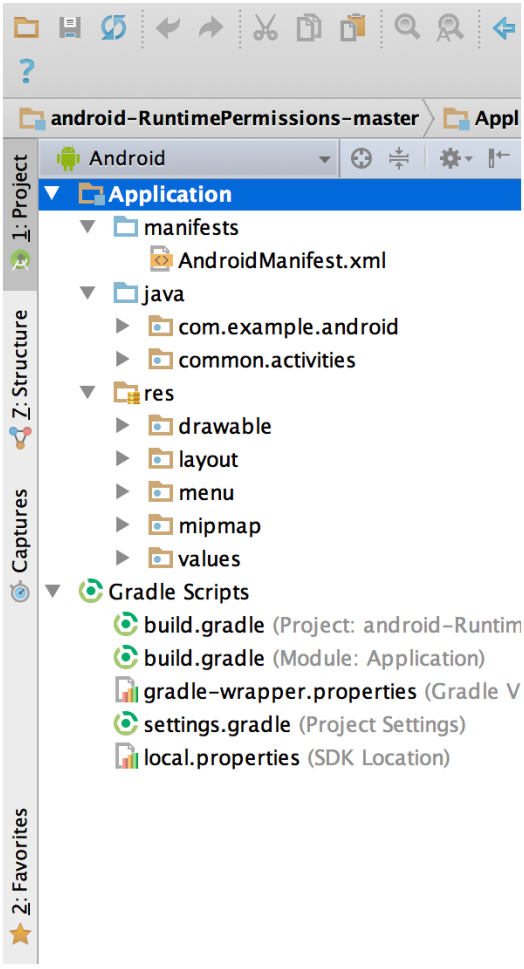

Figura 9: Vista proyecto

Como se observa, los proyectos Android en el IDE Android Studio se distribuyen en módulos, donde se encuentran los diferentes recursos necesarios para la creación y funcionamiento de éste. Se diferencian tres tipos de módulos [18]:

- Módulos de aplicaciones para Android.
- Módulos de bibliotecas.

- Módulos de Google App Engine

La manera predeterminada de Android Studio de mostrar los recursos de un proyecto es la que se puede observar en la Figura 10. Esto facilita al usuario un rápido acceso a los recursos.

Aunque esta sea la manera de mostrar los recursos al usuario no significa que en el disco los recursos se almacenen igual. Para ver la estructura de archivos real del proyecto se debe seleccionar la opción "Project" de la lista desplegable Android que se muestra en la Figura 10.

Una vez vista la estructura de un proyecto, centrémonos en la interfaz de usuario que los desarrolladores deberán de saber utilizar. A continuación, la Figura 10 muestra una imagen de esta [18]:

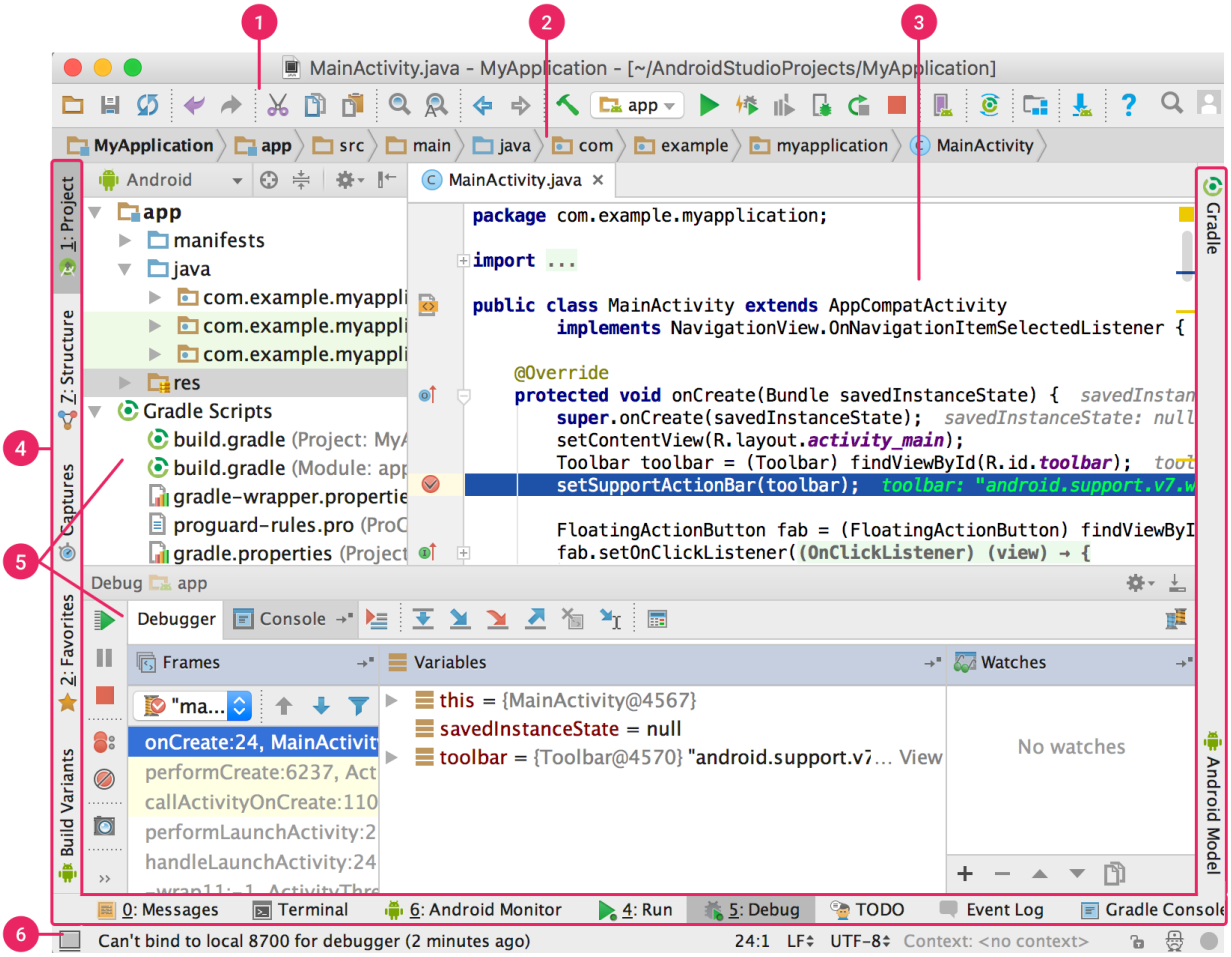

Figura 10: Interfaz Android

1. La **barra de herramientas**: permite realizar acciones sobre el código ejecutado.

2. La **barra de navegación**: permite navegar entre los diferentes recursos del proyecto.
3. La **ventana del editor**: área dedicada a la creación y la modificación de código.
4. La **barra de la ventana de herramientas**: contiene los botones que te permiten expandir o contraer ventanas de herramientas individuales.
5. Las **ventanas de herramientas**: permiten acceder a tareas específicas.
6. La **barra de estado**: muestra el estado del proyecto, así como mensajes y advertencias.

Todas las barras y herramientas se pueden mostrar y ocultar a placer para que el desarrollador pueda organizar el espacio de trabajo a su gusto. Además, la mayoría de las funciones son accesibles mediante una combinación de teclas [18].

3.3. Java

Los lenguajes de programación son lenguajes formales mediante los cuales los seres humanos pueden decirle a las máquinas que deben hacer, es decir, son lenguajes diseñados para describir un conjunto de acciones que deben ser llevados a cabo por máquinas como los ordenadores. El lenguaje elegido para la implementación de nuestra aplicación es Java.

Este lenguaje orientado a objetos fue desarrollado y comercializado en 1995 por la empresa Sun Microsystems, y posteriormente adquirido por la empresa Oracle. Permite la realización de cualquier tipo de programa. Esta tecnología dispone, además del lenguaje propiamente dicho, de una máquina virtual Java que permite la ejecución del código compilado mediante este lenguaje en cualquier máquina en la que esta esté instalada. Debido a que está ampliamente extendido, y a la facilidad de utilizar estos programas en diferentes plataformas, en la actualidad se considera un lenguaje de uso común que cada vez está destacando más en la informática [20].

Las cinco características principales de Java son [21]:

- La programación orientada a objetos.
- La posibilidad de ejecutar un mismo programa en diversos sistemas operativos.
- La inclusión por defecto de soporte para trabajar en red.
- La opción de ejecutar el código en sistemas remotos de manera segura.
- La facilidad de uso.

A continuación, se va a entrar un poco en detalle en algunas de estas características:

La programación orientada a objetos (POO) es un paradigma de programación que utiliza los objetos en sus interacciones, para diseñar programas y aplicaciones informáticos. En ella se implementan diferentes objetos o clases que posteriormente se podrán reutilizar las veces que se desee.

Java permite que los programas desarrollados mediante este lenguaje se ejecuten sin ningún tipo de problema en varios sistemas operativos. Esto se debe a que, como se ha mencionado anteriormente, Java no es solamente un lenguaje con el que desarrollar, sino que se complementa con una máquina virtual Java donde se ejecutarán los programas. Esta máquina está disponible para ser instalada en diferentes sistemas operativos, lo que permite la ejecución de cualquier programa Java en estos.

En la parte del soporte para trabajar en red, Java permite la creación de pequeñas aplicaciones conocidas como applets que se incrustan en el código HTML, y pueden ser ejecutadas directamente desde un navegador. No obstante, cabe destacar que, para su correcto funcionamiento, se precisa de la instalación de un plug-in. Sin embargo, esto no supone un problema para el usuario puesto que la instalación de éste es liviana y sencilla.

En cuanto a la seguridad que Java proporciona a los usuarios, se encuentra en la misma máquina virtual donde se ejecuta el código, ya que esta es la que controla qué se puede ejecutar y de qué modo, permitiendo controlar qué pueden hacer los programas al ejecutarse en nuestro sistema [21].

No obstante, dispone de otras características, que, aunque no se encuentren dentro de las cinco principales, no son por ello menos importantes. Estas características son [22]:

- Lenguaje simple y flexible: esto se debe a que elimina algunas de las características que sí están presentes en otros lenguajes como C++. Además, permite la reutilización de código de una forma muy sencilla para el desarrollador.
- Distribuido: es decir permite fácilmente la escalabilidad, es transparente y tolerante a fallos.
- Interpretado y compilado a la vez: las instrucciones que componen el programa son compiladas a bytcodes, y éstas son traducidas por el intérprete a un lenguaje entendible por la máquina instrucción por instrucción.

- Robusto: permite crear un software altamente fiable, por lo que proporciona numerosas comprobaciones en compilación y en tiempo de ejecución.
- Multihilo: permite la ejecución de código en paralelo mediante diferentes hilos.
- Con recolector de basura.
- De alto rendimiento: con la aparición de hardware especializado su rendimiento mejora.
- Para su utilización no es necesario un desembolso económico.

El amplio uso se debe, por tanto, a todas las características anteriormente descritas, así como al gran trabajo de testeo y ajuste realizado por los desarrolladores. Por tanto, no es de extrañar que una plataforma tan ampliamente adoptada y reconocida actualmente cómo Android haga uso del lenguaje Java como lenguaje nativo en el desarrollo de sus aplicaciones.

3.4. Interfaz OBDII

Hoy en día, el mundo está cada vez más automatizado y todos los elementos de la vida cotidiana se estás informatizando cada vez más.

Actualmente, el medio de transporte más estandarizado es el automóvil que, desde 1996, ha establecido el sistema OBDII como sistema estandarizado para obtener datos del sistema de propulsión del vehículo.

Pero, ¿qué es la herramienta OBDII?

OBD o “On Board Diagnostics”, conocido en español como “Diagnósticos de abordo”, es un sistema de diagnóstico, monitoreo y control completo del motor y de otros dispositivos del vehículo que actualmente se encuentra en la mayoría de los vehículos en circulación.

Este dispositivo, que empezó con el estándar OBD-I, fue la respuesta al número creciente de dispositivos electrónicos en los vehículos, y debido a la necesidad de satisfacer las normas de emisión de los gases contaminantes EPA. Hoy en día se diferencian 3 estándares de OBD:

- OBDII: es el estándar utilizado en Estados Unidos.
- EOBD (European On Board Diagnostics): es el estándar utilizado en Europa. Se diferencia del OBD en que no se monitorizan las evaporaciones del depósito de combustible. Sin embargo, los repuestos

necesarios para esta variación son de más alta calidad y específicos para el vehículo y el modelo.
- JOBD: es el estándar utilizado en Japón.

Por tanto, el OBDII es una herramienta que contiene un estándar encargado de monitorear todos los sistemas funcionales del automóvil y, además, almacena muchos de los códigos de error que el vehículo puede sufrir [23].

La implantación de este dispositivo se debe a que, en 1979, preocupados por la calidad del aire, The Society of Automotive Engineers (SAE) recomienda que se instale en los automóviles un conector de diagnóstico estandarizado y un conjunto de señales de prueba de diagnóstico para poder llevar a cabo un control más riguroso de las emisiones.

A finales de los 80, en 1991, como consecuencia de la recomendación hecha por la SAE en 1979, el California Air Resources Board (CARB) pide que todos los vehículos nuevos vendidos en California en 1991 tengan alguna capacidad básica de OBD. Este conjunto de requisitos se conoce a posteriori como OBD-I. No obstante, esta tecnología no define el conector ni su posición dentro del vehículo, ni tampoco un protocolo de datos estandarizados, por lo que no tiene éxito. Sin embargo, debido a las alertas de *smog* que hay en los Ángeles, se decide mejorar esta tecnología y hacerla obligatoria para todos los automóviles comercializados a partir del 1 de enero de 1996 en Estados Unidos. Esta nueva tecnología seria conocida como OBDII y esta vez sí se empezaría a incluir en los nuevos vehículos.

Aunque este dispositivo tuvo éxito, en cuanto a la estandarización del conector se refiere, no lo tuvo tanto respecto a las señales enviadas por los vehículos, o lo que es lo mismo, se crearon 5 “lenguajes” diferentes: los protocolos SAE J1850 PWM, SAE J1850 VPW, ISO9141-2, ISO14230-4 e ISO 15765-4/SAE J2480, para estas señales. Aunque en general un dispositivo OBDII es capaz de leer los 5 tipos de señales, hay dispositivos especializados en un solo tipo de señal.

Actualmente ya se trabaja en el estándar OBDIII, en el cual se plantea que sean los propios automóviles los encargados de avisar a las autoridades en caso de que las emisiones de gases nocivos aumenten mientras el coche se desplaza [24].

Figura 11: Interfaz OBDII

A continuación, se ven las características de las que dispone la actual herramienta OBDII. La primera y más importante es la estandarización de la ubicación del puerto OBDII: este debe de estar a 91cm del conductor dentro de la cabina, y se debe poder manipular sin la necesidad de utilizar ninguna herramienta especial. Esta toma de contacto cuenta con espacio para 16 pines, aunque no es necesario que los tenga todos. De hecho, este factor depende exclusivamente de la marca del vehículo y del "lenguaje" que utilice.

Figura 12: Terminales del conector OBDII

Además de controlar las emisiones realizadas por los automóviles, el OBDII dispone de otras funcionalidades debido a la situación privilegiada en que se encuentra, ya que se conecta directamente a la unidad electrónica central o ECU del vehículo. Entre ellas destaca la posibilidad de monitorizar en tiempo real todo tipo de errores y de información desde el teléfono móvil mediante una aplicación destinada a leer los datos. Entre los datos disponibles para su lectura se encuentra: la mezcla aire-combustible, el velocímetro, el consumo de combustible y la potencia, entre muchos otros.

También guarda un registro de los fallos y las condiciones en las que éste se dio para poder ofrecer al mecánico la máxima información posible sobre el mismo. Para ello se asigna a cada error un código que posteriormente el mecánico puede leer con un dispositivo que envía comandos al sistema OBDII llamados PID. La conexión para el envío de estos comandos actualmente se puede llevar a cabo mediante Bluetooth, Wifi, y USB.

Como se ha explicado anteriormente hay cinco “lenguajes” o protocolos que se utilizan para la trasmisión de señales en el vehículo, pero solo tres de ellos son básicos y se encuentran en uso: el protocolo ISO 9141, utilizado por los vehículos Chrysler, los vehículos europeos y los asiáticos, el protocolo SAE J1850 VPW, utilizado por los vehículos GM y el protocolo SAE J1850 PWM, utilizado por los vehículos Ford.

Aun así, el conjunto de comandos y de códigos de error se fija según el estándar SAE J1979, utilizado por los fabricantes de automóviles. En cuanto a los códigos de error, su formato es siempre el mismo y consta de 5 caracteres, que son una letra seguida de cuatro números.

Dependiendo de la letra, el código de error está indicando lo siguiente:

- P: Tren motriz o motor y transmisión (Powertrain)
- B: Carrocería (Body)
- C: Chasis (Chassis)
- U: No definido (Undefined)

El siguiente carácter indica si el código es o no genérico:

- 0: Genérico para todas las marcas y definido por la SAE.
- 1: Específico, definido por el fabricante del vehículo, por lo que suele ser diferente en cada fabricante.

El tercer carácter indica el subsistema del vehículo:

- 0: Sistema electrónico completo.
- 1 y 2: Control combustión.
- 3: Sistema de encendido.
- 4: Control de emisión auxiliar.
- 5: Control de velocidad y ralentí.
- 6: ECU, entradas y salidas
- 7: Transmisión.

Los caracteres cuarto y quinto indican el error. La Figura 13 muestra el sistema de nombrado de códigos de error descrito:

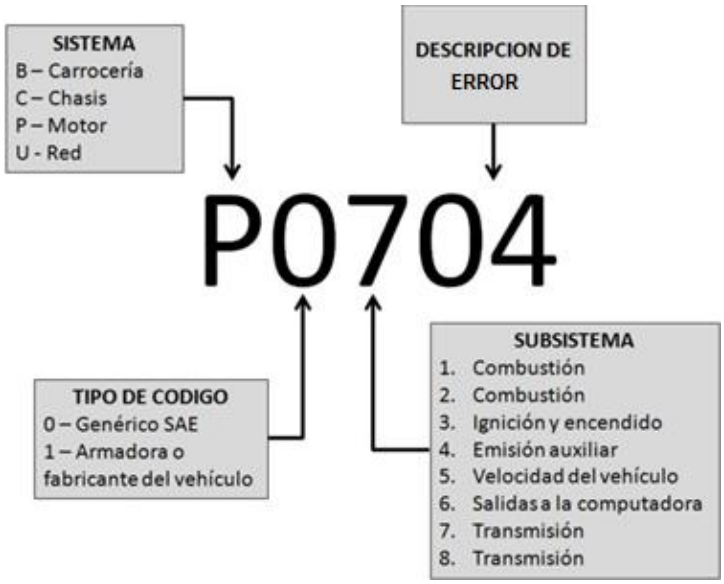

Figura 13: Código de error OBDII

3.5. Bluetooth

Hoy en día hay disponibles diferentes tecnologías de comunicación para conectar dispositivos entre sí. Entre ellas se debe destacar la comunicación inalámbrica ya que, debido al número en aumento de conexiones que actualmente se está produciendo, realizar todas ellas mediante cableado sería extremadamente complejo. La tecnología conocida popularmente como Bluetooth es el estándar de comunicación inalámbrica IEEE 802.15.1 que pertenece a este tipo de tecnología, y que tiene un conjunto de características que pueden servir para simplificar las diferentes conexiones a realizar.

Por tanto, el Bluetooth es básicamente una tecnología inalámbrica de comunicación de dispositivos a corta distancia mediante ondas de radio, que permite enviar y recibir archivos entre dos dispositivos [25].

La banda de transmisión del Bluetooth va de los 2,4 a 2,48 GHz de amplio espectro, pudiendo transmitir hasta 1600 saltos/s con un total de 79 frecuencias con intervalos de 1Mhz.

Aunque esta tecnología se asocia principalmente a los teléfonos móviles, actualmente se encuentra en otros muchos dispositivos como tablets, portátiles, teclados, impresoras, ratones, auriculares, televisores, cámaras digitales…

Así mismo, esta tecnología inalámbrica suele ser comúnmente confundida con otra tecnología inalámbrica conocida como WiFi. Por tanto, se debe partir de la base de que ambas son tecnologías inalámbricas diferentes y compatibles entre sí, y que cubren acciones y campos completamente diferentes.

La idea básica para la creación de esta tecnología fue la de reemplazar los cables como forma de comunicación entre dos terminales móviles por una tecnología de comunicación inalámbrica de bajo consumo y barata, con lo que en la década de los 90 fue desarrollada por la compañía Ericsson, empresa a la que, en poco tiempo, se le sumaron Sony, Intel, Toshiba, IBM, Nokia, Microsoft y Motorola. Finalmente se creó un consorcio para regular esta tecnología llamado Bluetooth Special Interest Group [26].

A partir de su creación, y como se ha mencionado anteriormente, su uso ha ido expandiéndose cada vez más, llegando a diferentes dispositivos de todo tipo.

Un dato curioso es que el nombre de la tecnología tiene un origen nórdico, y se debe a que, hace referencia al rey danés y noruego Harald Blåtand, traducido como Harold Bluetooth. Esto se debe, a que, debido a sus grandes dotes como orador y comunicador, consiguió unir las diferentes tribus noruegas, suecas y danesas [27].

En cuanto a la parte hardware de la tecnología Bluetooth, se tiene el dispositivo de radio que modula y transmite la señal y el controlador digital.

Así mismo hay diferentes tipos de Bluetooth los cuales, según su radio de alcance, se diferencian en 3 tipos [27]:

- Clase 1: Tiene un alcance de 100 metros.
- Clase 2: Tienen un alcance de entre 5 y 10 metros. Suelen ser los más habituales.
- Clase 3: Tienen un alcance de tan solo 1 metro.

Los especialistas en este tipo de tecnología creen que, en los próximos años, todos los equipos tecnológicos serán capaces de comunicarse entre sí gracias al estándar, aunque como se observa, una de sus grandes desventajas actuales es el corto alcance radio de esta tecnología.

Finalmente, los diferentes estándares de esta tecnología inalámbrica que se encuentran disponibles son los siguientes [28]:

- Bluetooth 1.0 y 1.0b: Primeros emisores y receptores de Bluetooth. Actualmente están obsoletos. Presentaban numerosos problemas de

interoperabilidad, y el anonimato de los dispositivos no era posible debido a la necesidad de conocer la dirección de los dispositivos.

- Bluetooth 1.1: Utiliza el estándar IEEE 802.15.1-20022. Sirvió para corregir numerosos errores de las versiones previas. Se añadieron los canales encriptados y las direcciones de los dispositivos ya no eran necesarias.
- Bluetooth 1.2: Utiliza el estándar IEEE 802.15.1-2005. Mejoró la velocidad de conexión y trasferencia de datos.
- Bluetooth 2.0: Mejora la velocidad de trasmisión de datos con la tecnología EDR o "Enhanced Data Rate". No obstante, la inclusión de esta tecnología era opcional y decisión del fabricante.
- Bluetooth 2.1: Mejora el consumo de energía, el emparejamiento entre dispositivos y la seguridad de la tecnología.
- Bluetooth 3.0: Aumenta la velocidad de transferencia de datos hasta 24Mb/s utilizando WiFi para el envío y la recepción de grandes paquetes.
- Bluetooth 4.0: Es la versión más reciente, creada en 2010, y reúne en uno solo dispositivo el Bluetooth clásico, el de alta velocidad con conexión vía WiFi y el de bajo consumo.

4. Visión general de la aplicación SmartCars

SmartCarsNet [29] es una aplicación Android dirigida a todo tipo de conductores. Su objetivo es, por un lado, dar a conocer al conductor cuál es el estado de su vehículo y si este necesita algún tipo reparación o revisión, y, por otro lado, poner en contacto a diferentes conductores para facilitar así, a la hora de hacer un viaje, la planificación de la ruta y las diferentes paradas de repostaje. La Figura 14 muestra su interfaz:

Figura 14: Menú SmartCar

La aplicación SmartCarsNet está pensada tanto para dispositivos móviles, smartphones o tablets, como para ser utilizado en la centralita de serie del vehículo, ya que se trata de una aplicación Android que para poder ser utilizada sólo requiere de la instalación de la misma en el dispositivo deseado.

