Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE
TELECOMUNICACIÓN

Arquitectura software de un vehículo autónomo usando ROS 2

Autor:
D. Víctor Rueda Domínguez
Tutor:
Dr. D. Juan Carlos Aguado Manzano
D. Adrián Mazaira Hernández

VALLADOLID, JULIO 2025

TRABAJO FIN DE GRADO

TÍTULO:	**Arquitectura software de un vehículo autónomo usando ROS 2**
AUTOR:	**D. Víctor Rueda Domínguez**
TUTOR:	**Dr. D. Juan Carlos Aguado Manzano** **D. Adrián Mazaira Hernández**
DEPARTAMENTO:	**Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE:	**Dr. D. Ignacio de Miguel Jiménez**
VOCAL:	**Dr. Dª. Noemí Merayo Álvarez**
SECRETARIO:	**Dr. D. Juan Carlos Aguado Manzano**
SUPLENTE:	**Dr. D. Ramón José Durán Barroso**
SUPLENTE:	**Dr. D. Juan Blas Prieto**

FECHA:

CALIFICACIÓN:

Resumen

En este Trabajo Fin de Grado se presenta el rediseño e implementación de la arquitectura *software* de un vehículo autónomo, basado en un Renault Twizy, utilizando como plataforma de desarrollo Robot Operating System 2 (ROS 2). Partiendo de un prototipo inicial compuesto por dos microordenadores HummingBoard –uno dedicado a comunicaciones y otro, denominado módulo de control, encargado de ejecutar las tareas de percepción, planificación y actuación–, se introduce un nuevo módulo de mayor rendimiento, denominado PC Fusión, encargado de procesar los datos de los sensores y ejecutar la fusión de información, liberando de carga computacional al módulo de control original. Se propone además una reorganización física de conexiones USB y CAN, así como la implementación de una conexión lógica basada en MQTT para integrar el PC Fusión en el sistema. Se diseñan y desarrollan nodos ROS 2 especializados (percepción de cámara, ultrasonidos, RFID, recepción y emisión CAN, fusión de datos y planificación de ruta) que se orquestan desde un nodo principal (`main_node`), mientras que el módulo de control queda dedicado exclusivamente a la actuación sobre acelerador, marchas y dirección.

Abstract

This Final Degree Project presents the redesign and implementation of the software architecture of an autonomous vehicle, based on a Renault Twizy, using Robot Operating System 2 (ROS 2) as the development platform. Starting from an initial prototype consisting of two HummingBoard microcomputers -one dedicated to communications and the other, called the control module, responsible for executing perception, planning and action tasks- a new, higher-performance module called PC Fusion is introduced, responsible for processing sensor data and executing information fusion, freeing the original control module from computational load. A physical reorganisation of USB and CAN connections is also proposed, as well as the implementation of an MQTT-based logical connection to integrate the Fusion PC into the system. Specialised ROS 2 nodes (camera perception, ultrasound, RFID, CAN reception and transmission, data fusion and route planning) are designed and developed, which are orchestrated from a main node (`main_node`), while the control module is dedicated exclusively to acting on the accelerator, gears and steering.

Palabras clave

Arquitectura *software*, CAN, MQTT, Módulo de control, PC Fusión, Percepción, Planificación, Renault Twizy, Robot Operating System 2, Vehículo autónomo.

Agradecimientos

Quisiera expresar mi más sincero agradecimiento a todas las personas que me han acompañado y apoyado durante la carrera y en la realización de este Trabajo Fin de Grado.

En primer lugar, agradezco a Juan Carlos, mi tutor, por su disponibilidad, orientación y compromiso a lo largo de todo el proyecto. Su apoyo y disposición para ayudar siempre que ha sido necesario han sido fundamentales para la consecución de este trabajo.

A David Manso, por su ayuda en momentos clave del proyecto, especialmente cuando me encontraba más perdido. Su conocimiento del sistema, fruto de su trabajo previo en el proyecto, fue de gran utilidad. En este contexto, también quiero mencionar a Ignacio Royuela y Adrián Mazaira por su apoyo puntual, pero útil y acertado.

A mi familia, por estar siempre a mi lado, por su cariño incondicional, por ser un apoyo constante, especialmente en los momentos más necesarios, y por creer en mí en cada etapa de este camino.

A mis amigos, por su compañía, ánimo y los buenos ratos compartidos, tan necesarios especialmente en los momentos más intensos del grado. En especial, a mis compañeros de piso, con quienes he pasado un año excelente y lleno de recuerdos.

Y, de forma muy especial, a mi padre, por su ejemplo, apoyo, dedicación y el orgullo constante con el que me ha acompañado siempre, así como por todo lo que me ha enseñado a lo largo de la vida.

A todos, gracias.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

1.1. Introducción y motivación

Este Trabajo Fin de Grado forma parte de un conjunto de trabajos que han surgido a raíz del proyecto TwizyLine. El proyecto TwizyLine fue iniciado por cuatro estudiantes de la Universidad de Valladolid en 2020 con el objetivo de presentar una forma de movilidad sostenible en la que se consiguiese mediante un funcionamiento autónomo, el aparcamiento y posterior carga de vehículos eléctricos, en concreto para los Juegos Olímpicos de París 2024, dentro del contexto del concurso internacional TwizyContest 2020 patrocinado por Renault Group [1]. Desde este proyecto inicial se ha continuado el trabajo a través de una serie de Trabajos de Fin de Grado (TFG) y Trabajos de Fin de Máster (TFM). Estos nuevos proyectos han realizado varias modificaciones respecto al proyecto inicial, mejorando en cada uno de ellos aspectos del primer prototipo que desde un principio habían sido introducidos como soluciones temporales, o bien mejorando aspectos que por obsolescencia era conveniente cambiar.

Figura 1. Logotipo del proyecto Twizyline [2]

El primer prototipo realizado partió de la donación de un Renault Twizy por parte de la Fundación Renault a la Universidad de Valladolid tras haber ganado la fase nacional del concurso TwizyContest 2020. Esta donación supuso una valiosa oportunidad para el desarrollo de nuevos proyectos en el ámbito académico. En este contexto, el equipo de estudiantes que se presentaron al concurso comenzó a trabajar en el vehículo, realizando una serie de modificaciones orientadas a convertir el Twizy en un vehículo autónomo. Estas intervenciones abarcaron aspectos de *hardware*, comunicaciones, seguridad y lógica en los distintos nodos del sistema, permitiendo dotar al Renault Twizy de funcionalidades de conducción autónoma, tanto a nivel longitudinal (aceleración y frenado) como lateral (control de la dirección).

Figura 2. Renault Twizy [3]

Desde este primer prototipo el vehículo autónomo conseguido ha estado limitado de varias formas. Aunque el conjunto de Trabajos Fin de Grado ha mejorado algunos aspectos de estas limitaciones como por ejemplo implementar el cambio automático de marchas o implementar un dispositivo controlador de aceleración más fiable que su primer prototipo, sin embargo sigue teniendo importantes limitaciones. Una de las más importantes es que por su *hardware* de procesamiento de percepción y control su funcionamiento está limitado a velocidades de hasta 3 km/h [4]. Esta limitación tenía dos causas, por un lado el propio *hardware* como acabamos de decir, que consistía en un miniordenador industrial que, aunque era seguro de cara a soportar ambientes eléctricos inestables como los de un vehículo, tenía capacidades de cómputo muy limitadas. Por otro lado, la propia limitación del *hardware* limitaba el tipo de *software* que se podía utilizar. El resultado era que el rendimiento de computación para procesar los datos de percepción era extremadamente pobre, principalmente por la carga introducida por la cámara [5].

Ante esta limitación, surge la necesidad del presente Trabajo de Fin de Grado: mejorar el rendimiento del sistema mediante una reestructuración tanto a nivel de *hardware* como de *software* del sistema de control y percepción al completo.

En el ámbito del *hardware*, se produce una mejora que ya se había considerado desde el primer prototipo, pero que por falta de medios no se abordó. Consiste en la incorporación de un nuevo módulo de procesamiento que permita manejar de forma más eficiente la carga computacional generada por los sensores, especialmente la cámara.

Dado que el procesamiento de los datos percibidos del entorno se realizará ahora en un nuevo módulo, es necesario replantear la arquitectura física y *software* del sistema.

Previamente, el módulo de control asumía la totalidad de las tareas críticas: la adquisición de datos del entorno a través de los sensores, la gestión de fallos y de los diferentes estados del vehículo, el procesado de dicha información para determinar las acciones a realizar a nivel lateral y longitudinal, y, finalmente, la comunicación de las órdenes resultantes a los actuadores encargados del control físico del vehículo, como el giro de las ruedas, el cambio de marchas y la aceleración.

Con la introducción de este nuevo módulo, denominado PC Fusión, se obtiene una mejora significativa en la capacidad de procesado, pues se conseguirá liberar al módulo de control de las tareas más demandantes en términos computacionales. Siendo imprescindible una reorganización de la arquitectura del sistema. Con esta nueva reorganización se conseguirá reducir la latencia en la toma de decisiones, un requisito fundamental para garantizar la fiabilidad y precisión en sistemas de conducción autónoma, y mejorando ese parámetro crítico del que se ha hablado al principio de esta introducción que consiste en una limitación de la velocidad a 3 km/h.

A nivel de *software*, los sistemas de control robóticos han evolucionado enormemente desde el año 2020, siendo imprescindible incorporar este tipo de *software* en el proyecto para mejorar en el futuro las prestaciones del vehículo. Por lo tanto, esta mejora se refiere a una mejora por obsolescencia de la solución original, que básicamente consistía en comunicación de los diferentes módulos *software* a través de subprocesos mediante la librería `multiprocessing` de Python. Concretamente, se ha optado por la implementación de una arquitectura *software* que usa como plataforma el ecosistema Robot Operating System 2 (ROS 2). Esta arquitectura se desplegó en el nuevo módulo de percepción que recibe el nombre de PC Fusión. Este es un marco de trabajo ampliamente utilizado en el desarrollo de sistemas robóticos y en aplicaciones de conducción autónoma. Esta elección responde tanto a su uso en proyectos similares como a las ventajas que ofrece en cuanto a modularidad, escalabilidad y la compatibilidad con el lenguaje de programación Python en que están programados los módulos. Su arquitectura distribuida permite organizar el *software* en nodos especializados, lo que facilita la adaptación del sistema a funcionalidades futuras, como podría ser la implementación de un sensor LiDAR. Además, aporta ventajas clave para este tipo de aplicaciones, como el soporte para comunicaciones en tiempo real, adaptabilidad e interoperabilidad y una mayor fiabilidad en la ejecución de procesos críticos [6]. Estas características lo convierten en una plataforma adecuada para reorganizar las tareas lógicas del sistema, logrando un funcionamiento más estable, robusto y escalable en escenarios variables y exigentes propios de la conducción autónoma.

1.2. Objetivos

El objetivo principal de este proyecto es conseguir una mejora en el rendimiento actual del vehículo autónomo desarrollado sobre el Renault Twizy, en términos de capacidad de percepción que debería influir en la velocidad máxima del vehículo.

Para conseguir este objetivo principal será necesario cumplir los siguientes objetivos secundarios:

- Diseño de una nueva arquitectura *hardware* que incluya el PC fusión como módulo de percepción más potente que el actual módulo de control.
- Diseño de la arquitectura *software* para el reparto de tareas entre el módulo de control y el PC Fusión.
- Introducción del ROS 2 como sistema general de control en el PC fusión, lo que implica crear los nodos necesarios que sustituyen a las aplicaciones anteriores.

Es importante notar que los dos primeros objetivos están relacionados entre sí, dado que la estructura *hardware* (no solo la comunicación entre el módulo de control y el PC fusión, sino con el resto de los elementos de percepción) y la estructura *software* están intrínsecamente relacionadas.

1.3. Fases y métodos

El trabajo realizado en este proyecto ha seguido varias fases que han permitido desarrollar el trabajo de manera eficiente:

- Estudio del estado del proyecto: En esta primera fase se estudió el material más importante relacionado con el proyecto que han generado estudiantes anteriores. Además, se estudió el nuevo *software* que se quería instalar para el control de vehículo. Concretamente se realizaron los siguientes trabajos:
    - Análisis y estudio de Trabajos de Fin de Grado (TFG) y Trabajos de Fin de Máster (TFM) previos para comprender el estado actual del proyecto y el funcionamiento de sus diferentes módulos.
    - Estudio teórico y práctico de Robot Operating System 2 (ROS 2) con el objetivo de su aplicación al proyecto.
    - Lectura y estudio teórico del funcionamiento de los coches autónomos, tratando la obtención, el procesado y el manejo de los datos del entorno.

- Fase de preparación del trabajo: En esta fase se evaluaron cuestiones prácticas del proyecto. En concreto se accedió a los ordenadores instalados en el vehículo para conocer su estado actual y evaluar qué debía ser cambiado. Además, se realizó una evaluación de la versión *software* de ROS con la que merecía la pena trabajar. Los trabajos realizados en esta fase fueron:

- Obtención de los códigos Python de los módulos *software* del coche actuales, con su posterior lectura y estudio teórico.
- Evaluación de las diferentes versiones de *software* a utilizar con el objetivo de elegir versiones compatibles y soportadas.
- Diseño de la nueva arquitectura *software*, definiendo el mapa de tareas que cada elemento realizará.
- Análisis de los cambios a implementar en el *software* antiguo para ser transcrito al nuevo *software* y nueva estructura electrónica.

- Implementación de los cambios: A partir de este momento, se comenzaron a implementar los cambios que se plantearon en la fase anterior. Concretamente:

  - Instalación y configuración del *software* seleccionado en el nuevo módulo a implementar en el sistema del vehículo (PC Fusión).
  - Búsqueda e instalación de los drivers para el control de sensores en el PC Fusión.
  - Reestructuración del código Python del módulo de control para su división en distintos módulos ROS en el PC Fusión.
  - Readaptación de la forma de comunicación entre los diferentes módulos, mediante el uso de tópicos ROS, usando una estructura de publicador-subscriptor.
  - Desarrollo de un código Python no ROS en el módulo de control, con la función de actuadores en los módulos físicos del vehículo.
  - Implementación de una comunicación MQTT entre el PC Fusión y el módulo de control.
  - Creación de scripts para simular la recepción de mensajes por parte del PC Fusión a través de CAN, provenientes del módulo de comunicaciones y de la OBD del vehículo.
  - Prueba y depuración de los distintos módulos ROS por separado, así como del código Python del módulo de control y del sistema en su conjunto.
  - Análisis de la carga computacional generada por los diferentes módulos y redistribución de funciones entre ellos para optimizar el rendimiento del sistema.

- Implementación en el Renault Twizy: Una vez se obtuvieron resultados prometedores en cuanto a rendimiento mediante simulación, se conectaron los módulos con las conexiones reales en el propio coche. Para ello ha sido necesario:

  - Investigación de los pasos necesarios para la puesta en marcha del sistema previo a las modificaciones actuales sobre el propio vehículo, y posterior prueba de su funcionamiento.
  - Implementación física del nuevo módulo y la reestructuración del cableado entre módulos, sensores y actuadores.
  - Unificación de las comunicaciones MQTT.
  - Estudio y mejora de la conexión CAN entre la ECU y el hub CAN.

- Modificaciones a nivel *software* sobre diferentes nodos ROS para solventar problemas que no se dieron en simulación.
- Implementación total sobre SSH para un manejo totalmente inalámbrico.
- Transmisión de la cámara tras el procesado con latencia mínima.
- Pruebas del nuevo rendimiento adquirido

En este Trabajo Fin de Grado, como los anteriores, se han encontrado múltiples errores propios de trabajar con un prototipo, como por ejemplo, errores de conexión en el cableado, fallo de inicialización de módulos, etc. Como aspecto nuevo que se incluye en este trabajo y que no aparecía en los anteriores, se ha desarrollado una sección de *troubleshooting*, esperando que los compañeros que continúen este proyecto no tengan que resolver los mismos problemas con los que me he encontrado a lo largo de su desarrollo.

1.4. Recursos

Se han empleado los siguientes materiales a nivel *hardware* para el desarrollo del proyecto:

- DSD TECH Adaptador USB a Bus CAN basado en Open Hardware Canable: Necesario para la conexión CAN con el nuevo módulo PC Fusión. Se muestra en la Figura 3.

Figura 3. DSD TECH Adaptador USB a Bus CAN [7]

- Hub USB de 6 Puertos: Proporciona conectividad USB adicional.
- Humming Board CBi (x2): Se utilizarán dos Humming Board CBi (módulos de comunicaciones y de control) como Unidades de Control Electrónico (ECU) en el vehículo. Estos dispositivos ejecutarán el SO Linux Debian 10.0. Se muestran en la Figura 4.

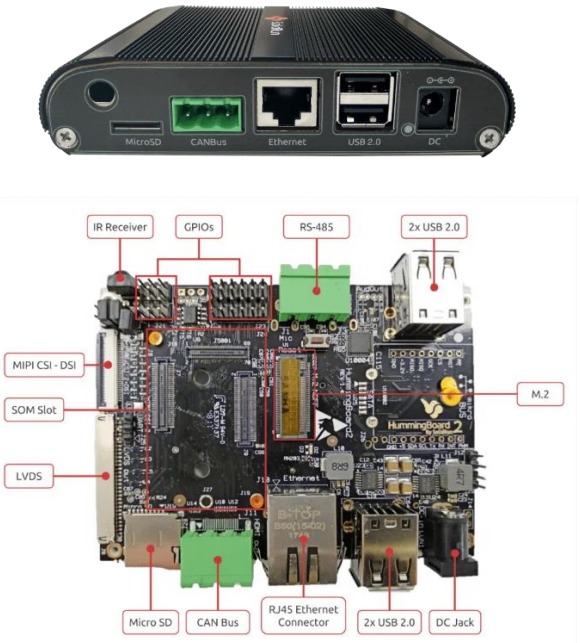

Figura 4. Humming Board CBi [8]

- Maxon EPOS4: Controladora del motor lateral.
- Motor Maxon: Incluye un motor de 100 W con su encoder MILE y una reductora con relación 1:81. Se encarga del control longitudinal del vehículo.
- Osciloscopio: Necesario para comprobar el estado de la conexión CAN.
- PowerBox: Caja que alberga 6 fusibles en su interior junto con sus respectivos interruptores, permitiendo la conexión de diversos equipos, siendo capaces de controlar alimentación de manera modular, como se detallará más adelante.
- Receptor GPS Garmin GPS18x USB: Para la obtención de datos de posición y navegación.
- Renault Twizy: Vehículo base para el proyecto, se muestra en la figura 3.
- Router: Gracias al *router* son posibles las diferentes comunicaciones en el vehículo que van sobre la red local, permitiendo de esta manera el control total a través del protocolo SSH.
- Sensor de RFID: Utilizado para la detección de etiquetas RFID ubicadas en el suelo.
- Sensores de Ultrasonidos de Proximidad (x3): Utilizados para la detección de objetos cercanos.
- SpyBox CAN: Es un hub de buses CAN, el cual dispone de 5 puertos DB9 hembra y 1 puerto DB9 macho en cada canal.[5]
- ZOTAC ZBOX E Series Magnus (PC Fusión): Es el ordenador donde se llevan a cabo las tareas de procesado de datos recibidos por los sensores, se le ha implementado ROS 2. Cuenta con las siguientes especificaciones:
  1 procesador Intel Core I7-10750H de 6 núcleos, 1 gráfica NVIDIA GeForce RTX 2080 SUPER, 1 disco duro 256GB SSD, 1 RAM DDR4 16GB, 6 puertos USB 3.0, 1 puerto USB tipo C, 2 puertos HDMI y 2 puertos Ethernet [3]. Se ha instalado

sobre él un sistema operativo Linux Ubuntu 22.04 LTS, se muestra en la Figura 5.

Figura 5. ZOTAC ZBOX E Series Magnus (PC Fusión) [9]

1.5. Estructura de la memoria

Tras esta introducción al proyecto, la memoria se estructura en cuatro capítulos adicionales, el primero constituye el estado del arte y del proyecto, y plantea las modificaciones a realizar, de manera introductoria a los siguientes capítulos que se centrarán en la implementación y las pruebas. Finalmente, el último capítulo recoge las conclusiones y líneas futuras.

- Capítulo 2: expone los fundamentos de los vehículos autónomos, incluyendo una descripción general de sus componentes principales y de las arquitecturas software habitualmente empleadas. Presenta el estado actual del proyecto, prestando especial atención a los distintos módulos *software* existentes y, en particular, al módulo de control. A partir de este análisis, se plantea el rediseño conceptual de la arquitectura global, justificándose las modificaciones a realizar sobre el sistema previo.
- Capítulo 3: constituye el núcleo del trabajo realizado. Se introducirá el marco de trabajo Robot Operating System 2 (ROS 2), describiéndose de forma detallada, nodo por nodo, las modificaciones realizadas a nivel *software* para su implementación en el sistema.
- Capítulo 4: se detallan las nuevas tareas que tendrá que encargarse ahora el módulo de control, especificando la implementación de la comunicación MQTT con el PC Fusión y las tareas específicas de cada "nodo actuador" fuera del marco ROS 2. Se muestran los resultados finales conseguidos tras la implementación completa en el vehículo así como una sección dedicada a la resolución de incidencias (*troubleshooting*).
- Capítulo 5: recoge las conclusiones del proyecto, junto a posibles líneas de mejora para desarrollos futuros.

2. Estado del arte y del proyecto

2.1. Introducción

El objetivo de este capítulo es doble. Por un lado, se realizará una breve descripción de los conceptos generales relacionados con los vehículos autónomos, haciendo especial hincapié en los módulos *software* que los componen y las plataformas o ecosistemas *software* en los que se suelen desarrollar.

Esta introducción al vehículo autónomo nos servirá de manera muy adecuada para introducir el estado actual del proyecto TwizyLine y por qué son necesarios los cambios que se proponen en el mismo.

2.2. Coches autónomos

2.2.1. Introducción a los coches autónomos

Los coches autónomos han emergido como una de las tecnologías más transformadoras en el ámbito de la movilidad y el transporte. Estos sistemas están diseñados para operar sin la intervención directa de un conductor humano, apoyándose en una combinación de sensores avanzados, algoritmos de aprendizaje automático y arquitecturas de *software* altamente especializadas. Entre las principales ventajas de los coches autónomos destacan la mejora en la seguridad vial, la optimización del tráfico y la inclusión de sectores de la población que anteriormente enfrentaban barreras para conducir debido a la edad o discapacidades físicas. Además, su implantación tiene el potencial de transformar el diseño urbano al reducir la necesidad de estacionamiento y fomentar un enfoque más sostenible en la gestión del espacio público [10] [11].

La autonomía de los vehículos está clasificada en seis niveles según el estándar J3016 de SAE (Society of Automotive Engineers), que va desde el nivel 0, donde el control es completamente manual, hasta el nivel 5, que representa la autonomía completa sin intervención humana. Esta clasificación permite identificar el grado de automatización de cada sistema y las responsabilidades del conductor en cada etapa de desarrollo. Actualmente, la mayoría de los vehículos en desarrollo se encuentran entre los niveles 2 y 4 donde el sistema puede asumir ciertas tareas de conducción bajo supervisión humana [11]. En la Figura 6 se pueden observar los diferentes niveles con un poco más de detalle.

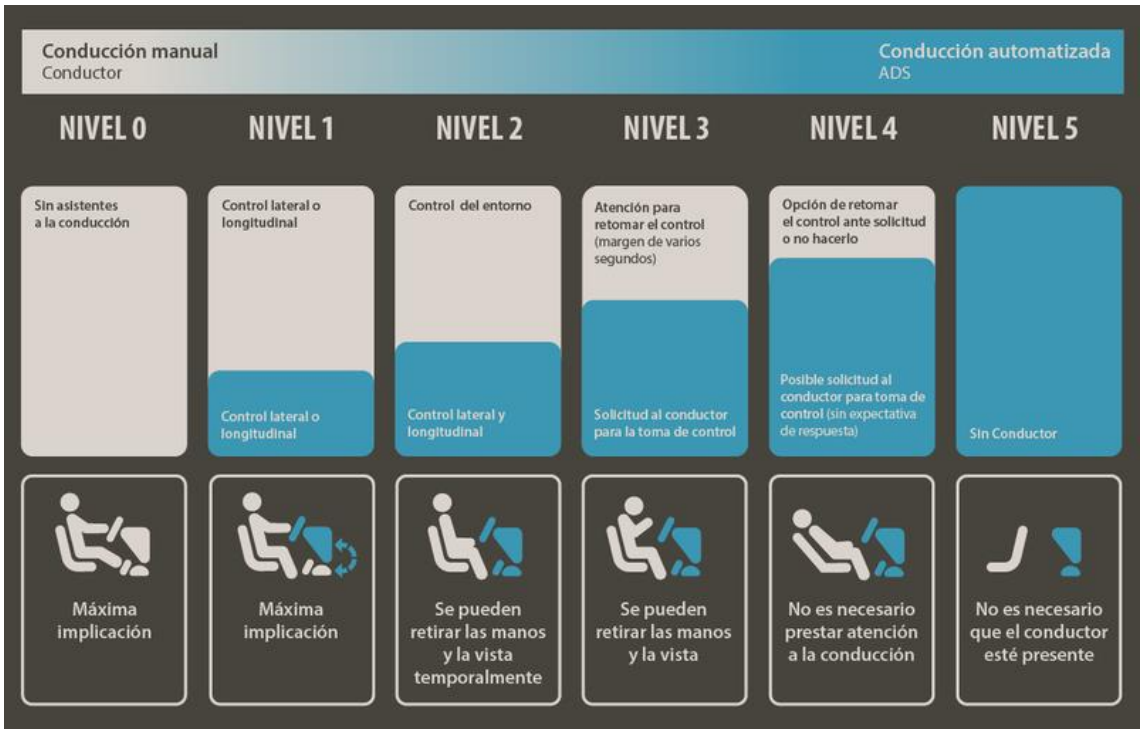

Figura 6. Niveles de conducción autónoma según el estándar SAE J3016 [12]

2.2.2. Componentes fundamentales de un coche autónomo

El funcionamiento de los coches autónomos se puede descomponer en tres módulos principales: percepción, planificación y control. Cada uno de estos desempeña un papel crucial en garantizar la autonomía y la seguridad del sistema [11].

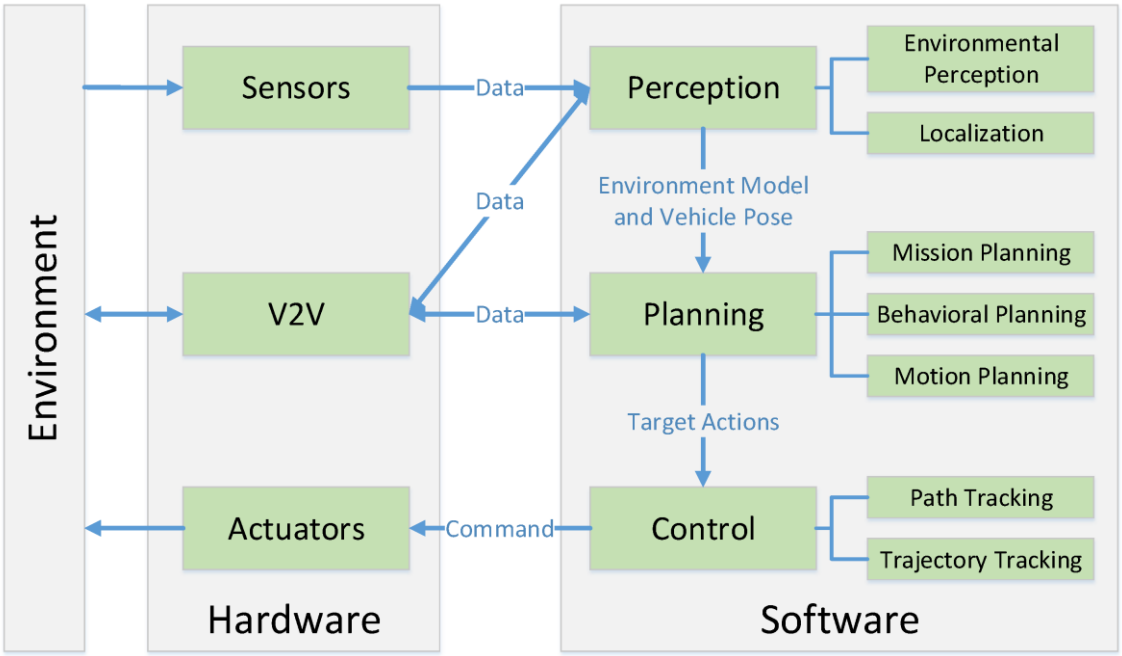

Figura 7. Estructura básica de un vehículo autónomo [13]

- Percepción: Este módulo es responsable de captar información del entorno mediante sensores como láser LIDAR, cámaras y radares. Los datos obtenidos se fusionan para generar una representación coherente y en tiempo real del entorno,

incluyendo la detección de peatones, otros vehículos y señales de tráfico. La percepción también incluye la localización precisa del vehículo en mapas de alta definición (HD maps), utilizando algoritmos como SLAM (localización y mapeo simultáneos) para garantizar una navegación segura incluso en entornos complejos.

