UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

"MODELADO Y SIMULACIÓN DE CONECTIVIDAD V2X EN LA PLATAFORMA DE CONDUCCIÓN AUTÓNOMA CARLA"

TRABAJO FIN DE GRADO

Septiembre – 2023

AUTOR: Rubén Andrés Servin
DIRECTOR/ES: Miguel Sepulcre Ribes

RESUMEN

La conducción autónoma se prevé clave para el futuro del transporte gracias a mejoras en la seguridad y eficiencia del tráfico. Para conseguir circular de forma autónoma sin intervención humana, los vehículos autónomos emplean sensores para percibir su entorno local de conducción y maniobrar consecuentemente. La tecnología de los sensores ha mejorado significativamente en los últimos años en cuanto a rango de percepción y precisión. Sin embargo, las capacidades de la sensorización aún pueden verse mermadas debido a la presencia de obstáculos, condiciones climáticas adversas o a las condiciones de iluminación, entre otros factores. Estas limitaciones pueden influir negativamente en la seguridad y la eficiencia de los vehículos autónomos.

En este contexto, las comunicaciones V2X (*Vehicle-to-Everything*) pueden reducir este impacto negativo y mejorar la percepción o las capacidades de detección de los vehículos conectados y autónomos al facilitar el intercambio de datos de sensores entre vehículos. Este proceso se denomina generalmente percepción cooperativa, percepción colectiva o sensado cooperativo. La percepción cooperativa permite a los vehículos intercambiar los datos que captan sus sensores. Esto proporciona a los vehículos información adicional sobre el entorno de conducción. Por ejemplo, los vehículos receptores serán capaces de detectar objetos que, de otro modo, no serían detectables localmente. La percepción cooperativa también ayuda a mitigar el impacto negativo de las condiciones climáticas adversas o las condiciones de iluminación. La percepción cooperativa es, por lo tanto, esencial para que los vehículos autónomos tengan una percepción precisa del entorno.

Para el estudio de los sistemas de comunicación V2X en general, y la percepción cooperativa en particular, se requieren simuladores que permitan emular de forma realista el entorno para producir resultados adecuados. A pesar de los avances en simuladores de tráfico y red, estos a menudo no consideran la percepción y las comunicaciones como aspectos interrelacionados, lo que puede limitar su utilidad para el estudio y desarrollo de vehículos autónomos conectados. Además, estas simulaciones pueden requerir una considerable cantidad de tiempo, esfuerzo y capacidad de cálculo, dependiendo de los parámetros de comunicación y la densidad del tráfico.

Este Trabajo de Fin de Grado aborda estos desafíos mediante la implementación de un módulo de comunicaciones V2X en un simulador realista que permite simular escenarios de conducción autónoma. Este enfoque unifica la simulación del entorno y las comunicaciones, permitiendo un estudio más completo y eficiente de los vehículos autónomos conectados. Con este módulo, se pueden analizar y estudiar las comunicaciones vehiculares en un entorno similar al de la realidad, aprovechando las simulaciones para mejorar la percepción basándose en modelos analíticos de comunicaciones configurados para tener en cuenta una gran variedad parámetros, como también errores de comunicaciones que pueden surgir en las comunicaciones.

ABSTRACT

Autonomous driving is expected to be key to the future of transportation thanks to improvements in traffic safety and efficiency. To achieve autonomous driving without human intervention, autonomous vehicles use sensors to perceive their local driving environment and maneuver accordingly. Sensor technology has improved significantly in recent years in terms of sensing range and accuracy. However, sensing capabilities can still be impaired due to the presence of obstacles, adverse weather conditions or lighting conditions, among other factors. These limitations can negatively influence the safety and efficiency of autonomous vehicles.

In this context, V2X (Vehicle-to-Everything) communications can reduce this negative impact and improve the perception or sensing capabilities of connected and autonomous vehicles by facilitating the exchange of sensor data between vehicles. This process is generally referred to as cooperative perception, collective perception or cooperative sensing. Cooperative sensing allows vehicles to exchange data captured by their sensors. This provides vehicles with additional information about the driving environment. For example, receiving vehicles will be able to detect objects that would otherwise not be detectable locally. Cooperative sensing also helps mitigate the negative impact of adverse weather or lighting conditions. Cooperative perception is therefore essential for autonomous vehicles to have an accurate perception of the environment .