La arquitectura de esta aplicación consta de dos servicios que se ejecutan paralelamente a la aplicación propiamente dicha. En el primer servicio la parte que se encarga de ejecutar el GPS y recabar los datos de éste. En el segundo servicio se encuentra tanto la parte encargada de ejecutar el GPS como la parte encargada de comunicarse con el OBDII y recabar de éste los datos necesarios. La comunicación entre este servicio y la herramienta OBDII se hará mediante el estándar de comunicación Bluetooth, como ya se ha mencionado en anteriores ocasiones. Estos dos servicios se comunicarán con una base de datos donde se almacenarán todos los datos de interés que posteriormente serán recuperados por la aplicación para su funcionamiento. Los servicios se lanzarán de forma excluyente, dependiendo de si la conexión Bluetooth con el dispositivo OBDII ha tenido éxito o no.

Para hacer uso de la aplicación se requiere un registro inicial de la persona que va a hacer uso de ésta, así como la inclusión por parte del usuario de los datos relativos al vehículo que utiliza.

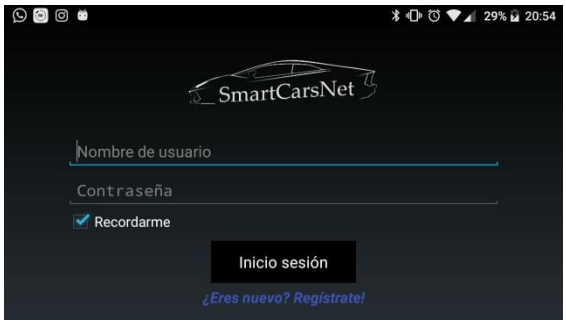

Figura 16: Página inicio

Figura 15: Página registro

Figura 17: Formulario registro vehículo

Una vez el usuario se ha registrado y accede a la aplicación se encuentra con un menú que contiene 8 opciones, cada una con una funcionalidad diferente y que serán las que permitirán navegar por la aplicación y hacer uso de ésta. Las diferentes opciones, con sus respectivas funcionalidades, son las siguientes:

Mapa, permite hace uso de un navegador de tipo GPS. Esta funcionalidad, similar al software Google Maps, irá indicando la ruta a seguir para poder llegar al destino. Además, esta interfaz permite ver en pantalla la velocidad a la que se está viajando, así como analizar el estilo de conducción. Así mismo, la aplicación irá almacenando los datos que vaya recolectando el GPS, tanto de geoposición como de velocidad, para que posteriormente el conductor pueda consultarlos.

Mantenimiento, permite llevar un registro del mantenimiento del vehículo realizado como cada vez que se han cambiado los neumáticos, de las diferentes revisiones que se le han hecho al vehículo, y de los diferentes trabajos de mantenimiento eléctricos realizados a éste. Por lo tanto, se tiene acceso a todos los datos referentes a los trabajos de mantenimiento hechos sobre el vehículo, como por

ejemplo la fecha en la que se realizó, cuál era el problema y el coste, entre otros. Así mismo, avisa de cuando sería necesario que el vehículo pasara una revisión. Sin embargo, esta funcionalidad no es automática: el usuario debe encargarse de insertar manualmente todas las revisiones llevadas a cabo en las diferentes partes del vehículo para que ésta sea una herramienta fiable.

Repostaje, permite tener una lista de las diferentes gasolineras en las que se ha repostado, junto a su localización y precio. Esto facilita, a la hora de hacer un viaje, la planificación de la ruta en base a las diferentes gasolineras que se encuentran a lo largo del trayecto, lo que permite repostar en aquellas que son más baratas. Al igual que la funcionalidad anterior, el usuario debe registrar de forma manual las diferentes gasolineras en las que para a repostar, junto con sus datos. Los datos introducidos en el módulo de repostaje pueden ser accedidos por todos los usuarios que utilicen la aplicación, facilitando así a los conductores el acceso a la información de gasolineras que no se hayan visitado con anterioridad.

Sincronizar datos, permite sincronizar los datos recolectados por la aplicación con la nube, que es la parte de ésta que permite ofrecer servicios en red. Esto facilita el poder utilizar la aplicación en diversos dispositivos sin perder los datos de los que ya se disponía anteriormente, por ejemplo, al cambiar de smartphone.

Trámites, permite llevar un registro de los trámites realizados del vehículo, desde un trámite por accidente de tráfico a un trámite para la homologación de alguna pieza. Al igual que las anteriores opciones debe ser el usuario quien registre en la aplicación todos los trámites realizados sobre el vehículo.

Diagnóstico, permite hacer un análisis a fondo de los fallos sufridos por el vehículo, y avisa de la gravedad de éste, y de los talleres más cercanos y su precio. También avisa de lo que puede suceder en caso de que no se repare la avería. Este trabajo se centra en el desarrollo de esta funcionalidad, por lo que en el apartado siguiente “5. Ampliación propuesta de la aplicación” se explicarán en detalle las diferentes acciones llevadas a cabo para su implementación.

Ajustes, permite configurar a gusto del usuario las diferentes características que presenta la aplicación, desde datos sobre el vehículo y los quilómetros totales realizados, hasta las alertas que desea recibir el conductor, pasando por diferentes opciones de configuración del OBDII, del Bluetooth y del GPS.

Finalmente, la opción Cerrar sesión, no es una funcionalidad como tal, sino que permite cerrar la sesión abierta con nuestras credenciales para que la aplicación de un mismo dispositivo pueda ser utilizada por diferentes usuarios.

Todas estas funcionalidades se han pensado con el objetivo de crear una aplicación lo más completa posible, y que sea capaz de competir y hacerse un hueco en el mercado actual de este tipo de aplicaciones.

Está aplicación que presenta varios módulos de código diferenciados, está en proceso de desarrollo, por lo que algunos de sus módulos ya han sido desarrollados. Este trabajo, por tanto, como se ha mencionado anteriormente, se centrará de realizar la implementación del módulo de diagnóstico, del cual no hay nada implementado.

5. Ampliación propuesta de la aplicación

Como se avanza en el capítulo anterior, este capítulo se centra en la implementación que se ha llevado a cabo en el módulo de "Diagnóstico" de la aplicación SmartCarsNet desarrollada.

El objetivo general de este módulo de la aplicación es el de realizar un diagnóstico completo del estado del vehículo mediante el análisis de los PID de error, ayudándose para ello de la herramienta de lectura de datos OBDII que será quien aporte a la aplicación los PID leídos.

Para ello, las diferentes características que debe cumplir y que serán implementadas son las siguientes:

- Debe indicar el código de error que ha producido la avería.
- Debe explicar de forma sencilla y clara los códigos de error mostrados.
- Debe informar de las consecuencias que puede acarrear la no reparación de la avería.
- Debe informar del coste medio que supone reparar dicha avería.
- Debe indicar la localización de la avería que se ha producido.
- Debe permitir resetear los indicadores de fallo en el salpicadero del vehículo.

Todo esto se debe implementar de forma que, al finalizar, se obtenga una aplicación amigable, sencilla para el usuario, además de una interfaz cuidada que facilite la navegación entre las diferentes funcionalidades.

Para llevar a cabo esta ampliación se ha implementado la aplicación en el entorno de desarrollo Android Studio, herramienta que se explicó anteriormente. También, se ha usado un Smartphone para poder ir probando la aplicación mientras se implementaba. Las pruebas se han realizado desde un Smartphone debido a la facilidad de trabajo que éste proporciona. No obstante, hay que tener en cuenta que la aplicación está pensada para ser instalada en la centralita Android de un vehículo.

Para simplificar la implementación que se debía llevar a cabo, se han diseñado dos diagramas de flujo que permiten separar en dos bloques diferenciados las dos grandes funcionalidades a implementar en el bloque; éstas son:

1) Selección y conexión del dispositivo OBDII:

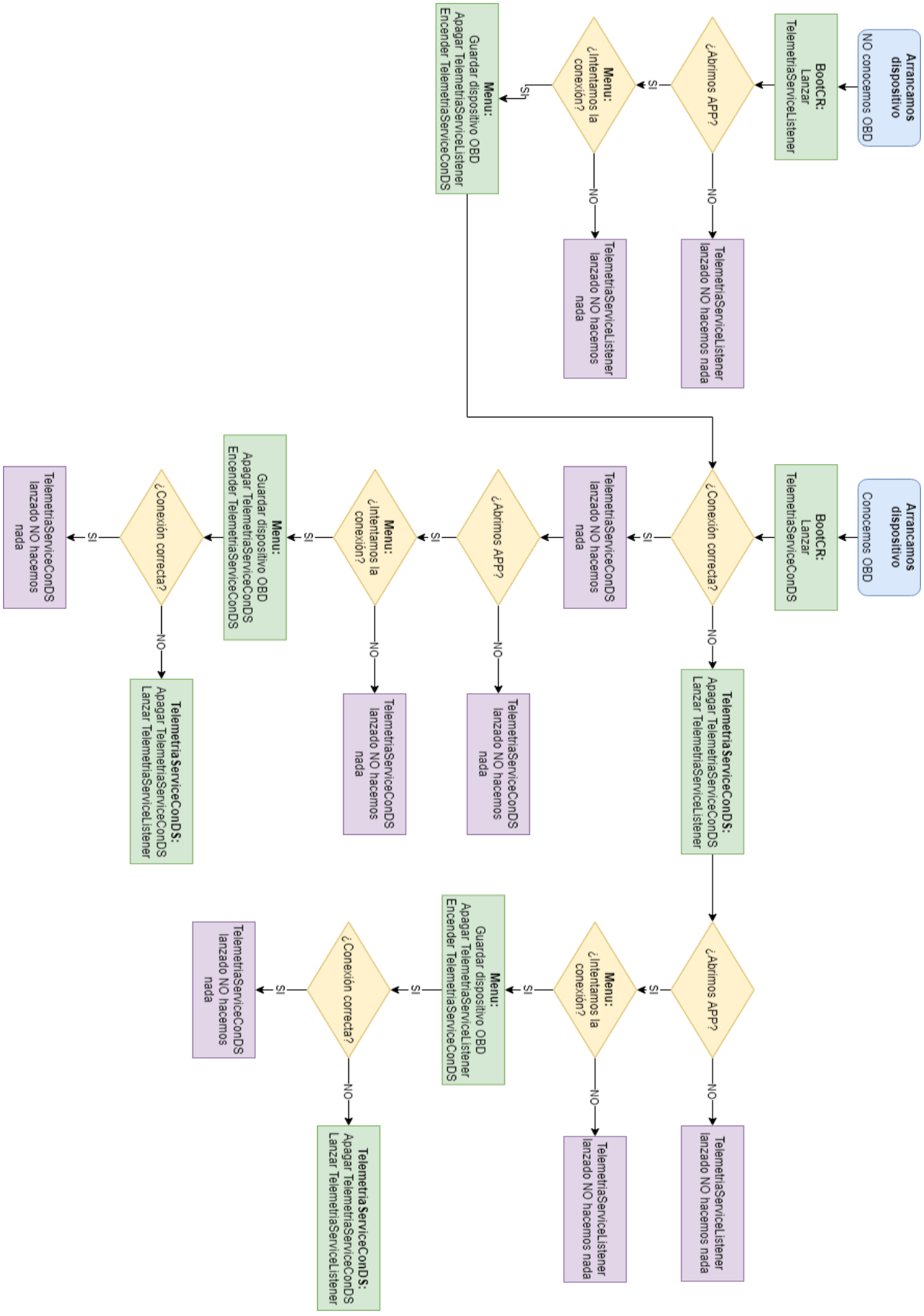

Figura 18: Diagrama flujo descubrimiento OBDII

Como se ve en el diagrama, la selección del dispositivo OBDII se realiza de dos maneras claramente diferenciadas: desde la clase "Menu", o desde la clase "BootCompletedReceiver". Este procedimiento permitirá a la aplicación saber si se ha podido obtener o no la MAC de un dispositivo OBDII. En caso de obtenerlo lanzará el servicio que permite leer datos del OBDII, conocido como "TelemetriaServiceListenerConDS", y dónde se realizará la conexión con el dispositivo. En caso contrario lanzará el servicio "TelemetriaServiceListener", que solo dispone de lo necesario para hacer el seguimiento del vehículo por GPS.

2) Obtención y visualización de los errores

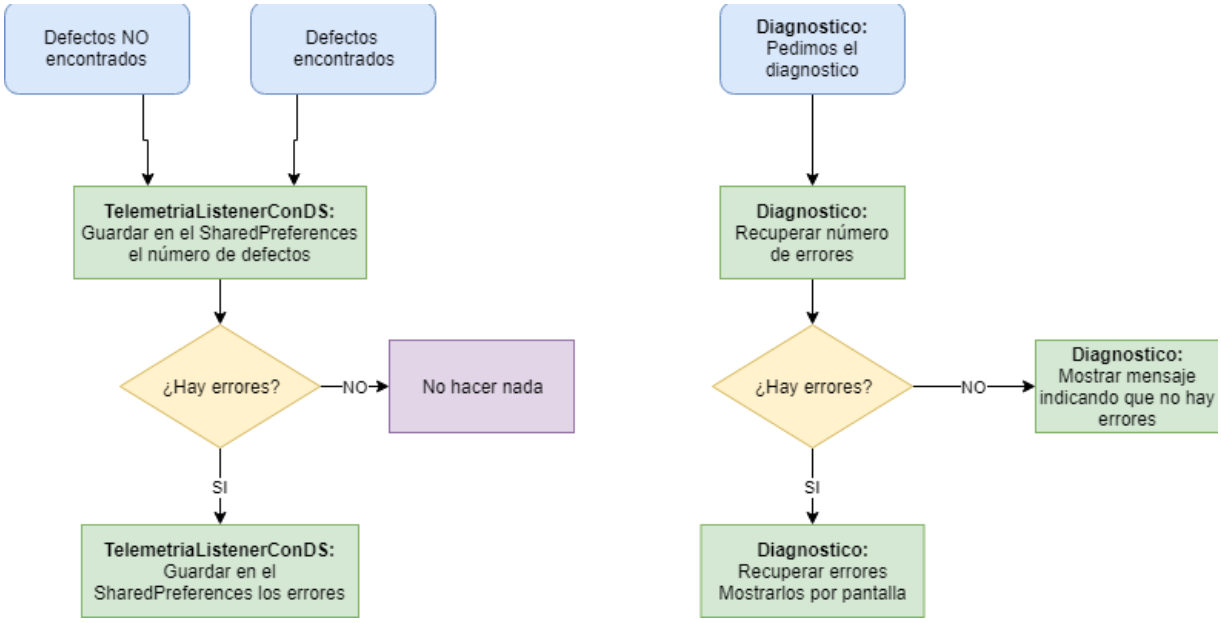

Figura 19: Diagrama flujo lectura de errores

Esta funcionalidad se activará si se consigue conectar al OBDII con éxito. En el diagrama se observa que la aplicación verifica si hay o no errores a través del servicio "TelemetriaServiceListenerConDS". Posteriormente, la clase "DiagnosticoActivity" permite listarlos en la interfaz de usuario si los hubiera, o avisa al usuario mediante un mensaje de que no existen errores que mostrar.

La primera parte que se ha implementado en la aplicación es la selección y la conexión del OBDII y el smartphone mediante Bluetooth. Como se ha explicado anteriormente, hay dos formas de realizar esta selección. La primera, cuando la aplicación todavía ni se ha conectado con el dispositivo OBDII deseado, con lo cual se debe acceder obligatoriamente a la aplicación para realizar manualmente la conexión desde la clase "Menu". La segunda se realiza automáticamente desde la

clase “BootCompletedReceiver” cuando la aplicación ya conoce el dispositivo. En cuanto a la conexión, esta se realizará siempre mediante el servicio “TelemetriaServiceListenerConDS”, independientemente de la forma de selección del dispositivo OBDII utilizada.

Hay que tener en cuenta que, para que el OBDII aparezca en la aplicación como un dispositivo apto para la conexión, éste debe haber sido previamente sincronizado con el smartphone, estado que indica la propia aplicación mediante la utilización de un mensaje emergente.

El código encargado de realizar el descubrimiento del dispositivo OBDII desde la clase Menu, y que permite seleccionarlo, es el siguiente:

Algoritmo 1: Descubrimiento del OBDII

```
//Método onCreate de la clase Menu
protected void onCreate(Bundle savedInstanceState) throws InterruptedException
{

//Inicializa los intents de los servicios que lanza la aplicación
i = new Intent(context, TelemetriaListenerService.class);
j = new Intent(context, TelemetriaListenerServiceConDS.class);

//Al iniciarse la aplicación se lanza un diálogo que le pregunta al usuario si
//quiere conectarse con el OBDII
AlertDialog.Builder blue_alert = new AlertDialog.Builder(this);
blue_alert.setMessage("No está conectado a ningún dispositivo Bluethooth
OBDII. ¿Desea conectarse?.").setTitle("Información")

//Si el usuario pulsa el botón Aceptar se buscarán los dispositivos bluetooth
//para conectar
.setPositiveButton("Aceptar", new DialogInterface.OnClickListener() {

  //Al detectar el click sobre “Aceptar”
  public void onClick(DialogInterface dialog, int id) {

    //Se cancela el mensaje anterior
    dialog.cancel();

    //Apaga los servicios que puedan estar ejecutandose
    context.stopService(i);
    context.stopService(j);

    //Verifica el estado del Bluetooth y en caso de estar apagado lo
    //activa
    BTAdapter = BluetoothAdapter.getDefaultAdapter();
    if (!BTAdapter.isEnabled()){BTAdapter.enable();}

    //Caso con ningún dispositivo sincronizado previamente
    if(BTAdapter.getBondedDevices().size() == 0) {

      //Se inicia el servicio “TelemetriaServiceListener”
      context.startService(i);
      //Se avisa al usuario de que no hay dispositivos sincronizados
      Toast.makeText(getApplicationContext(),"Ningun dispositivo
      sincronizado",Toast.LENGTH_LONG).show();

   //Caso con dispositivos sincronizado previamente
```

```
    }else{

       //Se avisa al usuario de que si su dispositivo no aparece debe
       //sincronizarlo previamente
       Toast.makeText(getApplicationContext(),"Si tu OBD no aparece,
       sincronizalo",Toast.LENGTH_LONG).show();
       //Se listan todos los dispositivos que estén sincronizados con el
       //teléfono
       contador = 0;
       pairedDevices = BTAdapter.getBondedDevices();
       numDisp = pairedDevices.size();
       arrayObjetos = new BluetoothDevice[numDisp];

       for (BluetoothDevice device : pairedDevices) {
       //Se van añadiendo los dispositivos sincronizados a la lista que se
       //mostrará por pantalla
         deviceStrs.add(device.getName() + "\n" + device.getAddress());
         devices.add(device.getAddress());
         arrayObjetos[contador]=device;
         contador++;
       }

       //Método que lista los OBD descubiertos
       listOBD();

    }})

    //En caso negativo se cierra el menu emergente
    .setNegativeButton("Cancelar", new DialogInterface.OnClickListener() {

     public void onClick(DialogInterface dialog, int id) {
       dialog.cancel();
     }
    });

    //Pone título al primer dialogo
    blue_alert.setTitle("ODB-2 Bluethooth");
       //Muestra el primer dialogo
       blue_alert.show();
    }
  }
}
```

Este código se encarga de crear una ventana emergente en la aplicación dónde se pregunta al usuario si quiere realizar la conexión con el OBDII. Dependiendo de la respuesta, su funcionalidad será diferente. Si se pulsa “Aceptar”, se mostrará una lista con los dispositivos Bluetooth sincronizados con el dispositivo multimedia; de esta parte se encargará el método “listOBD()”, dónde se podrá elegir el OBDII al que conectarse. Esta selección lanzará el servicio “TelemetriaServiceListenerConDS”, y cerrará la ventana emergente. Si se pulsa “Cancelar” la ventana emergente se cerrará sin realizar ninguna acción complementaria.

El código encargado de realizar la selección del dispositivo OBDII desde la clase “BootCompletedReceiver” es bastante más simple, puesto que en este caso solo se realiza una comprobación para ver si se conoce la MAC del dispositivo a

conectar, o no, y dependiendo del caso se lanza el servicio adecuado. Este código se va a lanzar cada vez que se inicie el dispositivo en el que la aplicación está instalada, cosa que debería ocurrir siempre, puesto que, como se ha comentado anteriormente, aunque las pruebas se hayan realizado desde un smartphone por comodidad, la aplicación está pensada para que se encuentre instalada en el ordenador de a bordo del vehículo, con lo que este dispositivo arrancaría siempre que se arrancara el vehículo. En el Algoritmo 2 se observa la implementación:

Algoritmo 2: Receiver que lanza los servicios al iniciarse el dispositivo.

```
public class BootCompletedReceiver extends BroadcastReceiver {

 //Inicializa las variables
 private SharedPreferences prefs;
 private SharedPreferences.Editor editor;
 private String macOBD;
 private Boolean exito;
 private Boolean destroy;

 //Al detectar que el dispositivo ha sido iniciado
 public void onReceive(Context context, Intent intent) {

  //Apaga los servicios que puedan estar ejecutándose
  Intent i = new Intent(context, TelemetriaListenerService.class);
  context.stopService(i);
  Intent j = new Intent(context, TelemetriaListenerServiceConDS.class);
  context.stopService(j);

  //Inicializa el espacio compartido y recupera la variable macOBD que
  //contiene en caso de que la aplicación la conozca la MAC del OBDII con el
  //que conectarse
  prefs = context.getSharedPreferences("FKPrefs",Context.MODE_PRIVATE);

  //Inicializa el editor de variables
  editor = prefs.edit();
  //Le indica a la aplicación desde la clase que se está ejecutando
  editor.putString("ACTIVITY", "BCR");
  editor.commit();

  macOBD = prefs.getString("MACOBD", "");

  // Si la variable está vacía
  if(macOBD.equals("")){
    //Lanza el servicio "TelemetriaServiceListener"
    context.startService(i);

  // Si la variable devuelve una MAC
  }else{
    //Lanza el servicio "TelemetriaServiceListenerConDS"
    context.startService(j);
  }
 }
}
```

Este método, transparente al usuario, detiene los servicios "TelemetriaServiceListener" y "TelemetriaServiceListenerConDS" al ser lanzado. Además, comprueba el valor de la variable "MACOBD" y, dependiendo de su estado, lanza un servicio u otro. Si está vacía, iniciará "TelemetriaServiceListener", y si está informada, iniciará "TelemetriaServiceListenerConDS".

Finalmente, una vez seleccionado el OBDII manualmente desde la clase "Menu", o de forma automática desde la clase "BootCompletedReceiver", faltaría realizar la conexión con el dispositivo desde el servicio "TelemetriaServiceListenerConDS".

A continuación, se presenta la implementación llevada a cabo dentro del servicio "TelemetriaServiceListenerConDS", pero primero se analiza la forma de comunicarse del OBDII.

El OBDII utiliza dos formas de comunicación diferenciadas: la primera mediante "Comandos AT" que permiten configurar el intérprete ELM327, y la segunda mediante los denominados "Parámetros ID" o "PID", para comunicarse con el sistema de diagnóstico de a bordo de un vehículo. A continuación se muestra un resumen tanto de los comandos AT como de los PID utilizados en este trabajo.

Comandos AT

Un comando AT es el parámetro utilizado por el intérprete ELM327 para realizar la configuración de éste. Aunque la mayoría de ellos son autoconfigurables, debido a la verificación que realiza el intérprete para conocer el protocolo OBDII con el que está trabajando, algunos también admiten personalización por parte del usuario. El intérprete no admitirá ningún parámetro diferente a éstos para configurarse, y estos comandos siempre se encontrarán precedidos por las letras AT, ya que de otra forma no serían reconocidos por el intérprete. Algunos de los parámetros AT utilizados durante este trabajo son:

Valor	*Descripción*
AT Z	Reinicia el dispositivo
AT E0/E1	Echo off/on.
AT M0/M1	Memoria off/on.
AT AT 0,1,2	Timeout Adaptativo off, auto 1, auto 2.
AT ST hh	Utiliza el Timeout hh x 4 msec.