- Planificación: Utilizando la información procesada por el módulo de percepción, el sistema de planificación toma decisiones sobre la ruta a seguir y las maniobras necesarias para evitar obstáculos. Este módulo también considera aspectos de seguridad y comodidad para los pasajeros. Se subdivide en tres niveles: planificación de misiones (ruta global), planificación de comportamiento (decisiones a nivel de maniobra) y planificación de movimiento (trayectorias específicas a corto plazo) [13].
- Control: El módulo de control traduce las decisiones del módulo de planificación en acciones físicas concretas. Esto incluye el control de la aceleración, el frenado y la dirección, asegurando que el vehículo se mantenga en su trayectoria planificada. Los algoritmos de control predictivo y los controladores PID son herramientas clave para lograr una conducción suave y eficiente [10] [11].

2.2.3. Arquitecturas de Software en coches autónomos

La arquitectura de *software* es una parte crucial en los coches autónomos. Una tendencia destacada es la consolidación de plataformas unificadas de *software* que permiten la gestión integrada de todos los módulos y datos. Estas arquitecturas permiten la escalabilidad y la actualización de los sistemas de manera eficiente, utilizando estándares como AUTOSAR para facilitar la interoperabilidad entre componentes de diferentes fabricantes [10].

En este contexto, las arquitecturas suelen apoyarse en sistemas operativos en tiempo real y diseños distribuidos, donde los datos capturados por los sensores son procesados por unidades dedicadas para tareas específicas, como la detección de peatones o la clasificación de objetos. La gestión de funciones críticas como el frenado o la dirección requiere cumplir con estrictos estándares de seguridad funcional, como el ISO 26262, que garantizan la fiabilidad del sistema incluso en situaciones de fallo parcial [10].

Una de las mayores complejidades en estos sistemas es la gestión eficiente del volumen masivo de datos generados en tiempo real. En este aspecto, cobra una gran importancia la necesidad de infraestructuras de comunicación internas robustas y de baja latencia. En los primeros desarrollos se empleaban soluciones como ROS 1 (Robot Operating System), ampliamente utilizada en robótica por su flexibilidad, pero con limitaciones claras en cuanto a rendimiento en tiempo real y capacidad de operación en entornos distribuidos [10].

Estas limitaciones motivaron la creación de soluciones específicas como Project Cocktail, una infraestructura de transmisión de datos orientada a vehículos autónomos. Project Cocktail actúa como intermediario entre módulos, gestionando los datos como

flujos binarios independientes mediante UDP, lo que lo hace más adecuado que ROS 1, superándolo en términos de rendimiento y retardo [14].

Sin embargo, Project Cocktail se limita a funciones de middleware y no contempla aspectos clave como la interoperabilidad, la escalabilidad o la certificación funcional [14]. Es en este punto donde se evidencia el valor añadido de ROS 2 como alternativa, pues este resuelve los problemas que motivaron la creación de Cocktail, pero ofreciendo una solución integral y estandarizada, con soporte para comunicaciones en tiempo real mediante DDS, ejecución distribuida, modularidad y compatibilidad con estándares como ISO 26262 [15]. Gracias a su arquitectura más robusta y su amplia adopción en la industria, ROS 2 representa una plataforma más adecuada para el desarrollo del presente sistema de coche autónomo.

2.3. Arquitectura *software* del proyecto TwizyLine al inicio de este trabajo

2.3.1. Introducción

En el desarrollo del presente Trabajo de Fin de Grado (TFG), se ha llevado a cabo un análisis completo de los módulos de *software* actuales que componen el sistema en estudio. Este análisis ha tenido como objetivo principal entender el papel de cada módulo dentro del sistema, detallando las funciones específicas que desempeñan, los datos que intercambian con otros módulos y las formas en las que se comunican entre sí. Este proceso ha permitido poder identificar las áreas del sistema que pueden ser optimizadas, especialmente en lo que respecta al rendimiento y la eficiencia computacional.

Uno de los principales hallazgos de este análisis es que el módulo de control presenta una serie de limitaciones en términos de rendimiento, lo que puede afectar negativamente al desempeño global del sistema. Dado que el módulo de control es responsable de una gran parte de realizar las operaciones de percepción y control que conllevan una gran carga computacional, se ha decidido centrar el esfuerzo de optimización en este módulo en particular. La idea es aligerar su carga de trabajo, lo que permitirá mejorar el rendimiento del sistema en su conjunto. Para lograr esto, se ha planteado la opción de delegar gran parte de las funciones que actualmente realiza este módulo a otro componente, en este caso, al PC Fusión, el cual tiene la capacidad de asumir estas tareas sin afectar negativamente la estabilidad y el rendimiento del sistema, de hecho, mejorándolo.

El enfoque para abordar esta optimización implica un estudio detallado de las funcionalidades del módulo de control, para así determinar qué tareas pueden seguir siendo gestionadas por él y cuáles son más convenientes trasladar al PC Fusión. Este proceso no solo busca mejorar el rendimiento, sino también garantizar que la transición de funciones entre los módulos sea lo más fluida posible, sin introducir nuevos cuellos de botella ni afectar la integridad del sistema.

Resulta interesante señalar que este cambio también permite aproximar el diseño de la arquitectura *software* del proyecto TwizyLine a un diseño más estándar de vehículo autónomo, como el que se ha visto en la sección anterior, y sobre el que se incidirá más adelante.

2.3.2. Esquema general previo

En cuanto a la arquitectura *hardware* con capacidad de computación, el vehículo contaba con dos microordenadores HummingBoard que actúan como los módulos principales del esquema: el módulo de control y el módulo de comunicaciones. En las siguientes secciones se describen sus funciones y conexiones.

2.3.2.1. Módulo de Control

El módulo de control era responsable de recopilar información del entorno, procesar dicha información, recibir mensajes por CAN desde el módulo de comunicaciones, procesar el control de errores y el manejo de estados del vehículo, y en base a toda esta información enviar las decisiones de control lateral y longitudinal a los actuadores. De forma directa, desde los puertos USB del módulo de control, estaban conectados los siguientes controladores:

- Controlador del acelerador: Este se encarga del control longitudinal del vehículo, gestionado mediante un Arduino, en el cual se activan o desactivan unos relés permitiendo el control manual o autónomo del acelerador.
- Controladora EPOS4: Maneja un motor Maxon permitiendo el giro del volante, consiguiendo de esta manera un control preciso de la dirección del vehículo.

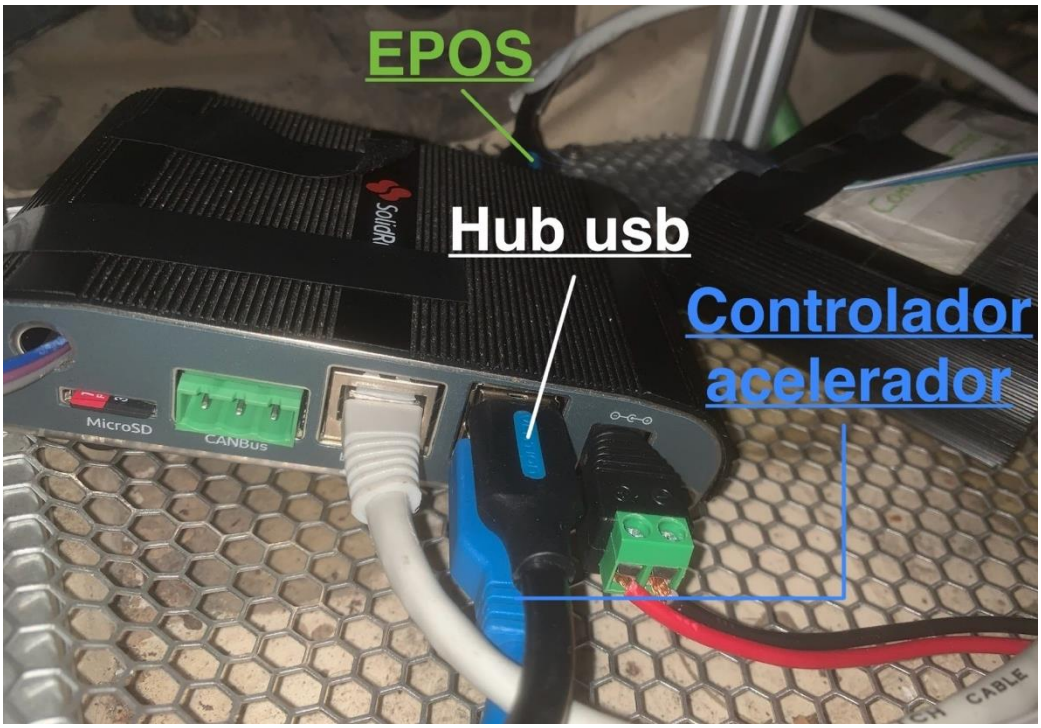

Figura 8. Conexiones previas al módulo de control

Se muestra en la Figura 8 las conexiones directas a este módulo. Dado que este no cuenta con suficientes puertos, se conectó un hub USB para ampliar dicha capacidad. A este se conectarán una serie de dispositivos, como se puede observar en la Figura 9. Se concretan estos a continuación:

- Cámara y sensores: Proporcionan información visual y ambiental. Entre los sensores conectados se encuentran:
    - Un sensor de ultrasonidos, utilizado para detectar objetos cercanos.
    - Un sensor RFID, se usa para detectar etiquetas RFID (actualmente no se utiliza en la implementación).

- Controlador de marchas: Este dispositivo, gestionado mediante un Arduino, regula el cambio de marchas del vehículo, accionando la marcha D cuando el sistema entra en modo autónomo.

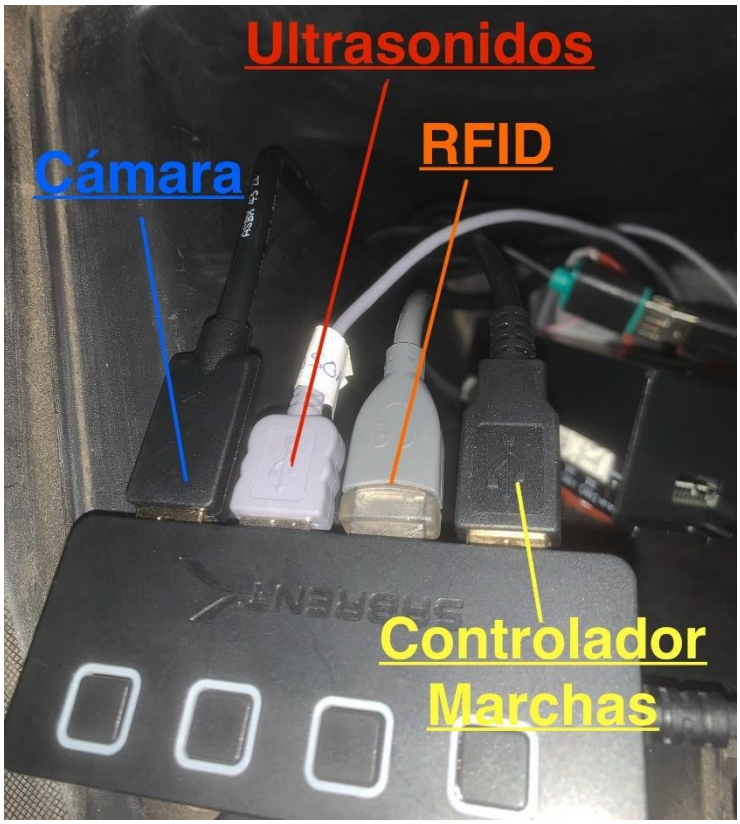

Figura 9. Conexiones previas al hub USB

2.3.2.2. Módulo de Comunicaciones

El módulo de comunicaciones tiene como función principal gestionar las señales externas e indicar el estado operativo del vehículo. Sus conexiones principales incluyen:

- Receptor GPS: Conectado por USB, proporciona datos de posicionamiento para la navegación del vehículo.

- Indicadores LED: A través de sus pines GPIO, el módulo controla los LEDs que informan sobre el modo operativo del vehículo, como conducción autónoma, normal o standby.

Este módulo tiene funciones como recibir mensajes MQTT para modificar el estado del vehículo o la velocidad objetivo entre otros, comunicar estos mensajes recibidos al módulo de control y el mantenimiento de la conexión con este mediante mensajes periódicos.

Este módulo tiene las mismas funciones que las modernas TCU (Telecommunication Control Unit) de los vehículos conectados. Sirve como punto conexión del vehículo con el mundo exterior, y facilita las labores de aseguración de dicha comunicación. El módulo tiene como misión principal la comunicación con internet para que el cliente pueda de esta manera solicitar el servicio mediante su aplicación desde un móvil. Sin embargo, dado que en este momento no se está usando el módem 4G que se comunicaba con un servidor ubicado en un laboratorio de la universidad, sino que la operación se realiza a través de un *router* Wi-Fi incorporado al vehículo, no habrá salida hacia internet [5].

2.3.2.3. Interconexión de módulos y comunicación externa

Ambos módulos, el de control y el de comunicaciones, estaban conectados entre sí mediante un bus CAN, que permite una comunicación eficiente y confiable entre ellos. Es mediante esta comunicación como se transmiten los mensajes de control de forma periódica, para que el sistema detecte en todo momento si alguno de los módulos ha dejado de funcionar.

En la Figura 10 se muestra el esquema físico de conexiones tal y como se encontraba al inicio de este Trabajo Fin de Grado.

Además, en dicha figura se puede observar que ambos módulos también están interconectados a un *router* a través de cables Ethernet. Este *router* desempeña un papel fundamental al permitir acceso remoto a los módulos mediante SSH, facilitando la monitorización, depuración y actualización desde ubicaciones externas al vehículo. Es importante señalar que esta conexión Ethernet no se utiliza para la comunicación entre módulos. Esto se debe a que el vehículo Twizy no tiene red Ethernet y, además, era necesario acceder a información del vehículo que se transmite por su bus CAN, por lo que desde el proyecto inicial se pensó que era mejor opción mantener todas las comunicaciones dentro del bus CAN.

La arquitectura general del sistema, esto es, no solo el vehículo, sino la infraestructura para dar el servicio de aparcamiento automático, contaba con un servidor MQTT externo, que se utiliza para la transmisión de mensajes hacia y desde el módulo de comunicaciones. Este servidor recibe mensajes de un smartphone a través del protocolo MQTT, los cuales son posteriormente procesados y enviados al módulo de comunicaciones, para ejecutar las acciones necesarias. Este esquema de comunicaciones se muestra en la Figura 11.

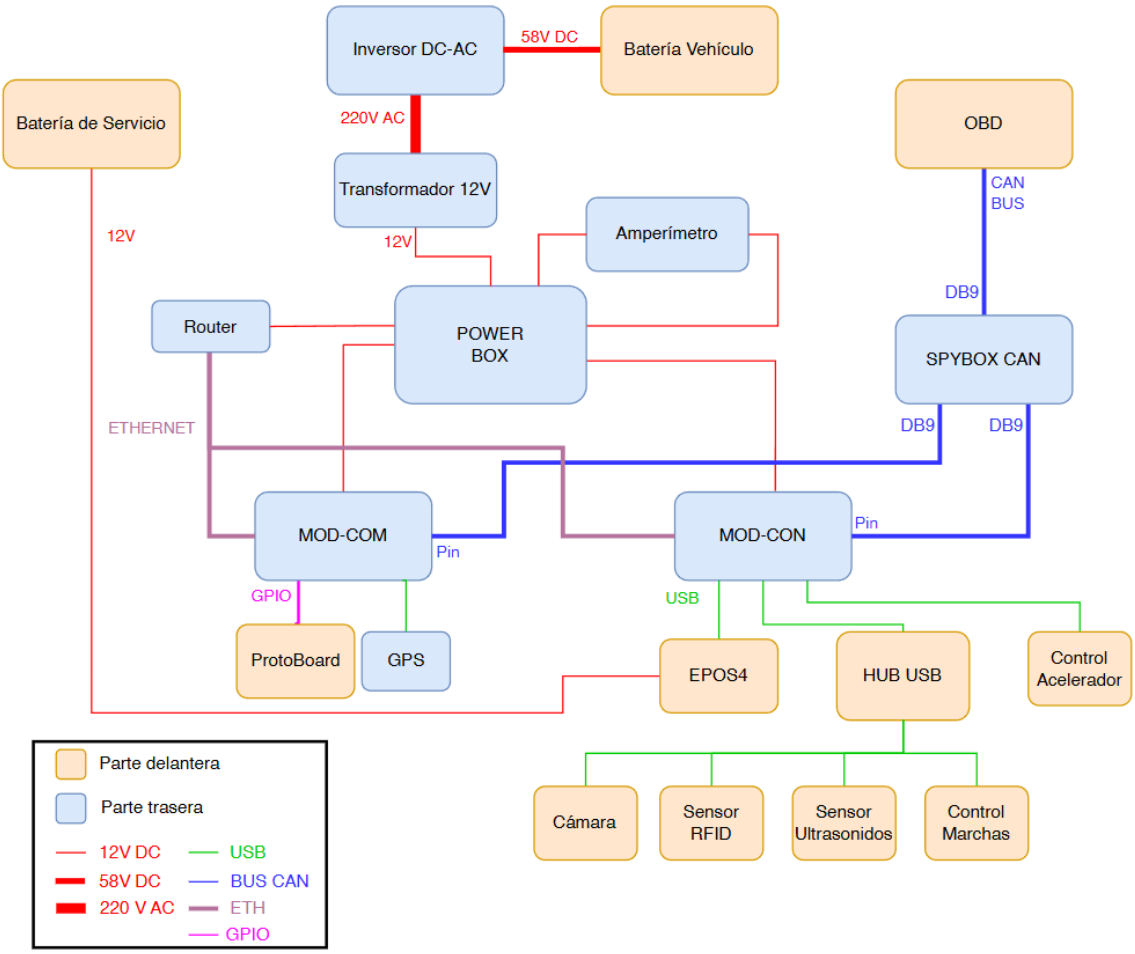

Figura 10. Esquema físico previo del vehículo [5]

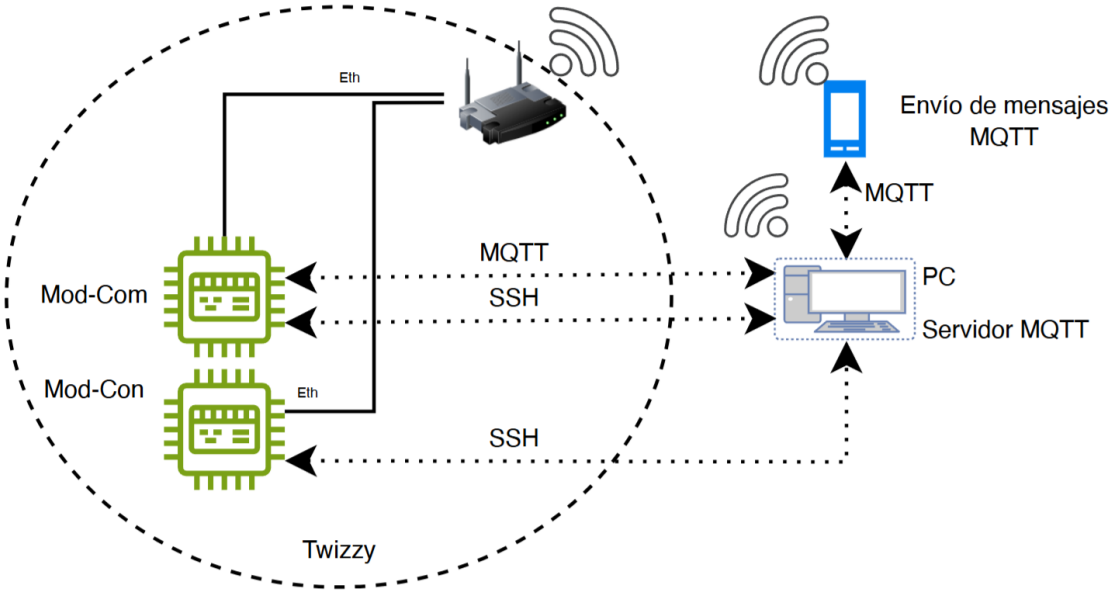

Figura 11. Esquema de comunicaciones previo del vehículo [5]

2.3.3. Módulo de control. Estado previo

Con el objetivo de mantener la línea temporal del desarrollo del proyecto, antes de detallar las funciones específicas previas del módulo de control, fue necesario acceder a su sistema de ficheros para poder obtener el código Python que contenía. En primer lugar, se intentó acceder a través de los puertos USB y directamente a la tarjeta microSD del dispositivo. Sin embargo, en ambos casos surgieron inconvenientes que impidieron la extracción de los datos. Ante estas dificultades, se optó finalmente por intentar el acceso mediante SSH.

Para ello, se intentó establecer una conexión SSH mediante un cable Ethernet conectado directamente entre el ordenador personal y el módulo. Sin embargo, surgieron complicaciones, ya que el dispositivo no era detectable en la red. Herramientas como Nmap y Wireshark no lograban identificar su presencia, lo que llevó a la conclusión de que el módulo no tenía asignada una dirección IP, impidiendo así la comunicación a través de este protocolo.

Finalmente, la solución consistió en conectar el módulo al *router* mediante un cable Ethernet y conectar el ordenador personal a esta misma red. De esta forma, el *router* asignó automáticamente una dirección IP al módulo a través de DHCP. Al encontrarse ambos dispositivos en la misma subred, fue posible identificar la dirección IP asignada al módulo mediante el comando `nmap -sn <dirección_subred/mascara>`. Una vez obtenida la IP, se estableció la conexión con el módulo mediante SSH y, posteriormente, se procedió a la transferencia de los datos al ordenador personal mediante el comando `scp`.Dentro del propio módulo de control, la puesta en marcha del programa final requería la ejecución previa de un script de inicialización, denominado `initial_script.sh`, encargado de habilitar la interfaz de comunicación CAN. A priori, tiene sentido integrar este script dentro del programa principal, como sí ocurre por ejemplo en el módulo de comunicaciones Sin embargo, no se ha llevado a cabo este proceso en la implementación actual debido a que la ejecución del script de inicialización requiere permisos de superusuario, cosa que no sucede para la ejecución del programa principal. Por lo tanto al tratar de iniciar el programa principal como superusuario provocaba modificaciones en los permisos de ciertos ficheros, corrompiendo la ejecución del programa.

Una vez se había ejecutado este script inicial, el módulo de control ya estaría preparado para realizar sus tareas asignadas. Su flujo de funcionamiento era el siguiente:

- Se ejecutaba el programa principal en Python, `Mod_con_final.py`, el cual constituía el componente central responsable de la conducción del vehículo. Este programa se estructura en torno a un proceso principal (`main`), encargado de gestionar los diferentes estados operativos del vehículo, que se clasifican en cinco modos: Startup, Normal, Autónomo, Standby y Fallo.
- El sistema arranca en el estado Startup, desde el cual espera la recepción de un mensaje de conexión por parte del módulo de comunicaciones.

- Una vez recibido, se transita al estado Normal, donde la conducción es completamente manual por parte del conductor, sin asistencias a la conducción a mayores de las presentes por defecto en el vehículo.
- A la recepción de un mensaje MQTT AM-ON, se activa el estado autónomo, el cual recibirá el enfoque principal de este proyecto. En este estado, se lanza un proceso principal denominado `main_driver`, el cual, a su vez, genera varios procesos secundarios o hijos encargados de gestionar las distintas funcionalidades del control longitudinal y lateral del vehículo. En los apartados 2.3.3.1. y 2.3.3.2. se detallan estos procesos y su funcionamiento.

Previo a entender cómo era el funcionamiento de estos procesos y de qué acciones se encargaban, es necesario mencionar la tecnología que usaban, junto a las limitaciones que esta presentaba.

Para la creación de los diferentes hilos, se utilizaba la librería Multiprocessing, que permite ejecutar varios procesos en paralelo aprovechando múltiples núcleos del procesador. Esta crea procesos independientes con su propia memoria y espacio de ejecución, lo que evita las limitaciones del Global Interpreter Lock (GIL). La comunicación entre estos hilos se realizaba mediante colas implementadas con la misma librería, lo cual permitía un intercambio sencillo de datos entre procesos concurrentes.

Este enfoque resultó útil en fases iniciales de desarrollo, pero muestra ciertas limitaciones:

- Falta de modularidad y aislamiento de fallos: Al estar todo el sistema concentrado en un solo *script* con múltiples procesos, la estructura era poco mantenible. La falta de separación clara entre módulos complicaba la depuración en caso de ocurrir algún fallo en el sistema, pues un error en cualquier hilo afectaría al sistema completo.
- Falta de escalabilidad y distribución: Debido a la falta de modularidad, dificulta las posibles actualizaciones y extensiones del sistema. De hecho imposibilitaría realizar de forma directa con esta tecnología el distribuir el trabajo en diferentes módulos *hardware*.
- Herramientas limitadas de supervisión y configuración: No se disponía de un sistema estándar para observar el comportamiento en tiempo real.

2.3.3.1. Control Longitudinal

El control longitudinal se encarga de regular la velocidad y el movimiento hacia delante del vehículo. Dentro de este ámbito, se crean los siguientes procesos hijos:

- `Sensor_controller`: Este proceso se encarga de gestionar los datos recibidos desde los sensores de ultrasonido del vehículo. La información recopilada por estos sensores es transmitida al proceso principal `main_driver` mediante la variable `UltrasonicData`. Esta información resulta esencial para detectar

obstáculos y ajustar el comportamiento del vehículo en función de las condiciones del entorno.

- `Throttle_controller`: Este proceso permite que el proceso principal `main_driver` se comunique con el microcontrolador Arduino responsable del control del acelerador del vehículo. Para ello se usa un PID (Proporcional-Integral-Derivativo), el cual se ajusta en base a las variables `currentSpeed` y `TargetSpeed`. Estas variables definen respectivamente la velocidad actual del vehículo, y la velocidad objetivo en función de las decisiones tomadas por el sistema.
- `Set_gear`: Al entrar en modo autónomo, se inicia este proceso con el argumento 'D', lo que indica al Arduino encargado del control de marchas que el vehículo está listo para comenzar a conducir. Este paso asegura que el Arduino accione la marcha D en el vehículo, y por lo tanto permita su movimiento hacia delante.
- `CAN_receiver`: Este proceso se encarga de recibir mensajes por parte de la ECU del vehículo y del módulo de comunicaciones. Estos mensajes pueden ser la velocidad actual del vehículo (`currentSpeed`), la marcha actual o mensajes de control `COM_STATUS`.

2.3.3.2. Control Lateral

El control lateral es responsable de mantener el vehículo en la trayectoria adecuada, determinada por una línea situada en el suelo, mediante la gestión de la dirección a través de la controladora Epos 4 sobre el motor Maxon. Para ello, se utilizan los datos proporcionados por la cámara integrada en el vehículo y procesados mediante la librería `cv2`.

Esta tarea no se realiza mediante la ejecución de procesos hijos en paralelo, sino que se ejecuta al completo y directamente en el proceso `main_driver` de la siguiente manera: en cada iteración del proceso `main_driver` la cámara captura imágenes del entorno, las cuales se transmiten recurrentemente a este proceso en forma de un array de datos denominado `frame`. Este array representa la imagen capturada en cada instante.

En el mismo proceso, se utiliza la clase `artemis_autonomous_car`, ubicada en el fichero `artemis_autonomous_car16.9.py`, la cual contiene funciones que aplican transformaciones a las imágenes capturadas para obtener información útil. Estas transformaciones permiten calcular dos parámetros clave para el control lateral:

- `Offset`: Desplazamiento lateral respecto al centro de la trayectoria deseada.
- Ángulo de la ruta: Orientación de la trayectoria deseada respecto al vehículo.

A partir del `offset`, el ángulo de la ruta, la velocidad actual del vehículo y un parámetro de ganancia k=1.2, se calcula el ángulo objetivo del volante (`delta`) mediante la función `calculo_stanley`. Este ángulo objetivo determina los pasos necesarios para ajustar la posición del volante, el cálculo para obtener estos pasos es realizado por la función `delta2steps`.

Adicionalmente, un controlador PID supervisa la posición actual del volante. Basándose en el error de ángulo (`delta_error`), se calcula y envía a la unidad de control del volante (Epos4) la variable `rpm_target`, que define la velocidad a la que debe girar el volante para corregir su posición.

Todos estos procedimientos llevados a cabo para el control lateral se realizaban de forma sucesiva en cada una de las iteraciones del proceso principal en el modo autónomo: `main_driver`.

2.3.3.3. Integración de los Controles Longitudinal y Lateral

El módulo de control combinaba al completo los sistemas de control longitudinal y lateral para gestionar la conducción autónoma. Mientras el control longitudinal regula la velocidad y la aceleración en función del entorno y las decisiones del sistema, el control lateral se asegura de que el vehículo mantenga su trayectoria deseada y ajuste su dirección según las condiciones del camino, determinado por una línea situada en el suelo.