For the study of V2X communication systems in general, and cooperative perception in particular, simulators that allow realistic emulation are required to produce adequate results. Despite advances in traffic and network simulators, these often do not consider perception and communications as interrelated aspects, which may limit their usefulness for the study and development of connected autonomous vehicles. In addition, these simulations can require a considerable amount of time, effort and computational power, depending on communication parameters and traffic density.

This Final Degree Work addresses these challenges by implementing a V2X communications module in a realistic simulator that allows simulating autonomous driving scenarios. This approach unifies the simulation of the environment and

communications, allowing a more complete and efficient study of connected autonomous vehicles. With this module, vehicular communications can be analyzed and studied in a realistic environment, leveraging simulations to improve perception based on analytical communications models configured to account for a wide variety of communications errors that can arise in communications.

AGRADECIMIENTOS

Este Trabajo de Fin de Grado representa el resultado de muchos meses de investigación, dedicación y aprendizaje. Su culminación no habría sido posible sin la ayuda de varias personas a las que deseo expresar mi gratitud. En primer lugar, a mi tutor, Miguel Sepulcre, por su disponibilidad y dedicación, brindándome su apoyo siempre que lo necesité, no solo en su papel de tutor, sino también como profesor.

Quiero agradecer al Departamento de Ingeniería de Comunicaciones, específicamente a los miembros del laboratorio UWICORE, por proporcionarme la oportunidad de acceder al equipo y las herramientas necesarias para el desarrollo de este trabajo.

Además, quisiera expresar mi gratitud a los compañeros y compañeras que me apoyaron en todo momento a lo largo de la carrera, compartiendo tanto alegrías como tristezas. También a aquellas personas que me alentaron a continuar durante todo este tiempo.

Finalmente, y no menos importante, a Esther, mi novia y mejor amiga. Por su apoyo incondicional durante todos estos años y por brindarme la oportunidad de alcanzar metas que no habría podido lograr solo.

A todos vosotros, gracias.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABLAS

LISTA DE ACRÓNIMOS Y ABREVIATURAS

ADAS	Advanced Driver Assistance Systems
API	Application Programming Interface
CAM	Cooperative Awareness Messages
CARLA	Car Learning to Act
CAV	Connected Automated Vehicle
CPM	Collective Perception Message
C-V2X	Cellular V2X
DENM	Decentralized Environmental Notification Messages
ETSI	European Telecommunications Standards Institute
FIFO	First In - First Out
IVC	Inter Vehicular Communication
IDE	Integrated Development Environmen
LTE-V2X	LTE-Vehicular
MCS	Modulation and Coding Scheme
PDR	Packet Delivery Ratio
PID	Proportional–Integral–Derivative
ROS	Robot Operating System
RSU	Road Side Unit
SAE	Society of Automotive Engineers
TM	Traffic Manager
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle
V2X	Vehicle-to-Everything

1 INTRODUCCIÓN

El futuro del transporte tiende a estar marcado por el protagonismo de los vehículos autónomos. Estos vehículos se caracterizan por su capacidad para percibir su entorno y maniobrar de manera independiente, lo cual tiene un potencial considerable para mejorar la seguridad y la eficiencia del tráfico. No obstante, su funcionamiento se basa en gran medida en sensores que pueden verse limitados por la presencia de obstáculos, condiciones climáticas adversas, condiciones de iluminación y otros factores. Estas limitaciones pueden influir negativamente en la seguridad y eficiencia de los vehículos autónomos.

Para compensar estas limitaciones, se recurre a las comunicaciones V2X (*Vehicle-to-Everything*), que permiten el intercambio de datos de sensores entre vehículos para mejorar su percepción del entorno. Este proceso se denomina percepción cooperativa y es esencial para una conducción autónoma segura y eficiente. No obstante, el estudio y desarrollo de estas tecnologías suponen desafíos considerables.

La investigación en el ámbito de la conducción autónoma enfrenta retos importantes en términos de logística e infraestructura necesarios para evaluar estos sistemas en entornos reales. Las pruebas en escenarios del mundo real se ven limitadas por los altos costos y la complejidad inherente a la diversidad de situaciones que pueden surgir, dificultando la replicación y creación de infraestructuras que reproduzcan con exactitud, por ejemplo, una red de carreteras con tráfico.

Para abordar este desafío, el enfoque actual consiste en emplear herramientas que simulan redes de comunicación y tráfico vehicular. Sin embargo, las herramientas y plataformas de comunicaciones vehiculares disponibles en la actualidad no modelan de manera detallada el entorno idóneo que permita estudiar fielmente aspectos relacionados con la conducción autónoma y los CAVs (*Cooperative Autonomous Vehicles)*.