AT SP h	Utiliza el protocolo h y lo guarda.
AT DP	Descripción del protocolo.

Tabla 4: Comandos AT

Todos los comandos AT se pueden encontrar en el Apéndice B de esta memoria.

PID

En cuanto a los "Parámetros ID" o "PID" son los códigos que utiliza el intérprete ELM327 para comunicarse con el sistema de diagnóstico de a bordo de los vehículos. Estos códigos identifican los distintos parámetros que se pueden medir dentro de éste.

El estándar OBDII SAE J1979 se encarga de definir 10 modos de funcionamiento, que se observan en la tabla 5, y dentro de los cuales se definen los códigos PID en el estándar J1939. Estos códigos PID, que fueron definidos por la Sociedad de Ingenieros Automotrices (SAE), no son de uso obligatorio para los fabricantes de vehículos, es decir, cada fabricante puede implementar en sus vehículos aquellos que considere necesarios, así como definir nuevos códigos propios.

Modo (hex)	Descripción
01	Muestra los parámetros disponibles.
02	Muestra los datos almacenados por evento.
03	Muestra los códigos de fallas de diagnóstico (Diagnostic Trouble Codes, DTC).
04	Borra los datos almacenados, incluyendo los códigos de fallas (DTC).
05	Resultados de la prueba de monitoreo de sensores de oxígeno (solo aplica a vehículos sin comunicación Controller Area Network, CAN).
06	Resultados de la prueba de monitoreo de componentes/sistema (resultados de la prueba de monitoreo de sensores de oxígeno en vehículos con comunicación CAN).
07	Muestra los códigos de fallas (DTC) detectados durante el último ciclo de manejo o el actual.
08	Operación de control de los componentes/sistema a bordo.

09	Solicitud de información del vehículo.
0A	Códigos de fallas (DTC) permanentes (borrados).

Tabla 5: Modos funcionamiento PID

A continuación se mostraran aquellos códigos PID utilizados durante el trabajo realizado. No obstante, un resumen de todos los PID implementados en el estándar J1939 se puede encontrar en el Apéndice C de esta memoria.

El PID 01 del modo 01 es utilizado para la implementación del algoritmo que permite descubrir si hay algún error almacenado, su definición es la siguiente:

PID (hex)	Data bytes devueltos	Descripción
01	4	Estado de los monitores de diagnóstico desde que se borraron los códigos de fallo DTC; incluye el estado de la luz indicadora de fallo, MIL, y la cantidad de códigos de fallo DTC.

Tabla 6: PID 01

Para indicar al OBDII que se quiere recuperar la información de este PID, se transmite la secuencia 0101, dónde el primer “01” hace referencia al modo de funcionamiento requerido, y el segundo “01” al PID utilizado.

La respuesta típica a esta petición es un código del tipo 41 01 81 06 56 04, donde el 41 01 indica que se está respondiendo a la petición inicial, y el 81 indica el número de errores detectados. Hay que tener en cuenta que 81 no es el número real de errores, sino que hay que tratar ese dato para obtenerlos.

Para calcular los errores reales obtenidos del vehículo hay que tener en cuenta que el bit más significativo del hexadecimal 81 recuperado, se utiliza para indicar si la lámpara indicadora del mal funcionamiento (MIL o Lámpara Verificadora del Motor) está encendida. El resto de los bits indican el número de errores. Para calcularlos solo hay que pasar el 81 de hexadecimal a decimal, lo cual correspondería a 129, y restarle el primer bit (128), con lo que se observa que, en el caso anterior, la lámpara está encendida y hay un error en el motor.

El modo 03 se utiliza para descubrir cuales son los errores almacenados en el OBD en caso de que el modo anterior haya detectado algún fallo.

PID (hex)	Data bytes devueltos	Descripción	Nota
N/A	N*6	Solicita los códigos de fallo.	3 códigos por bloque de mensaje.

Tabla 7: PID 03

La particularidad de este modo es que no necesita PIDs para funcionar, simplemente se le envía una petición al intérprete que contiene el modo de funcionamiento 03, y una respuesta típica podría ser 43 01 53 00 00 00 00.

El 43 indicaría que se está respondiendo al pedido inicial del modo 03. El resto de los bytes deben leerse en pares: 0153 0000 0000. Cabe destacar que la norma SAE define que, de forma predefinida, y en caso de no haber datos, se deben rellenar los mensajes con ceros hasta obtener una respuesta de 8 bytes, con lo que los 0000 no representarían fallos reales del motor. Por lo tanto, se obtiene un error 0153, como indicaba el mensaje anterior.

Al igual que en el caso anterior, hay que traducir el código para obtener el error correspondiente, ya que el primer bit contiene información adicional. Para ello hay que comparar el primer bit 0 con la siguiente tabla, y juntar el resto al valor obtenido:

Primer bit	Se reemplaza por	Códigos que hacen referencia a errores en el	Definidos por
0	P0	Tren de potencia	SAE
1	P1		Fabricante
2	P2		SAE
3	P3		SAE + Fabricante
4	C0	Chasis	SAE
5	C1		Fabricante
6	C2		Fabricante
7	C3		Reservados para el futuro

8	B0		SAE
9	B1	Cuerpo	Fabricante
A	B2		Fabricante
B	B3		Reservados para el futuro
C	U0		SAE
D	U1	Red	Fabricante
E	U2		Fabricante
F	U3		Reservados para el futuro

Tabla 8: Correspondencia de bits

Al reemplazar el bit se obtiene el código de error P0153, que estaría indicando que hay una respuesta lenta del circuito sensor de oxígeno.

El modo 04 sirve para borrar todos los errores almacenados en el dispositivo OBDII. Su funcionamiento es igual al anterior, pero en este caso se envía el mensaje 04 y no hay respuesta del intérprete.

PID (hex)	Data bytes devueltos	Descripción
N/A	0	Borra todos los códigos de fallos y apaga la luz indicadora de fallo (Malfunction Indicator Lamp, MIL).

Tabla 9: PID 04

Un requisito obligatorio del estándar seria pedir al usuario que confirme que realmente quiere borrar los códigos de error almacenados en el dispositivo.

Una vez visto como se comunican, se presenta a continuación la implementación del servicio TelemetriaListenerServiceConDS realizada, que es la siguiente:

Algoritmo 3: Creación del servicio TelemetriaListenerServiceConDS.

```
public void onCreate() {

  //Inicializa el broadcast que se utilizará durante la lectura de errores
  broadcaster = LocalBroadcastManager.getInstance(this);
  bcIntent.setAction(ACTION_FIN);

  // Se inicia la conexión del OBD y su configuración
  try {
```

```
  //Función encargada de la conexión y la configuración, definida más abajo
   startDevice();
  } catch (IOException e) {
   e.printStackTrace();
  } catch (InterruptedException e) {
   e.printStackTrace();
  }

  Thread thread = new Thread(null,backgroundConection);
  thread.start();
 }
```

Ésta se encarga de crear el servicio "TelemetriaServiceListenerConDS" la primera vez que se inicia, y a continuación lanza el método "startDevice()", cuya implementación es la siguiente:

Algoritmo 4: Conectar y configurar el OBD.

```
private void startDevice() throws IOException, InterruptedException {

  //Recupera las preferencias de configuración de la aplicación
  prefs =context.getSharedPreferences("FKPrefs",Context.MODE_PRIVATE);
  editor = prefs.edit();

  //Recupera la MAC
  macOBD = prefs.getString("MACOBD", "");
  //Recupera el device asociado a esa MAC
  mDevice = returnDevice(macOBD);

  //Si se encuentra un device asociado
  if(!(mDevice==null)){

   //Lanza el método que permitirá conectar con el OBD, en caso
   //de que la conexión sea un éxito se realizará la configuración
   if (ConnectWithOBD()){

    try{
     // Serie de métodos para realizar la configuración
     resetDevice();
     configureDevice();
     displayDevice();
     searchProtocol();
     protocolSelected();

     //En caso de que ninguna función falle le indicamos a la app que
     //el proceso ha tenido éxito
     editor.putBoolean("EXITO", true);
     editor.commit();

     Thread.sleep(2000);

    }catch (InterruptedException e) {
      //Si algo ha fallado se lo indica a la aplicación y
      //se destruye el servicio
      editor.putBoolean("EXITO", false);
```

```
      editor.commit();
      e.printStackTrace();
      onDestroy();
     }
    //La conexión no ha tenido éxito
    }else{
      //Se lo indica a la aplicación para destruir el servicio
      editor.putBoolean("EXITO", false);
      editor.putBoolean("DESTROY", true);
      editor.commit();
    }
   //No se ha encontrado ningún dispositivo ligado a esa MAC
  }else{
    //Se lo indica a la aplicación para destruir el servicio
    editor.putBoolean("EXITO", false);
    editor.putBoolean("DESTROY", true);
    editor.commit();
  }
 }
```

Este método se encarga de llamar a otros métodos encargados de realizar la conexión y configuración del OBD. En caso de que alguno de esos métodos falle se le indicará a la aplicación que han fallado para que así destruya al servicio.

Con esto se consigue que se ejecute la siguiente secuencia de pantallas, que permite la conexión del OBDII con el Smartphone deseado:

Figura 20: Pop-up conexión

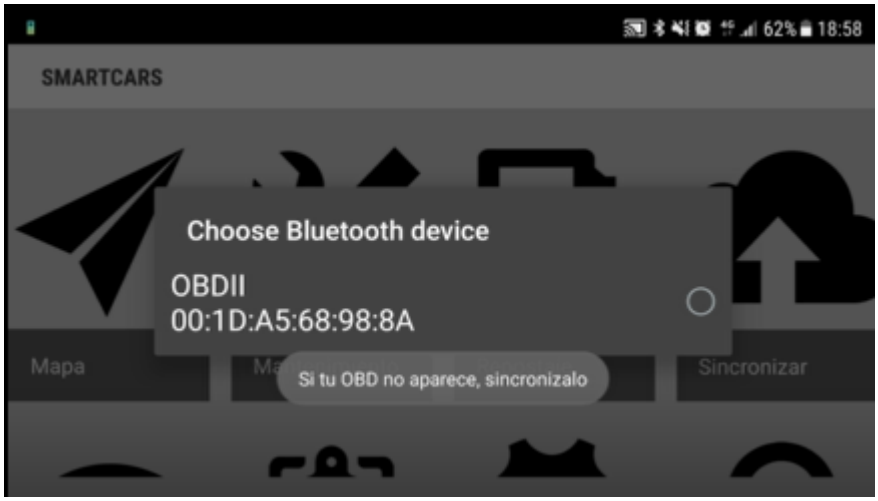

Figura 21: Dispositivos Bluetooth sincronizados

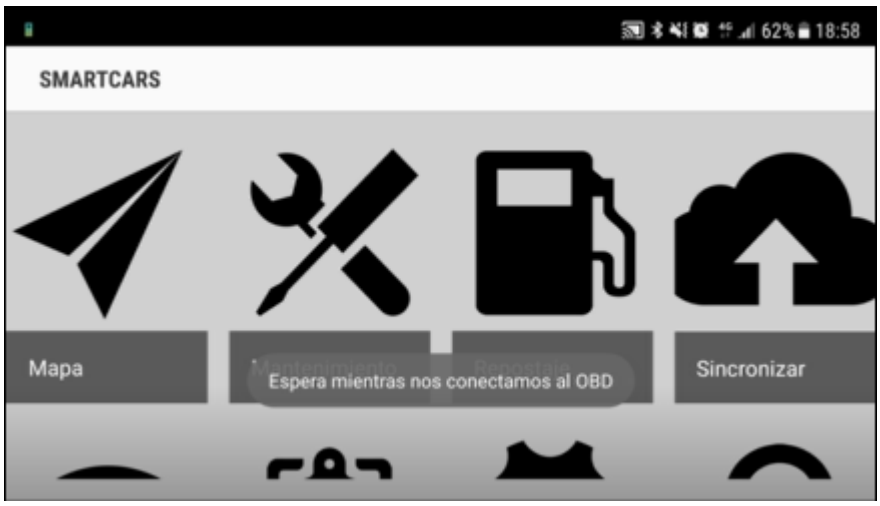

Figura 22: Aviso de espera

Figura 23: Aviso de conexión

Una vez comprobada la funcionalidad del primer bloque, se ha implementado la parte de la lectura y visualización de los códigos de error. Esta parte también utiliza la comunicación entre dispositivos vista anteriormente.

Si finalmente todo el proceso ha tenido éxito, la visualización de los errores a través de la interfaz se implementará en la clase "DiagnosticoActivity", aunque toda la comunicación con el OBDII la continuará haciendo el servicio "TelemetriaServiceListenerConDS". A continuación se muestra la parte implementada para la recuperación de los errores detectados por el servicio y su visualización:

Algoritmo 5: Recupera los errores detectados por el servicio y los muestra en la app.

```
public void realizarDiagnostico(){

 //Comprueba si el servicio TelemetriaListenerServiceConDS se está
 //ejecutando
 DSoperativo =
 isMyServiceRunning(TelemetriaListenerServiceConDS.class);

 //En caso afirmativo
 if(DSoperativo){

  //Obtiene el número de errores
  numErrores = prefs.getString("NUM_ERR", "0");
  //Si es igual a cero
  if(numErrores.equals("0")){
   //Le indica al usuario que no hay errores
   makeToastLong(getString(R.string.text_noerrors));
  //Si se obtienen errores
  }else{
   //Convierte la string con los errores obtenidos en un integer
   numPIDErr = Integer.parseInt(numErrores);
   //Por cada error
   for(contador=0;contador<numPIDErr;contador++){
    //Lo almacena en las variables persistentes
    PIDerr = prefs.getString("ERROR"+contador, "NO_DATA");
    //Obtiene los datos almacenados en la base de datos referente al
    //vehículo del usuario y al error obtenido
    String marca =
vehiImpl.getVehiculoByOwner(usrImpl.getUsuarioActivo().getEmail()).get
Marca();
    //Por cada error se verifica si ya lo habíamos detectado antes y
    //está guardado en caso de que ya exista se devuelve true
    for(int j=0;j<diagnostico.length;j++){
     if (PIDerr.equals(diagnostico[j])) {
      diagnExiste = true;
     }
    }
    //Si no existe se crea un nuevo diagnóstico y se añade a la
    //lista de diagnósticos
    if(!diagnExiste){
     diag1 = new Diagnostico(
             vehiImpl.getVehiculoByOwner(
             usrImpl.getUsuarioActivo().getEmail()).getMatricula(),
             new Date(), new CodigoOBD(PIDerr, "Descripcion", marca)
     );
     diagImpl.add(diag1);
    }
    //Lo almacena en las variables persistentes de diagnóstico
```

```
    diagnostico[contador] = PIDerr;
    editor.putString("DIAGNOSTICO"+contador,diagnostico[contador]);
    //Actualizamos el contador de diagnósticos
    editor.putInt("CONT_DIAG",contador);
    editor.commit();
    contador++;
   }
   //Muestra por pantalla los errores
   initView();
  }
  //Si no se está ejecutando el servicio
  }else{
    //Se lo indica al usuario
    makeToastLong("Ups! Parece que no estas conectado a tu OBD");
  }
}
```

Este método se encarga de verificar que el servicio "TelemetriaServiceListenerConDS" está iniciado, y en caso de estarlo recupera todos los errores detectados por el servicio, y los muestra por pantalla al usuario.

A continuación, se observa cómo la interfaz de la aplicación está vacía, y seguidamente se muestra el error obtenido del OBDII:

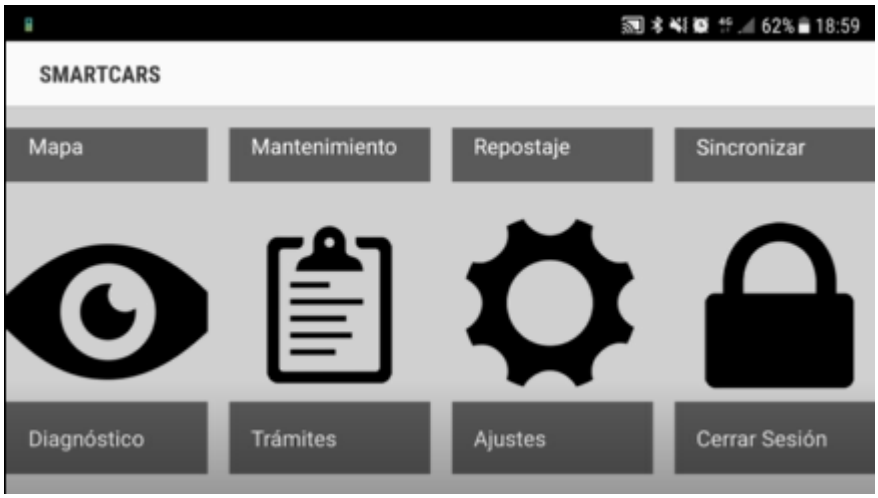

Figura 24: Menú SmartCarsNet

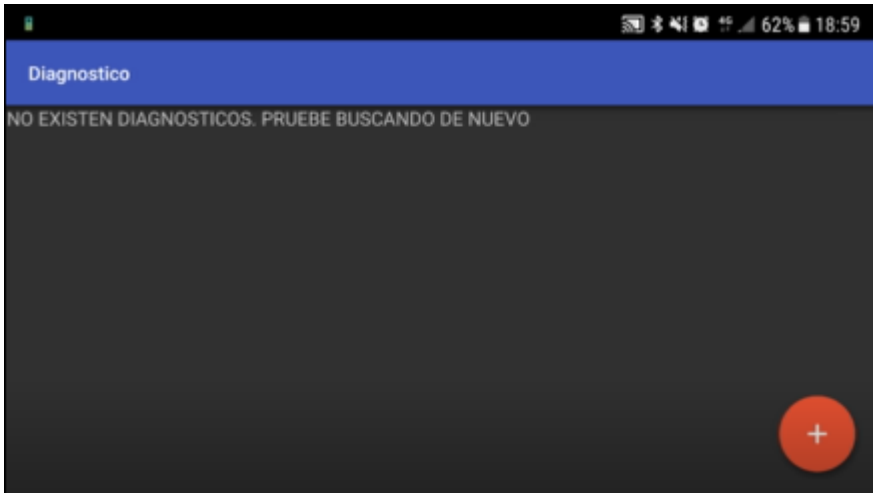

Figura 25: Interfaz diagnóstico

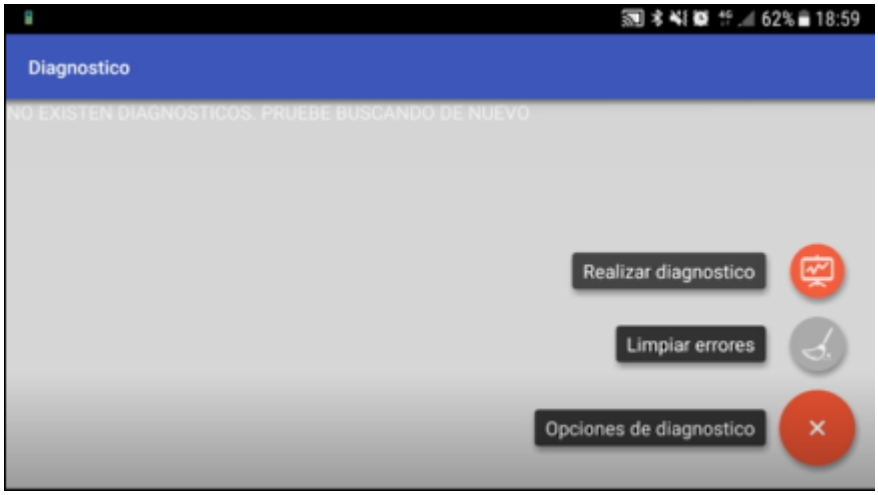

Figura 26: Menú Diagnóstico

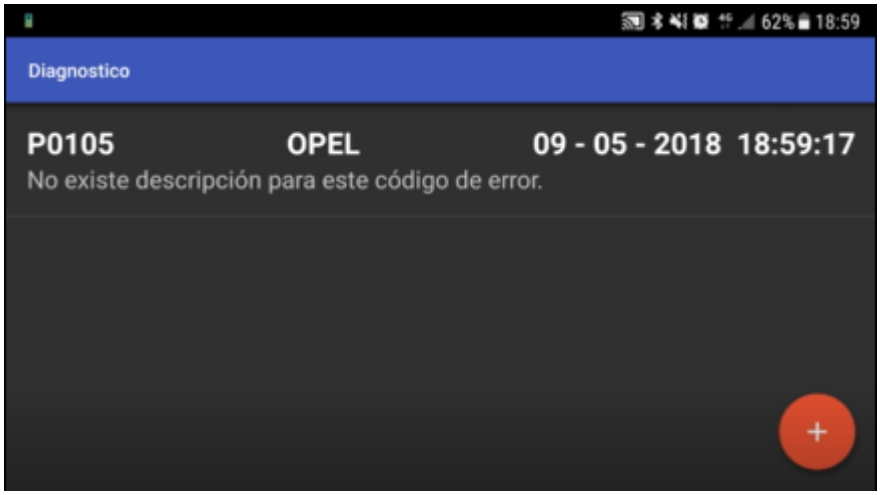

Figura 27: Ejemplo de PID de error

Como se observa en las figuras anteriores, para transmitir al usuario la información sobre los errores de la forma lo más clara posible, se opta por mostrarla mediante una lista. Cada elemento de la lista proporciona 4 campos que dan la información arriba descrita. Estos campos son:

- Código del error.
- Marca del vehículo.
- Fecha en que se ha producido.
- Descripción: campo dónde aparecerá toda la información mencionada anteriormente.

Finalmente, se podrán borrar los errores detectados por el OBDII mediante el siguiente código, también implementado en la clase "DiagnosticoActivity", y gestionado por el servicio "TelemetriaServiceListenerConDS":

Algoritmo 6: Gestión el borrado de errores.

```
public void clearCodes(){
 //Crea un dialogo para preguntar al usuario si está seguro de
 //querer borrar los errores
 AlertDialog.Builder blue_alert = new AlertDialog.Builder(this);
 blue_alert.setMessage("Vas a borrar los errores. ¿Estas seguro?")
 .setTitle("Información")
 //Si acepta
 .setPositiveButton("Aceptar", new DialogInterface.OnClickListener() {
  public void onClick(DialogInterface dialog, int id) {
   //Le indica a la aplicación que el usuario ha aceptado
   editor.putBoolean("CLEARCODES",true);
   editor.commit();
   //Obtine los diagnosticos
   int delCont = prefs.getInt("CONT_DIAG", 9999);
   //En caso de haberlos se borran de las variables persistentes
   if(delCont!=9999){
    for(int l = 0; l<=delCont; l++){
     editor.putString("DIAGNOSTICO"+l,"NO_DATA");
     editor.commit();
    }
   }
  //Se borran los diagnósticos de todas las variables
 diagImpl.deleteAllByCar(vehiImpl.getVehiculoByOwner(usrImpl.
 getUsuarioActivo().getEmail()).getMatricula());
 listaDiagnosticos.clear();
 mapaCodigos.clear();
 Arrays.fill( diagnostico, null );
 //Se refresca la vista de la interfaz y se cierra el dialogo
 initView();
 dialog.cancel();
}})
 .setNegativeButton("Cancelar", new DialogInterface.OnClickListener(){
  //Si ha clicado sobre cancelar se cierra el dialogo sin hacer nada
  public void onClick(DialogInterface dialog, int id) {
   dialog.cancel();
  }});

 blue_alert.setTitle("Limpiar Errores");
 blue_alert.show();

}
```

Este método, siguiendo la norma del SAE J1979, espera a que el usuario pida un borrado de los errores encontrados para preguntarle al usuario mediante una ventana emergente si efectivamente desea borrarlos. En caso de que la contestación del usuario sea afirmativa, los errores serán borrados. En caso de que sea negativa, se cerrará la ventana emergente sin realizar ninguna acción más.