Sin embargo, el módulo presenta ciertas limitaciones que pueden afectar su rendimiento debido principalmente a su capacidad de procesamiento. Para abordar estas limitaciones y mejorar el rendimiento general del sistema, se ha llevado a cabo la incorporación de un módulo adicional denominado PC Fusión. Este módulo integrará y procesará datos provenientes de los diferentes sensores, liberando así carga del módulo de control y mejorando la capacidad del sistema para tomar decisiones más robustas y precisas.

Para analizar adecuadamente estas limitaciones, es necesario identificar los cuellos de botella del sistema, es decir, aquellos elementos que restringen su rendimiento global. En este contexto, el proceso `main_driver`, responsable de gestionar y procesar los datos provenientes de los sensores, debe tener una capacidad de procesamiento igual o superior a la velocidad con la que los sensores capturan la información del entorno.

Esta condición no se cumplía en esta implementación, ya que la cámara es capaz de capturar hasta 30 *frames* por segundo (*fps*), sin embargo, según lo observado en la versión previa del código del módulo de control, el proceso `main_driver` solo era capaz de procesar 6 *fps*. Este valor limita directamente la frecuencia de procesado del sistema a 0.1667 Hz. En consecuencia, la posición del vehículo, tanto en sentido lateral como longitudinal, solo se actualiza cada 0,1667 segundos.

Se tiene certeza de que este es el principal cuello de botella del sistema, ya que el resto de los datos obtenidos de los sensores presentan un tamaño significativamente menor. Esto permite que puedan ser transmitidos y procesados en intervalos de tiempo considerablemente más cortos, sin generar retrasos apreciables en el flujo de datos ni en el rendimiento general del sistema. Concretamente:

- A través del bus CAN se recibía la velocidad longitudinal del vehículo cada 0.01 segundos. Este valor representado como un tipo *double* en Python mediante `multiprocessing.Value` ocupa 8 bytes en memoria.

- El Arduino que recibe los datos de los sensores de ultrasonidos envía un valor cada 0.005 segundos al proceso hijo `Sensor_Controller`. Este recibe un array de tipo int en Python a través de `multiprocessing.Array`, el cual contiene tres enteros con signo de 4 bytes cada uno, sumando un total de 12 bytes.

La posición actual del volante es recibida por el proceso `main_driver` en cada iteración de este. Este valor es un entero que varía entre -1.857.945 y 1.857.945, lo cual implica un tamaño de aproximadamente entre 28 y 32 *bytes*. El tamaño de cada *frame* enviado por la cámara se puede calcular a partir de la resolución y el formato de color utilizados. La cámara captura imágenes con una resolución de 640×360 píxeles en formato de color BGR, es decir, con 3 canales y un tipo de dato por defecto de 1 *byte* por canal. Por tanto, cada píxel ocupa 3 *bytes*.

Multiplicando la cantidad de píxeles por el tamaño por píxel, obtenemos:

$$640\ x\ 360\ =\ 230400\ píxeles$$

$$230400\ píxeles\ x\ 3\ bytes/píxel\ =\ 691200\ bytes$$

Luego cada *frame* ocupa unos 675KB en memoria. Esta diferencia de varios órdenes de magnitud con respecto al resto de datos implica que el procesamiento y la transmisión de las imágenes representan el principal cuello de botella del sistema, limitando la frecuencia con la que pueden gestionarse los datos en tiempo real por parte del proceso `main_driver`. Esto se menciona de manera indirecta en anteriores trabajos [5], pues se toma como retardo de procesamiento el tiempo que tarda en procesar la Humming Board cada *frame* capturado por la cámara, sin hacer referencia al resto de elementos a procesar de los otros sensores, siendo este un valor de 0.2 segundos.

Con la integración del mencionado PC Fusión, se conseguirá procesar un mayor número de *frames* por segundo de la cámara, permitiendo de esta manera actualizar la posición del vehículo lateral y longitudinalmente con una mayor frecuencia, y logrando por tanto reducir el retardo de procesamiento de cada imagen.

2.3.3.4. Estado previo arquitectura software de coche autónomo

En la sección 2.2.3. se explicaba la arquitectura *software* ideal que debería adoptar un coche autónomo. Esta se basa en una separación clara entre los módulos encargados de la percepción del entorno (sensores), la planificación de la ruta (procesamiento de la información sensorial) y el control de los actuadores (volante, acelerador, marchas, etc.).

En el presente capítulo se ha expuesto la arquitectura previa del sistema, y sus limitaciones, sin embargo no se había abordado este problema, pues en el estado previo del sistema, no se contaba con esta separación de las tareas a nivel *software*:

El módulo de control realizaba las siguientes tareas dentro del mismo proceso `main_driver`:

- Obtenía los datos de percepción del entorno relacionados con el control lateral (imágenes de la cámara).

- Procesaba todos los datos del entorno obtenidos de los diferentes sensores, correspondiente a las tareas de planificación de la ruta.
- Enviaba al actuador del control lateral las diferentes órdenes de control que se deben ejecutar.

Por lo tanto, únicamente algunas tareas cumplían este principio de separación modular:

- La percepción del entorno de los sensores relacionados con el control longitudinal, pues se realizaba en procesos separados.
- Las tareas de control del acelerador y las marchas, pues también se ejecutaban en procesos separados.

Con el desarrollo del presente proyecto, se ha logrado una transición hacia una arquitectura *software* mucho más alineada con el modelo ideal de un vehículo autónomo. La introducción de ROS 2 ha sido clave en este avance, ya que ha permitido la reorganización modular de las tareas de percepción y planificación en distintos nodos ROS 2. Además, las tareas de control se han delegado a un dispositivo *hardware* diferente, en el cual se han implementado procesos separados para cada actuador a controlar. Rediseño de la arquitectura global.

2.4. Modificaciones propuestas en la arquitectura modular

En el marco de este trabajo, se propone una reorganización modular que introduce un nuevo componente denominado PC Fusión, el cual reconfigura las responsabilidades de los módulos existentes:

- PC Fusión: Este módulo centraliza el procesamiento de los datos provenientes de los sensores, como podrían ser LIDAR, las cámaras y los radares. En la implementación actual se ha diseñado para funcionar con cámara, ultrasonidos y RFID. Su función principal es obtener estos datos y realizar la fusión de ellos para proporcionar al sistema una representación única y coherente del entorno. Esto reduce la carga computacional en los otros módulos, pero permite realizar las tareas de percepción y predicción mejorando el rendimiento global del sistema.
- Módulo de Control: Con esta modificación, el módulo de control queda enfocado exclusivamente en las funciones de actuación, como el control del acelerador, el freno y las marchas, liberándolo de tareas relacionadas con la percepción y la planificación. Al liberarle de las tareas que requerían una capacidad de procesado superior a sus especificaciones, se consigue una reducción de la latencia en la toma de decisiones, factor crítico en la conducción autónoma.
- Módulo de Comunicaciones: Este módulo permanece inalterado, manteniendo su función de gestionar las interacciones entre el coche y otros sistemas externos, en nuestro caso un smartphone que hace de cliente MQTT. Con este enfoque se asegura la compatibilidad con redes vehiculares modernas, como V2X.

Mediante estas modificaciones se busca que la nueva arquitectura se acerque más a la arquitectura óptima de un vehículo autónomo planteada en la sección 2.2. Para ello se reparten las tareas de percepción del entorno, planificación de la ruta y control de los actuadores entre diferentes nodos *software* especializados.

Cabe destacar que esta reorganización libera al módulo de control de parte de su carga computacional, un aspecto importante dado que sus limitaciones de procesamiento restringían previamente la capacidad del sistema en tareas de percepción del entorno

2.5. Nueva arquitectura física

Con la implementación de este nuevo módulo, ha sido necesario una reestructuración a nivel físico en el cableado del coche, para conseguir que cada uno de los módulos puedan realizar las nuevas tareas a las que están asignados. Comenzaremos con el nuevo módulo PC Fusión.

2.5.1. PC Fusión

Este módulo al encargarse de las tareas relacionadas con la percepción del entorno será necesario que tenga una conexión con los diferentes sensores: cámara, ultrasonidos y RFID.

Dado que tanto la cámara como los sensores de ultrasonidos se sitúan en la parte delantera del coche, y el cableado de estos llega hasta la guantera delantera, lo ideal sería que la conexión a estos sea en la propia guantera. Sin embargo, puesto que el PC Fusión tiene un tamaño demasiado grande, es necesario situarlo en la parte trasera del coche donde está localizado el rack, en el mismo sitio que estaba localizado el módulo de control. Debido a esta distancia, a pesar de que el PC cuenta con suficientes puertos para conectar todos los sensores, se ha utilizado el hub USB, pues permite conectar en este todos los sensores dentro de la guantera, y conectar el otro extremo a uno de los puertos usb del PC. Por lo tanto, las modificaciones han sido cambiar el hub del módulo de control al PC Fusión, y de este hub desconectar el controlador de marchas.

En cuanto a la conectividad CAN, ahora será este PC el que se comunique por CAN con el módulo de comunicaciones, para enviar y recibir mensajes relacionados con el control de estado y fallos. Además, será necesaria esta conectividad CAN para que reciba desde la ECU la velocidad del vehículo y la marcha introducida en todo momento, ambos valores necesarios para las tareas de planificación. Puesto que el PC no contaba con un puerto CAN de serie, fue necesaria la obtención de un adaptador CAN a USB, el cual permite las velocidades de 500Kbps que se utilizaban con el módulo de control.

Durante la fase de implementación en el vehículo del programa final, se dieron varios problemas relacionados con estas comunicaciones por CAN. Se consiguió detectar utilizando un osciloscopio, que las tramas CAN entre el conector OBD y el PC estaban contaminadas con un ruido excesivo, por lo tanto, fue necesario rehacer la conexión utilizando conectores más fiables, concretamente Sub-DB9, para bus CAN entre la

conexión de la Spybox CAN hacia el hub CAN. Se muestra dicha conexión en la Figura 12.

Figura 12. Conexión CAN entre OBD y PC Fusión mediante conector Sub-DB9

El PC Fusión cuenta con puerto Ethernet y con dos antenas Wi-Fi, se decide conectar un cable ethernet hacia el *router* para una conexión más rápida, estable y con menos latencia. Se necesita esta conexión de red para permitir una comunicación entre el PC Fusión y el módulo de control, la cual se ha decidido realizar mediante el uso de MQTT (esta decisión se justifica en el apartado 252.6.

En cuanto a esta comunicación MQTT, anteriormente se utilizaba un dispositivo externo al coche que actuase como servidor MQTT, como se mostraba en la Figura 11, y este se encargaba de comunicar las órdenes enviadas por el usuario a través de un smartphone hacia el módulo de comunicaciones, pues en ambos dispositivos se alojaban clientes MQTT. Mediante la introducción del PC, ha sido necesario crear dos clientes MQTT más, uno en el PC, y otro en el módulo de control, para que se consiga una comunicación MQTT entre ambos.

Con el objetivo de agilizar el proceso de pruebas en el vehículo, se ha integrado por completo la comunicación MQTT en el propio vehículo. Para ello, se ha implementado un *broker* Mosquitto dentro del propio PC Fusión, siendo únicamente necesario este bróker. Sin embargo, esta implementación es únicamente útil para este periodo de pruebas, pues para el proyecto final en el que habría una gran cantidad de vehículos siendo controlados simultáneamente y un servidor externo controlando el servicio en general, no tendría sentido.

2.5.2. Módulo de control

Ahora este módulo procede a encargarse únicamente de las tareas relacionadas con el control, operando sobre los diferentes actuadores. Por lo tanto, no será necesario su conexión a los sensores, como antes sucedía, sino solamente a los actuadores. Para ello, mantendrá sus conexiones directas hacia el controlador del acelerador, y hacia la controladora de la dirección EPOS4. Previamente tenía una conexión con el controlador de marchas a través del hub USB, esta pasará a ser de forma directa.

Para sus nuevas tareas recibirá los datos necesarios como la velocidad, ángulo de giro o marcha objetivos a través de MQTT desde el PC Fusión, por lo tanto, seguirá siendo necesaria su conexión ethernet con el *router* interno. No ocurrirá lo mismo con la conexión CAN, pues en la implementación actual no será utilizada.

Esto es debido a que previamente, dicha conexión CAN era utilizada en gran medida para el control de estados y de errores entre este módulo y el módulo de comunicaciones. Sin embargo, estos mecanismos han pasado a ser entre el módulo de comunicaciones y el nuevo PC.

En la implementación actual, no se darán estos mecanismos de forma bilateral entre ambos, sino solo del PC Fusión hacia el módulo de control, mediante órdenes a través de MQTT de tipo START y TERMINATE a los diferentes procesos que controlan los actuadores. Sin embargo, un desarrollo completo de este control de estados y errores puede ser una línea futura de trabajo, en la cual sería necesario realizar modificaciones en el *software* de los 3 módulos, siendo interesante una estandarización de estos mecanismos sobre CAN.

2.6. Justificación de los cambios propuestos

La introducción del módulo PC Fusión responde a la necesidad de optimizar el procesamiento de datos y mejorar la arquitectura general del sistema. Anteriormente, el módulo de control se encargaba de la percepción, procesamiento y control, asumiendo un volumen significativo de tareas, lo que incrementaba la complejidad, retardos y el riesgo de errores. Al centralizar el procesamiento en el PC Fusión, se libera capacidad de cómputo de los módulos, se ceden las tareas con mayores requerimientos computacionales al nuevo módulo con mejores características y se mejora así la capacidad de respuesta del sistema en situaciones críticas.

La especialización del módulo de control permite un enfoque más preciso en las tareas de actuación, al conseguir liberarse de gran parte de la carga de procesado consigue reducir la latencia en la ejecución de maniobras, aumentando así la seguridad en la conducción. Esta estructura modular también facilita la actualización y mantenimiento del sistema, al aislar funciones específicas en componentes dedicados.

Ha sido necesaria la implementación de una comunicación entre ambos módulos, y como se mencionó en los otros apartados de este capítulo, se decidió optar por MQTT.

La justificación de emplear MQTT como protocolo de comunicación se justifica por diferentes motivos, entre los que destacan:

- Reutilización de infraestructura ya implementada: El proyecto ya contaba con una implementación funcional de comunicación basada en MQTT. Aprovechar esta infraestructura permite reducir significativamente el tiempo de desarrollo, depuración y validación, facilitando la integración de los módulos y asegurando una continuidad técnica dentro del sistema.
- Modelo de comunicación basado en tópicos, similar a ROS 2: MQTT, al seguir exactamente el mismo patrón de comunicación, facilita una integración natural con los nodos de ROS 2 presentes en el PC Fusión. De esta manera se consigue una transición fluida de los datos entre sistemas ROS (PC) y no ROS (módulo de control) sin necesidad de romper el paradigma de diseño original. Esto favorece la mantenibilidad, la coherencia del sistema y la comprensión para siguientes desarrolladores. Al igual que ROS 2 cuenta con diferentes herramientas para la depuración y diagnóstico de fallos.
- MQTT es un protocolo ligero, que introduce una baja sobrecarga, siendo una ventaja en términos de eficiencia y latencias, especialmente cuando los mensajes son pequeños y frecuentes, como ocurre en esta comunicación.
- Escalabilidad y desacoplamiento de componentes: MQTT permite que múltiples publicadores y subscriptores de datos se comuniquen sin necesidad de conocer detalles específicos de su implementación. Esto introduce un alto grado de desacoplamiento entre los módulos, lo cual es deseable desde el punto de vista de arquitectura de sistemas distribuidos como el presente. Además, facilita la ampliación futura del sistema, ya que se pueden añadir nuevos módulos sin afectar la lógica existente.

Esta escalabilidad, además de ser aportada por el protocolo de comunicación, también se da directamente mediante la reestructuración de ambos módulos. El PC Fusión ofrece una plataforma flexible capaz de integrar tecnologías emergentes, como algoritmos de inteligencia artificial más avanzados u otros tipos de sensores como LiDAR, sin requerir un rediseño completo de la arquitectura del sistema. Esto permite que el vehículo autónomo permita ser actualizado de forma eficiente con la llegada de otras tecnologías.

Esta capacidad de adaptación se verá incrementada con la implementación del entorno de trabajo ROS 2, el cual proporciona una arquitectura de *software* inherentemente distribuida y modular, en la que cada tarea o subsistema se encapsula en un nodo independiente. Esta estructura proporciona varias ventajas directas que favorecen la escalabilidad, entre las cuales destacan:

- Reusabilidad y extensibilidad: facilita la integración de futuros desarrollos, aspecto especialmente relevante en la industria de los sensores donde los cambios, mejoras y actualizaciones son continuas [6].

- Flexibilidad y variabilidad: permite que la arquitectura se ajuste a distintos tipos de vehículos y entornos operativos, lo que incrementa su aplicabilidad en numerosos escenarios [6].

En conclusión, la introducción del módulo PC Fusión, la especialización del módulo de control y la adopción de MQTT como protocolo de comunicación permiten una distribución más eficiente y segura de las tareas del sistema, reduciendo la latencia y mejorando la capacidad de respuesta en situaciones de alta exigencia operativa. Esta reestructuración no solo optimiza el rendimiento operativo del vehículo autónomo, sino que también mejora su mantenibilidad al separar responsabilidades entre módulos especializados, alineándose con el modelo ideal, previamente descrito en el apartado 2.2.2. 2.2.3. y representado en la Figura 7. Además, la integración del entorno ROS 2 en el PC Fusión (donde se aloja el *software* encargado de las tareas de percepción y planificación) refuerza la escalabilidad y flexibilidad del sistema, facilitando en el futuro la incorporación de nuevas tecnologías sin necesidad de rediseñar la arquitectura. Todo ello establece una base sólida y preparada para adaptarse a las necesidades cambiantes del mercado, mediante futuras ampliaciones que se adapten a las expectativas de los usuarios.

3. Implementación o actualización de los diferentes módulos a ROS 2

3.1. ROS 2

ROS 2 es un *framework* de *software* modular y distribuido. Surge como el sucesor de ROS 1. Mientras que ROS 1 fue muy beneficioso para el desarrollo de sistemas robóticos en general, no siempre cumplía con los estrictos requisitos de fiabilidad y rendimiento en tiempo real necesarios para aplicaciones críticas de seguridad como los vehículos autónomos [6].

ROS 2 se desarrolló para superar estas limitaciones, ofreciendo la fiabilidad y el rendimiento en tiempo real que se requieren para la conducción autónoma [6]. A pesar de sus mejoras, ROS 2 conserva muchas de las ventajas de ROS 1, como su arquitectura distribuida y sus tipos de mensajes estandarizados [6].

ROS 2 es considerado una solución prometedora para la conducción automatizada y autónoma por varios aspectos:

- Rendimiento en tiempo real: Los vehículos autónomos deben tomar decisiones basadas en la información de sus sensores en tiempo real. Esto implica cumplir con límites de tiempo muy estrictos, especialmente en el bucle de control del vehículo (alrededor de 10 ms con un *jitter* de unos pocos microsegundos). ROS 2 está diseñado para ofrecer este rendimiento en tiempo real, lo cual es crucial para garantizar un control estable y una reacción rápida ante situaciones de tráfico. Para lograr esto, ROS 2 permite el uso de parches de *kernel* de Linux en tiempo real (PREEMPT RT), requiere la asignación estática de memoria y utiliza un middleware en tiempo real basado en el estándar Data Distribution Service (DDS) para la comunicación dentro de la red ROS 2 [6] [15].

- Fiabilidad: La seguridad funcional es una exigencia primordial en los vehículos autónomos debido a su interacción directa con personas. Los sistemas de automatización de la conducción deben ser altamente fiables para evitar fallos que puedan poner en riesgo la vida de los pasajeros y otros usuarios de la vía. ROS 2 consigue mejorar la fiabilidad en comparación con ROS 1 [6]. Entre las características que permiten definirlo como fiable para estas aplicaciones, son por ejemplo:
    - Cumple con el estándar de seguridad requerido en automoción ISO 26262, permitiendo de esta manera ser utilizado en aplicaciones de vehículos autónomos [15].

    - ROS 2 utiliza DDS, lo cual es fundamental para su fiabilidad, ya que DDS es un estándar utilizado en infraestructuras donde la seguridad es crucial, como sistemas militares y espaciales [15].
    - Contiene sistemas integrados de seguridad como encriptación, control de acceso y autenticación pudiéndose configurar diferentes aspectos de la comunicación, utilizando para ello políticas sobre el tráfico entrante y saliente, pudiéndose configurar también calidad de servicio (QoS) [15].

- Arquitectura distribuida y modular: ROS 2 permite diseñar sistemas de *software* como una red de nodos distribuidos, donde cada nodo se encarga de una tarea específica (por ejemplo, la percepción de un sensor concreto, la planificación del control lateral o el control del vehículo), y esos nodos se puede comunicar mediante tópicos, servicios o acciones. Esta modularidad facilita:

    - La reutilización del *software*, ya que diferentes partes pueden ser intercambiadas o adaptadas para diferentes vehículos y escenarios.
    - La escalabilidad del sistema, pues se pueden introducir fácilmente nuevos nodos con diferentes funciones, sin modificar la arquitectura general.
    - El aislamiento de fallos, permitiendo de esta manera el funcionamiento general del sistema a pesar de pequeños fallos parciales, y además facilitando así la depuración de los programas.

3.1.1. Elección de ROS 2 Humble

Para la elección de la versión adecuada de ROS 2, se ha considerado como criterio principal que se trate de una versión oficialmente soportada por la comunidad de ROS. Bajo esta premisa, las versiones elegibles se reducen a ROS 2 Humble Hawksbill (soportada hasta mayo de 2027) y ROS 2 Jazzy Jalisco (soportada hasta mayo de 2029) [16].

La decisión de optar por ROS 2 Humble se fundamenta en la compatibilidad con el *hardware* disponible, específicamente un sensor LiDAR de la marca RoboSense, el cual no ha sido implementado aún en el sistema de percepción del vehículo autónomo, pero representa una opción interesante a considerar gracias a su potencial y disponibilidad. La última versión publicada por el fabricante de dicho sensor (año 2022) indica soporte explícito para ROS 2 Humble, sin referencias a versiones posteriores como ROS 2 Jazzy, pues Jazzy fue lanzada después de dicha *release* del LiDAR [17].

En consecuencia, se ha decidido instalar ROS 2 Humble en el sistema operativo Ubuntu 22.04 LTS, siendo esta la versión más reciente compatible con dicha versión de ROS 2. Esta instalación se llevará a cabo exclusivamente en el equipo denominado PC Fusión, ya que los módulos restantes del sistema (control y comunicaciones) operan sobre versiones más antiguas de Linux, las cuales no son compatibles con ROS 2 [16]. Por lo tanto, para comunicar los diferentes módulos se seguirán usando tecnologías ya implementadas previamente: MQTT y *sockets* TCP/IP, lo que permite mantener una integración coherente y funcional entre los componentes.

3.1.2. Implementación de ROS 2

El proceso de instalación de ROS 2 en el PC Fusión, fue llevado a cabo mediante instalación de paquetes binarios para Ubuntu. Para ello, se siguió paso a paso la guía disponible en la página web oficial de ROS, disponible de forma pública en la dirección web https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html.

Tras seguir dicha guía se consigue tener instalados todos los paquetes necesarios. El siguiente paso consiste en la creación y configuración de un espacio de trabajo en el cual se crearán los diferentes nodos ROS 2. Para ello se siguen los pasos presentados en la sección oficial de tutoriales de ROS 2, disponible en https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Configuring-ROS2-Environment.html.

Una vez ya se cuenta con el espacio de trabajo creado, el cual ha sido alojado dentro del PC Fusión en el directorio `~/pcfusion_ros2`, se puede ver que dentro de este se encuentran 4 directorios diferentes: `build`, `install`, `log` y `src`. Dentro de estos directorios, es importante mencionar la localización de los diferentes nodos y ficheros creados, y del fichero de configuración `setup.py`, el cual se modifica en base a dichos nodos. Este fichero de configuración se encuentra en el directorio `src/twizy_pcfusion`, y los diferentes ficheros y nodos modificados se encuentran en el directorio `src/twizy_pcfusion/twizy_pcfusion.`

Cada vez que se va a iniciar la ejecución del programa completo, es necesario seguir una serie de pasos. En caso de haberse modificado algo en el código, se deberán seguir los pasos 2 y 3 obligatoriamente para que tengan efecto dichos cambios, en caso contrario no es necesario:

- Finalizar la ejecución de los posibles nodos ROS 2 que se hayan quedado ejecutando (por ejemplo por una finalización forzosa del programa), para ello se ha creado un *script* de *Shell* llamado `limpiar_nodos.sh`, localizado también en el directorio home.

- La compilación del espacio de trabajo (*workspace*), mediante el comando `colcon build`.
- Lanzar el entorno (*enviroment*) con el comando `source install/setup.bash`. En este momento ya se puede lanzar el nodo principal, llamado `main_node.py`, mediante el comando `ros2 run twizy_pcfusion main_node`.

Tras este procedimiento, como se explicará más adelante en el capítulo, dicho nodo llamará al resto de nodos en base al estado en que se encuentre el coche. Con el objetivo de agilizar el proceso, se ha creado un *script* de `bash` llamado `ejecRos.sh`, localizado en el directorio personal o `home` (~) del propio PC, el cual al ejecutarse realiza los tres últimos pasos anteriores.

Los diferentes nodos creados, realizan diferentes tareas especializadas, ya sean de percepción del entorno, o bien de planificación de la ruta, siguiendo de esta manera la separación de tareas que se buscaba, para acercarse más a la estructura de coche autónomo explicada en la sección 2.2. En cuanto a los diferentes nodos creados, todos estos siguen una misma estructura. En primer lugar, en el constructor de la clase que define el nodo (función `__init__`):

- Se inicializan las diferentes variables a usar.
- Se crean las diferentes colas, tanto de salida como de entrada de datos, utilizando para ello la librería Python `queue`.
- Se definen los diferentes publicadores ROS 2, usando la función `create_publisher`, en la cual se especifica el tipo de datos a publicar, el tópico sobre el que se publican, y el tamaño máximo del *buffer* de salida.
- Se definen los subscriptores ROS 2, mediante la función `create_subscription`, especificando el tipo de datos a recibir, el tópico al que subscribirse, el *callback* al que llegarán los datos y el tamaño máximo del *buffer* de entrada.
- Se crean los diferentes *timers*, los cuales ejecutan una función determinada cada un tiempo concreto especificado. Dependiendo del uso requerido, se han usado *timers* mediante la función `create_timer`, o bien hilos (*threads*) usando la librería `threading`. Para el caso de los hilos no se especifica la frecuencia de ejecución, pues llama directamente a funciones que ejecutan un bucle *while*, por lo tanto esta frecuencia la determinará la frecuencia máxima de la CPU.
- En el caso de ciertos nodos, se configura la conexión con el Arduino o cámara que proporciona los datos del entorno, la conexión CAN con el módulo de comunicaciones o la conexión MQTT con el módulo de control.

A continuación, se especifican los diferentes *callbacks* de salida y de entrada de datos por y desde tópicos, para ello en una serie de casos determinados, se utilizan *buffers*, para que al llamar a estos *callbacks*, o bien se guarde el mensaje recibido en un *buffer*, o bien se lea el mensaje a enviar desde un *buffer*.

Los casos en que se han utilizado dichos *buffers* han sido en todos los *callbacks* de subscripción, con el objetivo de no perder ningún dato que ya se haya recibido. Para el caso de los *callbacks* de publicación, solo se han usado para los mensajes de control, es decir los mensajes no periódicos y constantes enviados a frecuencias altas, como podrían ser datos de sensores.

Se ha realizado este proceso con el objetivo de seguir la lógica de colas implementada previamente en el programa no ROS. Sin embargo, puesto que los propios tópicos ROS cuentan de por sí con colas, se podría plantear como una línea futura la eliminación de estos *buffers*, para evitar así esta dependencia con la librería `queue`, y quizá conseguir reducir latencias eliminando este procesado extra.

3.2. Nuevo esquema de comunicación entre nodos

Con la implementación de ROS 2 en el módulo PC Fusión y la migración de las funciones de actuación al módulo de control, se ha llevado a cabo una reestructuración del esquema de comunicaciones entre nodos. Esta reorganización ha permitido una distribución más eficiente de las tareas, mejorando la modularidad y el mantenimiento del sistema.

Actualmente, el sistema cuenta con un total de siete nodos principales:

- `camera_node`
- `ultrasonic_node`
- `RFID_receiver_node`
- `main_driver_node`
- `main_node`
- `CAN_receiver_node`
- `CAN_sender_node`

Los tres primeros nodos (`camera_node`, `ultrasonic_node` y `RFID_receiver_node`) actúan como nodos sensores, responsables de la percepción del entorno. Por su parte, los nodos `main_node` y `main_driver_node` procesan los datos proporcionados por los sensores y gestionan el tratamiento de errores. Finalmente, los nodos `CAN_receiver_node` y `CAN_sender_node` se encargan de la comunicación con el módulo de comunicaciones y con la unidad OBD (On-Board Diagnostics) del vehículo. Se explicará en detalle el funcionamiento de cada uno de estos nodos en los siguientes apartados del capítulo.