Considerando que los vehículos circularán en un entorno real, estas herramientas y plataformas de comunicación deberían ser capaces de simular una infraestructura que

reproduzca con fidelidad el mundo real, particularmente para investigar y desarrollar no sólo aspectos vinculados a la comunicación en la conducción autónoma sino también aquéllos relacionados con la percepción cooperativa.

A raíz de esto, es esencial contar con una herramienta que permita tanto modelar como simular las comunicaciones y el tráfico vehicular de forma detallada y lo más cercana posible a la realidad. De esta manera, se podrán abordar de manera más efectiva los desafíos asociados al estudio y desarrollo de sistemas de conducción autónoma y percepción cooperativa en entornos reales.

La representación de manera tridimensional con un alto nivel de detalle y precisión que represente la infraestructura puede ser de gran ayuda en las investigaciones sobre conducción autónoma, ya que permite crear escenarios complejos y dinámicos que se asemejan fielmente a la realidad, superando así las limitaciones de replicar estos entornos en el mundo físico. En un entorno tridimensional, se pueden generar una amplia variedad de escenarios virtuales que imitan la realidad para contribuir al ajuste y validación de sistemas de conducción autónoma, al mismo tiempo que se integra la modelización y simulación de comunicaciones vehiculares.

En estos escenarios, resulta esencial contar con un modelado preciso de sensores como cámaras, radares o LIDARs, y a la vez, es crucial emplear motores de simulación de alta fidelidad para reproducir las complejas situaciones que se encuentran en el mundo real en un escenario controlado.

Una simulación realista de un entorno controlado puede incluir, por ejemplo, un escenario donde se simulen diversas condiciones meteorológicas, así como distintos momentos del día que permitan hacer un estudio de las comunicaciones y la percepción de los CAVs. El escenario no solo es fundamental para simular las diferentes condiciones del entorno, sino también para crear situaciones que involucren peatones, otros usuarios de la vía como ciclistas, y vehículos con características diversas.

Además, el escenario debe posibilitar la simulación de eventos inesperados, como un vehículo realizando maniobras erráticas o un peatón cruzando la carretera corriendo.

En tales escenarios, se puede evaluar la habilidad de los vehículos autónomos para identificar rápidamente soluciones ante conflictos imprevistos, mientras se analizan las comunicaciones vehiculares y su interacción con las herramientas de conducción autónoma en situaciones como, por ejemplo, en alta congestión vehicular.

Teniendo en cuenta lo mencionado anteriormente, se puede deducir intuitivamente que la mejor solución es combinar ambos conceptos, es decir, realizar co-simulaciones. Esto implica simular un entorno real por un lado y las comunicaciones por el otro, uniendo ambos aspectos a través de un *bridge*. Sin embargo, este enfoque presenta ciertas limitaciones, ya que las simulaciones, por naturaleza, consumen tiempo y recursos de cálculo considerables, especialmente si se configuran con múltiples parámetros.

En casos como la simulación de entornos con una alta densidad de tráfico, se requiere una gran cantidad de recursos y tiempo de simulación. Además, hay que tener en cuenta el *hardware* necesario para llevar a cabo estas tareas. La complejidad aumenta aún más si se busca incluir el aspecto perceptivo en una simulación de conducción autónoma donde intervenga un simulador que utilice un motor gráfico potente para obtener escenarios realistas. Aunque la co-simulación ofrece una solución prometedora para abordar los desafíos en el ámbito de la conducción autónoma, es necesario considerar las limitaciones, los requisitos de recursos y tiempo asociados con esta técnica.

En el presente trabajo se propone emplear la plataforma de simulación tridimensional de alta fidelidad CARLA (*Car Learning to Act*) para recrear una infraestructura vial virtual realista que facilite el modelado de comunicaciones vehiculares en contextos de conducción autónoma. Para ello, se utilizarán modelos analíticos del rendimiento de las comunicaciones vehiculares C-V2X o LTE-V Modo 4, que presentan una PDR (*Packet Delivery Ratio*) en función de la distancia entre el transmisor y el receptor, y los diferentes tipos de errores de transmisión que pueden encontrarse en C-V2X Modo 4.