A continuación se observa cómo la interfaz de la aplicación tiene un error, y continuamente se muestra vacía al borrar el error obtenido:

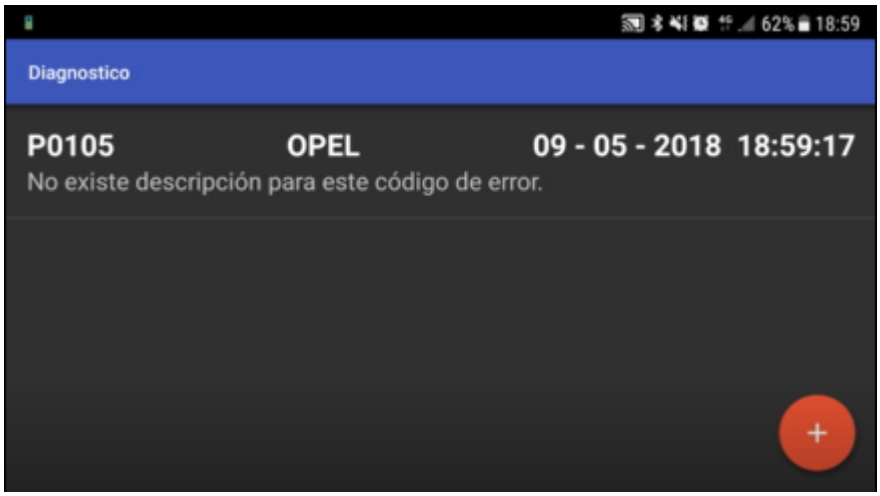

Figura 28: Interfaz con error

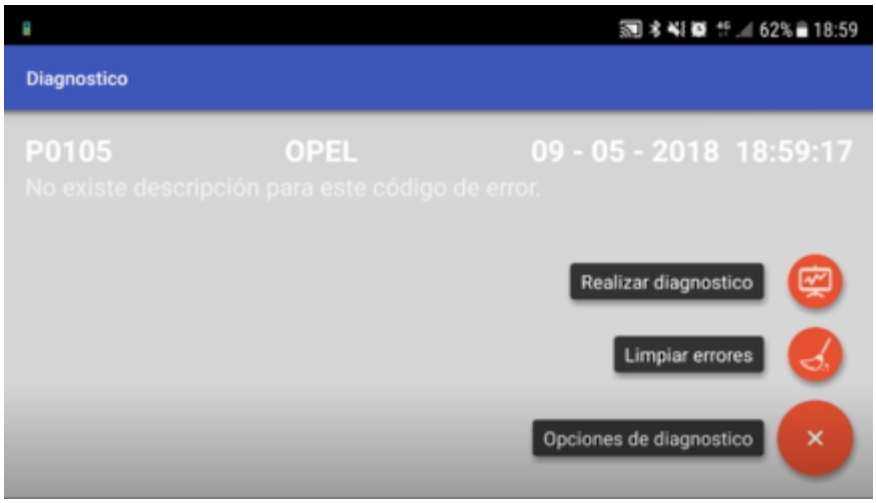

Figura 29: Opción "Limpiar errores"

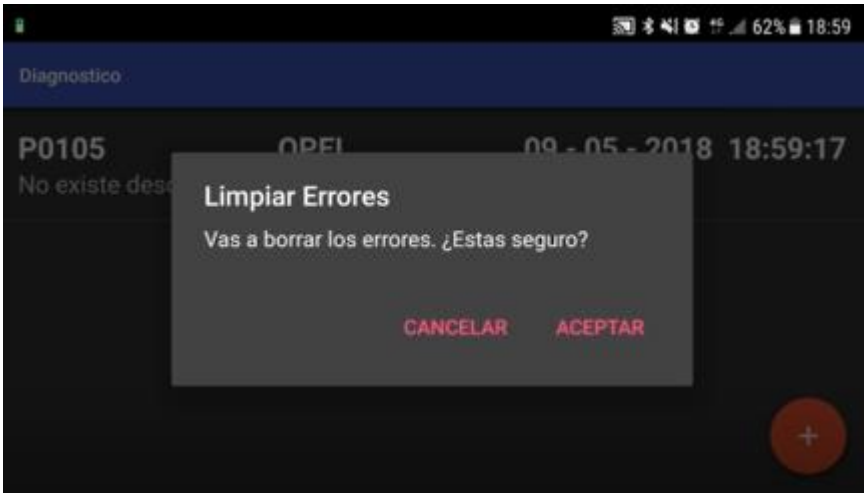

Figura 30: Pop-up limpiar errores

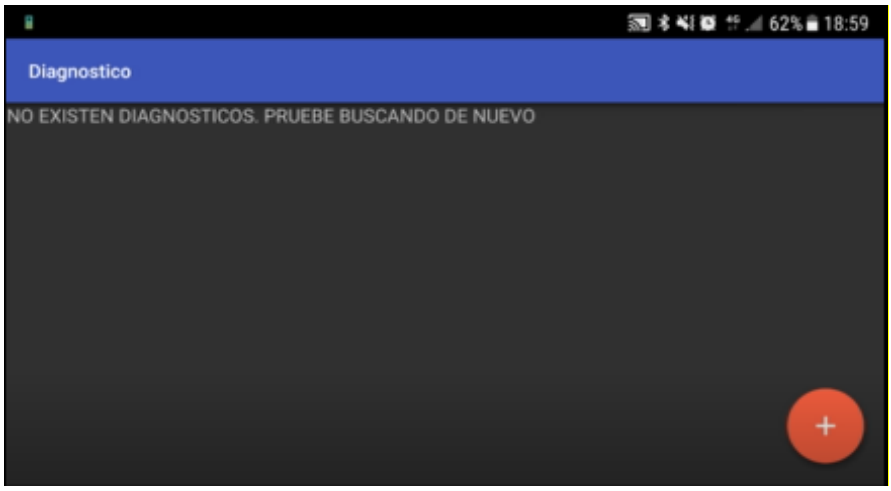

Figura 31: Interfaz sin errores

Como se ha podido ver, el servicio "TelemetriaServiceListenerConDS" es el encargado de llevar a cabo toda la comunicación con el dispositivo OBDII, quien a su vez se encargará de la lectura de los errores. Para la obtención de éstos se ha implementado dentro del método "onStartCommand" del servicio el siguiente código:

Algoritmo 7: Inicia el servicio y se encarga de leer/borrar los errores.

```
public int onStartCommand(Intent intent, int flags, int startId) {
 //Se verifica si se ha obtenido algún mensaje para destruir el
 //servicio
 destroy = prefs.getBoolean("DESTROY", false);
 //en caso de haberlo obtenido el mensaje de destruir el servicio
 if(destroy){
  //Se destruye el servicio
  onDestroy();
  return START_NOT_STICKY;
 }else{
  //Se verifica si se ha obtenido algún mensaje para destruir el
```

```
//servicio
clearCodes = prefs.getBoolean("CLEARCODES", false);
//en caso de haberlo obtenido
if(clearCodes){
 try {
  while (!stop) {
   //Se le envía el comando 04 y se obtiene la respuesta
   result = runATCommands(new OBDCode("04")).replace(" ","");
   if (result != null) {
    break;
   }
   Thread.sleep(2000);
  }
 } catch (InterruptedException e) {
    e.printStackTrace();
 }
 //Como se han borrado los errores se actualiza la variable
 //persistente a 0
 editor.putString("NUM_ERR","0");
 editor.putBoolean("CLEARCODES",false);
 editor.commit();
 //Cerramos la conexión
 return START_STICKY;

//en caso de no haber obtenido el mensaje de destruir el servicio
}else{
  try {
   //Se obtiene la MAC del dispositivo a conectar
   deviceAddress = mDevice.getAddress();
  }catch (NullPointerException  e) {
    deviceAddress ="";
  }

  try {
   //Se verifica si hay errores
   numPIDError = numPIDError();
   //si el valor es diferente al flag 9999 que se ha definido lo que
   //quiere decir que si hay errores
   if(numPIDError!=9999){

     allErrorsClean = new String[numPIDError];
     Errors = new String[numPIDError];
     //Se obtiene el número de errores a recuperar
     PIDredondeado = numPIDErrorRedondeado(numPIDError);
     allMessage = new String[PIDredondeado+1];
     allMessageClean = new String[PIDredondeado];
     //si no hay errores
     if(numPIDError == 0){
      //Se actualiza la variable persistente indicándolo
      editor.putString("NUM_ERR", "0");
      editor.commit();
     //Si hay errores
     }else{
     //Se actualiza la variable persistente con el número de errores
      editor.putString("NUM_ERR", String.valueOf(numPIDError));
      editor.commit();
       while (!stop) {
        //Se recuperan los errores
        result = runATCommands(new OBDCode("03")).replace(" ","");
        allMessage[i] =result;
```

```
        i++;
        if (result != null && result.contains("0000")) {
         break;
        }
        Thread.sleep(2000);
       }
       //Limpia el array de errores obtenido para dejar solo los
       //errores de cada mensaje
       allMessageClean =
                  limpiaArrayMensajes(allMessage,PIDredondeado);
       //Separa los errores
       allErrorsClean = separaErrores(allMessageClean, numPIDError);
       Errors = hexACode(allErrorsClean,numPIDError);

       for(int j=0; j<Errors.length;j++){
        error = Errors[j];
        editor.putString("ERROR"+contador, error);
        editor.commit();
        contador++;
        var1 = var1 + Errors[j] + " ";
       }

       editor.putString("ERRORES", var1);
       editor.commit();
      }
     }
    } catch (InterruptedException e) {
       //Si hay algún error lo indica
       e.printStackTrace();
       editor.putBoolean("EXITO", false);
       editor.putBoolean("DESTROY", true);
       editor.commit();
       onDestroy();
     }
     //Verifica el activity desde donde se ha lanzado el servicio
     activity = prefs.getString("ACTIVITY","BCR");
     //Si se ha lanzado desde MENU se lanza el broadcast
     if(activity.equals("MENU")){
      sendBroadcast(bcIntent);
     //Sino
     }else{
       //Verifica si el procedimiento a terminado con éxito
       exito = prefs.getBoolean("EXITO", false);
       //En caso afirmativo
       if(exito){
        //Avisa al usuario de que el OBD se ha conectado y
       //configurado con éxito
        Toast.makeText(context, "OBD conectado y configurado",
        Toast.LENGTH_SHORT.show();
       //Sino indica que algo ha fallado y se debe finalizar el
       //servicio
       }else {
        Toast.makeText(context, "La conexión y configuración del OBD
        ha fallado", Toast.LENGTH_SHORT).show();
        destroy = prefs.getBoolean("DESTROY", false);
        if(destroy){
         onDestroy();
        }
       }
      }
```

```
            return START_STICKY;
           }
       }
}
```

Este método se encarga de iniciar el servicio y gestionar toda la parte de la comunicación con el OBDII dependiendo de las acciones que se quieran llevar a cabo. Estas acciones serán la obtención y el borrado de errores. En caso de que hubiera algún fallo se le indicaría a la aplicación para poder destruir el servicio.

En caso de que algo vaya mal se destruirá el servicio mediante el siguiente código:

Algoritmo 8: Destruye el servicio

```
public void onDestroy() {
 Log.i(TAG,"DataLiveService onDestroy");
 //Le indica a la aplicación que algo ha fallado
 editor.putBoolean("EXITO", false);
 editor.putBoolean("DESTROY", true);
 editor.commit();

 //Obtiene la clase desde la cual se ha lanzado el servicio
 activity = prefs.getString("ACTIVITY","BCR");
 //Si es desde Menu
 if(activity.equals("MENU")){
  //Lanza un Broadcast que se encargará de gestionar la situación

  sendBroadcast(bcIntent);

//Si es desde BootCompletedReceiver
}else{
  //Lanza el servicio TelemetriaServiceListener
  Intent msg2Intent = new Intent(context,
  TelemetriaListenerService.class);
  context.startService(msg2Intent);
 }
 //Destruye el servicio TelemetriaServiceListenerConDS y cierra
 //las conexiones
 super.onDestroy();
 this.endingService();

 //Si quedara algún receiver registrado lo desregistra
 if (serviceReceiver!=null) {
  unregisterReceiver(serviceReceiver);
  serviceReceiver=null;
 }
}
```

Este método se encarga de verificar si la aplicación ha fallado en algún momento y, en caso afirmativo, dependiendo de si el servicio "TelemetrisServiceListenerConDS" ha sido lanzado desde la actividad "BootCompletedReceiver" o "Menu", llama a los métodos necesarios para finalizarlo.

La implementación de todos los algoritmos llamados desde los algoritmos principales se encuentra descrita en el Apéndice D de la memoria.

6. Validación

Una vez realizada la implementación, para poder validar el correcto funcionamiento de la aplicación, se han realizado diferentes pruebas funcionales sobre la aplicación.

Las pruebas se han basado en la funcionalidad de la aplicación definida por las siguientes especificaciones:

6.1. Especificaciones técnicas

En este apartado se describen las variables utilizadas durante la implementación, así como las funcionalidades que presenta el módulo de diagnóstico implementado.

6.1.1. Variables definidas

Las variables persistentes utilizadas por la aplicación durante su ejecución son las siguientes:

VARIABLES SHAREDPREFERENCES			
Variable	**Tipo**	**Valores que puede tomar**	**Clase en las que se encuentra definida**
ACTIVITY	STRING	"BCR", "MENU"	BootCompletedReceiver Menu
MACOBD	STRING	Null o una MAC	BootCompletedReceiver
DESTROY	BOOLEAN	"True", "False"	TelemetriaListenerServiceConDS
EXITO	BOOLEAN	"True", "False"	TelemetriaListenerServiceConDS
CLEARCODES	BOOLEAN	"True", "False"	TelemetriaListenerServiceConDS Menu

NUM_ERR	STRING	Numero de errores detectados durante el análisis	TelemetriaListenerServiceConDS Menu
ERROREX	STRING	Errores detectados durante el análisis	TelemetriaListenerServiceConDS Menu
DIAGNOSTICOX	STRING	Errores que se muestran en el diagnostico	Menu
CONT_DIAG	Int	Numero de errores del diagnostico	Menu

Tabla 10: Variables SharedPreferences

6.1.2. Funcionalidad

En este apartado se explican las diferentes funcionalidades que el módulo de diagnóstico implementado posee. Para facilitar la comprensión se separarán las diferentes funcionalidades del módulo en actividades y servicios desarrollados.

6.1.2.1. Actividad "BootCompletedReceiver"

Se parte de la base de que el usuario se ha registrado en la aplicación y tiene la sesión iniciada.

El usuario arranca el coche con lo que se inicia la centralita dónde la aplicación se encuentra instalada, arrancando el receiver **BootCompletedReceiver (BCR)**. Como precaución, por si hubiera algún servicio iniciado primero se detienen, estos servicios son **TelemetriaServiceListener** y **TelemetriaServiceListenerConDS**. Se almacena mediante la variable persistente **ACTIVITY** que la actividad que ha lanzado el servicio es **BootCompletedReceiver**. Y se obtiene el valor almacenado en la variable **MACOBD**:

- Si la variable está vacía se lanza **TelemetriaServiceListener**

- Si la variable está informada se lanza **TelemetriaServiceListenerConDS**

Caso 1: Se lanza **TelemetriaServiceListener**

Este servicio se encarga de obtener los datos provenientes del GPS. No obstante, el funcionamiento de la clase **TelemetriaServiceListener** no entra dentro del TFM, por lo que no se va a comprobar su correcto funcionamiento, aunque sí se comprobará que éste se lanza de forma correcta, y no da ningún error.

Caso 2: Se lanza **TelemetriaServiceListenerConDS**

6.1.2.2. Servicio "TelemetriaServiceListenerConDS"

La primera vez que se instancia el servicio se lanza el método **onCreate** encargado de crear el servicio; en posteriores llamadas el servicio se lanzará el método **onStartCommand** encargado de iniciarlo.

En el método **onCreate** se configura el servicio y se lanza el método **startDevice**. En el método **startDevice** se define la variable DESTROY a *false*, variable que será modificada más adelante dependiendo del comportamiento de la aplicación, y se obtiene la variable MACOBD:

- En caso de que la variable esté informada → Se intenta la conexión con el OBD:
    - Si la conexión tiene éxito → Se configura el OBD:
        - Si la configuración tiene éxito → Se actualiza la variable **EXITO** a *true*. La variable **DESTROY** ya se encuentra definida a *false*.
        - Si la configuración NO tiene éxito → Se actualiza la variable **EXITO** a *false* y **DESTROY** a *true.*
    - Si la conexión NO tiene éxito → Se actualiza la variable **EXITO** a *false* y la variable **DESTROY** a *true.*
- En caso de no estarlo (este estado no debería darse nunca, pero se controla por si acaso) → Se actualiza la variable **EXITO** a *false* y la variable **DESTROY** a *true*.

Se lanza el método **onStartCommand**, que obtiene la variable **DESTROY:**

- Si **DESTROY** = *true* → se lanza el método **onDestroy**
- Si **DESTROY** = *false* → se obtiene la variable **CLEARCODES**
    - Si **CLEARCODES** = *true* → se limpian los códigos de error y se actualiza la variable **NUM_ERR** a 0 y **CLEARCODES** a *false.*
    - Si **CLEARCODES** = *false* → se obtiene el número de errores. Si el valor obtenido es diferente de 0, se almacena el valor obtenido en **NUM_ERR** y se le pide al OBDII estos errores que se procesan

y almacena en la variable **ERRORES.** Sí es igual a 0, se actualiza la variable **NUM_ERR** a 0.

Si durante este proceso algo fallara, se actualizarían las variables **EXITO** a *false* y **DESTROY** a *true*, y se llamaría al método **onDestroy**. Si el proceso finaliza con éxito se obtiene el **ACTIVITY** que ha lanzado el servicio.

- Si se ha llamado desde **BCR** →se obtiene la variable **EXITO**:
  - Si **EXITO** = *true* → se lanza el mensaje de aviso: "OBD conectado y configurado".
  - Si **EXITO** = *false* → se lanza el mensaje de aviso: "La conexión y configuración del OBD ha fallado". Se obtine la variable **DESTROY** y si el valor obtenido es igual a *false* se lanza el método **onDestroy**.
- Si se ha llamado desde **MENU** → se envía un Broadcast que se explicará en la clase MENU

El método **onDestroy** actualiza la variable **EXITO** a *false* y la variable **DESTROY** a *true*. Se obtiene la actividad que ha lanzado el servicio:

- Si lo lanza la actividad **BCR** → se lanza el servicio **TelemetriaServiceListener.**
- Si lo lanza la actividad **MENU** → se envía un Broadcast que se explicará en la clase MENU.

Por último, se destruye el servicio, se finaliza la comunicación del socket y se desregistra el receiver.

6.1.2.3. Actividad "Menu"

Siempre se lanzará después de la actividad **BCR**.

Se le indica a la aplicación que la actividad que ha sido lanzada es **MENU**, y se definen los servicios **TelemetriaServiceListener** y **TelemetriaServiceListenerConDS**. Se registra el receiver para el segundo servicio.

Se pregunta al usuario si quiere conectarse a un OBD:

- Botón **ACEPTAR** → Se detienen ambos servicios, en caso de que el Bluetooth esté apagado se enciende:
  - No se encuentra ningún dispositivo sincronizado → Se muestra el mensaje de aviso: "Ningún dispositivo sincronizado", se cierra

la caja del diálogo y se lanza el servicio **TelemetriaServiceListener.**

- Se encuentran dispositivos sincronizados → Se lanza el mensaje de aviso "Si tu OBD no aparece, sincronízalo". Aparece una lista con los dispositivos sincronizados. Se guarda en la variable **MACOBD** la MAC del dispositivo seleccionado de la lista.
  - Si **MACOBD** = vacío → Se lanza **TelemetriaServiceListener**
  - Si **MACOBD** = informado → Se lanza **TelemetriaServiceListenerConDS** y el mensaje de aviso "Espera mientras conectamos el OBD".

- Botón **CANCELAR** → No se realiza ninguna acción. Se queda encendido el servicio activo.

Cuando la actividad Menu recibe el mensaje lanzado por el broadcast se obtiene la variable **EXITO**:

- Si **EXITO** = *true* → Se lanza el mensaje de aviso: "OBD conectado y configurado".
- Si **EXITO** = *false* → Se lanza el mensaje de aviso: "La conexión y configuración del OBD ha fallado". Se obtiene la variable **DESTROY** y, si el resultado obtenido es igual a *true*, se detiene el servicio **TelemetriaServiceListenerConDS** y se lanza **TelemetriaServiceListener**.

6.1.2.4. Actividad "Diagnostico"

Se obtiene el diagnóstico de los errores mediante el método **realizarDiagnostico.** Para ello comprueba si el servicio **TelemetriaListenerServiceConDS** está iniciado:

- En caso de estar iniciado → Se comprueba el número de errores almacenado en la variable **NUM_ERR**:
  - Si **NUM_ERR** = 0 → Se lanza el mensaje de aviso: "No se han encontrado errores!".
  - Si **NUM_ERR** != 0 → Se recorre la variable **ERRORX** recuperando los errores almacenados. Se comparan los errores almacenados con los que hay almacenados en el Array **DIAGNOSTICO**. Si el error que se encuentra almacenado en la variable **ERRORX** NO se encuentra almacenada en el Array **DIAGNOSTICO** se crea un nuevo diagnóstico con el error, y se almacena en este Array. Se actualizan las variables de diagnóstico **DIAGNOSTICOX** con el

nuevo error. Se actualiza la variable **CONT_DIAG** con el número de variables **DIAGNOSTICOX** existentes. Se muestran los errores.

- Si NO lo está se lanza el mensaje de aviso: "Ups! Parece que no estás conectado a tu OBD".

Método **clearCodes**, realiza el borrado de errores. Aparece un mensaje emergente con el mensaje: "Va a borrar los errores. ¿Estás seguro?":

- Si se pulsa Aceptar → Primero actualiza la variable **CLEARCODES** a *true*, y seguidamente se obtiene la variable **CONT_DIAG**:
  - Si **CONT_DIAG** != 9999 → se recuperan todas las variables **DIAGNOSTICOX** actualizándolas a **NO_DATA,** y se actualiza la variable **CONT_DIAG** a 9999. Se lanza el servicio **TelemetriaServiceListenerConDS** para borrar los errores. Se limpia la lista de diagnóstico **listaDiagnostico** y la variable **mapaCodigos**, se actualiza **NUM_ERR** a 0 y **ERROREX** a NO_DATA. Se vacía el array **DIAGNOSTICO** y se informa a null. Se actualiza la vista.
  - Si **CONT_DIAG** = 9999 → Se muestra el mensaje de aviso: "No hay errores que borrar".

Si se pulsa Cancelar →El mensaje emergente se cierra sin realizar ninguna acción.

6.2. Pruebas funcionales realizadas

Una vez vistas las especificaciones técnicas ya se entiende mucho mejor las pruebas funcionales realizadas sobre la aplicación que conforman el siguiente plan de pruebas:

Test	Caso de test	Resultado esperado
1	Arrancar el vehículo, arranca la actividad BCR y la variable MACOBD está vacía (primera conexión).	Se lanza el servicio TelemetriaServiceListener.
2	Arrancar el vehículo, arranca la actividad BCR y la variable MACOBD está informada.	Se lanza el servicio TelemetriaServiceListenerConDS.
3	Se lanza el servicio TelemetriaServiceListenerConDS desde la	Se lanza el método onDestroy, lo que

	actividad BCR. La conexión con el OBD falla.	destruye el servicio TelemetriaServiceListenerConDS y lanza el servicio TelemetriaServiceListener.
4	Se lanza el servicio TelemetriaServiceListenerConDS desde la actividad BCR. La conexión con el OBD tiene éxito, no hay errores.	Servicio iniciado, no hay errores. Mensaje de aviso: “OBD conectado y configurado”.
5	Se lanza el servicio TelemetriaServiceListenerConDS desde la actividad BCR. La conexión con el OBD tiene éxito, hay errores.	Servicio iniciado, errores procesados y almacenados. Mensaje de aviso: “OBD conectado y configurado”.
6	Se abre la aplicación, se lanza la actividad Menu y se abre un mensaje emergente preguntando si se quiere conectar con el OBDII. Hay que contestar que no.	El mensaje emergente se cierra.
7	Se intenta realizar un diagnóstico sin estar conectados al OBD.	Mensaje de aviso: "Ups! Parece que no estas conectado a tu OBD".
8	Abrir la aplicación, se lanza la actividad Menu y se abre un mensaje emergente preguntando si se quiere conectar con el OBDII. Hay que contestar que sí, pero no hay ningún OBD sincronizado.	Mensaje de aviso: "Ningún dispositivo sincronizado". Se cierra el mensaje emergente y se lanza el servicio TelemetriaServiceListener.
9	Abrir la aplicación, se lanza la actividad Menu y se abre un mensaje emergente preguntando si se quiere conectar con el OBD. Hay que contestar que sí, hay un dispositivo sincronizado, pero no es el OBD.	Mensaje de aviso: "Si tu OBD no aparece, sincronízalo". Aparece una lista con los dispositivos sincronizados.
10	Abrir la aplicación, se lanza la actividad Menu y se abre un mensaje emergente preguntando si se quiere conectar con el OBD. Hay que contestar que sí, seleccionar el OBD y conectarse.	Se lanza TelemetriaServiceListenerConDS, mensaje de aviso: "Espera mientras conectamos el OBD".