Funciones	Estructura multiprocesos	Estructura nodos ROS 2
Percepción entorno - cámara	main_driver	camera_node
Percepción entorno - ultrasonidos	Sensor_Controller	ultrasonic_node
Percepción entorno - RFID	RFID_receiver	RFID_receiver_node
Recepción datos CAN	CAN_receiver	CAN_receiver_node
Transmisión datos CAN	CAN_sender	CAN_sender_node
Planificación ruta + Control de errores y estado	main_driver proceso principal main	main_driver_node main_node
Control – marchas	set_Gear	set_Gear
Control - longitudinal	Throttle_Controller	Throttle_Controller
Control - lateral	main_driver	Epos_Controller

Tabla 1: Comparación funciones estructura previa vs actual

Con el objetivo de relacionar las funciones de percepción del entorno y de planificación de la ruta que realizan los nodos ROS 2 actuales con respecto a los subprocesos multiprocessing de la estructura previa se ha realizado una tabla comparativa

(Tabla 1). En esta tabla también se han incluido las funciones de actuación, de las cuales en la implementación actual se encarga el módulo de control, este dado que no es capaz de soportar ROS 2 usará también multiprocesos.

En el diseño del sistema de comunicación entre los distintos nodos, se optó por utilizar el modelo publicador/suscriptor mediante tópicos de ROS 2, en lugar de servicios o acciones. Se muestra una representación de este sistema en la Figura 13. Esta elección responde a las exigencias del sistema en cuanto a eficiencia, modularidad y capacidad para gestionar flujos de datos continuos sin introducir latencias innecesarias.

- Frecuencia alta y flujo continuo de datos: Los sensores involucrados en el vehículo generan datos a frecuencias elevadas y de forma continua. El modelo de tópicos está diseñado específicamente para este tipo de flujo de datos constante, permitiendo una transmisión eficiente y asíncrona sin necesidad de una solicitud explícita para cada dato. Sin embargo,

    - los servicios en ROS 2 son síncronos, bloqueantes y orientados a tareas puntuales de petición-respuesta. Requieren que un nodo solicite explícitamente cada dato. Esto introduce latencia y no escala bien con fuentes de datos de alta frecuencia.
    - Las acciones están pensadas para operaciones prolongadas y controlables (como tareas que pueden cancelarse o dar *feedback*), pero implican una fase de negociación inicial (*handshake*) que no sería necesaria para simples flujos de datos constantes.

- El uso de tópicos implica una menor complejidad de implementación en comparación con acciones, especialmente cuando no se requiere gestión explícita del estado de la tarea ni retroalimentación intermedia. Esto facilita el mantenimiento, la depuración y la ampliación futura del sistema.
- Los tópicos permiten que múltiples nodos se suscriban a los mismos datos sin que el publicador deba conocer a sus consumidores, lo que proporciona una arquitectura más modular y escalable.

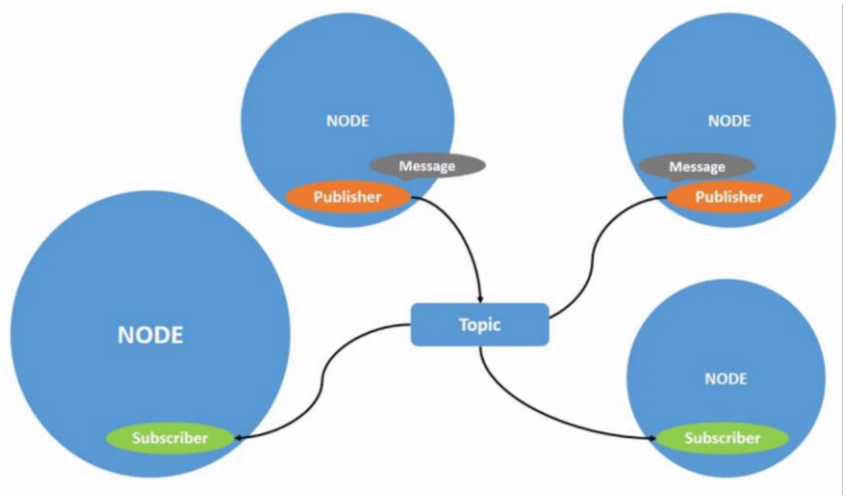

Figura 13. Esquema comunicación publicador/subscriptor [18]

Adicionalmente, la comunicación entre el módulo PC Fusión y el módulo de control se establece mediante el protocolo MQTT, como ya se justificó previamente en el apartado 2.6. Para ello, se implementan clientes MQTT tanto en el módulo PC Fusión como en el módulo de control. Esta comunicación también se basa en un modelo de publicador subscriptor y es unidireccional: el módulo PC Fusión transmite datos procesados al módulo de control, que utiliza esta información para llevar a cabo las tareas de actuación, tanto en el control lateral como en el longitudinal del vehículo.

El hecho de que esta comunicación sea unidireccional muestra una posible mejora a implementar en un proyecto futuro, pues sería de interés que el PC Fusión pudiese comprobar en todo momento el estado de los actuadores, como si ocurre en su conexión con el módulo de comunicaciones.

Para llevar este proceso a cabo, y siguiendo la estructura actual, sería interesante crear una serie de publicadores MQTT en los diferentes multiprocesos actuadores. Estos procederían a declarar su estado de forma periódica, y se añadiría en el nodo `main_driver_node` un suscriptor MQTT que procesara dichos estados. De esta manera se conseguiría aumentar la robustez del sistema, evitando discordancias no detectadas entre las diferentes partes de este.

Previo a la presentación del esquema general de comunicaciones, se describen en detalle las funcionalidades de cada uno de los nodos, especificando los datos que reciben, el procesamiento que realizan y los nodos destinatarios de la información generada. Con el objetivo de entender mejor la arquitectura de comunicaciones, se muestra en la Figura 14un esquema general simplificado.

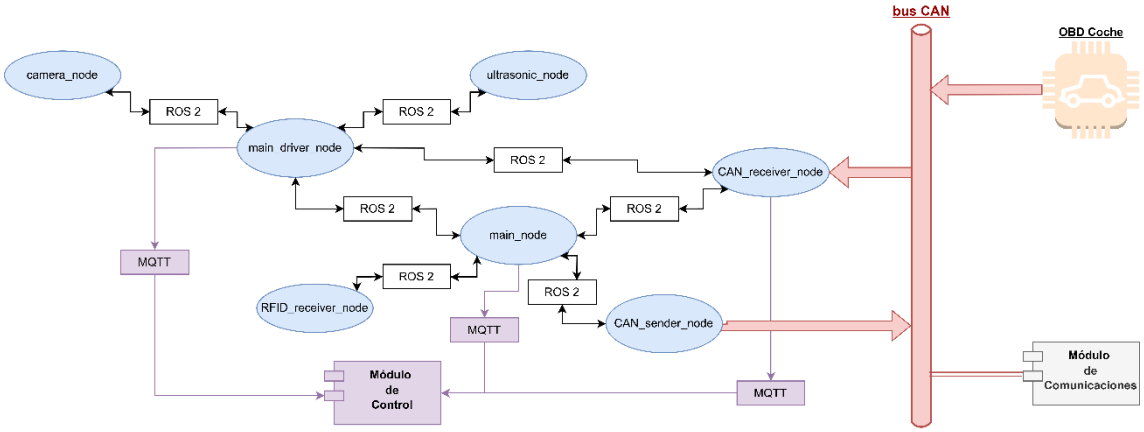

Figura 14. Esquema general de comunicaciones simplificado

3.2.1. Nodos sensores

Dentro del módulo PC Fusión contamos con 3 nodos que se encargan de la percepción del entorno: cámara, sensores de ultrasonidos y RFID (Identificador por radiofrecuencia).

En el proyecto TwizyLine, se instaló una cámara para reconocer una línea pintada en el suelo y seguir la trayectoria marcada por esta línea, calculando el ángulo del eje del

vehículo frente a la tangente de la trayectoria y la distancia normal entre el punto de referencia del vehículo y la trayectoria definida por la línea.

Este tipo de sensores, como son las cámaras y los LIDAR, tienen la desventaja de requerir una alta carga computacional para procesar las imágenes en tiempo real, de ahí surge la necesidad principal de introducir el PC Fusión en el sistema, poder manejar esta carga.

Los sensores de ultrasonidos funcionan mediante la emisión y detección de ondas ultrasónicas para medir la distancia entre el sensor y un objeto, son muy útiles para detectar objetos cercanos. Estos detendrán el vehículo en caso de detectar un objeto demasiado cerca, evitando de esta manera una posible colisión.

A diferencia de las cámaras y los ultrasonidos, que proporcionan información sobre objetos y distancias, el sensor RFID ofrece una identificación discreta de la ubicación basada en la lectura de etiquetas. En el proyecto TwizyLine, se utilizarían diferentes etiquetas RFID distribuidas por el parking, para saber así la posición del vehículo en este.

Cabe destacar que, aunque el código necesario para la utilización de los sensores de ultrasonidos y RFID ya ha sido implementado, no se han realizado pruebas reales de su funcionamiento dentro del sistema global. Esto se debe, por un lado, a problemas técnicos internos detectados en el sensor de ultrasonidos, y por otro, al hecho de que los sensores RFID no han sido empleados en los desarrollos más recientes del proyecto. No obstante, con el fin de facilitar una posible integración futura, el código correspondiente ha sido dejado en el proyecto debidamente comentado, de manera que pueda ser fácilmente habilitado cuando se decida incorporar estos sensores al sistema completo. Dicho esto, a continuación se muestran en detalle los diferentes nodos sensores con los que se cuenta en el sistema.

3.2.1.1. Nodo de la cámara: `camera_node`

El nodo `camera_node` es responsable de la captura y procesamiento de imágenes obtenidas a través de la cámara instalada en el sistema. También está capacitado para recibir mensajes de control que indican la finalización del proceso.

Además, se ha implementado un envío de las imágenes procesadas por la cámara mediante UDP con el objetivo de visualizar en tiempo real, desde un PC externo, las imágenes que el vehículo está capturando y procesando. Aunque el sistema principal del vehículo utiliza ROS 2 para la gestión y procesamiento de datos, el envío por UDP se ha añadido de forma complementaria para facilitar las pruebas y validación remota, ya que permite una transmisión con mínima latencia, algo esencial para una supervisión eficiente durante el desarrollo y testeo del sistema.

Se muestra en la Figura 15 un ejemplo de imagen procesada recibida en otro PC externo:

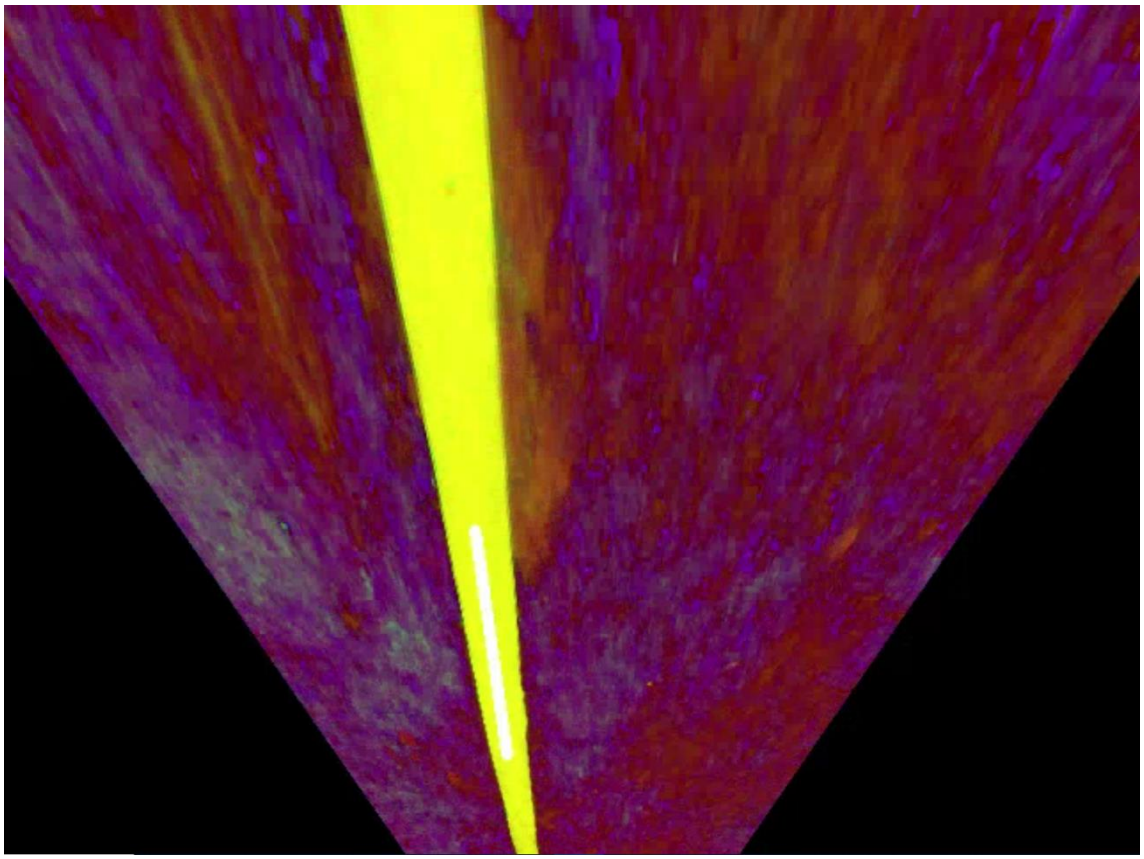

Figura 15. Imagen procesada recibida en PC externo

3.2.1.1.1. Instalación y configuración de la cámara

Para la correcta inicialización y funcionamiento del módulo de cámara, fue necesario realizar una serie de pasos de configuración en el sistema operativo. A continuación, se detallan los comandos ejecutados y su propósito:

- Clonado del repositorio con el firmware y herramientas necesarias:
    - `git clone https://github.com/digitalloggers/geocam-bin.git`

Este comando clona el repositorio que contiene los ficheros de firmware y utilidades necesarias para la inicialización del dispositivo de cámara.

- Creación del directorio de firmware y copia de archivos:
    - `sudo mkdir /lib/firmware/mxcam`
    - `sudo cp geocam-bin/* /lib/firmware/mxcam`
    - `sudo cp /lib/firmware/mxcam/mxcam /usr/bin`

Se crea el directorio `/lib/firmware/mxcam`, donde se almacenarán los ficheros del firmware. Luego, se copian los ficheros desde el repositorio clonado hacia dicho directorio. Finalmente, se copia el ejecutable `mxcam` a `/usr/bin` para que esté disponible globalmente como comando del sistema.

- Creación de la regla `udev`:
    - `sudo nano /etc/udev/rules.d/99-mxcam.rules`
    - Se crea un fichero de reglas `udev`, el cual permite que el sistema reconozca automáticamente el dispositivo cuando se conecte. En este archivo se añade la siguiente línea:
    - `ATTR{idVendor}=="29fe", ATTR{idProduct}=="b00c", MODE="0660", OWNER="root", GROUP="video", RUN+="/usr/bin/mxcam boot /lib/firmware/mxcam/gc6500_ddrboot_fw.img /lib/firmware/mxcam/config.json /lib/firmware/mxcam/sensor_ov2710_mayfield_le.bin"`

Esta regla indica que, cuando se detecte un dispositivo USB con identificador de proveedor 29fe y producto b00c, se le asignen permisos específicos y se ejecute automáticamente el comando `mxcam boot`, el cual carga el firmware correspondiente al sensor de la cámara.

- Recarga del demonio `udev` y aplicación de las nuevas reglas:
    - `sudo udevadm control --reload`
    - `sudo udevadm trigger`

Estos comandos recargan la configuración de `udev` y activan las reglas recién creadas sin necesidad de reiniciar el sistema.