Estos modelos han sido validados para una amplia gama de parámetros de transmisión y densidades de tráfico. Utilizando este modelo, se desarrolla una herramienta que permite analizar y estudiar las comunicaciones en una plataforma realista que funciona como simulador de conducción autónoma. Este enfoque aprovechará las simulaciones realistas para abordar el estudio de la percepción y, además, se basará en modelos analíticos de comunicaciones, proponiendo un enfoque donde la intervención de un simulador de comunicaciones no sea necesaria.

Este Trabajo de Fin de Grado se divide en secciones principales, cada uno de los cuales se aborda los aspectos relevantes para la realización de este proyecto.

En la sección 2, se explora el estado del arte, comenzando con un análisis de las herramientas de simulación, tanto simuladores de tráfico como de red, seguido de un estudio de las plataformas de simulación virtual de conducción autónoma.

La sección 3 está dedicada a CARLA, una plataforma de simulación de conducción autónoma. En esta sección, se detallan los diversos componentes y funcionalidades de CARLA, desde la arquitectura general hasta su API, destacando los distintos módulos que la componen. También se describe el gestor de tráfico, la creación y gestión de actores y planos, el ciclo de vida de los actores y se describen los principales mapas y escenarios. Finalmente, se analizan los sensores vehiculares y su integración en CARLA.

La sección 4 está dedicada al modelado y la simulación de la conectividad V2X (Vehicle-to-Everything). En esta sección, se presentan las implementaciones realizadas, incluyendo las implementaciones de escenarios, la descripción de la arquitectura adoptada y la presentación de un módulo específico de conectividad V2X. Además, se aborda el modelado de las comunicaciones V2X, incorporando un modelo de rendimiento analítico y representaciones gráficas para una mejor comprensión. Esta sección concluye con la configuración y el desarrollo de la simulación, mostrando cómo se seleccionan y ajustan los parámetros de la simulación.

Finalmente, en la sección 5, se presentan las conclusiones obtenidas a partir del desarrollo del módulo V2X y la experimentación realizada en los capítulos anteriores.

2 ESTADO DEL ARTE

En este capítulo, se abordan las herramientas de comunicación actuales empleadas en el campo de la conducción autónoma. Se mencionan las más utilizadas, se explica su funcionamiento y se destacan las funciones más sobresalientes de cada una, evidenciando las funciones adicionales requeridas para impulsar la investigación y el desarrollo en este sector. Además, se mencionarán las dos plataformas de simulación de conducción autónoma basadas en motores gráficos más prominentes encargadas de simular un entorno virtual, las cuales pueden complementar las herramientas actuales. Se discutirá también la elección de una de estas plataformas como simulador principal a utilizar en este trabajo, destacando sus ventajas y cómo contribuye éste al avance en el ámbito de la conducción autónoma.

2.1 SIMULADORES DE TRÁFICO Y RED

Los simuladores de tráfico son herramientas esenciales y altamente desarrolladas para evaluar sistemas de tráfico complejos, ya que pueden emular la variabilidad temporal de los fenómenos de tráfico. Entre los simuladores comerciales más populares se encuentran PTV Vissim [1] y Aimsun [2]. Ambos se basan en la simulación microscópica, lo que significa que cada entidad se simula individualmente, y ambos pueden proporcionar resultados estadísticos en escenarios de tráfico complejos. SUMO [3], por otro lado, es un simulador de tráfico de código abierto muy popular en la comunidad de investigación. Es compatible con formatos de otros simuladores, como Vissim, y proporciona una API (*Application Programming Interface*) llamada TraCI [4] que permite la comunicación bidireccional con otras aplicaciones, lo que convierte a SUMO en un candidato adecuado para ser un componente básico de un simulador integrado. Estos simuladores, incluido SUMO, permiten modelar sistemas de tráfico que abarcan desde vehículos en carretera, hasta transporte público y peatones, permitiendo analizar el comportamiento individual de cada vehículo o agente dentro del sistema para observar interacciones detalladas entre ellos. Además, incluyen herramientas que facilitan tareas como la búsqueda de rutas, visualización, importación de redes y cálculo de emisiones, proporcionando un enfoque integral para el estudio y desarrollo de la movilidad urbana.

Sin embargo, estos simuladores de tráfico, aunque pueden recrear escenarios de tráfico complejos, no son capaces de simular eventos físicos por sí mismos, lo que los hace insuficientes para simular y recrear escenarios que sean fieles a la realidad. La falta de simulación de eventos físicos limita su aplicabilidad en el ámbito de la conducción autónoma, ya que no pueden abordar aspectos importantes como el reconocimiento de objetos, la detección de obstáculos y la adaptación a diversas condiciones del entorno. Por lo tanto, aunque los simuladores de tráfico como SUMO, PTV Vissim y Aimsun son herramientas indispensables y maduras para el análisis y diseño de sistemas de tráfico, es necesario integrarlos con otras herramientas y plataformas que puedan simular eventos físicos y aspectos relacionados con la percepción, con el fin de obtener resultados más completos y realistas en el campo del vehículo autónomo conectado.