11	El servicio TelemetriaServiceListenerConDS se lanza desde la actividad Menu. La conexión tiene éxito, no hay errores.	Mensaje de aviso: “OBD conectado y configurado”.
12	Realizar un diagnóstico sin haber encontrado errores.	Mensaje de aviso: “No se han encontrado errores!”.
13	El servicio TelemetriaServiceListenerConDS se lanza desde la actividad Menu. La conexión tiene éxito, sí hay errores.	Mensaje de aviso: "OBD conectado y configurado".
14	Realizar un diagnóstico.	Se muestran los errores obtenidos.
15	Repetir el diagnóstico.	No se deben duplicar los errores.
16	Salir a la actividad Menu y volver a entrar a la actividad diagnóstico para repetir el diagnóstico.	No se deben duplicar los errores.
17	Salir de la aplicación y volver a entrar para realizar otro diagnóstico.	No se deben duplicar los errores.
18	Borrar los errores.	Los errores son borrados.
19	Salir a la actividad Menu y volver a entrar a la actividad diagnóstico para realizar un diagnóstico después del borrado de errores.	Los errores borrados no aparecen.
20	Salir de la aplicación y volver a entrar para realizar otro diagnóstico después del borrado de errores.	Los errores borrados no aparecen.
21	Desde la actividad Menu: Con el servicio TelemetriaServiceListenerConDS activo, sincronizar con otro OBD.	Mensaje de aviso: "La conexión con el OBD se ha detenido". Mensaje de aviso: "OBD conectado y configurado".
22	Desde la actividad Menu: Intertar la conexión con el OBD y falla.	Se lanza el método onDestroy lo que lanza el boadcast de Menu, se destruye el servicio TelemetriaServiceListenerConDS.

	Mensaje de aviso: "La conexión con el OBD se ha detenido". Se lanza TelemetriaServiceListener.

Tabla 11: Casos de prueba

El proceso que se ha seguido para la realización de las pruebas es la de probar mediante ciclos, es decir, en cada ciclo se prueban todos los test definidos anteriormente, arreglando en cada nuevo ciclo los errores que han surgido en el ciclo anterior.

En este caso, para la correcta implementación de la aplicación, han sido realizados dos ciclos completos. Los resultados obtenidos en cada ciclo son los siguientes:

Test	Resultado ciclo 1	Resultado ciclo 2
1	OK	OK
2	OK	OK
3	KO, el servicio no se destruye.	OK
4	OK	OK
5	OK	OK
6	OK	OK
7	OK	OK
8	OK	OK
9	OK	OK
10	OK	OK
11	OK	OK
12	OK	OK
13	OK	OK
14	OK	OK
15	KO, se duplican los errores.	OK
16	KO, se duplican los errores.	OK

17	KO, se duplican los errores.	OK
18	KO, no se borran los errores.	OK
19	Dependencia con el caso anterior, no se lanza.	OK
20	Dependencia con el caso anterior, no se lanza.	OK
21	KO, el primer mensaje se queda bloqueado en pantalla durante la fase de conexión.	OK
22	OK	OK

Tabla 12: Resultados pruebas

Solo han hecho falta dos ciclos debido a que el análisis de las especificaciones y la creación de un plan de pruebas se ha realizado para llevar a cabo solamente la validación final, cuando la mayoría de las funcionalidades ya funcionaban correctamente. Esto se debe a que, durante la implementación, se han ido haciendo pruebas parciales de la aplicación para ir comprobando el correcto funcionamiento de las funcionalidades desarrolladas. No obstante, durante esta primera parte, no se ha establecido un plan de pruebas ni se ha seguido ningún método de testeo, por lo que se entra en detalles más allá de algunos problemas relevantes para la implementación y la solución llevada a cabo.

Los principales problemas que se han encontrado durante la implementación de la aplicación y las soluciones implementadas han sido los siguientes:

El primer gran problema, que no se refleja en el anterior plan de pruebas, ha sido saber si se podían utilizar dos servicios en una misma aplicación. Después de realizar un estudio mediante la API de Android y diferentes pruebas con la aplicación, se descubre que esto es posible siempre que, antes de lanzar un servicio, se compruebe que no haya otros servicios lanzados que puedan entrar en conflicto. Para solucionar este problema se hace un control antes de lanzar cualquier nuevo servicio, y en este control se apagan ambos servicios antes de lanzar el deseado.

El segundo problema ha sido que no se conseguía destruir correctamente el servicio. Se ha solucionado controlando los errores mediante bloques try-catch y forzando la llamada al método onDestroy cuando era necesario, así como mediante la utilización de la variable DESTROY que se comprobaba en momentos clave de

ejecución para ver si el servicio había tenido algún comportamiento no esperado y debía ser destruido.

El tercer problema observado es que los errores eran duplicados cada vez que se hacía un diagnóstico. Esto se ha solucionado guardando un array con todos los errores ya diagnosticados. Así, cuando se pedía un nuevo diagnóstico, se comprobaban los errores que ya se encontraban en este array y que no volvían a ser creados y por tanto no se duplicaban.

El cuarto error a corregir fue que, aunque daba la impresión de que se conseguían borrar los errores (la lámpara de fallo motor se apagaba), estos continuaban mostrándose en la aplicación. Esto se debió a que la variable utilizada en la visualización del diagnóstico utilizaba una lista que no estaba siendo borrada durante el método de borrado de errores, con lo cual, aunque se estaba gestionando bien la parte de la comunicación con el OBD, los diagnostico anteriores seguían almacenados en dicha lista de forma persistente. Esto se solucionó borrando la lista junto al borrado de datos del OBD.

Finalmente, el último error a solucionar fue que, cuando la aplicación estaba conectada a un OBD y se intentaba conectar a otro, el mensaje de que el servicio TelemetriaServiceListenerConDS se había detenido permanecía fijo hasta que la conexión con el nuevo OBD terminaba, cosa que podía inducir a error ya que el usuario podía pensar que la aplicación había fallado. Se solucionó asociando el mensaje a la actividad Menu y no al mensaje emergente de conexión que se cerraba a posteriori, y era el causante de que éste permaneciera fijo.

7. Conclusiones

El objetivo principal de este Trabajo Fin de Máster ha sido es el desarrollo de un módulo de diagnóstico del vehículo para la aplicación SmartCar, aprovechando tecnologías punteras en el ámbito del "Smart Driving". Así mismo, algunos de los objetivos específicos alcanzados han sido los siguientes:

- Tener una interfaz amigable, sencilla y cuidada.
- Dar una explicación sencilla y clara de los códigos de error obtenidos.
- Definir la gravedad de la avería y su coste.
- Definir la localización de la avería.
- Realizar un estudio sobre la posibilidad de resetear los indicadores de fallo en el salpicadero del vehículo.

Para poder llevar a cabo todos los objetivos se ha seguido una metodología basada en testeo cíclico, es decir, se ha definido un plan de pruebas con todas las funcionalidades que la aplicación debía cubrir y, cada vez que se ha implementado una parte, se han realizado las pruebas definidas en el plan de pruebas para testear esa funcionalidad. En caso de que alguna funcionalidad no funcionara correctamente, se han realizado diferentes ciclos de testeo, acompañados de los cambios pertinentes en la implementación hasta llegar a un ciclo dónde no se obtuviera ningún error.

Considero, así mismo que ha sido un proyecto complejo desde la parte técnica, ya que ha permitido profundizar en tecnologías no utilizadas durante la carrera, y ha permitido entender el potencial de la integración de vehículos y aplicaciones móviles mediante un interfaz al bus del vehículo, tal y como ofrece el estándar OBDII.

7.1. Trabajo futuro

Aunque se puede echar un vistazo global del trabajo futuro a realizar sobre la aplicación en general, este apartado se centrará en el trabajo futuro que se podría realizar sobre el bloque de diagnóstico implementado durante el desarrollo de este TFM.

Hay que centrarse en 3 grandes rasgos a mejorar y o ampliar del bloque desarrollado: (i) enriquecimiento de la base de datos, (ii) ubicación visual del error y (iii) notificaciones de alerta.

Estas ampliaciones permitirían conocer todos los detalles relacionados con el error obtenido del vehículo, detalles que son necesarios para la correcta comprensión del error y de su gravedad.

Además, aunque actualmente la interfaz está pensada para dar toda la información del error en la descripción de éste, una forma de aligerar esta descripción y obtener una interfaz mucho más visual es la visualización de la ubicación del error mediante un mapa. Esta funcionalidad, con la cual se podría ampliar el proyecto, se encargaría de mostrar de forma gráfica, en un mapa del vehículo, dónde se encuentra ubicada la avería, y a que componentes afecta. Se podría añadir, así mismo, un código de colores que indicara al mismo tiempo la gravedad de ésta.

Finalmente, se podría implementar un sistema de alertas que recordara al usuario que se han detectado errores que afectan al funcionamiento del vehículo, y que se debe realizar una revisión para solucionar los problemas, permitiendo elegir al usuario cuando quiere recibirlas.

Apéndice A. Manual de usuario

En este apéndice, se explicará el funcionamiento de la aplicación desarrollada en forma de manual de usuario para facilitar la comprensión de esta y su uso.

A.1. Registro

Lo primero que se debe hacer antes de poder utilizar la aplicación es registrarse en ella. Para ello se debe pulsar el botón "¿Eres nuevo? Regístrate" de la página principal.

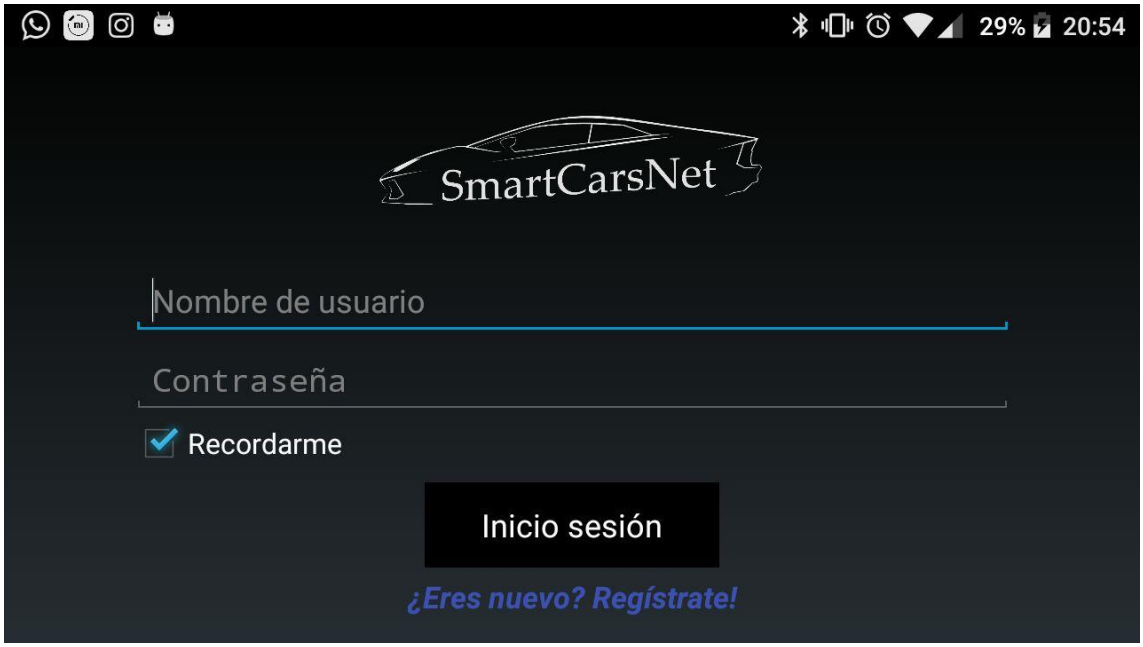

Figura 32: Log-in SmartCar

Este botón llevará a un cuestionario que se debe rellenar con los datos del usuario. Una vez rellenado se debe pulsar el botón "Siguiente".

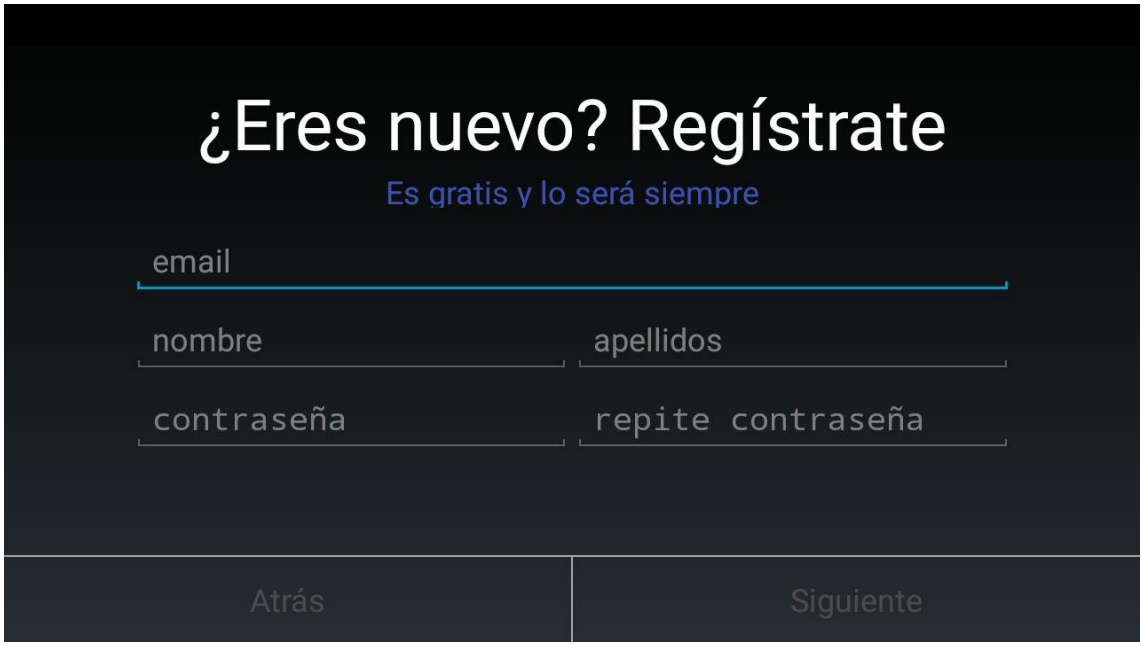

Figura 33: Formulario registro

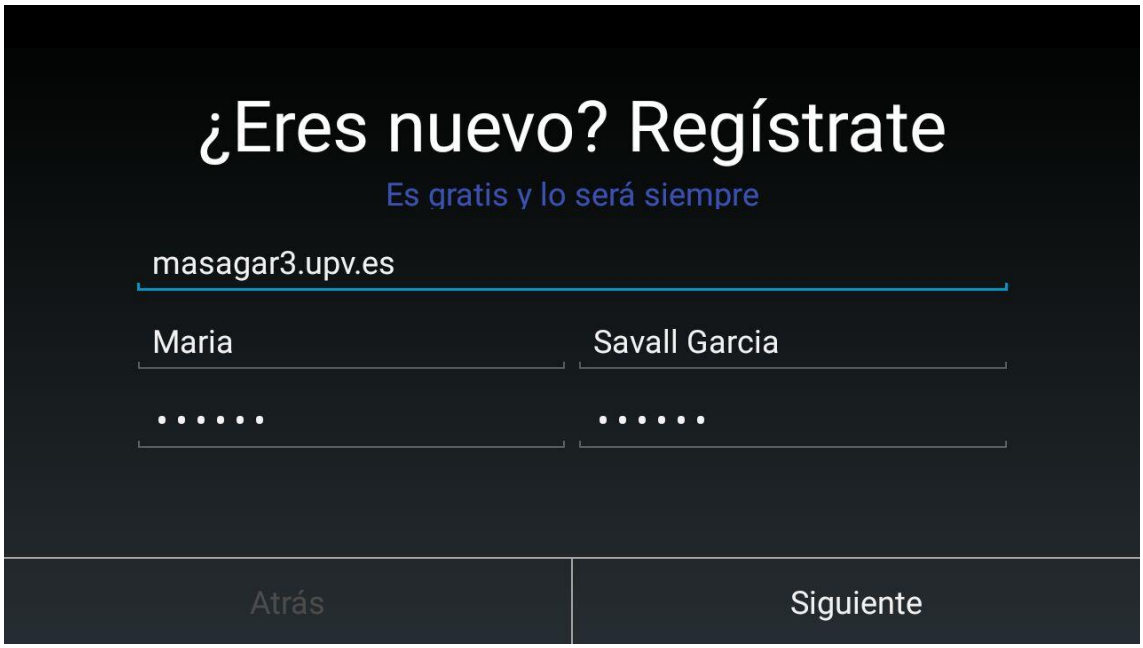

Figura 34: Formulario registro rellenado

Finalmente se debe rellenar un cuestionario con los datos del vehículo y pulsar "Registrar". Con esto el usuario ya está registrado y se podría empezar a utilizar la aplicación.

Figura 35: Formulario registro vehículo

Figura 36: Formulario registro vehículo relleno

A.2. Acceso a la aplicación

Una vez registrados, para el correcto funcionamiento de la aplicación se debe iniciar sesión. Para ello se rellena la pantalla de acceso con el correo y la contraseña elegida durante el registro. Si se quiere que la aplicación recuerde nuestros datos se debe seleccionar también la opción "Recordarme". A continuación, pulsar al botón "Inicio sesión".

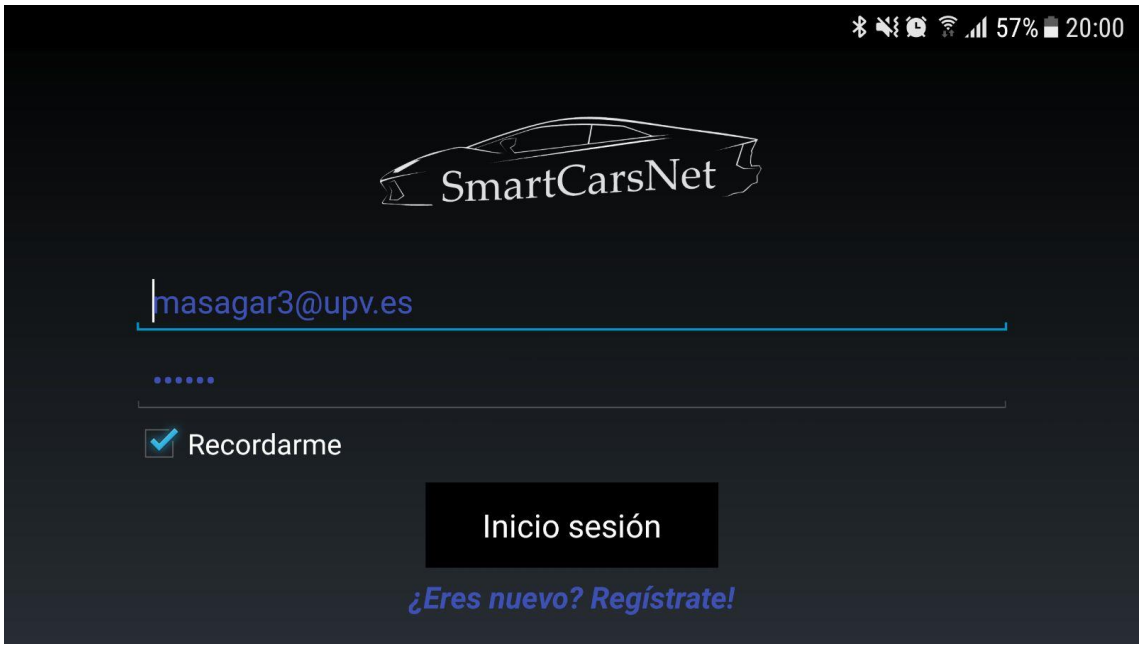

Figura 37: Log-in rellenado

A.3. Conexión con el OBDII

Durante la primera conexión a la aplicación se elige si conectarse o no al OBDII mediante un mensaje emergente que aparece; esto tendrá un impacto en el comportamiento de la aplicación de cara al lanzamiento posterior de los servicios. Es decir, durante la primera conexión se lanzará de forma predeterminada el servicio TelemetriaServiceListener, quien se encarga de recoger los datos del GPS. En caso de que la respuesta a la ventana emergente sea que no se quiere conectar al OBDII, este servicio permanecerá lanzado y será el servicio que se lance en cada reinicio del dispositivo sin necesidad de volver a abrir la aplicación. En caso contrario, se apagará el servicio TelemetriaServiceListener, y se lanzará el servicio TelemetriaServiceListenerConDS, encargado de recoger los datos del GPS y del OBDII. En este caso, este será igualmente el servicio que se lanzará en cada reinicio, cosa que permitirá la conexión automática con el OBDII seleccionado, pero con la

excepción de que, si falla, se apagará para lanzar el primero. Este funcionamiento se repetirá cada vez que se acceda al menú de la aplicación, ya que el mensaje emergente aparecerá cada vez por si se decide cambiar de OBD.

El procedimiento para conectarse al OBD es el siguiente:

Contestar “Aceptar” a la pregunta “No está conectado a ningún dispositivo Bluetooth OBDII. ¿Desea conectarse?”.

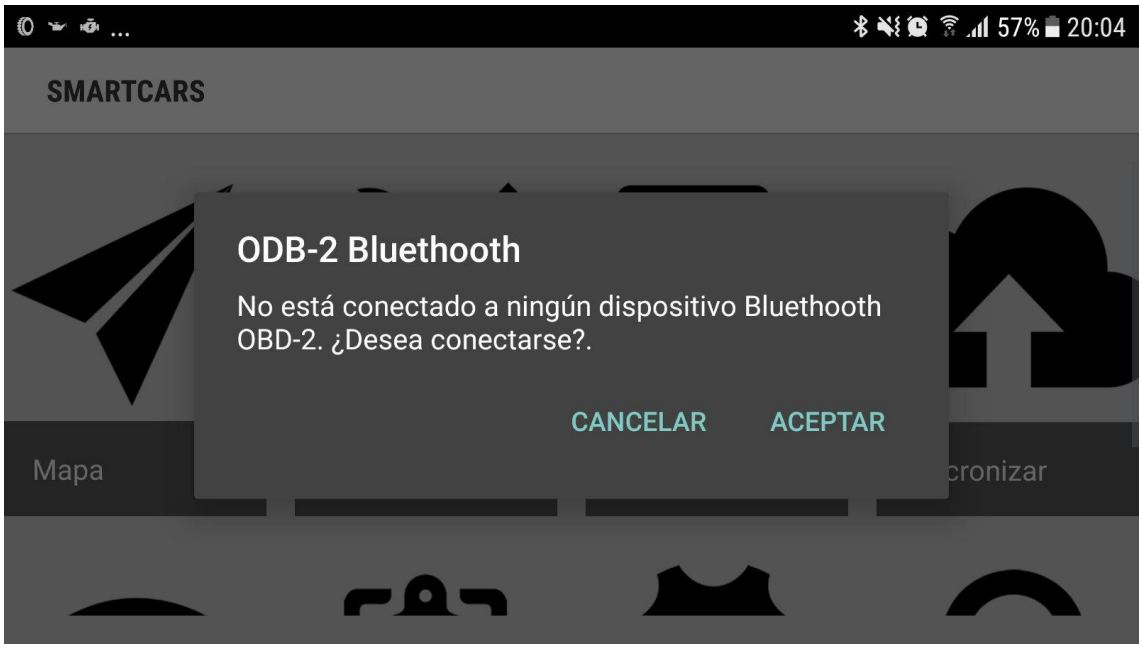

Figura 38: Pop-up de conexión OBDII

Esto abrirá una lista con los dispositivos Bluetooth previamente sincronizados con la Tablet del vehículo. En caso de que el OBD no se encontrara, se debe a que primero se deben sincronizar los dispositivos, cosa que la aplicación te indica.