Una vez se realizaron estos pasos, la cámara pudo conectarse a un puerto USB del módulo PC Fusión y, tras un periodo de inicialización de aproximadamente 30 segundos a un minuto, ser detectada por el sistema. La correcta detección se verificó mediante el comando `lsusb`, donde la cámara aparecerá con el identificador 29fe:4d53 GEO Semi Condor, conforme a lo establecido en la configuración. Se muestra en la Figura 16 la salida de dicho comando con la cámara detectada.

```
twizyline@twizyline-ZBOX:~$ lsusb
Bus 006 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 005 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 006: ID 8087:0029 Intel Corp. AX200 Bluetooth
Bus 001 Device 004: ID 0bda:0153 Realtek Semiconductor Corp. 3-in-1 (SD/SDHC/SDXC) Card Reader
Bus 001 Device 007: ID 1a86:7523 QinHeng Electronics CH340 serial converter
Bus 001 Device 008: ID 29fe:4d53 GEO Semi Condor
Bus 001 Device 003: ID 05e3:0610 Genesys Logic, Inc. Hub
Bus 001 Device 002: ID 1d50:606f OpenMoko, Inc. Geschwister Schneider CAN adapter
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Figura 16. Salida lsusb para verificar conexión de la cámara

3.2.1.1.2. Funcionamiento del nodo `camera_node`

El nodo hace uso de la librería cv2 para capturar los *frames* de vídeo. Se configuran diversos parámetros como la resolución de la imagen y la tasa de captura de

frames por segundo, la cual se ha conseguido aumentar desde los 6 *fps* que se contaba previamente hasta el máximo posible de la cámara, 30 *fps* . En cuanto a la comunicación, el nodo actúa tanto como publicador como suscriptor:

- Publicador: envía mensajes en formato Float32MultiArray a través del tópico `/camera_processed`.
- Suscriptor: recibe mensajes de tipo String desde el tópico `/camera_topic`, que se almacenan en el *buffer* `camera_buffer`.

Se define un *timer* para el bucle principal del nodo usando el sistema de *timer* nativo de ROS 2, el `create_timer`, este se ejecuta cada 1/30 segundos. Para cada iteración del bucle:

Se lee el *frame* obtenido de la cámara, al cual aplicando las funciones `transformar_perspectiva` y `calcular_ruta`, de la clase `artemis_autonomous_car`, se consigue obtener el ángulo de la ruta, el *offset* de la ruta y un parámetro de error. Es importante destacar en este punto otro de los problemas que surgieron a la hora de desarrollar el presente proyecto, pues una vez funcionaba correctamente el sistema, se vio que estos tres valores obtenidos no correspondían con la realidad. Después de analizar que estaba ocurriendo, se advirtió que se debían ajustar los parámetros `lower_promiscuos_color` y `upper_promiscuous_color` de la clase `artemis_autonomous_car`. Estos dos parámetros fijan el valor máximo y mínimo del filtro de color de la imagen, que estará codificada en formato HSV (Hue-Saturation-Value), esto es, el valor mínimo y máximo que se espera para:

- H (Hue): determina el color.
- S (Saturation): controla la pureza o intensidad del color.
- V (Value): indica la luminosidad.

Todos los píxeles que se encuentran dentro de ese rango pertenecen, en principio, a la línea que marca la trayectoria, permitiendo de esta manera detectarla fácilmente.

Para conseguir los valores correctos de estos dos parámetros, se utilizaron en primer lugar los dos valores que estaban establecidos previamente, se ejecutó de forma independiente el nodo, y se visualizó en tiempo real la cuarta salida de la función `calcular_ruta`. Esta representa en color negro lo que no detecta como línea, y en color blanco lo que sí detecta como línea. En base a esta salida, denominada `frameF` se ajusta de manera manual ambos parámetros, hasta conseguir que aplicando dicho filtro se vea en blanco la línea que marca la trayectoria. En la Figura 17 se muestra la imagen real captada por la cámara, y en la Figura 18 se muestra la salida `frameF` habiéndose dado unos valores correctos para la iluminación concreta de dicho entorno.

Estableciéndose estos 2 parámetros de forma correcta, los valores de ángulo de la ruta, el offset de la ruta y un parámetro de error obtenidos se ajustarán correctamente a la realidad. Estos valores se envían por el publicador de la siguiente manera:

- `array_cam.data = [float(anguloRuta),float(offsetRuta), float(error)]`
- `self.publisher2_.publish(array_cam)`

Figura 17. Imagen real cámara

Figura 18. Imagen filtrada cámara

Además, se gestionan los mensajes almacenados en el `camera_buffer`, los cuales provienen del nodo `main_driver_node.` Estos pueden ser mensajes TERMINATE para detener la ejecución del nodo, o bien mensajes con el parámetro `bif`, el cual determina la decisión a tomar en caso de bifurcaciones, lo especifica el usuario al enviar una señal MQTT de tipo GOTO, este es requerido por la función `calcular_ruta`.

En la Figura 19 se muestra el esquema de comunicación de este nodo con el resto del sistema.

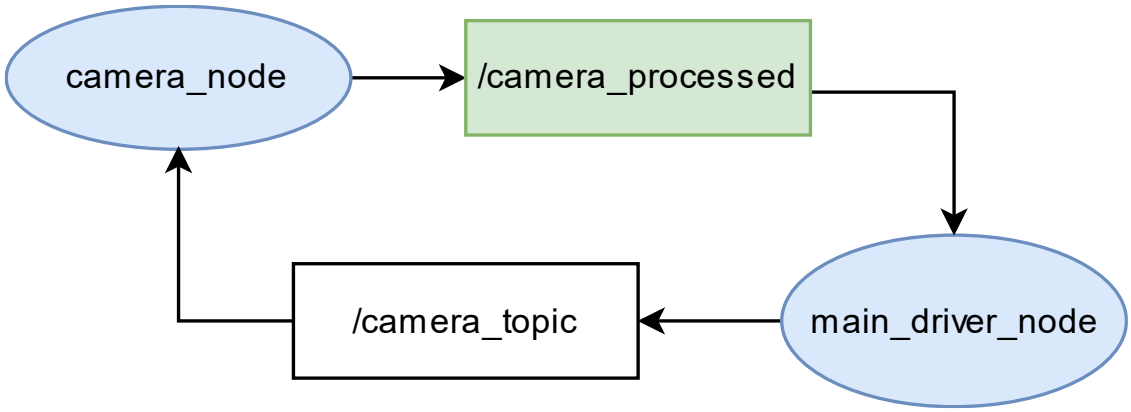

Figura 19. Esquema de comunicación `camera_node` *- resto del sistema*

A mayores, como se mencionó previamente en la sección 3.2.1.1. , se ha implementado en el propio nodo un sistema en el que se consigue enviar mediante UDP, utilizando la herramienta `ffmpeg` la salida de los *frames* de la cámara tras el procesado. Con esto se consigue ver con poca latencia desde cualquier ordenador, la salida procesada de la cámara, y por tanto la línea de dirección que sigue el coche, siendo de gran utilidad a la hora de realizar pruebas del funcionamiento del vehículo.

3.2.1.2. Nodo del sensor de ultrasonidos: ultrasonic_node

Este nodo recibe los datos del sensor de ultrasonidos, sensor que está conectado mediante un USB, el cual es necesario que sea detectado y asignado a este nodo por el PC. Para manejar las interfaces USB se crean variables de entorno del sistema operativo con el fin de almacenar en ellas las direcciones de los periféricos [5].

Para hacer esto posible, se diseñó anteriormente un *script* que va recorriendo todos los periféricos conectados. Este *script* abre la comunicación serie con cada periférico de manera individual y se queda esperando a que este envíe mensajes relacionados con el funcionamiento del controlador [5].

Originalmente, este *script* se ejecutaba en el módulo de control. Sin embargo, tras la reestructuración del sistema, ha sido trasladado al módulo PC Fusión. Dado que también era usado para detectar actuadores, ha sido necesario también una adaptación de este para que solo detectara los sensores del vehículo.

Una vez se ejecuta el *script*, tarea que se realiza manualmente al modificar la posición de los usb, se asigna al usb correspondiente a los ultrasonidos una variable de

entorno llamada `Sonidos_USB`. Para poder recoger esta información desde el programa principal necesitamos utilizar las funciones de la librería `os` de Python como es `os.environ()` [5].

Este *script* solo necesita ejecutarse una vez, siempre y cuando no se desconecten ni cambien de puerto los dispositivos USB. Si los USB permanecen conectados en el mismo orden, no es necesario volver a ejecutarlo incluso si el vehículo se apaga y se vuelve a encender.

3.2.1.2.1. Funcionamiento del nodo `ultasonic_node`

La comunicación entre el nodo y el sensor de ultrasonidos se realiza a través de un microcontrolador Arduino, el cual gestiona la lectura de los datos desde el sensor y su envío mediante la interfaz serie. Al iniciar el nodo, se envía una señal de encendido (ON) al Arduino para establecer la conexión.

En cuanto a la arquitectura de comunicaciones, el nodo funciona tanto como publicador como suscriptor:

- Publicador: envía mensajes en formato Int32MultiArray a través del tópico `/ultrasonicData_topic`.
- Suscriptor: recibe mensajes de tipo String desde el tópico `/sensorultra_topic`, que se almacenan en el *buffer* `sensorultra_buffer`.

Se definen dos *timers* usando `create_timer`, un bucle principal y un bucle de publicación.

- El bucle principal del nodo se ejecuta cada 0.001 segundos, y para cada una de sus iteraciones se comprueba el estado de la conexión con el microcontrolador Arduino, en caso de conexión activa, se leen los datos enviados por el puerto serie. Si los datos recibidos están en el formato esperado se habilitará el envío de dichos datos por el bucle de publicación, mediante la variable `publicar`.
- Este bucle de publicación se ejecuta cada 1/60 segundos, pues el factor limitante a la hora de procesar los datos en el `main_driver_node` serán los *frames* de la cámara (1/30 segundos), y ejecuta la siguiente instrucción cuando la variable `publicar` tenga el valor *True*.

    - ```
      self.publisher_.publish(Int32MultiArray(data=[int(val
      ues[1]) - 24, int(values[2]), 400]))
      ```

También se gestionan los mensajes almacenados en el `sensorultra_buffer`, los cuales provienen del nodo `main_driver_node`. Estos son mensajes TERMINATE cuyo objetivo es detener la ejecución del nodo.

En la Figura 20 se muestra el esquema de comunicación de este nodo con el resto del sistema.
      ```

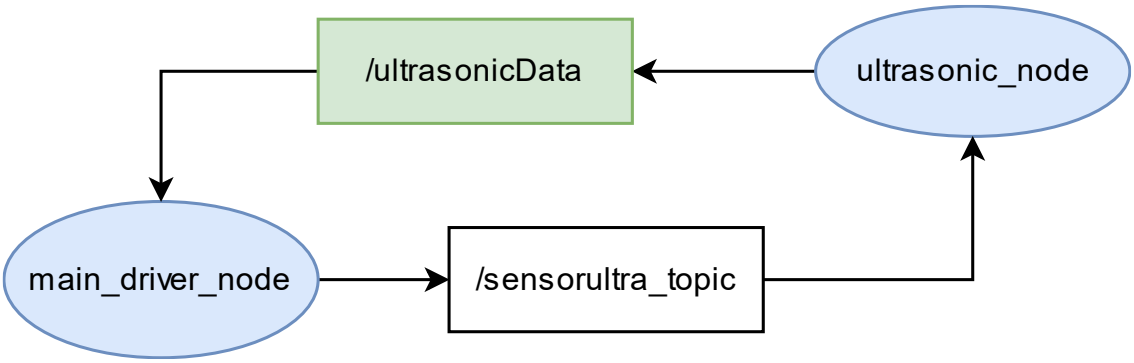

Figura 20. Esquema de comunicación `ultrasonic_node` *- resto del sistema*

Como se mencionó previamente en la sección 3.2.1. , en el diseño actual no se está utilizando este nodo, debido a problemas internos del propio *hardware* de los sensores de ultrasonidos. Por lo tanto el bucle publicador no se está ejecutando, en un trabajo futuro en el que se hayan arreglado los problemas de estos sensores, en lo respectivo a este nodo, simplemente será necesario volver a habilitar la ejecución del bucle publicador.

3.2.1.3. Nodo del sensor RFID: `RFID_receiver_node`

Este nodo se encarga de recibir los datos de las etiquetas RFID que detecta el sensor RFID. Para gestionar la comunicación entre el sensor y el nodo, se usa un microcontrolador Arduino. Este transmite dichos datos al nodo usando una conexión USB. Para ello, igual que ocurría con el anterior nodo, explicado en la sección 3.2.1.2. , será necesario que el sistema haya asociado dicho USB a en este caso la variable de entorno `RFID_USB`, para lo que es necesario que se haya ejecutado el *script* detector de terminales adaptado al PC Fusión.

3.2.1.3.1. Funcionamiento del nodo `RFID_receiver_node`

En cuanto a las comunicaciones, el nodo funciona tanto como publicador como suscriptor:

- Publicador: envía mensajes en formato String a través del tópico `/rfid_topic`.
- Suscriptor: recibe mensajes de tipo String desde el tópico `/rfidterm_topic`. Estos son mensajes TERMINATE enviados por el `main_node`, los cuales al ser recibidos finalizan la ejecución del nodo.

Se define un *timer* principal usando `create_timer` que ejecuta un bucle cada 0.25 segundos. Durante cada iteración de este bucle:

- Se procesan los datos provenientes del Arduino recibidos del lector RFID.
- El resultado de este procesamiento se almacena en una variable llamada `data`.
- Esta variable `data` se introduce en el *buffer* de salida `rfid_bufferOut`, el cual funciona como una cola temporal para organizar los mensajes a ser enviados.

De forma paralela, se define un segundo *timer* que ejecuta un bucle cada 0.1 segundos. Este bucle es responsable de leer los datos almacenados en el *buffer* de salida, y enviar estos al publicador.

Este enfoque basado en doble temporización, el cual también es usado en el nodo de ultrasonidos permite desacoplar el tiempo de procesamiento del tiempo de publicación, lo cual es especialmente útil en entornos donde los datos RFID pueden llegar de forma irregular o en ráfagas.

En la Figura 21 se muestra el esquema de comunicación de este nodo con el resto del sistema.

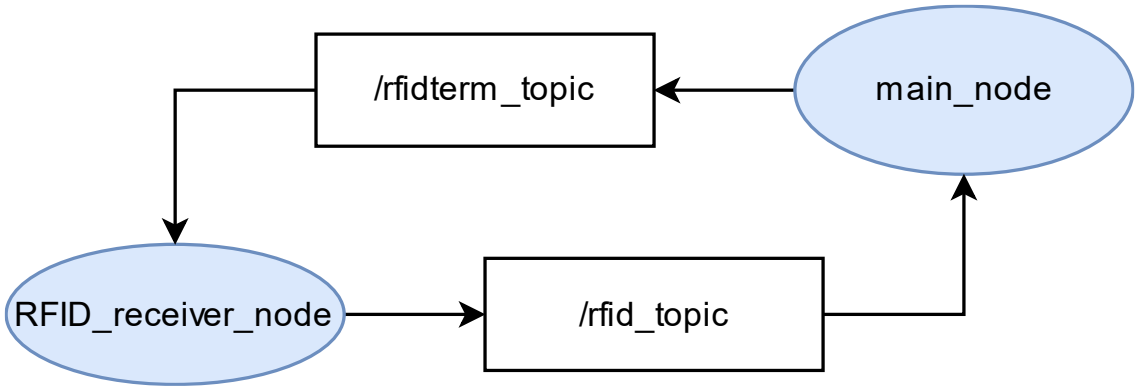

Figura 21. Esquema de comunicación RFID_receiver_node - resto del sistema

Como también se explicó previamente en la sección 3.2.1. , este nodo no se ha llegado a usar en la práctica debido a que no ha sido empleado en los desarrollos más recientes del proyecto. Sin embargo, el código tiene implementada toda la lógica para que pueda comenzar a usarse en un futuro desarrollo.

3.2.2. Nodos CAN (Controller Area Network)

Como se ha explicado en la sección 2.5. , anteriormente las comunicaciones CAN eran entre los módulos de control, de comunicaciones y el OBD del coche. Actualmente modificamos este esquema, pues el módulo de control no entrará en las comunicaciones CAN, y si lo hará el módulo PC Fusión.

Para ello son necesarios 2 nodos: `CAN_receiver_node` y `CAN_sender_node`. Como se puede intuir por su nombre, el primero se encargará de recibir los datos enviados por CAN del módulo de comunicaciones y del OBD, y el segundo de enviar datos hacia el módulo de comunicaciones.

Los datos intercambiados tendrán que ver con el estado de los diferentes módulos, el control de errores, datos del estado actual del vehículo y datos sobre el estado próximo del vehículo. Al igual que ocurría anteriormente, se seguirá recibiendo la velocidad actual del vehículo y la marcha introducida, siendo estas enviadas desde el OBD.

En cuanto a los estados posibles en que se encuentra el vehículo, se diferencian cinco, los cuales fueron definidos en el TFG de Ignacio Royuela [19]. Se explican resumidamente a continuación:

- Estado 0, Start-up: es el estado en el que el sistema se está iniciando o reiniciando debido a una falla. En este estado, el PC Fusión y el módulo de comunicaciones se inician al mismo tiempo, verifican que todos los dispositivos estén conectados y se sincronizan.
- Estado 1, Normal: "es el estado en el que se encuentra el sistema cuando el vehículo está siendo conducido por un conductor corriente. En este estado, la mayoría de los sistemas conectados al módulo de control permanecen apagados, ya que el vehículo es manejado por una persona" [19].
- Estado 2, Autónomo: "es el estado en el que se encuentra el sistema cuando opera sin conductor, es decir, cuando las tareas de conducción del vehículo son realizadas por el sistema instalado. En este estado, todos los dispositivos instalados en el vehículo están encendidos y en uso" [19].
- Estado 3, Stand-by: "es el estado de ahorro de energía del sistema. En este modo, los sensores, actuadores y el receptor GPS permanecen apagados" [19].
- Estado 4, Fallo: "es el estado al que se pasa cuando cualquiera de los dos módulos detecta un error. En este estado, si el último estado fue el modo Autónomo, el vehículo se detiene automáticamente. Mediante mensajes CAN, ambos módulos se colocan en este estado sin importar cuál de ellos haya provocado el error" [19].

3.2.2.1. Nodo receptor CAN: `CAN_receiver_node`

El nodo `CAN_receiver_node` tiene como función principal la recepción, procesamiento y publicación de información proveniente del OBD del vehículo y del módulo de comunicaciones, sirviendo como intermediario clave en el sistema de comunicación interna del vehículo, pues también intercambiará información con otros nodos, como el nodo principal `main_node`.

En cuanto a la tipología de los mensajes recibidos a través del bus CAN, el nodo recibe cuatro tipos principales de mensajes:

- Desde el OBD del vehículo:
    - Estado actual de las marchas.
    - Velocidad actual del vehículo.
- Desde el módulo de comunicaciones:
    - Mensajes `COM_status`, que informan del estado de dicho módulo.
    - Mensajes `datosGOTO` a través de una pila CAN ISO-TP. Estos se procesan para determinar la velocidad objetivo del vehículo.

3.2.2.1.1. Funcionamiento del nodo `CAN_receiver_node`

Al inicio de la ejecución, se procede a la inicialización tanto del bus CAN como del hilo TP-Thread. Para el bus CAN se definen tres filtros con el fin de escuchar exclusivamente los mensajes con identificadores asociados a: `COM_status`, velocidad y marchas.

En lo relativo a las comunicaciones con otros nodos, el nodo funciona como suscriptor y como publicador:

- Suscriptor: recibe mensajes en formato String a través del tópico `/can_rx_term_topic`, enviados por el `main_node` y los almacena en el *buffer* `terminate_buffer`.
- Publicador: El nodo enviará mensajes por varios tópicos:
    - Envía mensajes tipo String en los tópicos `/can_receiver_topic` y `/can_sender_topic.`
    - Mensajes tipo Float32 por el tópico `/currentSpeed_topic`.
    - Mensajes tipo Int32 por el tópico `/emergencyStop_topic`.
- También contará con un publicador pero esta vez de mensajes MQTT, estos irán por el tópico `/gear_topic`, y con destino el módulo de control. Los mensajes enviados por el tópico `/currentSpeed_topic` también se enviarán por MQTT a este módulo.

Previo a la explicación de los diferentes bucles, es importante recordar el uso de colas para el envío y la recepción de datos. Se han definido dos tipos de datos, datos de carácter más sensible, los cuales se prioriza para que no haya pérdidas, y datos más recurrentes en los que se prioriza que haya menos latencia.

Para todos los datos entrantes al nodo, siempre se almacenan en diferentes *buffers* de entrada, sin embargo para los datos salientes, en el caso de los datos más recurrentes, como son la velocidad actual del vehículo por ejemplo, no pasarán por un *buffer* de salida. De aquí en adelante, con el objetivo de economizar el lenguaje, se dirá que se envían los datos hacia el siguiente nodo sin mencionar estos *buffers* de salida.

Por el bus CAN se reciben diferentes tipos de mensajes, los mensajes recibidos en el PC Fusión que son enviados por el módulo de comunicaciones se muestran en la Tabla 2.

Mensaje	Explicación
CSR	Primera trama COMM-STATUS recibida
S0	STATE-COM señal=00 (Start Up)
S1	STATE-COM señal=01 (Normal mode)
S2	STATE-COM señal=10 (Autonomous mode)
S3	STATE-COM señal=11 (Stand by)
P0	PAUSE-COM señal=0
P1	PAUSE-COM señal=1
G (goto data)	mensaje GOTO ISO-TP recibido
W (código de error) [(atributo)]	Señalización de un error tipo *warning*
E (código de error) [(atributo)]	Señalización de un error tipo *error*

Tabla 2. Tipos de mensajes CAN enviados por el módulo de comunicaciones [19]

3.2.2.1.2. Estructura del nodo `CAN_receiver_node`

El nodo se estructura en torno a un hilo `threading.Thread`, el cual ejecuta el bucle principal del nodo de forma continua, a la frecuencia máxima proporcionada por la CPU. Se ha establecido este hilo en lugar de un `create_timer` estándar de ROS debido a que este nodo al estar recibiendo datos por CAN a una frecuencia muy alta, interesaba ampliar su capacidad de procesado.

A mayores de este bucle principal, se cuenta con dos bucles más, esta vez utilizando `create_timer`. El primero de ellos sirve para comparar tiempos de diferentes puntos del código, actuando a modo de temporizador pero adaptado a las herramientas de ROS. El segundo se encarga de la publicación de la velocidad (cada 0.1 segundos), concretamente envía la variable `CurrentSpeed` a través de dos tópicos:

- Un primer tópico `/currentSpeed_topic`, el cual va dirigido hacia el nodo `main_driver_node`.
- El segundo tópico tiene el mismo nombre, pero en este caso no es un tópico ROS 2, sino MQTT, con destino el módulo de control.

En cuanto al bucle principal, el cual lleva la mayor parte del procesado del nodo, en primer lugar espera a recibir un mensaje `COM_status`, cuando esto sucede, añade el mensaje "CSR" al *buffer* `can_receiver_bufferOut` y lo publicará por el tópico `/can_receiver_topic`. Este mensaje le llegará al `main_node`, el cual hará que el coche salga del estado de Start-up.

A continuación, iniciará el nodo `CAN_sender` e iniciará los temporizadores `COM_STATUS_reception_timer` y `Speed_reception_timer`. Estos se ejecutan y duran 0.5 y 0.3 segundos respectivamente, si en ese tiempo no se han cancelado, el sistema lo toma como que el módulo de comunicaciones ha dejado de mandarle estos datos, y por ello mandarán los errores 33 y 140 respectivamente hacia el `main_node`. El funcionamiento de estos temporizadores se realiza de forma diferente.

- El `Speed_reception_timer` funciona en base a la función Timer de la librería Threading, para ello se introducen dos argumentos:
    - Tiempo en segundos que establece la frecuencia de ejecución del temporizador.
    - Función que ejecutar en caso de cumplirse dicho tiempo sin haberse cancelado el temporizador antes.

- El `COM_STATUS_reception_timer` se basa en la ejecución del primer `create_timer`, mencionado anteriormente, el cual se ejecuta cada 0.5 segundos, y compara el tiempo actual con el tiempo capturado en el punto anterior del bucle principal.

Siguiendo con la ejecución del bucle principal tras la llegada de un mensaje `COM_status`, finalmente le da valor 1 a la variable `ready`, con esto conseguimos que en

la siguiente iteración del bucle no se ejecute este flujo, sino el siguiente a comentar. Este siguiente flujo se puede dividir en los siguientes mensajes recibidos:

- Mensajes ISO-TP

Ahora en cada iteración se estará consultando si hay mensajes ISO-TP disponibles en la pila CAN ISO-TP, estos mensajes son enviados desde el módulo de comunicaciones, y son mandados hacia este módulo para indicar la velocidad longitudinal objetivo del vehículo y el parámetro `bif`, que indica la acción a realizar en las bifurcaciones. Ambos parámetros son seleccionados por el usuario al enviar un mensaje MQTT de tipo GOTO. Si resulta que sí hay mensajes disponibles, se leerán y se enviarán al `/can_receiver_topic` como "G" más el mensaje, y se enviarán al `/can_sender_topic` como "G1". También se iniciará otro temporizador: `GOTOACK_off_timer`, el cual tendrá un segundo para ser cancelado, o se enviará el mensaje de error "G0" al `main_node`. Cada vez que se detecte un nuevo mensaje ISO-TP se reseteará dicho temporizador. El sentido de dicho temporizador viene de que el protocolo de transporte no garantiza una transmisión completamente fiable porque no todos los mensajes son confirmados (ACK). El receptor sabe cuántos bytes debe recibir y puede detectar si falta alguno, pero necesita avisar al emisor [19].

- Mensajes CAN

Se llama a la función `procesar_msg_can`, esta función para cada iteración mira el bus CAN intentando recibir un mensaje esperando como máximo 1 ms. Si no hay mensaje en ese tiempo, devuelve `None`, lo que evita bloquear la ejecución del programa., al leer el mensaje recibido de este, los filtramos entre los tres posibles identificadores para ver si es un mensaje de `COM_status` (0x100), de velocidad (0x19F) o marchas (0x59B).

  - `COM_status`

    Para el primer caso, diferencia los 4 submensajes:

    - STATE_COM, PAUSE_COM: si cambian respecto a la iteración anterior, se publican en `/can_receiver_topic` con los prefijos "S" y "P" respectivamente.
    - RFID_ACK, CON_ERR_ACK: si su valor es diferente de 0 se publican en `/can_sender_topic`.

    Cada vez que se recibe un mensaje `COM_status` se toma el valor de tiempo actual, para actualizar la comprobación en el temporizador `COM_STATUS_reception_timer`.

  - Marchas (*gear*)

    Para cada mensaje se extraerá la variable de frenos *brakes* y la variable de marcha *gear*. En primer lugar, se envía la variable *gear*

mediante MQTT por el tópico `/gear_topic` hacia el módulo de control, para que de esta manera los actuadores sepan la marcha actual introducida.

Después se maneja el mensaje de `EmergencyStop` en función de estas dos variables, cuando tenga un valor diferente al anterior se enviará por el tópico `/emergencyStop_topic`.

- Velocidad (`currentSpeed`)

Para este tipo de mensajes, se extrae la velocidad longitudinal actual del mensaje CAN mediante las siguientes líneas:

```
data = msg.data
signal = (data[2] << 4) + (data[3] >> 4)
speed = round(((signal-2000)*10/7250)*80, 1)
self.CurrentSpeed.data = speed
```

Por cada iteración se reseteará el temporizador `Speed_reception_timer`. Esta variable `currentSpeed` no se enviará cada vez que llegue la velocidad o cada vez que cambie, sino que se manda de manera recurrente en el último de los bucles.

- Mensajes TERMINATE

Lee los mensajes almacenados en el *buffer* `terminate_buffer`, si recibe un mensaje TERMINATE, se detiene la ejecución del nodo

Con el objetivo de concentrar y facilitar el entendimiento de la lógica del nodo, se muestra en la Figura 22 un diagrama de flujo del funcionamiento de este nodo.

3.2.2.2. Nodo emisor CAN: `CAN_sender_node`

El nodo `CAN_sender_node` tiene como función principal la comunicación mediante CAN desde el módulo PC Fusión hacia el módulo de comunicaciones. Enviará una serie de mensajes para el control de errores y del estado del vehículo, en concreto:

Mensajes `CON_status`, que informan sobre el estado del módulo PC Fusión.
Mensajes de error `CON_err`.
Mensajes RFID.

3.2.2.2.1. Funcionamiento del nodo `CAN_sender_node`

Al inicio de la ejecución, se procede a la inicialización del bus CAN, junto a variables, temporizadores y colas.

En lo relativo a las comunicaciones con otros nodos, el nodo funciona como suscriptor y como publicador:

Suscriptor: recibe mensajes en formato String a través del tópico `/can_sender_topic`, enviados por el `main_node` y el `CAN_receiver_node`. Estos los almacena en el *buffer* `can_sender_buffer`.

Publicador: El nodo enviará mensajes tipo String por el tópico `/can_receiver_topic`. Estos son mensajes de error tras haber saltado un temporizador `ACK_not_received`, que se envían hacia el `main_node`.

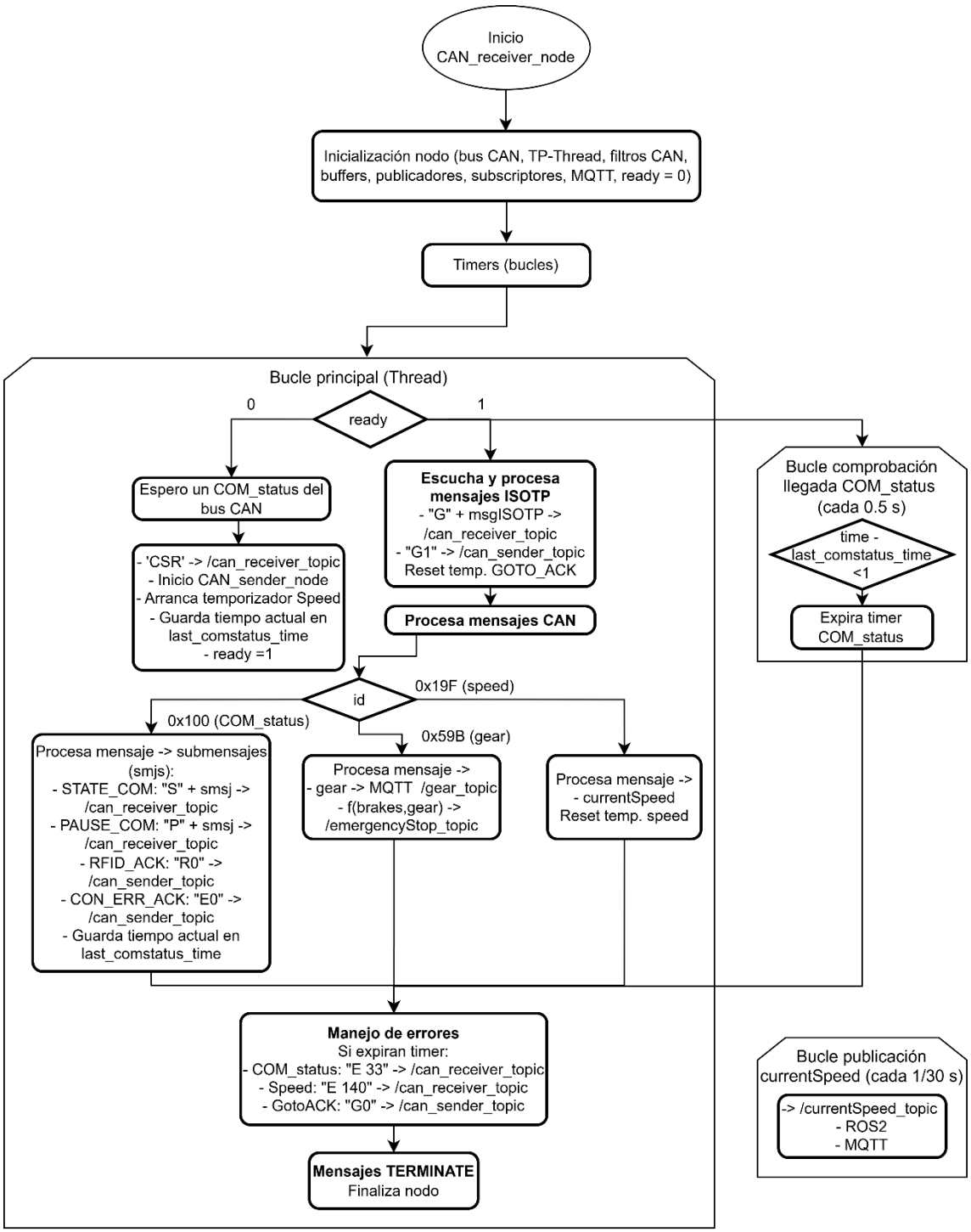

Figura 22. Diagrama de flujo `CAN_receiver_node`

3.2.2.2.2. Estructura del nodo CAN_sender_node

La estructura del nodo se basa al igual que ocurría con el anterior nodo CAN en un hilo `threading.Thread`, el cual ejecuta el bucle principal del nodo de forma continua, a la máxima frecuencia dada por la CPU. Se ha establecido este hilo en lugar de un `create_timer` estándar de ROS debido a que este nodo al estar enviando datos por CAN a una frecuencia muy alta interesaba ampliar su capacidad de procesado.

A mayores de este bucle principal, se cuenta con un bucle más llamado `publicar_can`, esta vez utilizando `create_timer`. Este bucle se encarga de publicar de forma recurrente los mensajes de tipo `CON_STATUS` por el bus CAN, concretamente cada 0.1 segundos. Esta ha sido una modificación importante respecto al código anterior, dado que previamente se realizaba al completo en un único bucle, y para que se enviaran de forma periódica estos mensajes se utilizaba la función `bus.send_periodic`, siendo `bus` definido como:

```
can.interface.Bus(can_interface, bustype='socketcan')
```

Al integrar ROS 2, esta función provocaba un funcionamiento incorrecto del nodo que se expandía al sistema completo, siendo este un error que presentó muchas dificultades a la hora de ser detectado, pues a priori nada apuntaba a que pudiese ser el causante del funcionamiento incorrecto del sistema. La solución fue usar este `create_timer` independiente ejecutándose de forma periódica y usar la función `bus.send` que simplemente envía el mensaje una vez. Para el resto de los mensajes periódicos a enviar se utilizaron también otros `create_timer` dentro del bucle principal, eliminando por completo el uso de la función `bus.send_periodic`.

Centrándonos en el bucle principal, durante cada iteración de este se dan los siguientes procesos:

En primer lugar espera a recibir un mensaje “READY” enviado por el `main_node`. A partir de este momento, tras recibir un mensaje que no fuese de tipo `CON_ERR` o RFID se empieza a mandar mensajes `CON_status` por CAN de forma periódica por el bucle anterior. Estos mensajes se envían con identificador 0x100, y su contenido depende de la variable `datos`, la cual irá cambiando en función del contenido del *buffer* `can_sender_buffer`, el cual almacena los datos recogidos del tópico `/can_sender_topic`.

Por el `can_sender_buffer` se maneja la recepción de diferentes señales, en función del identificador de estas:

- S0 (Coche entra en estado 0)
- S1 (Coche entra en estado 1)
- S2 (Coche entra en estado 2)
- S3 (Coche entra en estado 3)
- P0 (Modo autónomo sale de la pausa)
- P1 (Modo autónomo en pausa)

- G0 (Salta temporizador, luego no se reciben mensajes GOTO-ACK)
- G1 (Se reciben mensajes GOTO-ACK)

Al recibir estas señales se modificarán los datos que se envían de forma recurrente en los mensajes `CON_status`. Al *buffer* anterior también llegan otro tipo de mensajes, que se encargan de iniciar o detener el envío periódico de otros mensajes por el bus CAN:

- R1: Inicia el envío de forma periódica de mensajes RFID cada 0.1 segundos, también inicia un temporizador `RFID_ACK_not_received_timer`, el cual si llega a saltar provoca el envío de un mensaje de error E 34 por el tópico `can_receiver_topic`.
- R0: Finaliza el envío de mensajes RFID periódicos, y se encarga de detener el temporizador, de esta manera no se envía el mensaje de error.
- E1 o W1: Estos mensajes provocan el envío de forma periódica cada 0.1 segundos de los mensajes de error del módulo PC Fusión: `CON_ERR`. Inician también un temporizador `CON_ERR_ACK_not_received_timer`, el cual envía un mensaje de error E 35 por el tópico `can_receiver_topic` en caso de terminarse.
- E0: Finaliza el envío de mensajes `CON_ERR` periódicos, además de detener el temporizador anterior.

En la Figura 23, se muestra el esquema de comunicación de los dos nodos CAN con el resto del sistema:

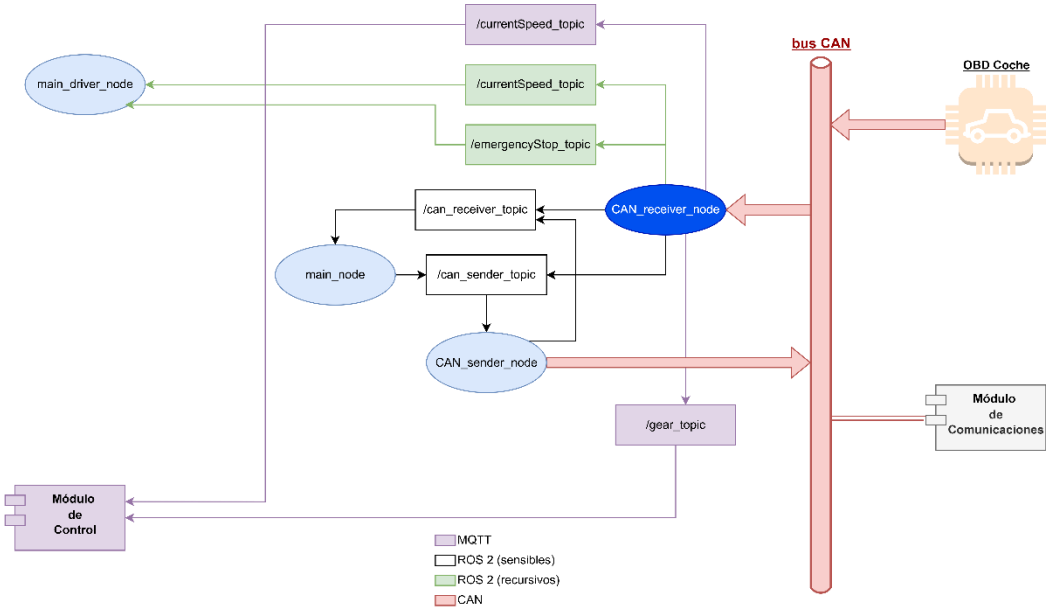

Figura 23. Esquema de comunicaciones CAN_receiver - resto del sistema

3.2.3. Nodos centrales

Todos los nodos anteriormente mencionados se tratan de nodos que se encargan de recibir datos o enviar datos, ya sea de sensores, otros nodos u otros módulos. Sin embargo para poder manejar todos estos datos provenientes de dichos nodos se utilizan 2 nodos centrales:

- Nodo central para el estado autónomo: `main_driver_node`
- Nodo central general: `main_node`

El coche cuenta con 5 estados, el `main_node` maneja en qué estado debe estar el vehículo en función de los datos recibidos, y en base a esto envía diferentes mensajes al resto de nodos.

Cuando el coche entra en el estado autónomo, es el `main_driver_node` quien toma protagonismo, pues se encarga de recibir los datos de percepción del resto de nodos, procesarlos, y finalmente enviarlos mediante MQTT a los actuadores, situados en el módulo de control. También se encargará de lanzar y finalizar la ejecución de varios de estos nodos.

3.2.3.1. Nodo central: `main_node`

En primer lugar se inicializan las diferentes variables, *buffers* y la conexión MQTT. En cuanto a las comunicaciones se cuentan con diferentes suscriptores y publicadores:

- Suscriptores: El nodo recibe mensajes de tipo String provenientes de tres tópicos:
    - El tópico `/can_receiver_topic`, tópico en el que publican mensajes los dos nodos CAN.
    - El tópico `/rfid_topic`, tópico desde el que recibe datos del nodo `RFID_receiver_node`.
    - El tópico `/main_driver_out_topic`, desde donde recibe mensajes del otro nodo central `main_driver_node`.

- Publicadores: Publica mensajes por 5 tópicos diferentes:
    - Tópico `/can_sender_topic`, donde manda mensajes de estado y error al nodo `/CAN_sender_node`.
    - Tópico `/rfidterm_topic`, para enviar mensajes 'TERMINATE' al nodo `RFID_receiver_node` y finalizar así su ejecución.
    - Tópico `/main_driver_in_topic`, para enviar mensajes al otro nodo central `main_driver_node`.
    - Tópico `/can_rx_term`, para finalizar la ejecución del nodo `CAN_receiver_node` enviando mensajes 'TERMINATE'.
    - Tópico MQTT `/actuadores_control`, el cual se utiliza para iniciar el proceso actuador `Gear` dentro del módulo de control.

Se ejecutará un `create_timer` cada 0.05 segundos para el bucle principal del nodo. En el bucle se diferencian los 5 posibles estados en los que puede entrar el vehículo:

- Start-up (estado 0)
- Normal (estado 1)

- Autónomo (estado 2)
- Standby (estado 3)
- Fallo (estado 4)

3.2.3.1.1. Arranque del sistema, Start-up (Estado 0)

En la primera iteración el nodo comienza en el estado 0, se inicia el nodo `CAN_receiver_node` y se espera a recibir un mensaje 'CSR', proveniente de dicho nodo. Una vez recibido se enviará un mensaje 'READY' por el tópico `/can_sender_topic` al nodo `CAN_sender_node`, nodo que como mencionamos antes, ha sido iniciado por el `CAN_receiver_node.` Desde este momento se espera a recibir del nodo creado un mensaje de cambio de estado: 'S' seguido del identificador numérico del estado, o bien de un mensaje de error 'E' seguido del identificador de la causa del error, que cambiaría el estado del nodo actual al estado de fallo.

En la Figura 24, se muestra el diagrama de flujo de este estado.

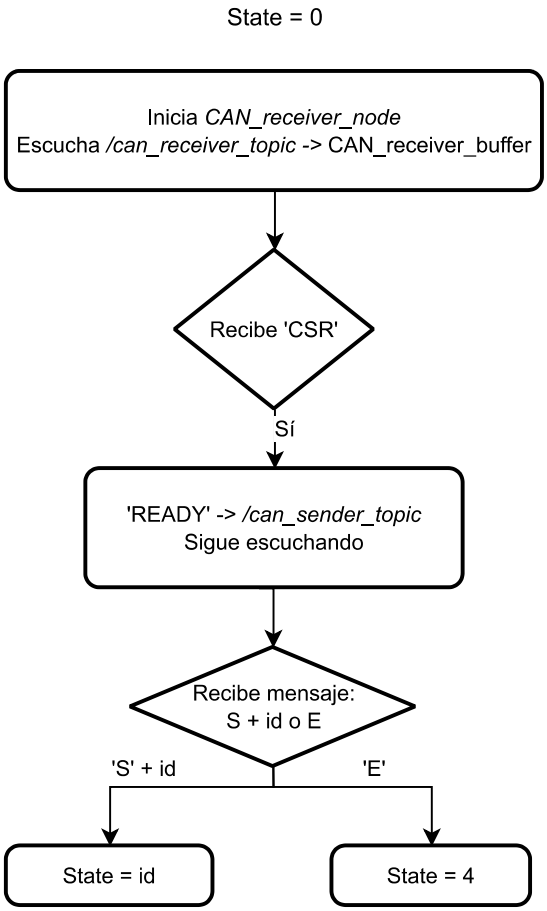

Figura 24. Diagrama de flujo estado de Start-up

3.2.3.1.2. Modo normal (Estado 1)

En este estado, el control del coche es ejercido completamente por parte del conductor, no se realizan tareas en ningún momento ni de control lateral ni longitudinal. Este estado se iniciará la primera vez tras el envío de una señal de CONNECTED a través

de MQTT, tras esto, para volver a este estado será necesario el envío de una señal de AM-OFF. Centrándonos en su funcionamiento, en primer lugar se realizan dos comprobaciones:

- Que el nodo `RFID_receiver_node` esté iniciado, si no es así lo inicia.
- Que el nodo `main_driver_node` esté detenido, si no es así mandará a dicho nodo una señal TERMINATE por el tópico `/main_driver_in_topic`.

Tras esto se envía una señal 'S1' por el tópico `/can_sender_topic` al nodo `CAN_sender_node` para indicar el nuevo estado. A continuación, se lleva a cabo el procesado de los mensajes recibidos en los respectivos *buffers* a través de la función `process_buffers_S1`.

Del *buffer* `can_receiver_buffer` se manejan algunos de los mensajes que se tratan en los capítulos 3.2.2.1. y 3.2.2.2. , enviados por los nodos CAN.

- 'S' + identificador estado: se cambia al estado especificado
- 'E' + identificador causa error: se cambia al estado de fallo
- 'W' + identificador causa error (+ identificador argumento error): se envía por el `/can_sender_topic` hacia el nodo `CAN_sender_node`

Del *buffer* `rfid_buffer` se manejan mensajes de tipo:

- 'E' + identificador causa error: se cambia al estado de fallo
- Datos RFID, estos se envían por el tópico `/can_sender_topic` hacia el nodo `CAN_sender_node`, con el prefijo 'R1 '.

El diagrama de flujo del estado se muestra en la Figura 25.

3.2.3.1.3. Modo autónomo (Estado 2)

Este será el modo que cobra una mayor importancia para el presente proyecto, en este modo es en el cual se dan las tareas de control lateral y longitudinal del vehículo. Para que se inicie dicho modo, será necesario que el vehículo haya entrado en primer lugar en el modo normal, tras esto será necesario el envío de un mensaje MQTT de tipo AM-ON. Tras esto ya se darán las tareas del control lateral del vehículo, y permanecerá estático, para que se mueva de forma longitudinal, se deberá enviar otro mensaje en el que se especificará la velocidad objetivo, este será otro mensaje MQTT del tipo GOTO (velocidad) (1). Tratando su funcionamiento en este nodo central `main_node`, en primer lugar se realizan tres comprobaciones:

- Que se haya iniciado el proceso actuador `Gear` dentro del módulo de control. Si no es así, envía un mensaje 'START_GEAR' por el tópico MQTT /`actuadores_control`.
- Que el nodo `RFID_receiver_node` esté iniciado, si no es así, lo inicia.
- Que el nodo `main_driver_node` esté iniciado, si no es así, lo inicia.

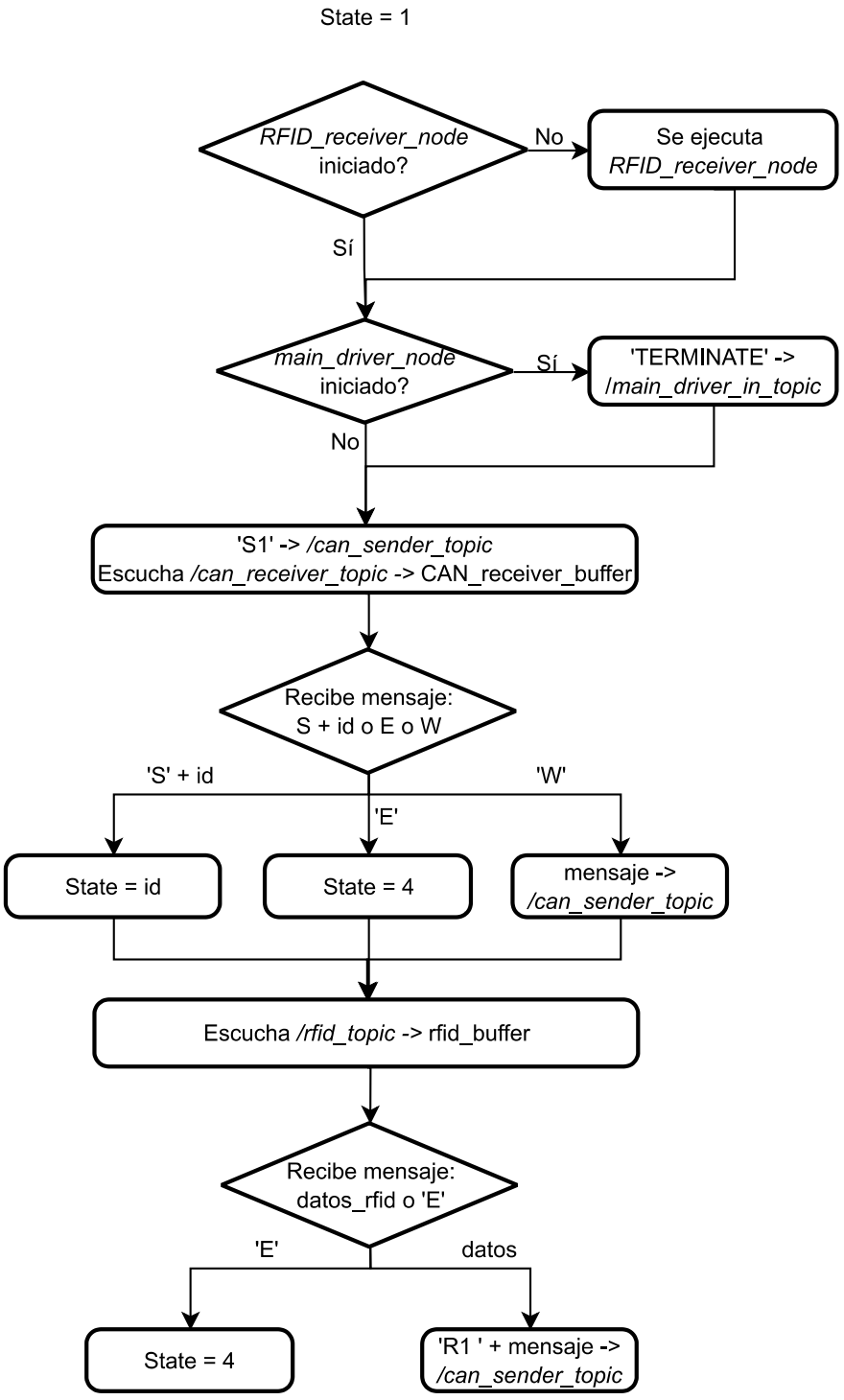

Figura 25. Diagrama de flujo estado normal

Tras esto se envía una señal 'S2' por el tópico `/can_sender_topic` al nodo `CAN_sender_node` para indicar el nuevo estado. A continuación, se lleva a cabo el procesado de los mensajes recibidos en los respectivos *buffers* mediante la función `process_buffers_S2`.

Del buffer `can_receiver_buffer` se manejan los mensajes enviados por los nodos CAN en el Modo normal (Estado 1), explicados en el apartado 3.2.3.1.2.

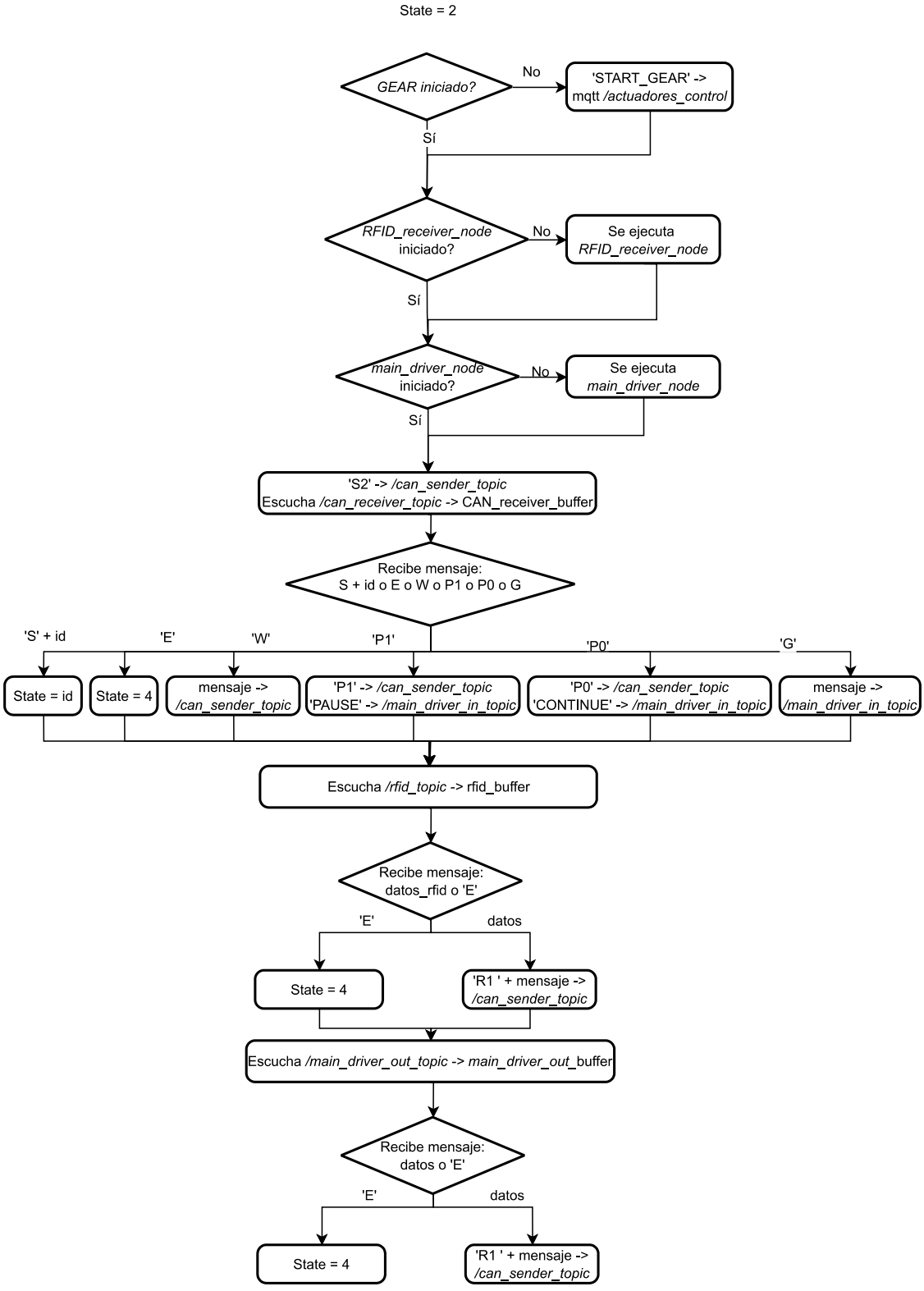

Figura 26. Diagrama de flujo estado autónomo

A mayores de estos, también se manejan los mensajes con identificador:

- 'P1': Envía por el tópico `/can_sender_topic` este mensaje, y por el tópico `/main_driver_in_topic` un mensaje 'PAUSE'.

- ‘P0’: Envía por el tópico `/can_sender_topic` este mensaje, y por el tópico `/main_driver_in_topic` un mensaje ‘CONTINUE’.
- ‘G’: Son mensajes GOTO, estos se reenvían por el tópico `/main_driver_in_topic`.

Del *buffer* `rfid_buffer` se manejan los mensajes RFID de la misma manera que en el Modo normal (Estado 1). Del *buffer* `main_driver_out_buffer` se lleva a cabo el mismo procedimiento que con el `rfid_buffer`.

El diagrama de flujo del estado se muestra en la Figura 26.

3.2.3.1.4. Modo Standby (Estado 3)

En este estado se finaliza la ejecución de los nodos `main_driver_node` y `RFID_receiver_node` mediante el envío de mensajes ‘TERMINATE’ a los tópicos `/main_driver_in_topic` y `/rfidterm_topic`. A continuación envía un mensaje del nuevo estado ‘S3’ por el tópico `/can_sender_topic`.

Después de esto se queda a la escucha de un posible mensaje de error ‘E 33’ del *buffer* `can_receiver_buffer`. Si se recibe este mensaje, el nodo mandará una señal ‘TERMINATE’ por los tópicos `/can_receiver_topic` y `/can_sender_topic`, para finalizar la ejecución de los nodos CAN. Tras esto cambia el estado del coche al estado de Start-Up (estado 0).

En la Figura 27 se representa el diagrama de flujo del estado.

3.2.3.1.5. Modo de fallo (Estado 4)

Debido al control de fallos implementado en el código, se da este modo de fallo. La inicialización de este modo puede darse por múltiples causas, como la finalización de temporizadores que representan fallos de conexión con el módulo de comunicaciones, fallos internos reportados desde dicho módulo, o fallos programados en casos de emergencia, como podrían ser pulsar el freno o realizar un cambio de marcha cuando el vehículo se encuentra en el estado autónomo. Para una visión más detallada, dentro del TFG de Ignacio Royuela [19] se puede encontrar en el anexo II una tabla completa con todos los fallos contemplados en el sistema.

En el estado de fallo, en primer lugar se envía un mensaje de error ‘E1’ junto a la causa de este y al atributo de este (en caso de existir) hacia el nodo `CAN_sender_node` mediante el tópico `/can_sender_topic`, y acto seguido de la misma manera se le envía una señal ‘S3’ para representar el cambio de estado.

Después se finaliza la ejecución de todos los nodos, enviando señales ‘TERMINATE’ por los tópicos: `/main_driver_in_topic`, `/rfidterm_topic`, `/can_receiver_topic` y `/can_sender_topic.`

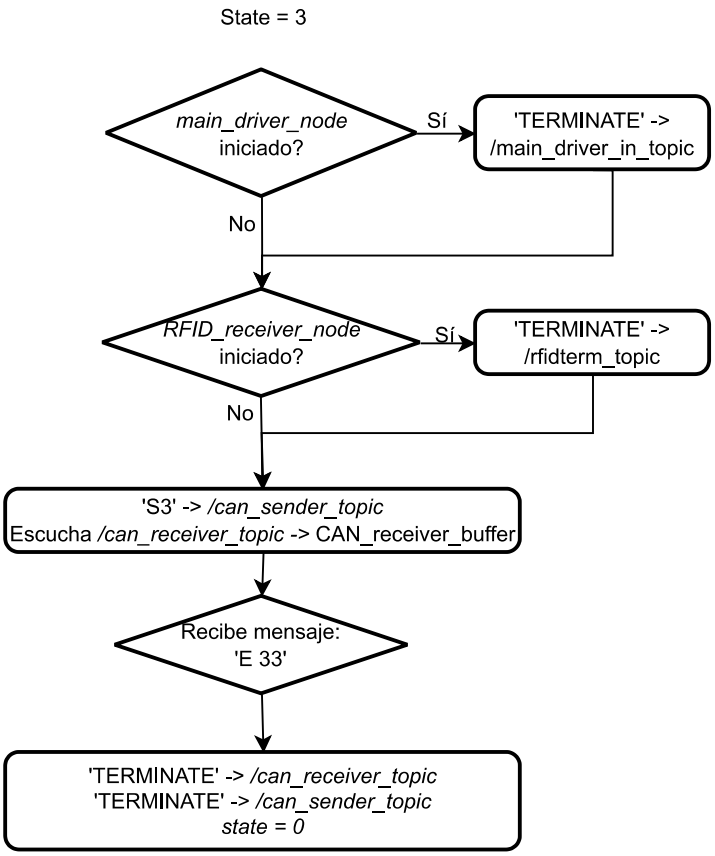

Figura 27. Diagrama de flujo estado Standby

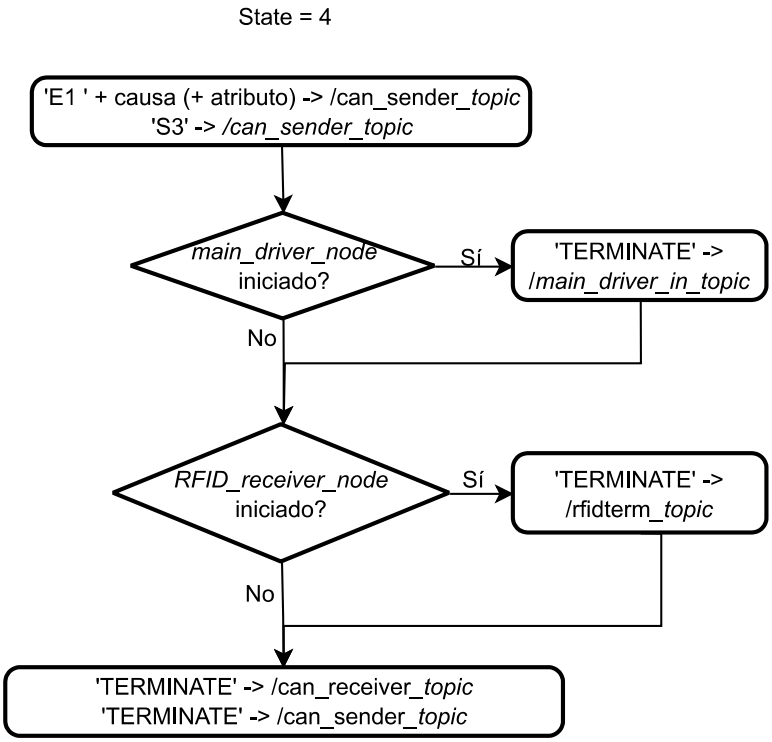

Figura 28. Diagrama de flujo estado de fallo

Al enviar esta señal al nodo `main_driver_node`, este se encarga de finalizar la ejecución del resto de nodos sensores (`camera_node` y `ultrasonic_node`), al igual que de los nodos actuadores, alojados en el módulo de control.

Se muestra el diagrama de flujo del estado de fallo en la Figura 28.

Finalmente, en la Figura 29 se muestra un esquema simplificado de las comunicaciones del nodo central *`main_node`* con el resto de los nodos del sistema. Además, uniendo los diagramas de los diferentes estados, se obtiene el diagrama de flujo final del mismo, que se muestra en la Figura 30.

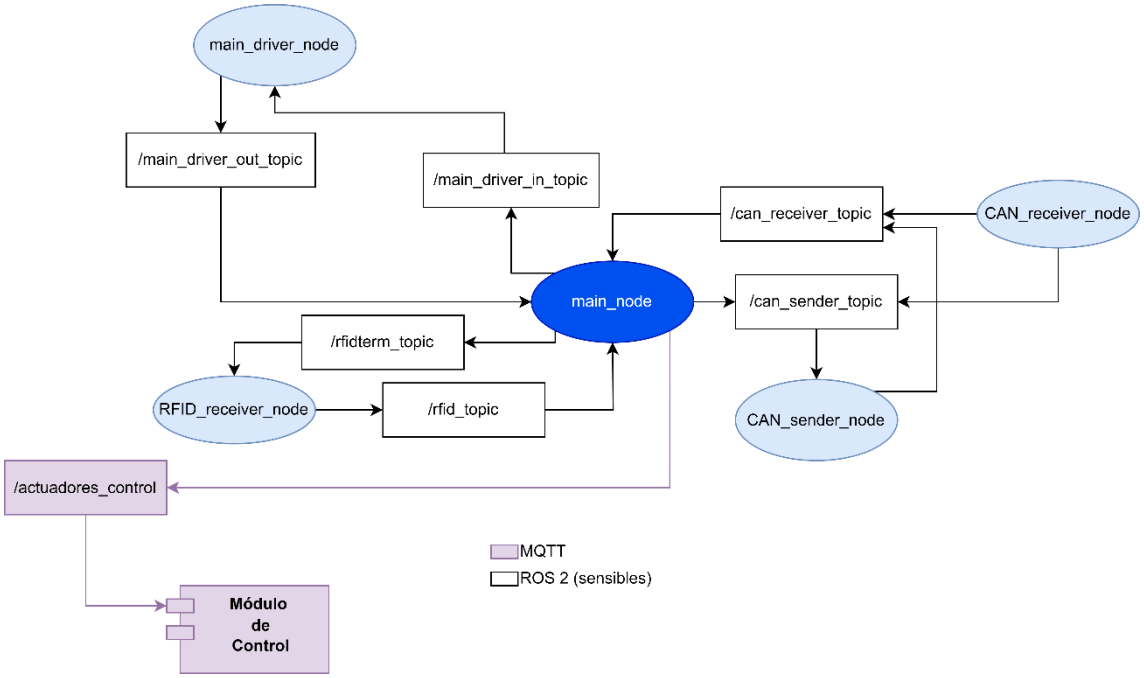

Figura 29. Esquema de comunicación `main_node` – resto del sistema

3.2.3.2. Nodo central modo autónomo: `main_driver_node`

Este es el nodo central encargado de las tareas de planificación de la ruta, es decir de llevar a cabo el procesado de los datos recibidos por los nodos sensores, y de su posterior envío del dato ya tratado hacia los nodos actuadores. De esta manera, representa uno de los tres bloques principales de la división planteada en la sección 2.2. que buscaba acercarse lo máximo posible a la arquitectura ideal de coche autónomo.

Centrándonos en su funcionamiento lógico, en primer lugar se inicializan las diferentes variables, el PID (Proporcional Integral Derivativo) para el control lateral y longitudinal, *buffers* y la conexión MQTT. También se inicia la ejecución de varios nodos:

- De percepción: `camera_node` y `ultrasonic_node`.
- De actuación: Se envían señales 'START_THROTTLE' y 'START_EPOS' hacia el módulo de control mediante el tópico MQTT `/actuadores_control`, para iniciar los procesos del control del acelerador y la Epos (control lateral).

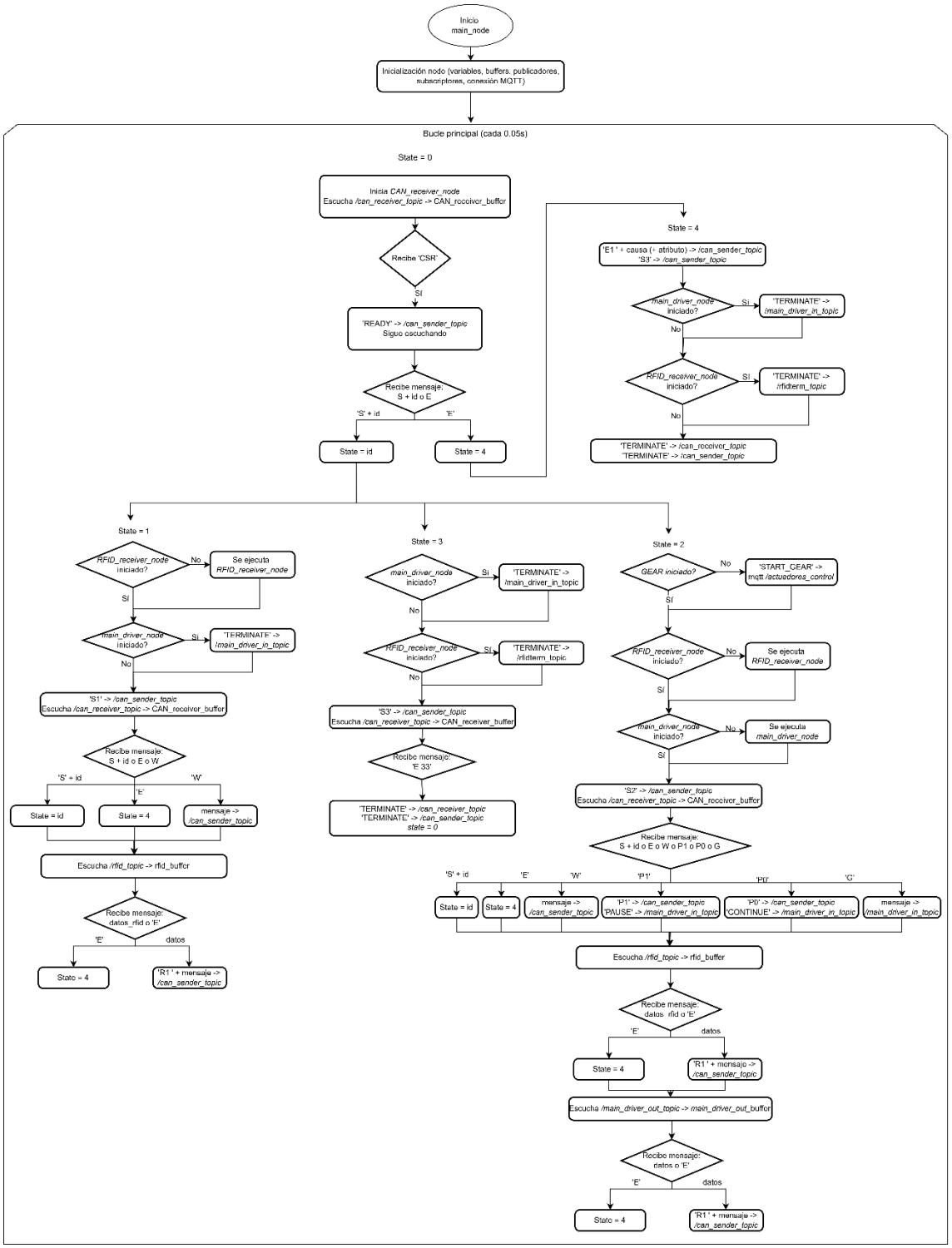

Figura 30. Diagrama de flujo nodo `main_node`

Cabe destacar de nuevo que actualmente no se realiza un control de errores en los actuadores, lo cual representa una limitación del sistema. Se considera recomendable implementar un mecanismo de gestión de errores en futuras versiones.

En cuanto a las comunicaciones se cuentan con diferentes suscriptores y publicadores:

- Suscriptores: El nodo recibe diferentes mensajes por los tópicos:

- Tópico `/camera_processed`, mensajes de tipo Float32MultiArray con los datos procesados de la cámara
- Tópico `/ultrasonicData_topic`, mensajes Int32MultiArray con los datos del sensor de ultrasonidos.
- Tópico `/currentSpeed_topic`, recibe la información de la velocidad actual del vehículo en formato Float32.
- Tópico `/emergencyStop_topic`, recibe datos del `CAN_receiver_node` al igual que el anterior tópico, esta vez en formato Int32 sobre la señal de emergencia.
- Tópico `/main_driver_in_topic`, datos en tipo String del *`main_node`*.

- Publicadores: Publica mensajes por 7 tópicos diferentes:

  Tópicos ROS 2:

  - Tópico `/camera_topic`, manda en formato String mensajes 'TERMINATE' o la variable bif al `camera_node`.
  - Tópico `/sensorultra_topic`, para enviar mensajes 'TERMINATE' al nodo `ultrasonic_node` y finalizar así su ejecución.
  - Tópico `/main_driver_out_topic`, para enviar mensajes de error tipo String al otro nodo central `main_node`.

  Tópicos MQTT (hacia el módulo de control):

  - Tópico `/targetSpeed_topic`, para enviar el valor de la velocidad objetivo al acelerador.
  - Tópico `/deltaTarget_topic`, para enviar el ángulo de giro objetivo a la Epos.
  - Tópico `/throttle_topic`, para enviar señales 'TERMINATE' al actuador del acelerador.
  - Tópico MQTT `/actuadores_control`, mencionado anteriormente su uso.

El funcionamiento del nodo se basa en un hilo `threading.Thread`, el cual ejecuta el bucle principal del nodo de forma continua, a la frecuencia máxima proporcionada por la CPU. Se ha establecido este hilo en lugar de un `create_timer` estándar de ROS debido a que este nodo será el nodo que gestione una mayor cantidad de datos provenientes de sensores a diferentes tasas, por lo tanto nos interesa que tenga la mayor capacidad de cómputo posible.

En el bucle se tratan los diferentes datos de percepción y del estado actual de vehículo obtenidos de los tópicos suscritos mencionados. En los siguientes subapartados se explica el procesado realizado sobre los diferentes datos que recibe el nodo.

3.2.3.2.1. Datos de ultrasonidos

Se reciben por el tópico `/ultrasonicData` un vector con 3 valores del sensor de ultrasonidos. A partir de estos se calcula la velocidad objetivo del vehículo, para ello se utiliza el PID longitudinal. Esta es la línea de código que calcula la `targetSpeed` en base a dichos datos:

```
self.targetSpeed=-
float((round(self.pidLongitudinal(min(self.UltrasonicData)),1)))
```

Después se enviará mediante MQTT por el tópico `/targetSpeed_topic` hacia el actuador del control longitudinal del vehículo: `Throttle_Controller`.

Como se explicó en secciones previas, el *hardware* de los sensores de ultrasonidos no funciona correctamente, y por lo tanto ha sido necesario adaptar ciertas líneas del código hasta que sea sustituido o arreglado en un proyecto futuro.

La adaptación se ha basado en la creación de un `create_timer` que se ejecuta cada 0.1 segundos, este publica la `TargetSpeed` por los mismos tópicos que se realizaba anteriormente, sin embargo la fórmula de calcular su valor es lo que cambia:

```
self.targetSpeed = -float((round(self.pidLongitudinal(500), 1)))
```

Como se ve, ahora dejan de intervenir los ultrasonidos, y se le asigna un valor fijo de 500, este es un valor grande que indicaría al PID lo equivalente a que los ultrasonidos no están detectando un objeto cercano y por tanto no se debería detener el vehículo.

3.2.3.2.2. Datos de la cámara y velocidad actual

Por el tópico `/camera_processed` se recibe un vector con los 3 valores procesados de la cámara: `anguloRuta`, `offsetRuta` y `error`.

Primero se manejan fallos usando el dato `error`. Si este vale 1, no se toman en cuenta los otros dos valores, se usan los de la anterior iteración, esto solo puede suceder diez veces, la siguiente vez se detendrá el vehículo, enviando velocidad objetivo nula. Originalmente, el número de fallos que se admitían era solo de dos, pero este número de veces se ha podido incrementar debido al aumento de la frecuencia de envío de *frames* de la cámara conseguido con el nuevo PC.

A continuación, se aplica el cálculo de Stanley para obtener el ángulo objetivo que debe girar el vehículo. Para ello se necesitan los otros dos datos obtenidos de la cámara (`anguloRuta` y `offsetRuta`) junto a la velocidad actual del vehículo, obtenida desde el tópico `/currentSpeed_topic`.

Estas son las líneas para obtener el ángulo objetivo `delta_target`:

```
self.velocidad_actual = max(self.currentSpeed_buffer.get(), 1)
self.delta_target = self.aac.calculo_stanley(self.anguloRuta, self.offsetRuta,
self.velocidad_actual, 1.2)
self.delta_target = max(min(self.delta_target, 0.7), -0.7)
```

En cuanto a las constantes que se ven en las anteriores líneas de código, son respectivamente:

- la constante de Stanley (1.2), la cual fue elegida en trabajos previos y por tanto no se ha modificado
- Los valores 0.7 y -0.7 representan el máximo ángulo de giro que podría dar el volante desde la posición central de este. Introduciendo estos límites se busca que la Epos nunca mande al motor Maxon sobrepasar las posiciones máximas de giro y evitando así ocasionar problemas a nivel de *hardware* en los actuadores.

Tras esto se envía la `delta_target` mediante MQTT por el tópico `/deltaTarget_topic` hacia el actuador del control lateral del vehículo: `EPOS_Controller`.

3.2.3.2.3. Datos de parada de emergencia

En el nodo `CAN_receiver_node` se obtenía la variable `EmergencyStop` en base al valor obtenido de las marchas y el freno. Se recibe esta variable desde el tópico `/EmergencyStop_topic`, y en caso de valer 1, se enviará una señal de error 'E 141' por el tópico `/main_driver_out_topic` hacia el nodo central `main_node`.

3.2.3.2.4. Datos del `main_node`

Se reciben 4 tipos diferentes de datos desde el tópico `/main_driver_in_topic` provenientes del `main_node`, que se listan a continuación.

- Mensajes 'TERMINATE', al recibir uno de estos, aparte de detener el propio nodo, se enviará este mismo mensaje a los nodos: `camera_node` y `ultrasonic_node` por medio de los tópicos: `/camera_topic` y `/sensorultra_topic`, y mediante el tópico MQTT `/throttle_topic` al actuador `Throttle_Controller`, con el objetivo de detenerlos también.
- Mensajes GOTO, con los cuales:
    - Se obtiene la variable `bif`, la cual se envía al `camera_node` mediante el tópico `/camera_topic`, esta es necesaria en dicho nodo para obtener el *offset* y el ángulo que debe girar el coche.
    - Se obtiene la variable `superTargetSpeed`, esta se utiliza para modificar el PID longitudinal, el cual como ya se ha mencionado, se aplica a los datos de ultrasonidos para obtener la velocidad objetivo.

Se reciben también mensajes 'PAUSE' y 'CONTINUE'. Al recibir un mensaje de pausa, se comienza a enviar una velocidad objetivo de 0 por el tópico MQTT `/targetSpeed_topic`. Una vez se ha recibido uno de estos, la única manera de salir de esta situación es al recibir un mensaje de continuación, volviendo a enviar la velocidad objetivo en función de los datos de percepción.

Se muestra en la Figura 31 el diagrama de flujo del nodo.

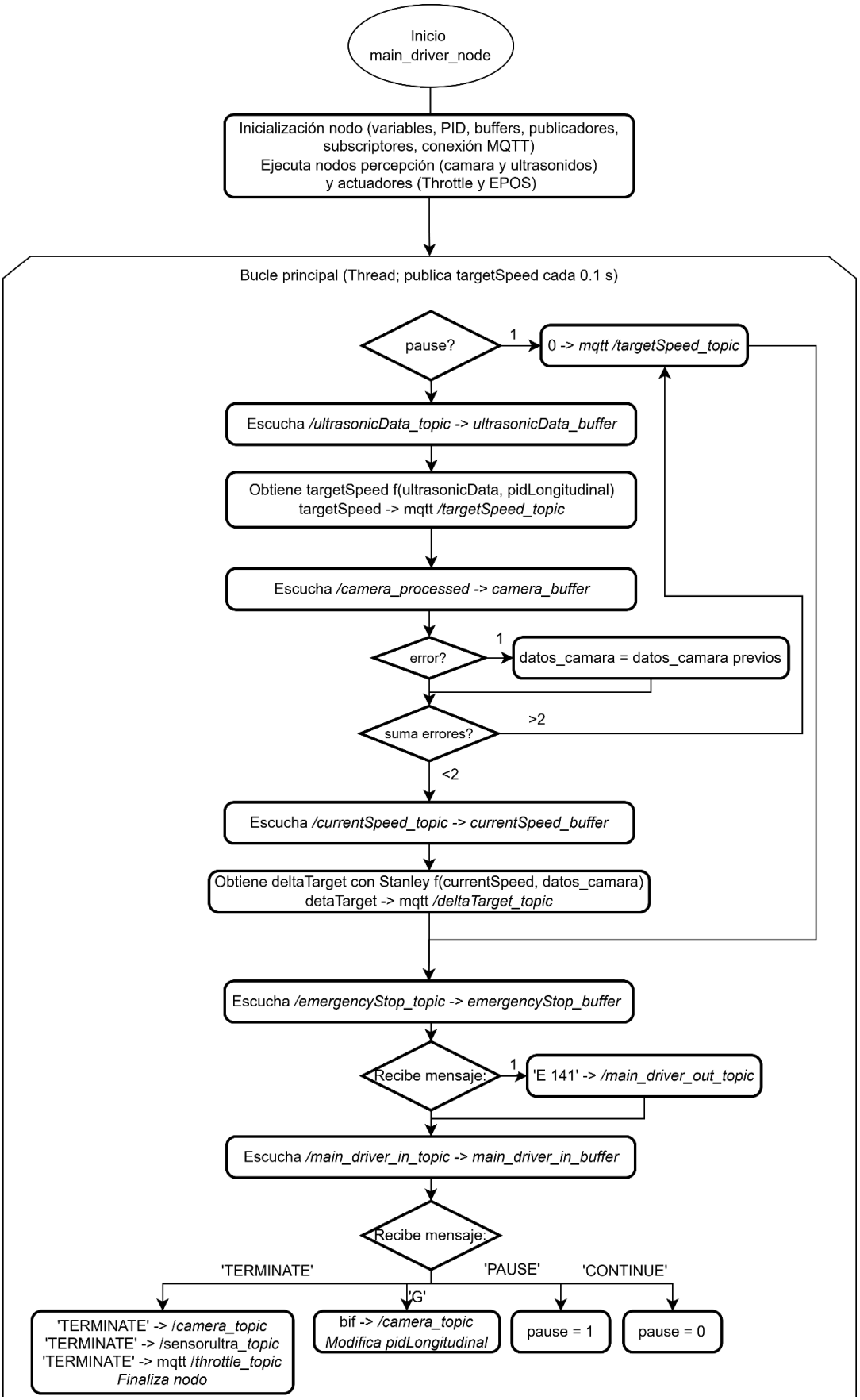

Figura 31. Diagrama de flujo `main_driver_node`

En la Figura 32 se muestra un esquema simplificado de las comunicaciones del nodo central `main_driver_node` con el resto de los nodos del sistema en la figura.

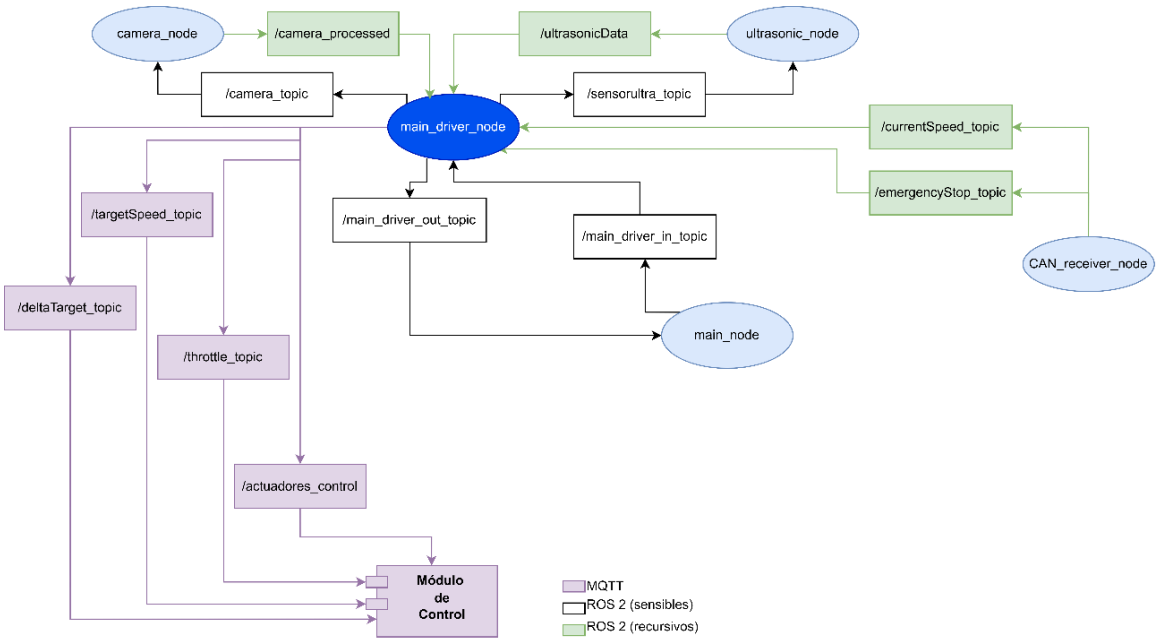

Figura 32. Esquema de comunicación `main_driver_node` – resto del sistema

Finalmente, en la Figura 33 se muestra el esquema general de conexiones entre los diferentes nodos ROS 2 del módulo PC Fusión, y las conexiones con los otros módulos del sistema.

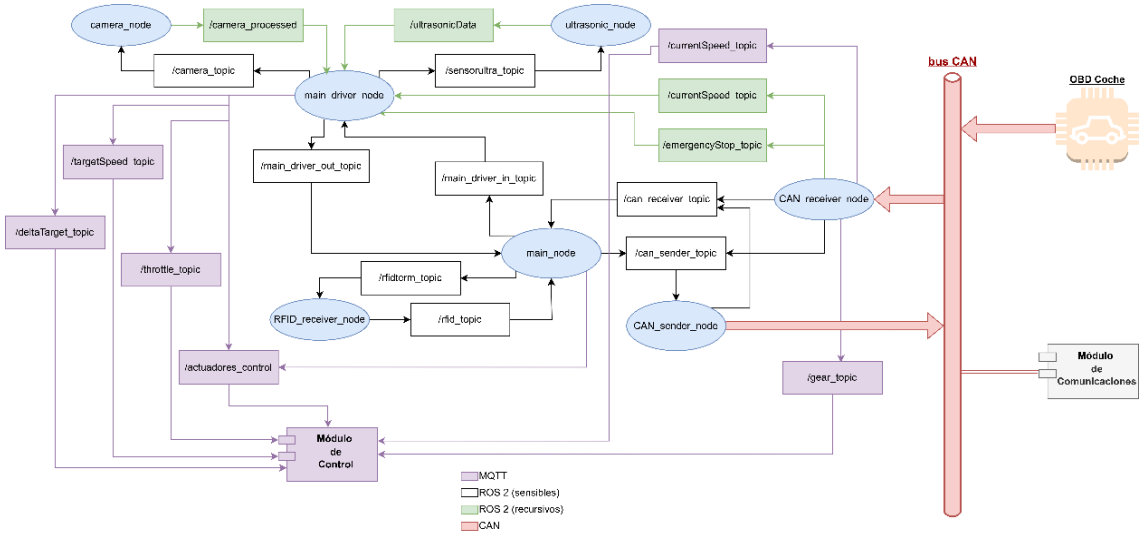

Figura 33. Esquema general de comunicaciones

4. Nuevo módulo de control, pruebas y troubleshooting

Actualmente el módulo de control se encarga únicamente de las tareas de actuación, por lo tanto los datos que reciben ya están procesados tras haber pasado las etapas de percepción y planificación.

Este módulo es un miniordenador HummingBoard del año 2019. Se trata por lo tanto de un *hardware* bastante antiguo que desde el principio tenía capacidades de cómputo muy limitadas. Por lo tanto, para su actualización no es posible utilizar ROS 2. Por el contrario, se seguirá utilizando módulos *software* programados en Python. Este código tendrá diferentes módulos actuadores que en base a los datos recibidos por MQTT mandan las correspondientes órdenes a los Arduinos que controlan el acelerador, las marchas y la Epos (que maneja la dirección del vehículo).

En primer lugar, se explicará la comunicación MQTT entre el módulo de control y el módulo PC Fusión.

4.1. Comunicación mediante MQTT con el módulo PC Fusión

Anteriormente, solo contábamos con dos clientes MQTT que se comunicaban entre sí, el módulo de comunicaciones y el smartphone que se encargaba del control remoto del vehículo. Para ello se necesitaba un servidor MQTT mosquitto, el cual se encontraba en un ordenador externo al vehículo.

Actualmente se han hecho modificaciones en este esquema de comunicaciones, pues habrá dos comunicaciones MQTT:

- Módulo de comunicaciones – Smartphone: Será la misma comunicación que había anteriormente para el control remoto del vehículo.
- Módulo PC Fusión – Módulo de control: Comunicación unilateral, el PC Fusión envía los datos del control longitudinal y lateral ya procesados para que los apliquen los actuadores. Sustituye a la antigua comunicación mediante colas entre los diferentes multiprocesos que se ejecutaban dentro del módulo de control.

Como se mencionó en la sección 2.5.1. con el fin de acelerar el proceso de pruebas en el vehículo, se ha integrado completamente la comunicación MQTT en el propio sistema del vehículo. Para ello, se ha configurado un *broker* Mosquitto directamente en el PC Fusión, siendo este el único *broker* usado durante esta etapa. No obstante, esta solución está pensada únicamente para la fase de pruebas, ya que en el proyecto final,

donde múltiples vehículos estarían operando simultáneamente y un servidor externo gestionaría el sistema global, este enfoque no sería adecuado.

Ya contando con el servidor MQTT, es necesario crear clientes MQTT en los módulos PC Fusión y de control. Dado que el módulo PC Fusión únicamente transmite datos y el módulo de control se limita a recibirlos, se deben implementar las líneas de código correspondientes para establecer la comunicación según el rol de cada módulo. Se muestran en las subsecciones 4.1.1. y 4.1.2. .

4.1.1. Módulo PC Fusión

Con el objetivo de establecer la comunicación entre los distintos nodos del sistema y el módulo de control, dentro del módulo PC Fusión, se ha implementado un cliente MQTT en cada uno de los nodos desarrollados con ROS 2 que requieren enviar información. Para ello, se ha utilizado la biblioteca de Python `paho.mqtt.client`, que permite una integración sencilla y eficiente del protocolo MQTT en aplicaciones Python. Esta biblioteca se importa con la siguiente instrucción:

```
import paho.mqtt.client as mqtt
```

La configuración de la conexión MQTT se lleva a cabo en varias etapas. En primer lugar, se crea una instancia del cliente MQTT:

```
self.mqtt_client = mqtt.Client()
```

Posteriormente, se configuran las credenciales de acceso y la dirección del servidor MQTT, estableciendo el usuario, la contraseña, la dirección IP del servidor y el puerto utilizado (por defecto, el puerto estándar MQTT es el 1883):

```
self.mqtt_client.username_pw_set(username=MQTT_user,password=MQTT_pass)
self.mqtt_client.connect(server_ip, 1883, 60)
```

Una vez establecida la conexión, el cliente está en condiciones de publicar mensajes en los distintos tópicos definidos. Por ejemplo, si se desea transmitir un mensaje con la velocidad objetivo almacenada en la variable `targetSpeed`, se realiza mediante:

```
self.mqtt_client.publish('/targetSpeed_topic', self.targetSpeed)
```

Este enfoque resulta especialmente adecuado ya que mantiene una arquitectura similar a la que emplea ROS 2, basada también en el intercambio de mensajes mediante tópicos. Así, se logra una transición transparente entre la comunicación interna en ROS 2 y la comunicación externa a través del protocolo MQTT. Cada nodo puede actuar como publicador o suscriptor de un determinado tópico, permitiendo una comunicación escalable, desacoplada y eficiente entre múltiples componentes del sistema distribuido.

4.1.2. Módulo de control

Dado que este módulo cuenta con un único *script* principal encargado de la gestión de mensajes entrantes, la creación del cliente MQTT se realiza una sola vez. Para ello, se

configura inicialmente la conexión mediante la creación del cliente con compatibilidad explícita con la versión de la API utilizada en la distribución de Linux empleada:

```
client_MQTT = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
```

Posteriormente, se ajustan ciertos parámetros de retardo para reconexiones automáticas en caso de pérdida de conexión con el servidor:

```
client_MQTT.reconnect_delay_set(min_delay=1, max_delay=1)
```

También se especifican las credenciales de acceso, las cuales deben coincidir con las establecidas en el servidor MQTT:

```
client_MQTT.username_pw_set(username=MQTT_user,password=MQTT_pass)
```

A continuación, se definen las funciones de *callback* que se ejecutarán en los distintos eventos del ciclo de vida del cliente MQTT, como son la conexión al *broker*, la recepción de un mensaje y la desconexión:

```
client_MQTT.on_connect = on_connect
client_MQTT.on_message = on_message
client_MQTT.on_disconnect = on_disconnect
```

Una vez establecida la conexión con el *broker* MQTT, el cliente se suscribe a los diferentes tópicos por los cuales se reciben mensajes desde el módulo PC Fusión. Por ejemplo, la suscripción al tópico que transporta la velocidad objetivo se realiza con:

```
client.subscribe("/targetSpeed_topic")
```

Cuando se recibe un mensaje, este se procesa utilizando la función `payload.decode()`, y se filtra en función del tópico de origen. Los datos se almacenan en variables específicas según el caso:

- `/targetSpeed_topic`: velocidad objetivo, se almacena en la variable `TargetSpeed`.
- `/gear_topic`: marchas, se almacena en `gear`.
- `/currentSpeed_topic`: velocidad actual, almacenada en `CurrentSpeed`.
- `/deltaTarget_topic`: ángulo de giro objetivo, se almacena en la variable `delta_target`.

En el caso particular de recibir un mensaje del tópico `/throttle_topic`, se almacenará en el *buffer* de entrada: `q_Throttle_Controller`.

Además, si se recibe un mensaje en el tópico `/actuadores_control`, se invoca la función `start_process`, la cual se encarga de lanzar el proceso concreto especificado en el mensaje. A diferencia de los nodos ROS 2, en este módulo se lleva a cabo la ejecución de subprocesos mediante la librería `multiprocessing`, siguiendo el mismo enfoque que en el módulo de comunicaciones.

De esta manera, por este tópico se puede esperar la recepción de mensajes de tipo:

- 'START_THROTTLE': ejecuta el proceso `Throttle_Controller`.
- 'START_GEAR': ejecuta el proceso `set_Gear`.
- 'START_EPOS': ejecuta el proceso `EPOS_Controller`.

Este enfoque en el intercambio de información mediante tópicos MQTT posibilita la activación dinámica de los distintos controladores del sistema, preservando una arquitectura modular y escalable.

4.2. Nodos actuadores

Una vez definida ya la conexión MQTT, se explica el funcionamiento de los diferentes nodos actuadores (procesos `multiprocessing`) presentes en el módulo de control:

- Control de las marchas: `set_Gear`
- Control del acelerador: `Throttle_Controller`
- Control de la dirección: `EPOS_Controller`

En el caso de los dos primeros nodos, los actuadores correspondientes se conectan al módulo de control mediante cables USB. De forma análoga a lo descrito previamente para los sensores en la sección 3.2.1.2. , relativa al PC Fusión, es necesario que el módulo de control identifique qué puerto USB corresponde a cada actuador.

Para ello, se ha adaptado el *script* mencionado en dicha sección con el objetivo de detectar exclusivamente los actuadores conectados y asociarlos a distintas variables de entorno. En concreto, se definen las siguientes variables:

- `Marchas_USB`
- `Acelerador_USB`

Estas variables de entorno permiten identificar los puertos USB correspondientes a cada actuador. Para acceder a ellas desde el programa principal, se hace uso de las funciones proporcionadas por la librería `os` de Python como es `os.environ()` [5].

Cabe destacar como con la implementación de estos nodos actuadores se sigue la estructura ideal de coche autónomo planteada en la sección 2.2. , pues todos estos nodos simplemente recibirán datos ya procesados como argumentos y, en base a estos datos, comunicarán al microcontrolador Arduino correspondiente del control longitudinal o de marchas o bien a la controladora Epos4 las señales a enviar al actuador *hardware*.

4.2.1. Nodo set_Gear

Este nodo se encarga del control de las marchas del vehículo. Su función principal es recibir, como argumento de entrada, la marcha que se desea introducir y comunicar

dicha orden al microcontrolador Arduino. Este último es el responsable de habilitar las señales necesarias para ejecutar el cambio de marcha [5].

En la implementación actual, su funcionalidad está limitada a introducir la marcha 'D' (Drive) al iniciar el modo autónomo del vehículo. Este proceso se activa mediante el envío de la señal 'START_GEAR' desde el PC Fusión a través del protocolo MQTT. Al recibir esta señal, el módulo de control mediante la función `start_processes`, que se encarga de lanzar los diferentes multiprocesos, ejecuta este nodo, pasándole como argumento la marcha 'D'. Tras esto se vuelve a habilitar el control manual de las marchas del vehículo, permitiendo que si el conductor en una situación de emergencia quisiera intervenir, pulsando uno de los botones de marchas, el vehículo responderá aun estando en este modo autónomo.

No se profundizará en los aspectos técnicos de este nodo, ya que no ha sido modificado respecto a su versión anterior. Para una descripción más detallada de su funcionamiento interno, se remite al Trabajo Fin de Máster de David Manso [5].

4.2.2. Nodo Throttle_Controller

Este nodo se encarga del control del acelerador del vehículo. Su funcionamiento se basa en habilitar dicho control, gestionado a través de un microcontrolador Arduino, y en modificar un controlador PID en función de la velocidad objetivo y la velocidad actual, las cuales se reciben desde distintos nodos del PC Fusión. A partir de este ajuste se obtiene un valor de aceleración que será transmitido al Arduino.

La ejecución del nodo se inicia al recibir un mensaje 'START_THROTTLE', enviado mediante MQTT desde el PC Fusión. Al recibir esta señal, la función `start_processes` lanza el nodo, pasándole como argumento la velocidad longitudinal objetivo (`TargetSpeed`), la cual se obtiene del nodo `main_driver_node` del PC Fusión a través del tópico `/targetSpeed_topic`. Además, se recibe también la velocidad actual (`CurrentSpeed`) desde el tópico `currentSpeed_topic`.

El funcionamiento del nodo se basa en la creación de un controlador PID con los siguientes parámetros (seleccionados y justificados en proyectos previos al presente):

```
pid = PID(20, 6, 0, output_limits=(0, 50))
```

A este controlador se le asigna inicialmente como `setpoint` el valor de la velocidad objetivo recibida:

```
pid.setpoint = TargetSpeed.value
```

A partir de ese momento, el nodo entra en un bucle con una frecuencia de ejecución de 0.1 segundos. En cada iteración, se actualiza el `setpoint` del PID con el valor más reciente de la velocidad objetivo mediante la misma línea anterior.

Seguidamente, se calcula la variable `throttle`, que representa la señal de aceleración a enviar al Arduino. Este valor se obtiene como la salida del PID al introducir

la velocidad actual del vehículo, y se redondea posteriormente. La instrucción encargada de este proceso es:

```
throttle = round(pid(CurrentSpeed.value))
```

En caso de finalizarse la ejecución del nodo `main_driver_node`, este enviará una señal de TERMINATE al presente nodo, a través del tópico MQTT `/throttle_topic`, provocando su terminación inmediata al recibir dicha señal.

No se entrará en más detalle del funcionamiento de este nodo, ya que, al igual que en el caso descrito en la Sección 4.2.1. , no se han realizado modificaciones respecto a la versión anterior.

4.2.3. Nodo EPOS_Controller

Este nodo es el encargado de comunicar a la controladora EPOS4 tanto la cantidad de pasos que debe girar el motor encargado de la dirección como la velocidad a la que debe hacerlo.

La ejecución del nodo se inicia al recibir, a través del tópico `/actuadores_control`, una señal 'START_EPOS'. En ese momento, la función `start_processes` lanza la ejecución del presente nodo.

Para establecer la comunicación entre el nodo y la controladora EPOS4 se continúa utilizando la metodología implementada en trabajos previos, basada en el uso de memoria compartida. Esta técnica permite establecer la comunicación entre el programa en Python (este nodo) y el programa en C++ que gestiona directamente la controladora EPOS4. Entre ambos programas se intercambian tres datos:

- Del programa C++ al programa Python:
    - La posición actual del motor respecto del centro, medida en pasos.
- Del programa Python al programa C++:
    - La posición objetivo del motor respecto al centro, también medida en pasos.
    - Las RPM (revoluciones por minuto) objetivo a las que deberá girar el motor.

Para llevar a cabo esta comunicación mediante memoria compartida, se siguen los siguientes pasos:

- Inicialización de la memoria compartida para permitir la escritura, mediante:

```
ID1, ID2 = motor.keyRPM(), motor.keyDEG()
```

- Envío de una señal inicial al motor (solo de 10 pasos) usando:

```
motor.rpm_launcher(10, ID1)
```

- Ejecución del programa que maneja la EPOS: `HelloEposCmd_byManso`.

Una vez establecida la comunicación, se procede a inicializar un controlador PID para el control lateral con los siguientes parámetros:

```
pidLateral = PID(5000, 0, 12500, output_limits=(-3000, 3000))
pidLateral.setpoint = 0
```

A continuación, el nodo entra en un bucle con una frecuencia de ejecución de 1/30 segundos, correspondiente a la tasa de envío de imágenes de la cámara. En cada iteración del bucle:

Se calcula la posición objetivo del motor en pasos utilizando la función `delta2steps`, la cual toma como entrada el ángulo de giro objetivo (`delta_target`) recibido a través del tópico `/deltaTarget_topic`.

Tras esto, se obtiene, mediante la memoria compartida, la posición actual del motor (`steps_current`). Este valor se convierte nuevamente en ángulo (`delta_current`) utilizando la función `steps2delta`, obteniéndose así la delta asociada a estos pasos, que será por tanto la delta actual.

A continuación, se calcula la diferencia entre el ángulo actual y el ángulo objetivo (`delta_error`), y a partir de esta diferencia se obtiene el valor de RPM objetivo (`rpm_target`) usando el PID lateral definido previamente:

```
rpm_target = abs(int(round(pidLateral(delta_error, 1))))
```

Tras esto se volverá a utilizar la memoria compartida, en este caso para enviar a la controladora las dos variables calculadas, que le indicarán cuántos pasos debe girar el motor, y a qué velocidad (RPMs) debe hacerlo, mediante:

```
motor.rpm_launcher(round(rpm_target), ID1)
motor.deg_launcher(steps_target, ID2)
```

De manera similar a como sucedía para el caso del nodo `Throttle_Controller`, si el nodo `main_driver_node` envía una señal TERMINATE, este nodo finalizará su ejecución al recibirla. Además, para cerrar correctamente la conexión con la EPOS4, se envía una señal de 8000 RPM, valor que la controladora interpreta como fin de operación.

4.3. Pruebas del vehículo, comparación

La evolución del presente proyecto siguió una serie de pasos en términos del funcionamiento del vehículo.

- En primer lugar, se consiguió que los diferentes módulos se comunicarán de forma directa sin estar conectados ni al vehículo ni a los diferentes sensores y actuadores (salvo la cámara), implementando las partes que faltaban mediante simulación.
- Una vez se consiguió un correcto funcionamiento por simulación, se pasó a probar el funcionamiento completo dentro del coche con todo el *hardware* real. Tras solucionar los diferentes problemas que fueron surgiendo, se consiguió una comunicación estable.
- En este punto se comenzó a probar uno a uno la integración de los diferentes sensores en el vehículo, modificando por el camino la lógica necesaria para un funcionamiento correcto.
- A continuación, se fueron implementando también de uno en uno el funcionamiento de los diferentes actuadores, los cuales manejaría íntegramente el módulo de control.
- Se consiguió que el control lateral del coche funcionase tanto a la tasa de envío de imágenes previa como a la máxima permitida por la cámara.
- Tras esto se probó de forma independiente que el control longitudinal también consiguiese un correcto funcionamiento.
- Una vez conseguido se pasó a incorporar ambos controles junto al control de marchas, resultando en una integración completa exitosa del sistema.

Llegado a este punto, se da el momento final de pruebas, en el cual se busca comparar el rendimiento obtenido en los proyectos anteriores con el del proyecto actual. Para la realización de estas pruebas, se preparó un recorrido a seguir por parte del vehículo mediante una banda azul en un trayecto sobre asfalto y con iluminación uniforme.

Para comparar con el rendimiento previo, es necesario mencionar una serie de detalles importantes que afectan a este. Hay varios factores que afectan a como el vehículo actúa en base a la información que obtiene del entorno, los cuales son valores constantes que fueron decididos previamente en base a la capacidad de procesado y velocidad del vehículo con que se contaba. Estos valores son:

- Los parámetros del controlador PID. Estos son 3 parámetros (Proporcional, Integral y Derivativo):

    - Proporcional (P): Este parámetro determina cuánto reacciona el sistema al error actual, es decir, a la diferencia entre la posición o ángulo real del vehículo y la trayectoria deseada. Si el valor de P es alto, el coche reaccionará más rápidamente a los errores, corrigiendo su dirección de forma agresiva. Sin embargo, si es demasiado alto, puede generar oscilaciones o movimientos bruscos en la dirección. Si es demasiado bajo, la respuesta será lenta e imprecisa.
    - Integral (I): Este parámetro se encarga de corregir errores acumulados en el tiempo, es decir, errores pequeños y persistentes que el término proporcional por sí solo no logra corregir. Este valor se ajustó a 0, luego no afectará a la toma de decisiones.

- Derivativo (D): Este parámetro considera la velocidad con la que cambia el error, anticipando su evolución. Ayuda a suavizar la respuesta del sistema, actuando como un amortiguador que evita sobrecorrecciones bruscas. Si se configura adecuadamente, mejora la estabilidad y reduce las oscilaciones. No obstante, un valor elevado puede hacer que el sistema reaccione de forma excesiva a pequeños cambios.

- Constante K del algoritmo de Stanley: Determina la intensidad con la que el vehículo corrige su desviación lateral respecto a la trayectoria deseada. Un valor bajo de K provoca correcciones suaves pero lentas, mientras que un valor alto genera respuestas más agresivas que pueden causar oscilaciones o inestabilidad, especialmente a bajas velocidades. En la corrección lateral en Stanley la velocidad influye directamente. A baja velocidad, una K alta puede causar sobrecorrecciones; a alta velocidad, una K baja puede resultar insuficiente para mantener la trayectoria.

Además, los datos de percepción del entorno no se enviarían con la seguridad necesaria, ya que uno de los factores que más afectaba al rendimiento era la posición variable de la cámara, pues al estar fijada únicamente con velcro en la parte frontal del vehículo, la cual además presentaba cierto juego debido a la imposibilidad de asegurar correctamente algunos tornillos, incluso pequeños movimientos de la cámara tenían un impacto significativo en la precisión del control de dirección.

Teniendo en cuenta todo lo anterior, es evidente que las pruebas a realizar estarán condicionadas y restringidas por estos factores. Por ello, resulta conveniente considerar como línea de trabajo futuro, además de una fijación de la cámara más segura y estable, el llevar a cabo un estudio exhaustivo de los parámetros anteriores, con el fin de adaptarlos a la nueva situación planteada tras la ejecución del presente proyecto.

4.3.1. Resultados de las pruebas

Se realizaron dos pruebas de validación en un mismo circuito. Este comienza de un punto inicial tomando un tramo recto de unos pocos metros, después una curva de unos 90° a la derecha, un segundo tramo recto de mayor longitud, terminando con una curva de unos 60° a la izquierda para llegar al punto final.

El vehículo saldría estando detenido desde el punto inicial, y la diferencia entre las dos pruebas radica en la velocidad operativa del vehículo:

- Prueba 1 (Velocidad: 3-4 km/h)

    - Tramo inicial y primera curva: El vehículo completó el tramo recto inicial correctamente. En la curva derecha, realizó una maniobra inicial excesivamente cerrada como se puede observar en la Figura 34, sugiriendo un cálculo inadecuado del radio de giro o ligera demora en el control. Sin embargo, realizó una corrección efectiva durante la propia curva, se observa en la Figura 35.

- Tramo intermedio y curva final: Durante el segundo tramo recto, el vehículo mostró oscilaciones laterales continuas ("zig-zag"), indicando un proceso constante de ajuste de trayectoria. A pesar de esta inestabilidad moderada, se mantuvo en la ruta y tomó con éxito la curva izquierda final (Figura 36).

Figura 34. Prueba 1 - Curva inicial demasiado cerrada

Figura 35. Prueba 1 - Rectificación correcta de curva inicial cerrada

Figura 36. Prueba 1 - Llegada exitosa al punto final

- Prueba 2 (Velocidad: 9-11 km/h)
    - Tramo inicial y primera curva: Desempeño óptimo en el tramo recto inicial y durante la primera curva derecha. Se puede observar en la Figura 37 , la cual si comparamos con la Figura 34 se observa que la rueda está a una distancia de la línea más adecuada al giro a realizar.
    - Transición al tramo intermedio: Se detectó un retraso significativo en enderezar las ruedas tras salir de la primera curva. Se puede observar en la Figura 38 como el vehículo está demasiado desplazado a la izquierda de la línea. Esto provocó que el vehículo iniciara el segundo tramo recto con las ruedas giradas, generando una desviación lateral inmediata.
    - Tramo intermedio: El sistema intentó corregir mediante maniobras correctivas rápidas, pero la mayor velocidad redujo el tiempo disponible para estabilizar la trayectoria. Las oscilaciones laterales no pudieron ser amortiguadas eficazmente.
    - Resultado: Debido a la inestabilidad no controlada en el tramo recto y la velocidad elevada, el vehículo se salió del recorrido definido antes de alcanzar la segunda curva. Se puede observar en la Figura 39 como el vehículo en ese punto debería estar con un mayor grado de giro, como sucedía exitosamente en la Figura 36.

Las pruebas comparativas muestran que el sistema de control autónomo funciona a baja velocidad (3-4 km/h), permitiendo recuperarse de errores puntuales (como el giro excesivo inicial en la curva derecha) y completar el circuito, aunque con oscilaciones laterales en tramos rectos que reflejan ajustes continuos. Sin embargo, a mayor velocidad (9-11 km/h), la rigidez de los parámetros de control (PID de ganancias fijas y constante K de Stanley) y la inestabilidad física de la cámara afectan demasiado y se convierten en limitantes críticos, evidenciando que estos valores no son los adecuados para la situación actual, pues un simple retraso en enderezar las ruedas tras la primera curva desencadenó una desviación lateral que, amplificada por la velocidad, derivó en correcciones bruscas y la salida del recorrido.

Figura 37. Prueba 2 - Primera curva tomada correctamente

Figura 38. Prueba 2 - Retardo en el alineamiento con la recta tras el giro

Otro aspecto relevante a destacar es que, si bien los controles lateral y longitudinal funcionan correctamente de manera independiente, durante las pruebas se observó un comportamiento irregular al implementar ambos de forma conjunta junto con el control de marchas. Esta irregularidad no se tradujo en una degradación del rendimiento

individual de los controles, sino en fallos intermitentes en los que, de forma repentina, uno de los dos dejaba de operar. Tras analizar el comportamiento del sistema, se concluye que el módulo de control empleado podría no disponer de la capacidad suficiente para gestionar de manera consistente el funcionamiento coordinado de todos los actuadores involucrados.

Figura 39. Prueba 2 - Llegada incorrecta al punto final

No obstante, esta conclusión no puede considerarse definitiva, ya que no se ha podido determinar con certeza que la causa del mal funcionamiento sea la limitación del módulo de control. Por ello, se plantea como posible línea futura de trabajo el análisis en profundidad del comportamiento del sistema cuando opera con todos los actuadores de forma simultánea, con el objetivo de comprender por qué se produce esta pérdida intermitente de respuesta. Cabe destacar que no se trata simplemente de que un actuador deje de funcionar, sino que el *software* deja de detectarlo por completo, como si hubiese dejado de estar presente en el sistema. Este comportamiento sugiere la necesidad de una investigación más detallada que permita identificar su origen y desarrollar una solución que garantice un funcionamiento conjunto estable y fiable.

4.4. Troubleshooting

Durante todo el desarrollo del proyecto, se han resuelto muchos problemas, una gran parte de estos han sido problemas que ya se habían dado de forma previa en otros proyectos, pero debido a que no se han documentado generan un grave perjuicio al estudiante que continua el proyecto en la forma de frustración y retraso. Con esta subsección se tratará de mostrar varios problemas que han surgido y la solución desarrollada para resolverlos. El primer problema para resolver y más claro es que no había una guía de cómo poner el sistema en marcha, se muestra en la siguiente subsección:

4.4.1. Arranque del vehículo y ejecución del sistema

A continuación, se describe el proceso de puesta en marcha del vehículo y del sistema completo:

- Secuencia de arranque
    - Encendido físico del vehículo:
        - Girar la llave de contacto hasta su posición máxima, debe aparecer GO en la pantalla.
        - Encender el *inverter*. Si los interruptores (de los módulos de control y comunicaciones y el del *router*) no están encendidos, encenderlos.
    - Inicio del subsistema computacional:
        - Encender el PC Fusión.
        - Establecer conexión SSH con los tres módulos del sistema desde un PC externo:
            - PC Fusión: 192.168.1.140
            - Módulo de Comunicación: 192.168.1.116
            - Módulo de Control: 192.168.1.122
    - Configuración previa:
        - Ejecutar *script* de inicialización en PC Fusión mediante:
            - `sudo ./initial_script.sh`
        - Ejecutar *script* de detección de terminales (opcional) y actualizar variables de entorno:
            - `python3 Victor/detectar_term_victor.py`
            - `source ~/.bashrc`

- Ejecución principal del sistema autónomo
    - Lanzo módulo de control:
        - `python3 Victor/actuadores_bueno.py`
    - Lanzar PC Fusión:
        - `. limpiarNodos.sh #Limpieza de nodos ROS 2 en ejecución`
        - `. ejecRos2.sh`
    - Lanzar el módulo de comunicaciones:
        - `sudo python3 mod-com.py`
    - Esperar a que aparezca en la salida del módulo de comunicaciones el mensaje: [INFO/MQTT_receiver_process] Connected to broker MQTT
    - Enviar desde el smartphone el comando CONNECTED al tópico `50/order` para cambiar al estado 1 (todos los mensajes que se envían desde el smartphone por MQTT se envían por este tópico).
    - Dejar el volante en la posición central para evitar sobrevueltas.

    - Enviar desde el smartphone el comando AM-ON para entrar en estado autónomo.
    - Solo la primera vez que se ejecuta el sistema tras encender el coche: ejecutar control + C en el módulo de control y después en los otros dos módulos, volver a realizar el proceso completo de ejecución del sistema autónomo.
    - Indicador de funcionamiento: Suenan los relés del controlador del acelerador, se pone de forma automática la marcha 'D', y se mueve el volante hasta colocarse en la posición adecuada.

- Mover el vehículo longitudinalmente

    - Se envían a través del smartphone un mensaje MQTT de tipo GOTO <velocidad> <comportamiento en bifurcaciones: poner 0 por defecto>

- Volver al modo normal (estado 1)

    - Se envía a través del smartphone un mensaje MQTT de tipo AM-OFF.

- Salir del modo fallo (estado 4)

    - Se da este modo por ejemplo al pisar el freno o cambiar las marchas manualmente durante el modo autónomo.
    - Para salir, y volver al modo normal, se envía a través del smartphone un mensaje MQTT de tipo RESTART.

- Ver salida de la cámara procesada en tiempo real en un PC externo

    - Ajustar en el nodo `camera_node` la dirección IP de tu PC externo
    - Descargar en el PC externo el programa `ffmpeg`,
    - Moverse hasta el directorio `bin`, dentro de la localización donde se realizó la instalación del *software*.
    - Ejecutar: `./ffplay.exe -fflags nobuffer -probesize 32 -sync ext -vf setpts=0 udp://127.0.0.1:5000`

4.4.2. Solución de errores

En la Tabla 3 se presentan una serie de errores recurrentes que pueden darse debido a la propia utilización de los equipos, junto a su causa o posibles causas, y las soluciones que se han utilizado para solventarlos.

Problema	Causa	Solución

Tras un funcionamiento correcto, repentinamente comienza a aparecer que no llega el mensaje `COM_status` al PC Fusión.	Desconexión del adaptador CAN.	Comprobar la desconexión ejecutando desde el PC Fusión:
		`candump can0`
		Si no hay salida, desconectar y volver a conectar el adaptador, comprobando la correcta conexión de los 3 cables: CAN L, CAN H y GND.
		Dar de alta la interfaz CAN ejecutando:
		`sudo ./initial_script.sh`
		Y comprobar que ahora si hay salida del comando:
		`candump can0`

Funcionamiento correcto y repentinamente mensaje de error (en color rojo) de tipo error 1.0 en la salida del programa en el PC Fusión.	La cámara no detecta la línea. Probablemente puede haberse movido la cámara de posición, haciendo que en su plano de visión aparezca el borde de la base donde se sujeta.	Comprobar la visión de la cámara ejecutando desde el PC Fusión: `. ejecCam.sh` Y ejecutar desde el PC externo el programa ffmpeg (según se explica en la sección 4.4.1. Si vemos que aparece el borde, y en efecto el segmento que delimita la dirección a seguir no aparece, recolocar manualmente la posición hasta que aparezca correctamente dicho segmento siguiendo la línea.

Cese del funcionamiento de los actuadores de aceleración o de marchas.	1.Incorrecta detección de los USB.
	2.Fallo detectado y explicado en la sección 4.3.1. en que no funcionan todos los actuadores de manera conjunta.
		Se volverán a detectar los terminales asociados a cada USB mediante la ejecución desde el módulo de control en el directorio ~/Victor de:
		`python3 detectar_terminales_victor.py`
		Se guardará la nueva asignación de forma permanente en el módulo mediante:
		`source ~/.bashrc`
		En caso de seguir el mismo fallo, cambiar manualmente la conexión a otro puerto USB y repetir el proceso.

Controladora Epos4 deja el volante bloqueado tras la salida del modo autónomo.	1.Salida incorrecta de la ejecución del programa
	2.Ejecución simultánea de todos los actuadores (fallo explicado en la sección 4.3.1.).
		Desde el módulo de control, comprobar que se haya cortado la ejecución del proceso `HelloEposCmd` mediante
		`ps aux | grep HelloEpos`
		Si aparece cortar su ejecución.
		Ejecutar a continuación:
		`python3 ~/Demo/epos4.py`
		Introducir el valor 9 para liberarla.

Cese del funcionamiento de la controladora Epos4.	1.Posible desconexión de o bien el USB que conecta esta con el módulo de control 2.Posible desconexion con el motor Maxon (por ejemplo el encoder) 3.La ejecución simultánea de todos los actuadores (fallo explicado en la sección 4.3.1.).	Para comprobar que la conexión con el módulo de control es correcta, ejecutar desde este el comando `lsusb`. Debe aparecer un dispositivo el cual no tiene asociado un nombre, esta es la Epos. Si no aparece, desconectar y volver a conectar en otro puerto USB del módulo o bien asegurar la conexión del puerto en la Epos (está algo dañada y tiene cierto juego). Una vez la detecta correctamente, si sigue fallando, comprobar las conexiones entre la Epos y el motor Maxon, si no se encuentra ninguna desconexión, instalar el programa Epos Studio. Desde el programa al ejecutar la propia Epos aparecerá el error asociado, del cual se podrán encontrar las posibles causas en los manuales de la Epos 4 70/15.
Error en el nodo de la cámara, no la detecta.	Se haya desconectado la cámara y el índice asociado a esta haya cambiado.	Comprobar en primer lugar que la reconoce el sistema mediante `lsusb`, aparece como en la Figura 16. Si así es, dentro del nodo `camera_node` del PC Fusión, en la línea: `self.cap = cv2.VideoCapture(0)` El 0 representa el índice del dispositivo de video asociado a la cámara, sustituirlo por un 1.

Tabla 3: Troubleshooting

5. Conclusiones y líneas futuras

5.1. Conclusiones

En este proyecto se ha logrado rediseñar completamente la arquitectura *software* del vehículo autónomo (Renault Twizy), migrando de un sistema más monolítico al paradigma modular basado en ROS 2. Se ha incorporado un nuevo módulo de mayor rendimiento (denominado PC Fusión) que se encarga de la percepción del entorno a través de sensores (cámara, ultrasonidos, RFID) y del procesado de dichos datos junto al control de errores y del estado del sistema, liberando al módulo de control original de las tareas más intensivas en cómputo.

Al especializar el módulo de control exclusivamente en las funciones de actuación (acelerador, marchas y dirección), se ha conseguido reducir significativamente la latencia en la toma de decisiones. Esta mejora es fundamental para aumentar la seguridad y la fiabilidad de la conducción autónoma, pues el nuevo sistema actúa con tiempos de respuesta más rápidos al eliminar las limitaciones del *hardware* previo.

Además, la reorganización tanto física (reubicación de conexiones USB y CAN) como lógica (comunicación mediante MQTT y tópicos de ROS 2) ha facilitado la integración eficiente de los componentes, y ha permitido acercar la arquitectura del vehículo a una más propia de un vehículo autónomo, en la cual idealmente se separarían en diferentes módulos *software* las tareas de percepción, planificación y actuación. Para la conexión entre el PC Fusión y el módulo de control se eligió MQTT como protocolo ligero, lo que introduce baja sobrecarga y latencias reducidas en el intercambio de mensajes frecuentes.

El resultado global es un sistema más robusto, mantenible y escalable; cada funcionalidad (percepción, planificación, actuación) se ejecuta en nodos especializados, lo cual facilita la ampliación o sustitución de partes del sistema sin afectar al resto. Los principales beneficios de la nueva arquitectura pueden resumirse en los siguientes puntos clave:

- Mejora del procesamiento de visión y datos: La introducción del PC Fusión multiplica la capacidad de cómputo disponible, permitiendo procesar flujos de *frames* a una mayor tasa sin sobrecargar el módulo original. Esto optimiza la percepción ambiental obteniéndose una representación más exhaustiva del entorno.
- Modularidad y escalabilidad: El uso de ROS 2 y MQTT permite organizar el *software* en nodos independientes, desacoplando los componentes. La arquitectura distribuida facilita añadir, sustituir o mejorar nodos (por ejemplo, nuevas cámaras o sensores) sin necesidad de rediseñar todo el sistema. Cada

módulo comunica mediante tópicos estándar, reforzando la reutilización de *software* y el aislamiento de fallos.

- Reducción de latencia: Al centralizar el procesamiento en el PC Fusión y descargar al módulo de control, se ha logrado acelerar la ejecución de las diferentes tareas de procesado, mejorando la capacidad de respuesta del vehículo autónomo. Este menor retardo en la ruta sensor–procesamiento–actuador es esencial para operar en tiempo real.
- Alineación con la arquitectura ideal: Con la separación clara entre módulos de percepción, planificación y control, la nueva arquitectura se acerca al modelo ideal descrito en la memoria (Figura 7). Tal como se destaca, esta estructura de tres bloques principales (nodo central de planificación conectado a nodos de sensores y de actuación) cumple los criterios de un vehículo autónomo óptimo. La adopción de estándares de comunicación como ROS 2 y MQTT no solo refuerza la modularidad, sino que facilita la integración con soluciones utilizadas en la industria de automoción, mejorando la mantenibilidad y la fiabilidad global.

En conclusión, el proyecto ha cumplido sus objetivos de mejorar el rendimiento del Twizy autónomo. La arquitectura ROS 2 desplegada en el PC Fusión ha proporcionado flexibilidad y eficiencia, permitiendo que el módulo de control quede dedicado exclusivamente al control de actuadores. La reorganización propuesta optimiza la gestión de recursos, facilita la incorporación de nuevos sensores y reduce las restricciones anteriores. En conjunto, esta reestructuración mejora significativamente el rendimiento del sistema y establece una base modular y escalable sobre la que desarrollar futuras mejoras, orientadas a afrontar escenarios más exigentes de conducción autónoma.

5.2. Líneas futuras

A partir de esta base robusta, se identifican varias líneas de desarrollo adicionales para seguir avanzando en el proyecto:

- Integración de sensores LIDAR y otros: Aprovechando la modularidad de ROS 2 y la mayor potencia del PC Fusión, el sistema puede ampliarse para incluir sensores más avanzados sin modificar la arquitectura principal. Esto enriquecería la percepción del entorno (detección de obstáculos en 3D) y permitiría desarrollar algoritmos de visión artificial más complejos, mejorando la seguridad en escenarios de poca visibilidad o alta variabilidad en el entorno.
- Algoritmos de inteligencia artificial: Se sugiere incorporar técnicas de aprendizaje automático e inteligencia artificial para tareas como reconocimiento de objetos o planificación predictiva. Gracia a ROS 2, el PC Fusión podría integrar modelos de IA y procesarlos en tiempo real. Por ejemplo, se podrían implementar redes neuronales para mejorar el procesamiento de imágenes de la cámara o algoritmos de planificación basados en aprendizaje reforzado.
- Mecanismos de detección y recuperación de fallos: Como se ha observado en la fase de implementación, actualmente no existe un control de errores entre el PC

Fusión y el módulo de control. Se considera recomendable el desarrollo de protocolos de diagnóstico y recuperación ante fallos en tiempo real. En concreto, sería interesante una estandarización de los mensajes de estado y error sobre el bus CAN que se aplicara en los tres módulos.

- Eliminación de *buffers* adicionales en PC Fusión: Actualmente se emplean colas o *buffers* intermedios para el paso de mensajes dentro del PC Fusión, pese a que ROS 2 ya incorpora mecanismos de cola y control de calidad de servicio (QoS) integrados. Sería conveniente estudiar la viabilidad de eliminar estas colas externas, lo que simplificaría el diseño del *software* y podría reducir retardos innecesarios.
- Revisión del *hardware* de los sensores de ultrasonidos: Durante las pruebas se detectaron fallos en la lectura de los sensores de ultrasonidos. Una línea de trabajo inmediata es verificar y corregir el conexionado y funcionamiento del *hardware* asociado, asegurando la correcta adquisición de datos de proximidad y su integración en el sistema.
- Fijación definitiva de la cámara: Para garantizar resultados estables en la detección visual y el control lateral, es prioritario ajustar la posición física de la cámara, fijándola de forma robusta. Esto evitará desviaciones entre pruebas permitiendo tener un rendimiento más consistente.
- Calibración precisa de los parámetros de control: Es necesario realizar un estudio sistemático de los parámetros de control. En concreto, se propone ajustar el valor del parámetro K del algoritmo Stanley, así como los coeficientes de los controladores PID asociados a dirección y aceleración. Esto permitirá mejorar la respuesta dinámica del vehículo y reducir oscilaciones o errores de seguimiento, y, en particular, aprovechar de forma óptima la mayor capacidad computacional y la reducción de latencia logradas con la nueva arquitectura propuesta en este proyecto.
- Activación concurrente de todos los actuadores: En la versión actual se ha identificado un problema que impide operar simultáneamente con todos los actuadores del vehículo. Se plantea resolver esta limitación para permitir una conducción autónoma fluida, en la que se pueda controlar dirección, aceleración y marchas de forma coordinada desde los comandos generados por el PC Fusión.
- Análisis y resolución del comportamiento irregular en la operación simultánea de actuadores: Aunque los controles lateral, longitudinal y de marchas operan correctamente de forma individual, se ha observado un comportamiento anómalo al activarlos simultáneamente, con pérdidas intermitentes de detección de alguno de los actuadores. Se propone como línea futura el análisis detallado del comportamiento del sistema en estas condiciones, con el objetivo de determinar si el origen del fallo reside en una limitación del módulo de control, en la gestión concurrente de procesos o en problemas de comunicación interna.

6. Bibliografía

[1] A. Mazaira Hernández, Front-end implementation for an automatized car parking, Valladolid: Universidad de Valladolid, 2020.

[2] «Twizyline,» 2020. [En línea]. Available: http://twizyline.com/. [Último acceso: 22 Junio 2025].

[3] «AutoBild,» [En línea]. Available: https://www.autobild.es/coches/renault/twizy. [Último acceso: 22 Junio 2025].

[4] S. Pilar Arnanz, «Servicios de vehículo conectado y conducción autónoma en un Twizy,» 2021.

[5] D. Manso Fernández, «Puesta a punto y optimización de un software de control lateral para un prototipo de vehículo autónomo,» 2024.

[6] M. Reke, «A Self-Driving Car Architecture in ROS2,» de *International SAUPEC/RobMech/PRASA Conference*, Cape Town, South Africa, 2020.

[7] «Amazon,» [En línea]. Available: https://www.amazon.es/DSD-TECH-Adaptador-Hardware-Canable/dp/B0BQ5G3KLR. [Último acceso: 2025 Junio 22].

[8] «SolidRun,» [En línea]. Available: https://www.solid-run.com/embedded-industrial-iot/nxp-i-mx6-family/hummingboard-cbi/. [Último acceso: 2025 Junio 22].

[9] «Zotac,» [En línea]. Available: https://www.zotac.com/es/product/mini_pcs/magnus-en173080c-barebone-0. [Último acceso: 2025 Junio 22].

[10] J. Ren y D. Xia., Autonomous driving algorithms and Its IC Design, Springer Singapore, 2023.

[11] Q. Zhou, Z. Shen, B. Yong, R. Zhao y P. Zhi, Theories and Practices of Self-Driving Vehicles, Elsevier, 2022.

[12] «km77,» [En línea]. Available: https://www.km77.com/reportajes/varios/conduccion-autonoma-niveles . [Último acceso: 14 Enero 2025].

[13] S. Drew Pendleton, H. Andersen, X. Du y X. Xen, «Perception, Planning, Control, and Coordination for Autonomous Vehicles,» 2017.

[14] W. Zong, C. Zhang, Z. Wang, J. Zhu y Q. Chen, «Architecture Design and Implementation of an Autonomous Vehicle,» 2018.

[15] S. Macenski, T. Foote, B. Gerkey, C. Lalancette y W. Woodall, «Robot Operating System 2: Design, Architecture, and Uses In The Wild,» 2022.

[16] O. Robotics, «ros.org,» [En línea]. Available: https://docs.ros.org/en/rolling/Releases.html#list-of-distributions. [Último acceso: 16 Abril 2025].

[17] RoboSense, «GitHub,» [En línea]. Available: https://github.com/RoboSense-LiDAR/rslidar_sdk. [Último acceso: 16 Abril 2025].

[18] O. Robotics, «ros.org,» [En línea]. Available: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html. [Último acceso: 14 Julio 2025].

[19] I. Royuela González, «Four level autonomous vehicle for an automatized parking,» 2020.

[20] C. Gómez Diego, «Guiado de vehículo autónomo mediante tecnología LiDAR,» 2022.