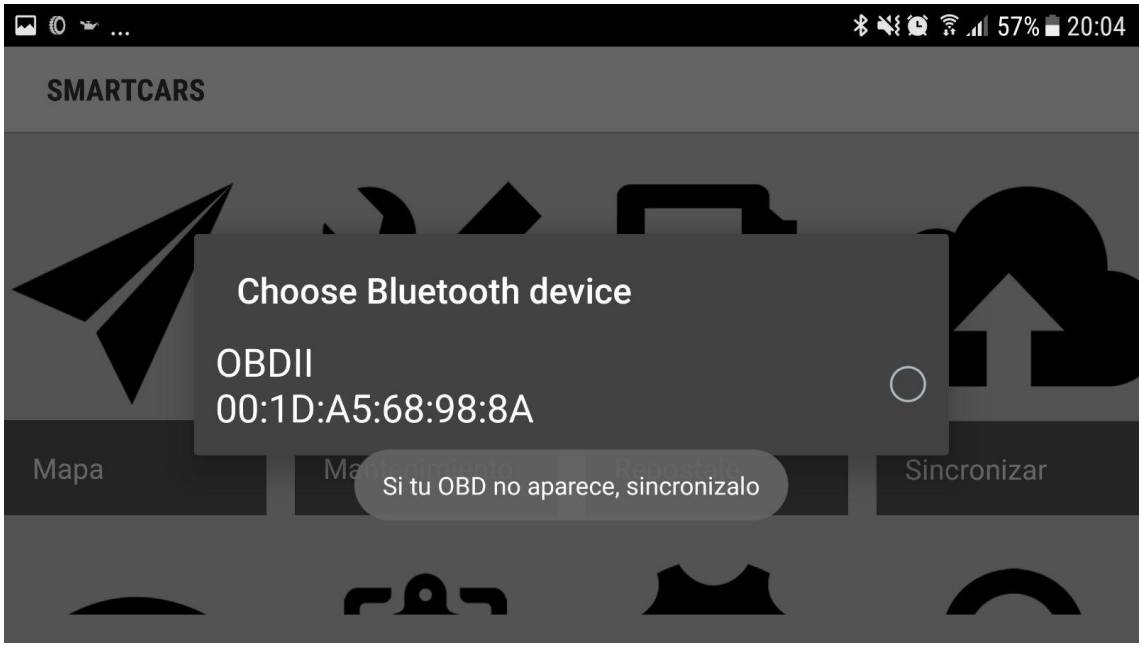

Figura 39: Dispositivos sincronizados

Seleccionar el dispositivo OBD deseado, y con esto ya se lanza de forma automática el servicio encargado de la conexión. El mensaje “Espera mientras nos conectamos al OBD” indica que la conexión se está llevando a cabo.

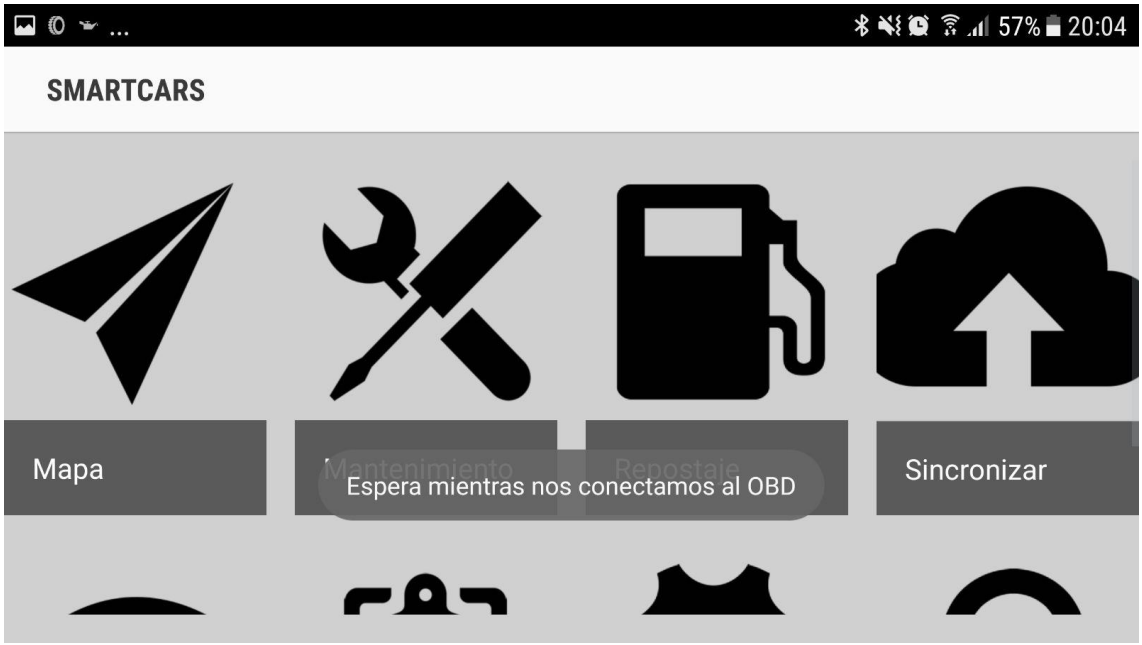

Figura 40: Pop-up de espera

Finalmente, si la conexión ha tenido éxito se obtiene el mensaje “OBD conectado y configurado”.

Figura 41: Pop-up de conexión

A.4. Realizar diagnóstico

Una vez se conecta con el OBD ya se puede realizar diagnósticos. La conexión es muy importante, puesto que de otra manera el diagnostico no se realizará.

Para realizar el diagnostico entrar en el apartado “Diagnostico”.

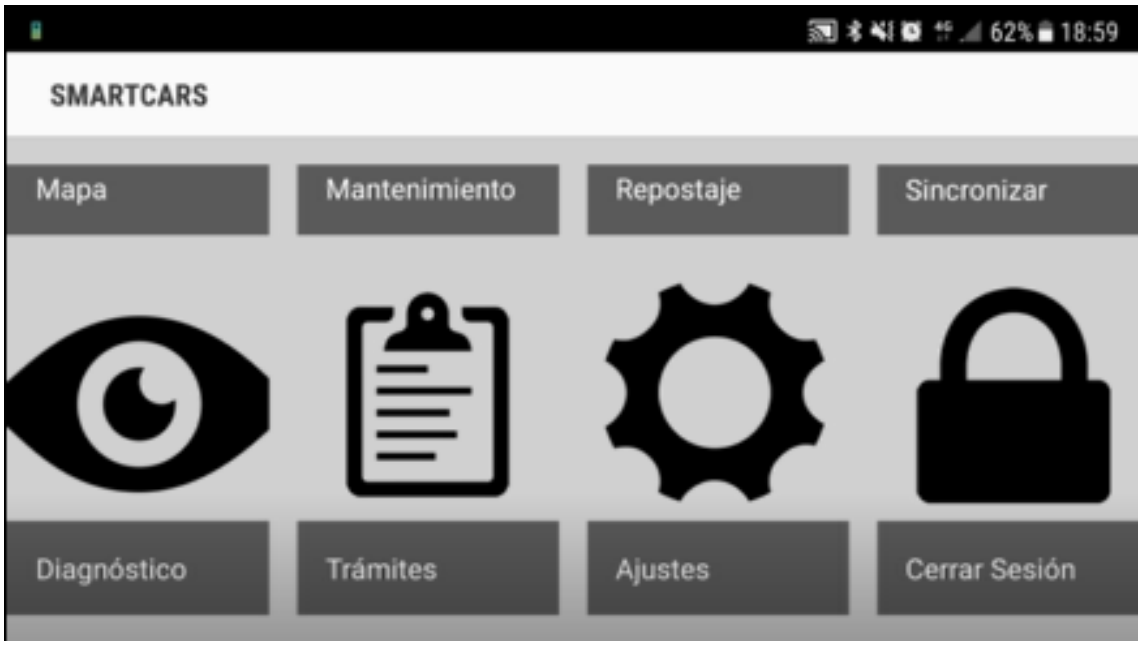

Figura 42: Menú

Pulsar el botón con el símbolo “+”.

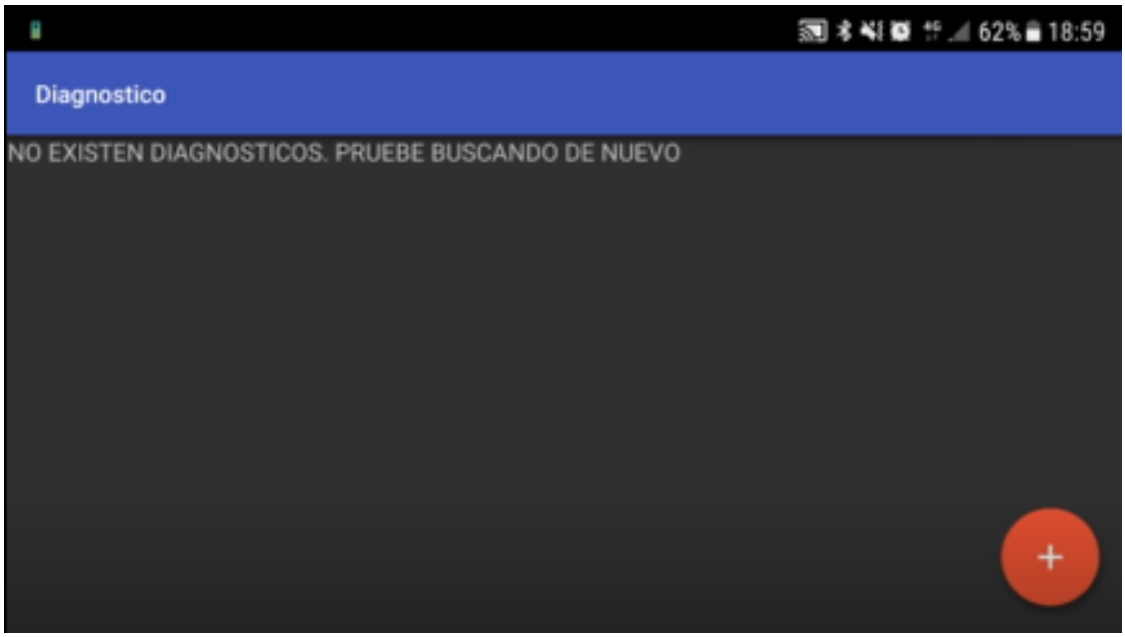

Figura 43: Interfaz diagnóstico

Y la opción “Realizar un diagnóstico”.

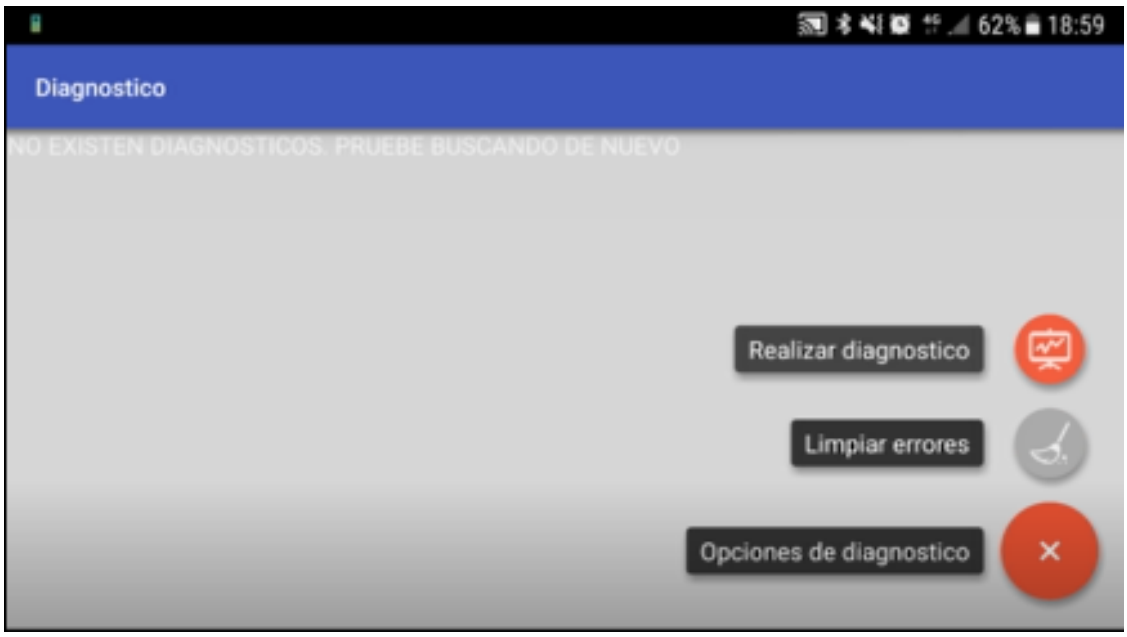

Figura 44: Menú diagnóstico

Con esto los errores detectados por el OBD aparecerán en pantalla en caso de haberlos. Sino se obtendrá el mensaje: “No se han encontrado errores!”

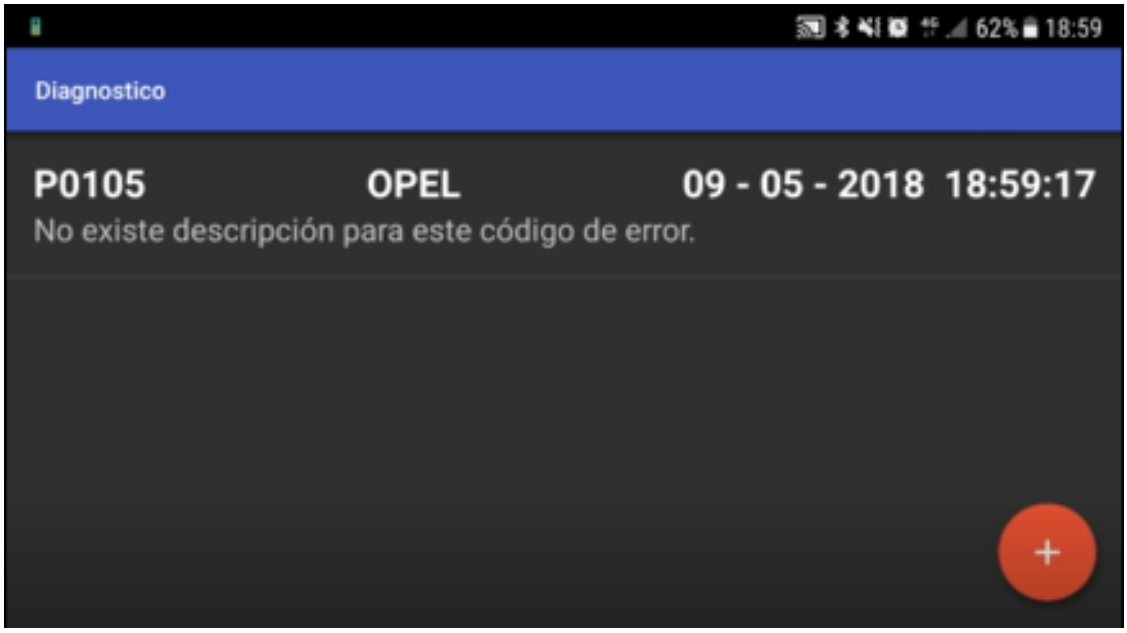

Figura 45: Interfaz con errores

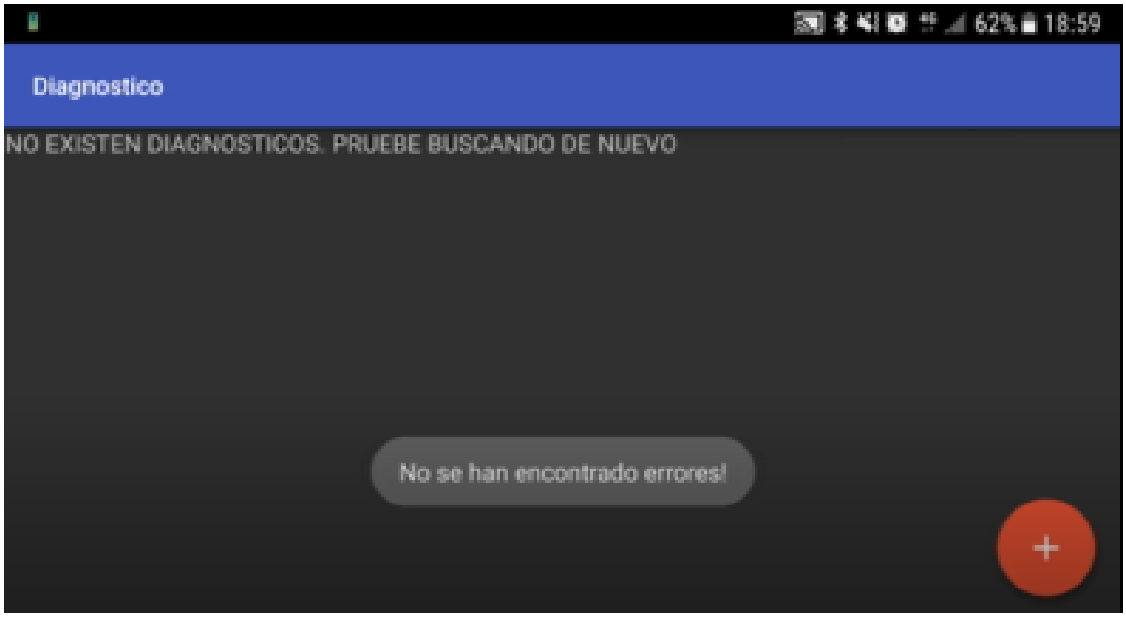

Figura 46: Pop-up sin errores

A.5. Borrar diagnóstico

Una vez detectados los errores se pueden borrar. Para ello volver al apartado “Diagnostico” y pulsar “+”.

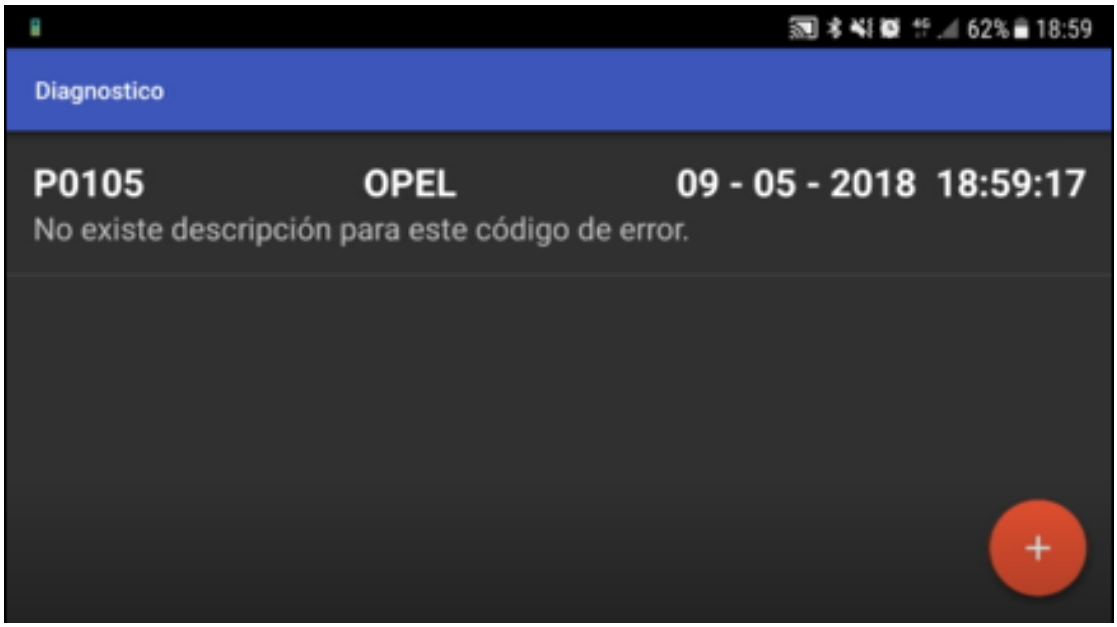

Figura 47: Ejemplo borrado (interfaz con errores)

En el menú desplegado seleccionar la opción “Limpiar errores”, y confirmar el mensaje emergente que aparece. Con esto se consigue borrar los errores tanto de la aplicación como del OBDII.

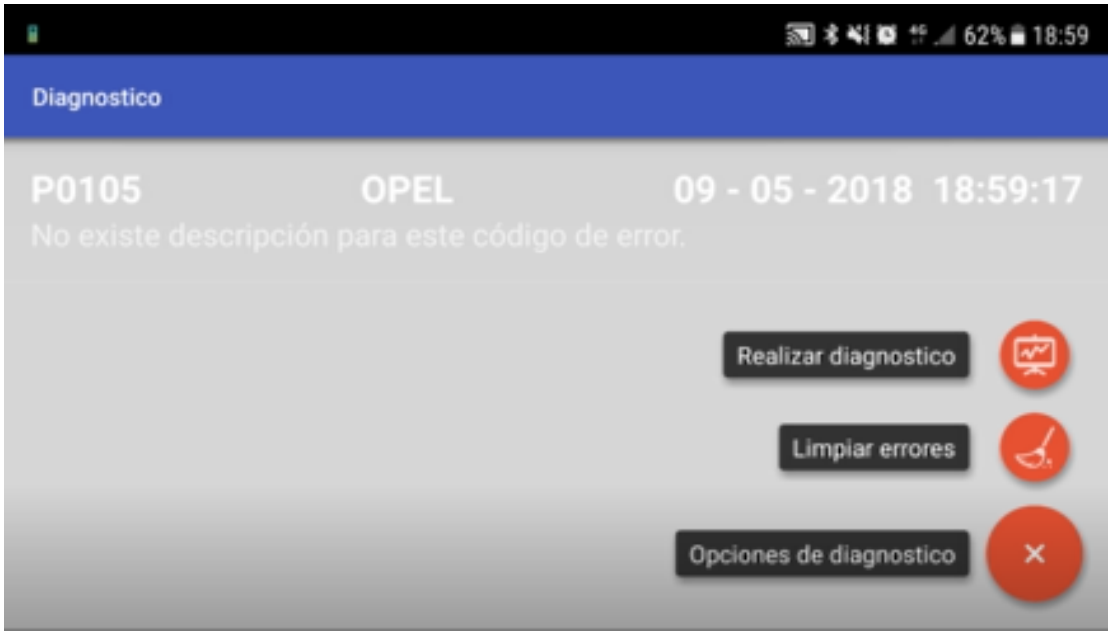

Figura 48: Ejemplo borrado (Menú)

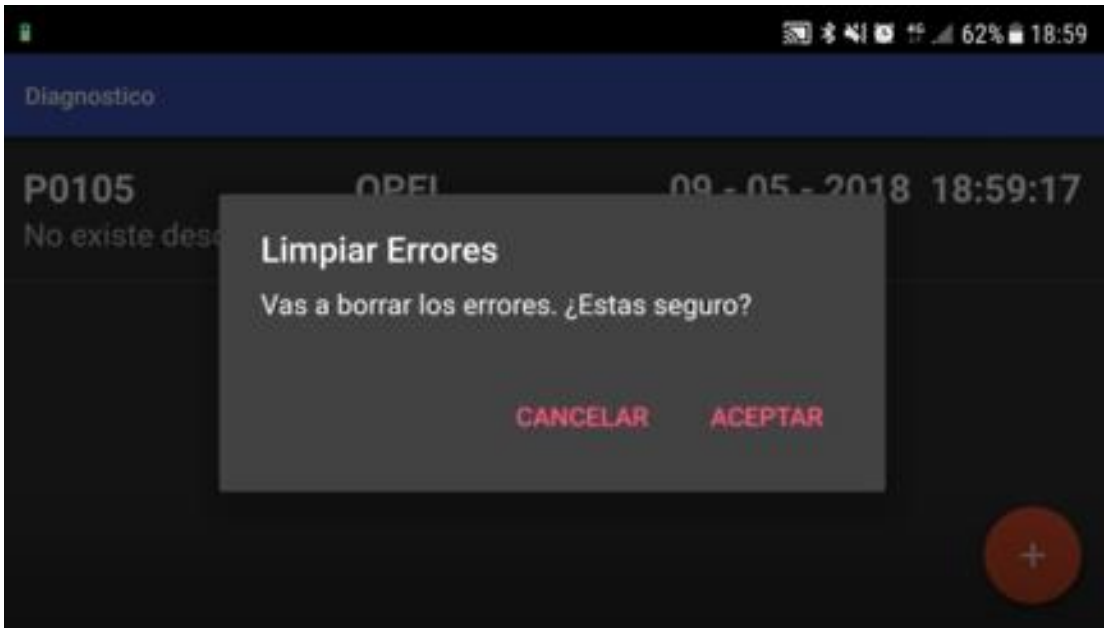

Figura 49: Borrado de errores (Pop-up confirmación)

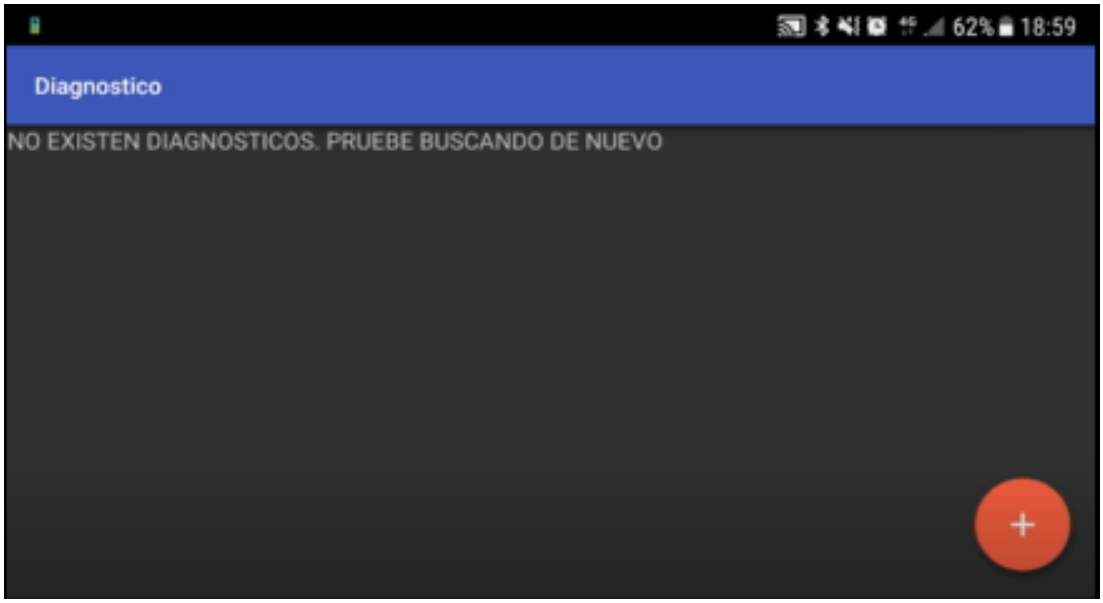

Figura 50: Borrado de errores (Interfaz limpia)

Apéndice B. Comandos AT

El OBDII, mediante el intérprete ELM327, puede ser configurado como el usuario desee. Para ello se utilizan los “Comandos AT”. Si la orden de configuración tiene éxito el intérprete obtendrá como respuesta la palabra “OK”, o en su defecto el valor solicitado durante la configuración. Los comandos se encuentran clasificados en varios tipos:

- Comandos AT Generales.
- Comandos AT de parámetros programables.
- Comandos AT OBD.
- Comandos AT Específicos ISO.
- Comandos AT Específicos CAN.
- Comandos AT Específicos J1939 CAN.

Valor	Descripción
<CR>	repeat the last command
BRD hh	try Baud Rate Divisor hh
BRT hh	set Baud Rate Timeout
D	set all to Default
E0, E1	Echo off, or on
FE	Forget Events
I	print the version ID
L0, L1	Linefeeds off, or on
LP	go to Low Power mode
M0, M1	Memory off, or on
RD	Read the stored Data
SD hh	Save Data byte hh
WS	Warm Start (quick software reset)

Z	reset all
@1	display the device description
@2	display the device identifier
@3	store the @2 identifier

Tabla 13: Comandos AT Generales

Valor	Descripción
PP xx OFF	disable Prog Parameter xx
PP FF OFF	all Prog Parameter disabled
PP xx ON	enable Prog Parameter xx
PP FF ON	all Prog Parameters enabled
PP xx SV yy	for PP xx, Set the Value to yy
PPS	print a PP Summary

Tabla 14: Comandos AT de parámetros programables

Valor	Descripción
AL	Allow Long (>7 bytes) message
ACM	display Activity Monitor Count
AMT hh	set the Activity Mon Timeout to hh
AR	Automatically Receive
AT0,1,2	Adaptative Timeout off, auto 1, auto 2
BD	perform a Buffer Dump
BI	Bypass the initialization sequence
DP	Describe the current Protocol
DPN	Describe the Protocol by Number
H0,H1	Headers off, or on

MA	Monitor All
MR hh	Monitor for Receiver = hh
MT hh	Monitor for Transmitter = hh
NL	Normal Length messages
PC	Protocol Close
R0,R1	Response off, or on
RA hh	set the Receive Address to hh
S0,S1	print of Space off, or on
SH xyz	Set Header to xyz
SH xxyyzz	Set header to xxyyzz
SP h	Set protocol to h and save it
SP Ah	Set protocol to Auto, h and save it
SP 00	Erase stored protocol
SR hh	Set the Receive address to hh
SS	use Standard Search order (J1978)
ST hh	Set Timeout to hh x 4 msec
TA hh	set Tester Address to hh
TP h	Try protocol h
TP Ah	Try protocol h with Auto search

Tabla 15: Comandos AT OBD

Valor	Descripción
FI	perform a fast Initiation
IB 10	set the Baud rate to 10400
IB 48	set the Baud rate to 4800
IB 96	set the Baud rate to 9600

IIA hh	set ISO (slow) Init Address to hh
KW	display the Key Words
KW0, KW1	Key Word checking off, or on
SI	perform a Slow (5 baud) Initiation
SW hh	Set Wakeup interval to hh x 20 msec
WM	set the Wakeup messages

Tabla 16: Comandos AT Específicos ISO

Valor	Descripción
CEA	turn off CAN Extended Addressing
CEA hh	use CAN Extended Address hh
CAF0, CAF1	Automatic Formatting off, or on
CF hhh	set the ID Filter to hhh
CF hhhhhhhh	set the ID Filter to hhhhhhhh
CFC0, CFC1	Flow Controls off, or on
CM hh	set the ID Mask to hhh
CM hhhhhhhh	set the ID Mask to hhhhhhhh
CP hh	set CAN Priority to hh (29bit)
CRA	reset the Receive Address filter
CRA hhh	set CAN Receive Address to hhh
CRA hhhhhhhh	set the Rx Address to hhhhhhhh
CS	show the CAN Status counts
CSM0, CSM1	Silent Monitoring off, or on
D0, D1	display of the DLC off, or on
FC SM h	Flow Control, Set the mode to h
FC SM hhh	FC, Set the Header to hhh

FC SH hhhhhhhh	Set the Header to hhhhhhhh
PB xx yy	Protocol B options and baud rate
RTR	send a RTR message
V0, V1	use of Variable DLC off, or on

Tabla 17: Comandos AT Específicos CAN

Valor	Descripción
DM1	monitor for DM1 message
JE	use J1939 Elm data format
JHF0, JHF1	Header Formatting off, or on
JS	use J1939 SAE data format
JTM1	set Timer Multiplier to 1
JTM5	set Timer Multiplier to 5
MP hhhh	Monitor for PGN hhhh
MP hhhhhh	Monitor for PGN hhhhhh

Tabla 18: Comandos AT Específicos J1939 CAN

Apéndice C. Parámetros ID

Los “Parámetros ID” o “PID” son los códigos que el intérprete ELM327 utiliza para comunicarse con el sistema de diagnóstico de a bordo del vehículo.

El estándar del protocolo OBDII conocido como SAE J1979, establece de forma predefinida los 10 modos de operar que se han visto anteriormente, y que se va a recordar:

Modo (hex)	Descripción
01	Muestra los parámetros disponibles.
02	Muestra los datos almacenados por evento.
03	Muestra los códigos de fallas de diagnóstico (Diagnostic Trouble Codes, DTC).
04	Borra los datos almacenados, incluyendo los códigos de fallas (DTC).
05	Resultados de la prueba de monitoreo de sensores de oxígeno (solo aplica a vehículos sin comunicación Controller Area Network, CAN).
06	Resultados de la prueba de monitoreo de components/sistema (resultados de la prueba de monitoreo de sensores de oxígeno en vehículos con comunicación CAN).
07	Muestra los códigos de fallas (DTC) detectados durante el último ciclo de manejo o el actual.
08	Operación de control de los componentes/sistema a bordo.
09	Solicitud de información del vehículo.
0A	Códigos de fallas (DTC) permanentes (borrados).

Tabla 19: Modos de funcionamiento

Además, como complemento a estos modos de funcionamiento, la Sociedad de Ingenieros Automotrices (SAE) definió, bajo el estándar J1939, una serie de PID estándares para cada uno de los modos de funcionamiento definidos anteriormente, aunque esto no quiere decir que éstos tengan que ser forzosamente implementados por los fabricantes de vehículos, ya que estos últimos tienen libertad para decidir cuáles van a implementar, e incluso pueden añadir sus propios códigos personalizados.

Los PID han sido implementados principalmente para el modo de funcionamiento 01 y 02. No obstante, y debido a que para llevar a cabo este trabajo son imprescindibles los modos 03 y 04, se ha decidido añadir la información de todos los PID estándar de estos 4 modos, completando así la información dada anteriormente:

C.1. Modo 01

PID (hex)	Data bytes devueltos	Descripción
00	4	PIDs implementados [01 - 20].
01	4	Estado de los monitores de diagnóstico desde que se borraron los códigos de fallo DTC; incluye el estado de la luz indicadora de fallo, MIL, y la cantidad de códigos de fallo DTC.
02	2	Almacena los códigos de fallo de diagnóstico DTC de un evento.
03	2	Estado del sistema de combustible.
04	1	Carga calculada del motor.
05	1	Temperatura del líquido de enfriamiento del motor.
06	1	Ajuste de combustible a corto plazo → Banco 1.
07	1	Ajuste de combustible a largo plazo → Banco 1.
08	1	Ajuste de combustible a corto plazo → Banco 2.
09	1	Ajuste de combustible a largo plazo → Banco 2.
0A	1	Presión del combustible.
0B	1	Presión absoluta del colector de admisión.
0C	2	RPM del motor.
0D	1	Velocidad del vehículo.
0E	1	Avance del tiempo.

0F	1	Temperatura del aire del colector de admisión.
10	2	Velocidad del flujo del aire MAF.
11	1	Posición del acelerador.
12	1	Estado del aire secundario controlado.
13	1	Presencia de sensores de oxígeno (en 2 bancos).
14	2	Sensor de oxígeno 1 A: Voltaje B: Ajuste de combustible a corto plazo
15	2	Sensor de oxígeno 2 A: Voltaje B: Ajuste de combustible a corto plazo
16	2	Sensor de oxígeno 3 A: Voltaje B: Ajuste de combustible a corto plazo
17	2	Sensor de oxígeno A: Voltaje B: Ajuste de combustible a corto plazo
18	2	Sensor de oxígeno 5 A: Voltaje B: Ajuste de combustible a corto plazo
19	2	Sensor de oxígeno 6 A: Voltaje B: Ajuste de combustible a corto plazo
1A	2	Sensor de oxígeno 7 A: Voltaje

		B: Ajuste de combustible a corto plazo
1B	2	Sensor de oxígeno 8 A: Voltaje B: Ajuste de combustible a corto plazo
1C	1	Estándar OBD implementado en este vehículo.
1D	1	Sensores de oxígenos presentes en el banco 4.
1E	1	Estado de las entradas auxiliares.
1F	2	Tiempo desde que se puso en marcha el motor.
20	4	PID implementados [21 - 40].
21	2	Distancia recorrida con la luz indicadora de falla (Malfunction Indicator Lamp, MIL) encendida.
22	2	Presión del tren de combustible, relativa al colector de vacío.
23	2	Presión del medidor del tren de combustible (Diesel o inyección directa de gasolina).
24	4	Sensor de oxígeno 1 AB: Relación equivalente de combustible – aire CD: Voltaje
25	4	Sensor de oxígeno 2 AB: Relación equivalente de combustible – aire CD: Voltaje
26	4	Sensor de oxígeno 3 AB: Relación equivalente de combustible – aire CD: Voltaje
27	4	Sensor de oxígeno 4 AB: Relación equivalente de combustible – aire CD: Voltaje

28	4	Sensor de oxígeno 5 AB: Relación equivalente de combustible – aire CD: Voltaje
29	4	Sensor de oxígeno 6 AB: Relación equivalente de combustible – aire CD: Voltaje
2A	4	Sensor de oxígeno 7 AB: Relación equivalente de combustible – aire CD: Voltaje
2B	4	Sensor de oxígeno 8 AB: Relación equivalente de combustible – aire CD: Voltaje
2C	1	EGR comandado.
2D	1	falla EGR.
2E	1	Purga evaporativa comandada.
2F	1	Nivel de entrada del tanque de combustible.
30	1	Cantidad de calentamientos desde que se borraron los fallas.
31	2	Distancia recorrida desde que se borraron los fallas.
32	2	Presión de vapor del sistema evaporativo.
33	1	Presión barométrica absoluta.
34	4	Sensor de oxígeno 1 AB: Relación equivalente de combustible – aire CD: Actual
35	4	Sensor de oxígeno 2 AB: Relación equivalente de combustible – aire

		CD: Actual
36	4	Sensor de oxígeno 3 AB: Relación equivalente de combustible – aire CD: Actual
37	4	Sensor de oxígeno 4 AB: Relación equivalente de combustible – aire CD: Actual
38	4	Sensor de oxígeno 5 AB: Relación equivalente de combustible – aire CD: Actual
39	4	Sensor de oxígeno 6 AB: Relación equivalente de combustible – aire CD: Actual
3A	4	Sensor de oxígeno 7 AB: Relación equivalente de combustible – aire CD: Actual
3B	4	Sensor de oxígeno 8 AB: Relación equivalente de combustible – aire CD: Actual
3C	2	Temperatura del catalizador: Banco 1, Sensor 1.
3D	2	Temperatura del catalizador: Banco 2, Sensor 1.
3E	2	Temperatura del catalizador: Banco 1, Sensor 2.
3F	2	Temperatura del catalizador: Banco 2, Sensor 2.
40	4	PID implementados [41 - 60].
41	4	Estado de los monitores en este ciclo de manejo.

42	2	Voltaje del módulo de control.
43	3	Valor absoluto de carga.
44	2	Relación equivalente comandada de combustible – aire.
45	1	Posición relativa del acelerador.
46	1	Temperatura del aire ambiental.
47	1	Posición absoluta del acelerador B.
48	1	Posición absoluta del acelerador C.
49	1	Posición del pedal acelerador D.
4A	1	Posición del pedal acelerador E.
4B	1	Posición del pedal acelerador F.
4C	1	Actuador comandando del acelerador.
4D	2	Tiempo transcurrido con MIL encendido.
4E	2	Tiempo transcurrido desde que se borraron los códigos de fallos.
4F	4	Valor máximo de la relación de equivalencia de combustible - aire, voltaje del sensor de oxígenos, corriente del sensor de oxígenos y presión absoluta del colector de entrada.
50	4	Valor máximo de la velocidad de flujo de aire del sensor de flujo de aire masivo.
51	1	Tipo de combustible.
52	1	Porcentaje de combustible Etanol.
53	2	Presión absoluta del vapor del sistema de evaporación.
54	2	Presión del vapor del sistema de evaporación.
55	2	Ajuste del sensor de oxígeno secundario de plazo corto. A: banco 1 B: banco 3

56	2	Ajuste del sensor de oxígeno secundario de plazo largo. A: banco 1 B: banco 3
57	2	Ajuste del sensor de oxígeno secundario de plazo corto. A: banco 2 B: banco 4
58	2	Ajuste del sensor de oxígeno secundario de plazo largo. A: banco 2 B: banco 4
59	2	Presión absoluta del tren de combustible.
5A	1	Posición relativa del pedal del acelerador.
5B	1	Tiempo de vida del banco de baterías híbridas.
5C	1	Temperatura del aceite del motor.
5D	2	Sincronización de la inyección de combustible.
5E	2	Velocidad del combustible del motor.
5F	1	Requisitos de emisiones para los que el vehículo fue diseñado.
60	4	PID implementados [61 - 80].
61	1	Porcentaje de torque solicitado por el conductor.
62	1	Porcentaje de torque actual del motor.
63	2	Torque de referencia del motor.
64	5	Datos del porcentaje de torque del motor.
65	2	Entrada / salida auxiliar implementada.
66	5	Sensor de flujo de aire masivo.
67	3	Temperatura del enfriador del motor.
68	7	Sensor de temperatura de aire de entrada.

69	7	EGR comandado y falla de EGR.
6A	5	Control comandado del flujo de aire de entrada de Diesel y posición relativa de la entrada del flujo de aire.
6B	5	Temperatura de recirculación del gas del escape.
6C	5	Control comandado del actuador del acelerador y posición relativa del acelerador.
6D	6	Sistema de control de presión del combustible.
6E	5	Sistema de control de presión de inyección.
6F	3	Presión de entrada del compresor del turbo cargador.
70	9	Control de presión de aumento.
71	5	Control del turbo de geometría variable (Variable Geometry Turbo, VGT).
72	5	Control de la compuerta de desperdicio.
73	5	Presión del escape.
74	5	RPM del turbo cargador.
75	7	Temperatura del turbo cargador.
76	7	Temperatura del turbo cargador.
77	5	Temperatura del enfriador del aire de carga (Charge Air Cooler Temperature, CACT).
78	9	Temperatura del gas del escape (Exhaust Gas Temperature, EGT) Banco 1.
79	9	Temperatura del gas del escape (Exhaust Gas Temperature, EGT) Banco 2.
7A	7	Filtro de partículas Diesel (Diesel Particulate Filter, DPF).
7B	7	Filtro de partículas Diesel (Diesel Particulate Filter, DPF).
7C	9	Temperatura del filtro de partículas Diesel (Diesel Particulate Filter, DPF).

7D	1	Estado del área de control NOx NTE.
7E	1	Estado del área de control PM NTE.
7F	13	Tiempo que el motor ha estado en marcha.
80	4	PID implementados [81 - A0].
81	21	Tiempo de marcha del motor para el dispositivo auxiliar de control de emisiones (Auxiliary Emissions Control Device, AECD).
82	21	Tiempo de marcha del motor para el dispositivo auxiliar de control de emisiones (Auxiliary Emissions Control Device, AECD).
83	5	Sensor de NOx.
84		Temperatua de superficie del colector.
85		Sistema reactivo NOx.
86		Sensor de partículas (Particular Matter, PM).
87		Presión absoluta del colector de admisión.

Tabla 20: PIDs modo 01

C.2. Modo 02

Se utilizan los mismos PID que se han definido para el modo 01 pero con una diferencia, ya que muestran la información del evento almacenado.

C.3. Modo 03

PID (hex)	Data bytes devueltos	Descripción	Nota
N/A	N*6	Solicita los códigos de fallo.	3 códigos por bloque de mensaje.

Tabla 21: PIDs modo 03

C.4. Modo 04

PID (hex)	Data bytes devueltos	Descripción
N/A	0	Borra todos los códigos de fallos y apaga la luz indicadora de fallo (Malfunction Indicator Lamp, MIL).

Tabla 22: PIDs modo 04

Apéndice D. Métodos implementados

La implementación de los métodos utilizados por las funciones más importantes, explicadas en el punto "5. Ampliación propuesta de la aplicación", se encuentra detallada en este apéndice.

D.1. Método listOBD

Se encarga de listar y mostrar en un pop-up todos los dispositivos sincronizados con el dispositivo multimedia del vehículo. Cuando el usuario selecciona el OBDII, este método se encarga de iniciar el servicio "TelemetriaServiceListenerConDS" encargado de la conexión.

Algoritmo 9: Listado de dispositivos BT.

```
public void listOBD(){
 //Crea un nuevo dialogo para mostrar los dispositivos sincronizados
 final AlertDialog.Builder alertDialog = new AlertDialog.Builder(Menu.this);

//Añade los dispositivos al dialogo
adapter = new
          ArrayAdapter(Menu.this,android.R.layout.select_dialog_singlechoice,
          deviceStrs.toArray(new String[deviceStrs.size()]));

 //Espera la selección de un dispositivo
 alertDialog.setSingleChoiceItems(adapter, -1, new
                                    DialogInterface.OnClickListener() {

//Al detectar el click
public void onClick(DialogInterface dialog, int which){

 //Cierra el dialogo
 dialog.dismiss();
 //Captura la posición seleccionada de la lista
 int position =
       ((AlertDialog)dialog).getListView().getCheckedItemPosition();
//Obtiene el dispositivo seleccionado por el usuario
mDevice = arrayObjetos[position];
//Guarda en una variable persistente la MAC que se va a conectar
editor.putString("MACOBD", mDevice.getAddress());
editor.commit();

//Obttiene la variable MAC
macOBD = prefs.getString("MACOBD", "");

//Comprueba su estado
//Si está vacía
if(macOBD.equals("")){

 //Lanza el servicio "TelemetriaServiceListener"
 context.startService(i);
```

```
//Si está informada
}else{

  //Lanza el servicio "TelemetriaServiceListenerConDS"
  context.startService(j);
 }
}});

 //Se añade título al dialogo
 alertDialog.setTitle("Choose Bluetooth device");
 //Se muestra el dialogo con los dispositivos
 alertDialog.show();

 //Al seleccionar un dispositivo se muestra por pantalla un mensaje que
 //indica al usuario que debe esperar a la conexión del dispositivo
 Toast.makeText(getApplicationContext(),"Espera mientras nos conectamos
  al OBD",Toast.LENGTH_LONG).show();
 }
```

D.2. Método ConnectWithOBD

Se encarga de realizar la conexión con el OBD; en caso de que la conexión tenga éxito devuelve *true* sino devuelve *false*.

Algoritmo 10: Realiza la conexión con el OBDII.

```
 public boolean ConnectWithOBD(){
  try {
   //Abre el socket
   btSocket =
mDevice.createInsecureRfcommSocketToServiceRecord(UUID.fromString("000
01101-0000-1000-8000-00805F9B34FB"));
   //Conecta con el OBD
   btSocket.connect();
   //Abre un canal de transferencia de datos de entrada y salida
   in  = btSocket.getInputStream();
   out = btSocket.getOutputStream();
   //Si tiene éxito devuelve true
   return true;
  } catch (IOException e1) {
     //Si falla devuelve false
     return false;
  }
 }
```

D.3. Método resetDevice

Le envía al OBD el comando "ATZ", que realiza un reseteo del dispositivo y responde con el nombre de éste cuando se vuelve a iniciar. Mientras no recibe respuesta se mantiene en espera.

Algoritmo 11: Resetear el intérprete.

```
private void resetDevice() throws InterruptedException {
 while (!stop) {
  //Obtiene la respuesta enviada por ALM al comando "ATZ"
  String result = runATCommands(new OBDCode("ATZ")).replace(" ","");
  //Si se obtiene respuesta se sale del bucle
  if (result != null && result.contains("ELM")) {
   break;
  }
  Thread.sleep(2000);
 }
}
```

D.4. Método configureDevice

Le envía al OBD los comandos "ATE0", "ATM0", "ATAT0" y "ATST11", encargados de realizar las configuraciones explicadas en el apéndice B. Mientras no recibe respuesta se mantiene en espera.

Algoritmo 12: Realiza las diferentes configuraciones vistas anteriormente.

```
private void configureDevice() throws InterruptedException {
 //Define los comandos AT a enviar al OBD
 String[] configuraciones = {"ATE0", "ATM0", "ATAT0", "ATST11"};
 //Para cada comando definido
 for (String command : configuraciones) {
  while (!stop) {
   // Obtiene la respuesta enviada por ALM al comando enviado
   String result = runATCommands(new OBDCode(command)).replace(" ",
   "");
   //Si obtiene respuesta se sale del bucle
   if (result != null && result.contains("OK")) {
    break;
   }
   Thread.sleep(2000);
  }
 }
}
```

D.5. Método displayDevice

Le envía al OBD el comando "ATI" encargado de preguntar al dispositivo su nombre. Mientras no recibe respuesta se mantiene en espera.

Algoritmo 13: Obtención del nombre del dispositivo utilizado.

```
private void displayDevice() throws InterruptedException {
 while (!stop) {
  // Obtiene la respuesta enviada por ALM al comando "ATI"
  String result = runATCommands(new OBDCode("ATI"));
  //Si se obtiene respuesta se sale del bucle
  if (result != null && result.contains("ELM")) {
   break;
  }
  Thread.sleep(2000);
 }
}
```

D.6. Método searchProtocol

Le envía al OBD el comando "ATSP0" encargado de acordar el protocolo a utilizar, en este caso el 0. Mientras no recibe respuesta se mantiene en espera.

Algoritmo 14: Busca un protocolo para comunicarse con el OBDII.

```
private void searchProtocol() throws InterruptedException {
 while (!stop) {
  // Obtiene la respuesta enviada por ALM al comando "ATSP0"
  String result = runATCommands(new OBDCode("ATSP0")).replace("
  ","");
  //Si se obtiene respuesta se sale del bucle
  if (result != null && result.contains("OK")) {
   break;
  }
  Thread.sleep(2000);
 }
}
```

D.7. Método protocolSelected

Le envía al OBD el comando "ATDPN" encargado de preguntar por el protocolo escogido. Mientras no recibe respuesta se mantiene en espera.

Algoritmo 15: Se obtiene el protocolo escogido.

```
private void protocolSelected() throws InterruptedException {
 while (!stop) {
  //Obtiene la respuesta enviada por ALM al comando "ATDPN"
  String result = runATCommands(new OBDCode("ATDPN")).replace("","");
  //Si se obtiene respuesta se sale del bucle
  if (result != null) {
   break;
  }
  Thread.sleep(1500);
 }
}
```

D.8. Método runATCommands

Abre los canales de comunicación con el OBDII, y es el encargado de transmitir el comando AT correspondiente al OBDII, así como de obtener la respuesta de éste y transmitirla.

Algoritmo 16: Envía los mensajes al OBDII en el formato adecuado.

```
public String runATCommands(OBDCode code)throws InterruptedException{
 //Abre los canales de comunicación
 code.setInputStream(in);
 code.setOutputStream(out);
 code.start();
 //Establece un contador de los intentos realizados
 int nVeces=0;

 while (!stop) {
  //Espera a que termine el thread o 300ms
  code.join(300);
  //Si el código no contesta sale del bucle
  if (!code.isAlive()) {break;}
  //Si se han realizado más de 20 intentos sale del bucle
  if (nVeces++ >20){break;}
 }
 //Si ha salido del bucle porque ha realizado más de 20 intentos
 //responde con null
 if (nVeces++ >20){
  return null;
 //Sino devuelve la respuesta del OBD
 }else{
   return code.formatResultATcommands();
```

```
 }
}
```

D.9. Método initView

Muestra por pantalla todos los errores recuperados del servicio "TelemetriaServiceListenerConDS".

Algoritmo 17: Visualización de los errores en la aplicación.

```
private void initView(){
 //Obtiene los diagnósticos creados en el método anterior
 listaDiagnosticos =
diagImpl.getAllByCar(vehiImpl.getVehiculoByOwner(usrImpl.getUsuarioAct
ivo().getEmail()).getMatricula());

 //Si se han recuperado diagnósticos
 if(listaDiagnosticos != null && !listaDiagnosticos.isEmpty()){

  //Recupera todos los diagnósticos con su información
  mapaCodigos =
codImpl.getAllCodes(vehiImpl.getVehiculoByOwner(usrImpl.getUsuarioActi
vo().getEmail()).getMarca());

  //Crea un nuevo adaptador dónde se encuentran todos los
  //diagnósticos encontrados previamente
  adaptador = new DiagnosticoAdapter(this,
              listaDiagnosticos.toArray(new
              Diagnostico[listaDiagnosticos.size()]),mapaCodigos);
  //Esconde la etiqueta de que no hay diagnósticos
  txtv_diagnostico.setVisibility(View.GONE);
  //Muestra en pantalla el adapter
  lv_diagnostico.setAdapter(adaptador);

 //Si no hay diagnósticos
 }else{
   //Recupera un adaptador vacío
   adaptador2 = new DiagnosticoAdapter(this,
                listaDiagnosticos.toArray(new
                Diagnostico[listaDiagnosticos.size()]),mapaCodigos);
   //Devuelve en pantalla el adaptador vacía
   lv_diagnostico.setAdapter(adaptador2);
   //Muestra la etiqueta de que no hay diagnósticos
   txtv_diagnostico.setVisibility(View.VISIBLE);
 }
}
```

D.10. Método numPIDError

Se comunica con el OBDII para preguntarle si hay errores en el vehículo o no. En caso afirmativo calcula cuantos errores hay.

Algoritmo 18: Verifica si hay errores o no. Si hay los calcula.

```
private int numPIDError() throws InterruptedException {
 int numPIDError;
 while (!stop) {
  //Pregunta al OBDII si hay errores
  String result = runATCommands(new OBDCode("0101")).replace(" ","");
  //Al recuperar una respuesta diferente de null se sale del bucle
  if (result != null) {
   break;
  }
   Thread.sleep(1500);
 }
 //Limpia el mensaje obtenido
 result = result.replace("SEARCHING...","");
 //Si no ha habido respuesta
 if(result.contains("UNABLETOCONNECT")){
  //Indica que el OBD no ha contestado y se debe destruir el
  //servicio
  editor.putBoolean("DESTROY",true);
  editor.putBoolean("EXITO",false);
  editor.commit();
  //Se define numPIDerror con el flag 9999 que indica que algo a
  //fallado
  numPIDError = 9999;
 //Si ha habido respuesta
 }else{
  //Recupera los bits que indican el número de errores
  result = result.substring(4,6);
  //Si el número obtenido es 0 lo indica
  if(result.equals("00")){
   numPIDError = 0;
  //Si es diferente de 0 calcula los errores
  }else{
   //convierte los bits recuperados en un integer
   numPIDError = Integer.parseInt(result);
   //Le resta la parte de información del primer bit
   numPIDError = numPIDError - 80;
  }
 }
 //Devuelve el valor obtenido
 return numPIDError;
}
```

D.11. Método numPIDErrorRedondeado

Calcula los mensajes que debe recuperar para obtener todos los errores calculados anteriormente, ya que cada mensaje contiene 3 errores.

Algoritmo 19: Obtiene el número de mensajes exacto que debe recuperar.

```
private int numPIDErrorRedondeado(int numPIDError){
 int PIDredondeado;
 BigDecimal PIDredondeadoBig;
 //Si el número obtenido en el método anterior es igual a 0 debemos
 //recuperar 0 mensajes
 if(numPIDError == 0){
  PIDredondeado = 0;
 //Sino
 }else{
   //Convierte los errores obtenidos a BigDecimal
   PIDredondeadoBig = BigDecimal.valueOf(numPIDError);
   //Los dividimos entre 3 y redondeamos
   PIDredondeadoBig = PIDredondeadoBig.divide(BigDecimal.valueOf(3),
                      RoundingMode.CEILING);
   //Convierte el valor obtenido en un Integer y lo asigna a la
   //variable que va a ser devuelta
   PIDredondeado = Integer.valueOf(PIDredondeadoBig.intValue());
  }
  //Devuelve el valor obtenido
  return PIDredondeado;
}
```

D.12. Método limpiaArrayMensajes

De todos los mensajes recuperados obtiene solo los que contienen errores.

Algoritmo 20: Filtra los mensajes que contienen errores.

```
private String[] limpiaArrayMensajes(String[] res, int PIDredondeado){
    //Obtiene todos los mensajes tengan o no errores
    String[] allMessages = res;
    //Crea una array con el tamaño de errores a recuperar
    String[] allMessagesClean = new String[PIDredondeado];
    //Recupera los primeros mensajes del primer array hata el tamaño
    //del segundo, ya que son los mensajes que contienen errores
    for(int i=0; i<allMessagesClean.length; i++){
        allMessagesClean[i] = allMessages[i];
    }
    //Devuelve el array con los mensajes que contienen errores
    return allMessagesClean;
}
```

D.13. Método separaErrores

Como cada mensaje contiene 3 errores, este método se encarga de separarlos.

Algoritmo 21: Separa los errores obtenidos en los mensajes.

```
private String[] separaErrores(String[] res, int numPIDError){

 //Array que recuperará los errores, se prevé un espacio adicional por
 //si los dos 8 últimos bits vinieran vacíos y no hubieran sido
 //contados como errores
 String[] PIDs = new String[numPIDError+2];
 //Guarda temporalmente los PID de cada mensaje
 String[] PIDsTemp = new String[3];
 //Recupera el array de mensajes con errores
 String[] allMessagesClean = res;
 int puntero = 0;

//Mientras haya mensajes
for(int i=0; i<allMessagesClean.length;i++){
 //Limpia la cabecera
 allMessagesClean[i] = allMessagesClean[i].replace("43","");
 for(int j=0; j<=2;j++){
  if(j==0){
   //Obtiene el primer error
   PIDsTemp[j] = allMessagesClean[i].substring(0,4);
  }else{
    if(j == 1){
      //Obtiene el segundo error
      PIDsTemp[j] = allMessagesClean[i].substring(4,8);
    }else{
    //Obtienr el tercer error
    PIDsTemp[j] = allMessagesClean[i].substring(8);
     }
    }
   }
   //Copia todos los errores separados en un array
   System.arraycopy(PIDsTemp, 0, PIDs, puntero, PIDsTemp.length);
   puntero = puntero+3;
  }
  //Devuelve el array con los errores separados
  return PIDs;
}
```

D.14. Método hexACode

Obtiene el primer bit de cada error y traduce la información adicional, obteniendo el código del error completo.

Algoritmo 22: Obtiene la información del primer byte traduciéndolo en un código de error.

```
private String[] hexACode(String[] res, int numPIDError){
 //Obtiene el array de PIDs
 String[] hex = res;
 //Crea un array para guardar los PIDs sin los bits últimos de
 //relleno
 String[] hexClean = new String[numPIDError];
 //Crea un array para guardar los PIDs traducidos
 String[] Errors = new String[numPIDError];

 for(int j=0; j<hex.length; j++){
  //Comprueba si el PID almacenado son bits de relleno si no lo
  //es lo copia en el nuevo array
  if(!hex[j].contains("0000")){
   hexClean[j]=hex[j];
  }
 }

 //Para cada PID en el array sin bits de relleno
 for(int i=0; i<hexClean.length; i++){
  //Obtiene el primer byte
  String primerByte = hexClean[i].substring(0,1);
  //Obtiene el resto de la string
  String resto = hexClean[i].substring(1);
  String code = "";

  //Si el primer byte es igual a 0
  if(primerByte.equals("0")){
   //Se traduce por P0
   code = "P0";
   //Se une la string P0 con el resto de bytes obtenidos anteriormente
   code = code + resto;
  }

  //Si el primer byte es igual a 1
  if(primerByte.equals("1")){
   //Se traduce por P1
   code = "P1";
   //Se une la string P1 con el resto de bytes obtenidos anteriormente
   code = code + resto;
  }

  //Si el primer byte es igual a 2
  if(primerByte.equals("2")){
   //Se traduce por P2
   code = "P2";
   //Se une la string P2 con el resto de bytes obtenidos anteriormente
   code = code + resto;
  }
```

```
//Si el primer byte es igual a 3
if(primerByte.equals("3")){
 //Se traduce por P3
 code = "P3";
 //Se une la string P3 con el resto de bytes obtenidos anteriormente
 code = code + resto;
}

//Si el primer byte es igual a 4
if(primerByte.equals("4")){
 //Se traduce por C0
 code = "C0";
 //Se une la string C0 con el resto de bytes obtenidos anteriormente
 code = code + resto;
}

//Si el primer byte es igual a 5
if(primerByte.equals("5")){
 //Se traduce por C1
 code = "C1";
 //Se une la string C1 con el resto de bytes obtenidos anteriormente
 code = code + resto;
}

//Si el primer byte es igual a 6
if(primerByte.equals("6")){
 //Se traduce por C2
 code = "C2";
 //Se une la string C2 con el resto de bytes obtenidos anteriormente
 code = code + resto;
}

//Si el primer byte es igual a 7
if(primerByte.equals("7")){
 //Se traduce por C3
 code = "C3";
 //Se une la string C3 con el resto de bytes obtenidos anteriormente
 code = code + resto;
}

//Si el primer byte es igual a 8
if(primerByte.equals("8")){
 //Se traduce por B0
 code = "B0";
 //Se une la string B0 con el resto de bytes obtenidos anteriormente
 code = code + resto;
}

//Si el primer byte es igual a 9
if(primerByte.equals("9")){
 //Se traduce por B1
 code = "B1";
 //Se une la string B1 con el resto de bytes obtenidos anteriormente
 code = code + resto;
}

//Si el primer byte es igual a A
if(primerByte.equals("A")){
 //Se traduce por B2
 code = "B2";
```

```
   //Se une la string B2 con el resto de bytes obtenidos anteriormente
   code = code + resto;
  }

  //Si el primer byte es igual a B
  if(primerByte.equals("B")){
   //Se traduce por B3
   code = "B3";
   //Se une la string B3 con el resto de bytes obtenidos anteriormente
   code = code + resto;
  }

  //Si el primer byte es igual a C
  if(primerByte.equals("C")){
   //Se traduce por U0
   code = "U0";
   //Se une la string U0 con el resto de bytes obtenidos anteriormente
   code = code + resto;
  }

  //Si el primer byte es igual a D
  if(primerByte.equals("D")){
   //Se traduce por U1
   code = "U1";
   //Se une la string U1 con el resto de bytes obtenidos anteriormente
   code = code + resto;
  }

  //Si el primer byte es igual a E
  if(primerByte.equals("E")){
   //Se traduce por U2
   code = "U2";
   //Se une la string U2 con el resto de bytes obtenidos anteriormente
   code = code + resto;
  }

  //Si el primer byte es igual a F
  if(primerByte.equals("F")){
   //Se traduce por U3
   code = "U3";
   //Se une la string U3 con el resto de bytes obtenidos anteriormente
   code = code + resto;
  }
  //Obtiene un array con todos los errores traducidos
  Errors[i] = code;
 }
 //Devuelve el array
 return Errors;
}
```

D.15. Método onReceiver

Gestiona los mensajes a mostrar al usuario y la destrucción del servicio "TelemetriaServiceListenerConDS" cuando éste ha sido lanzado desde la clase Menu.

Algoritmo 23: Broadcast de la clase Menu.

```
public class ProgressReceiver extends BroadcastReceiver {
 @Override
 public void onReceive(Context context, Intent intent) {
  //Comprueba que la acción asignada al intent es la correcta
  if(intent.getAction().equals(TelemetriaListenerServiceConDS.
  ACTION_FIN)) {
   //Comprueba si la aplicación ha tenido éxito o no (esto se debe a
   //que el broadcast no se lanza solo desde el método destroy)
   exito = prefs.getBoolean("EXITO", false);
   //Si ha tenido éxito
   if(exito){
    //Informa de que el OBD ha sido conectado y configurado
    //correctamente
    Toast.makeText(context, "OBD conectado y configurado",
    Toast.LENGTH_SHORT.show();
   // Si no ha tenido éxito
   }else{
    //Informa de que la conexión ha fallado
    Toast.makeText(context, "La conexión con el OBD se ha detenido",
    Toast.LENGTH_SHORT.show();
   }
   //Comprueba si la aplicación ha fallado en algún punto y ha sido
   //destruida
   if(destroy){
    //Si lo ha sido lanza el servicio TelemetriaServiceListener
    Intent msg2Intent = new Intent(context,
    TelemetriaListenerService.class);
    context.startService(msg2Intent);
   }
  }
 }
}
```

D.16. Método endingService

Cuando se destruye el servicio, se encarga de desbloquear los bucles y cerrar todas las conexiones abiertas.

Algoritmo 24: Cierra las conexiones abiertas.

```
public void endingService(){
 try{
  //Se desbloquean los bucles
  stop = true;
  //Se cierran los bluetoothSockets
  btSocket.close();
 } catch (IOException e) {
    e.printStackTrace();
 }
 //Se detiene el servicio
 this.stopSelf();
}
```

Referencias

[1] J. SANTOLOBO, «ABCdesevilla,» Las mejores aplicaciones Android para cuando se te avería el coche, 10 03 2015. [En línea]. Available: http://sevilla.abc.es/mobility/las_mejores_app/android/las-mejores-app-android/las-mejores-aplicaciones-android-para-cuando-se-te-averia-el-coche/. [Último acceso: 06 09 2017].

[2] F. RUIZ, «AndroidSis,» Torque OBD2, auténtico ordenador de a bordo a tiempo real, 12 05 2012. [En línea]. Available: https://www.androidsis.com/torque-obd2-autentico-ordenador-de-a-bordo-a-tiempo-real/. [Último acceso: 07 09 2017].

[3] J. ALVIZ, «clipset,» Torque Pro, tu coche monitorizado desde Android, 2014. [En línea]. Available: https://clipset.20minutos.es/torque-pro-tu-coche-monitorizado-desde-android/. [Último acceso: 11 09 2017].

[4] «Motorpasión,» Torque y REV, aplicaciones de diagnóstico OBD para Android e iPhone, 21 07 2011. [En línea]. Available: https://www.motorpasion.com/gadgets/torque-y-rev-aplicaciones-de-diagnostico-para-android-e-iphone. [Último acceso: 11 09 2017].

[5] «Palmer Performance Engineering,» DashCommand™, 2013. [En línea]. Available: https://www.palmerperformance.com/products/dashcommand/. [Último acceso: 11 09 2017].

[6] «Palmer Performance Engineering,» DashBoard Manual User, 07 10 2013. [En línea]. Available: https://www.palmerperformance.com/download/docs/DashCommand_User_Manual.pdf. [Último acceso: 12 09 2017].

[7] «Midas,» Midas Connect, [En línea]. Available: https://www.midas.es/midas-connect. [Último acceso: 20 09 2017].

[8] C. R. MOLINA, «tuexpertoapps.com,» Midas Connect, Uuna app para conocer el estado del coche desde el móvil, 14 03 2017. [En línea]. Available: https://www.tuexpertoapps.com/2017/03/14/midas-connect-una-app-para-conocer-el-estado-del-coche-desde-el-movil/. [Último acceso: 20 09 2017].

[9] M. Gaton, «Actualidad Gadget,» Midas Connect, lo último en conectividad para coches no conectados, 29 06 2017. [En línea]. Available: https://www.actualidadgadget.com/midas-connect-app/. [Último acceso: 20 09 2017].

[10] L. Gonzalez, «AudioLedCar,» Midas Connect: para qué sirve, cómo usarla y cómo descargarla, 20 04 2017. [En línea]. Available: https://audioledcar.com/blog/sobre-coches/midas-connect-sirve-usarla-descargarla/. [Último acceso: 20 09 2017].

[11] «Google Play,» iOBD2, 29 11 2016. [En línea]. Available: https://play.google.com/store/apps/details?id=com.xtooltech.ui&hl=es. [Último acceso: 18 09 2017].

[12] J. M. ULMEHER, «Ibertronica,» Android, ¿Qué es y cómo funciona?, [En línea]. Available: https://www.ibertronica.es/blog/tutoriales/android-sistema-operativo/. [Último acceso: 23 09 2017].

[13] «Android,» Versiones Android, 2014. [En línea]. Available: https://www.android.com/intl/es-419_mx/history/#/donut. [Último acceso: 23 09 2017].

[14] D. E. y A. , «Monografías,» Sistema Operativo Android, [En línea]. Available: http://www.monografias.com/trabajos101/sistema-operativo-android/sistema-operativo-android.shtml. [Último acceso: 23 09 2017].

[15] A. N. GONZALEZ, «Xataka,» ¿Que es Android?, 09 02 2011. [En línea]. Available: https://www.xatakandroid.com/sistema-operativo/que-es-android. [Último acceso: 23 09 2017].

[16] «Desde Linux,» Carácteristicas y cualidades de Android Studio, 24 05 2016. [En línea]. Available: https://blog.desdelinux.net/caracteristicas-y-cualidades-de-android-studio/. [Último acceso: 25 09 2017].

[17] «Prezi,» ¿Qué es y para qué sirve la android studio?, 23 06 2016. [En línea]. Available: https://prezi.com/5fiq7wucqt8z/que-es-y-para-que-sirve-la-android-studio/. [Último acceso: 25 09 2017].

[18] «Developers,» Meet Android Studio, 2017. [En línea]. Available: https://developer.android.com/studio/intro/. [Último acceso: 26 09 2017].

[19] «Academia Android,» Android Studio v1.0: características y comparativa con Eclipse, 11 12 2014. [En línea]. Available: https://academiaandroid.com/android-studio-v1-caracteristicas-comparativa-eclipse/. [Último acceso: 26 09 2017].

[20] M. A. ALVAREZ, «Qué es Java,» Descripción y características de este potente y moderno lenguaje de programación., 18 07 2001. [En línea]. Available: https://www.desarrolloweb.com/articulos/497.php. [Último acceso: 27 09 2017].

[21] J. PÉREZ PORTO y A. GARDEY, «Definición.de,» Definición de Java, 2013. [En línea]. Available: https://definicion.de/java/. [Último acceso: 27 09 2017].

[22] «Wikilibros,» Programación en Java/Características del lenguaje, 2017. [En línea]. Available: https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Java/Caracter%C3%ADsticas_del_lenguaje. [Último acceso: 27 09 2017].

[23] «OBDII ELM327,» Sistema OBDII, 2017. [En línea]. Available: http://obd2-elm327.es/sistema-obd2-historia-descripcion-futuro. [Último acceso: 28 09 2017].

[24] «Wikipedia,» OBD, 2017. [En línea]. Available: https://es.wikipedia.org/wiki/OBD. [Último acceso: 27 09 2017].

[25] E. MARTÍN, «tuexperto,» Qué es Bluetooth y para que sirve, 06 05 2013. [En línea]. Available: https://www.tuexperto.com/2013/05/06/que-es-bluetooth-y-para-que-sirve/. [Último acceso: 30 09 2017].

[26] «Culturación,» Todo lo que necesitas saber sobre el Bluetooth, [En línea]. Available: http://culturacion.com/todo-lo-que-necesitas-saber-sobre-el-bluetooth/. [Último acceso: 30 09 2017].

[27] J. PÉREZ PORTO y M. MERINO, «Definición.de,» Definición de Bluetooth, 2009. [En línea]. Available: https://definicion.de/bluetooth/. [Último acceso: 30 09 2017].

[28] «ValorTop,» ¿Qué es el Bluetooth y para qué sirve?, 03 2016. [En línea]. Available: http://www.valortop.com/blog/bluetooth. [Último acceso: 30 09 2017].

[29] J. C. CANO y C. TAVARES CALAFATE, «SmartCarsNet,» Vehicular Social Network, 2016. [En línea]. Available: http://www.smartcarsnet.com/. [Último acceso: 01 10 2017].