Los simuladores de red a diferencia de los simuladores basados en el tráfico o en motores gráficos como CARLA [5] permiten modelar y simular las comunicaciones. En este tipo de simuladores la simulación se basa en eventos discretos, lo que significa que el tiempo se divide en intervalos discretos. Uno de los simuladores más populares de este tipo es NS-3 (*Network Simulator 3*) [6], de código abierto, ofrece múltiples bibliotecas que permiten modelar canales y protocolos de comunicación, pudiendo abordar de esta manera las comunicaciones entre vehículos en distintos escenarios. Por otro lado, tenemos OMNeT++ [7], aplicación modular y gratuita de código abierto que incluye un entorno de desarrollo integrado en el IDE Eclipse. De OMNeT++ se extienden varios *frameworks* que brindan soporte para la simulación en tiempo real, implementación de protocolos y emulación de redes. Un ejemplo de estas extensiones es Artery [8], que inicialmente fue una extensión del *framework* Veins [9] basado en OMNeT++ y SUMO para la simulación IVC (*Inter-Vehicular Communication*), pero que ahora puede ser utilizado de manera autónoma. Artery permite el modelado e implementación de simulaciones V2X basadas en protocolos ETSI ITS-G5, pudiendo ofrecer servicios de mensajes CAM (*Cooperative Awareness*) y DENM (*Descentralized Notification*).

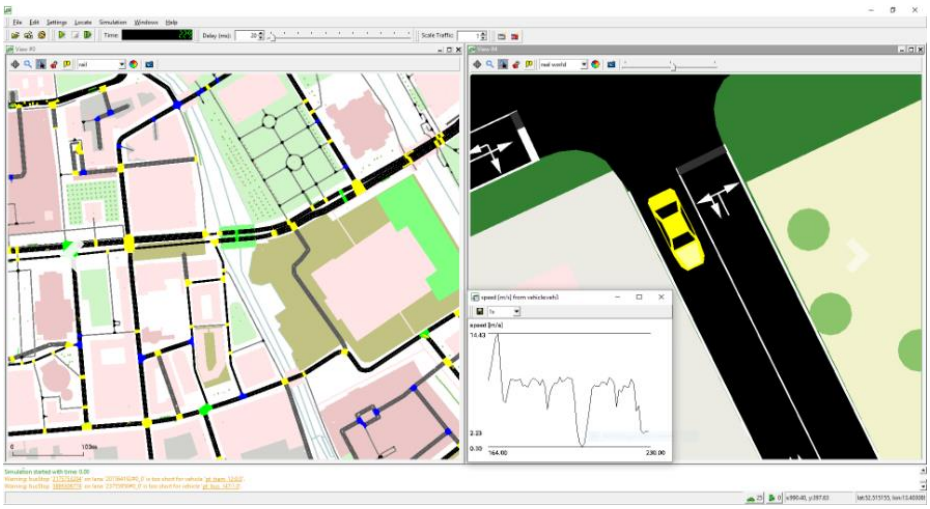


Figura 1. Simulación en SUMO.

Teniendo en cuenta lo expuesto anteriormente, es evidente que estos simuladores se enfocan en la simulación del tráfico y la comunicación entre vehículos, pero no abordan un aspecto fundamental en la investigación de la conducción autónoma: la percepción del entorno por parte del vehículo. Aunque estas herramientas cumplen con su propósito, su enfoque se limita principalmente a la simulación de redes y tráfico. En este contexto, incluir el factor perceptivo en las simulaciones podría enriquecer y complementar estos simuladores, permitiendo abordar la toma de decisiones basada en la percepción del entorno.

Es importante destacar que la mayoría de las investigaciones y aplicaciones actuales sobre conducción autónoma se centran en la percepción del entorno y la inteligencia del vehículo, es decir, su capacidad para moverse por el entorno de manera autónoma en base a la percepción de este. Sin embargo, generalmente los estudios abordan de manera separada la capacidad autónoma, la percepción y las comunicaciones. Esto conlleva limitaciones, ya que estos enfoques tienen carencias, como las comunicaciones en términos de redundancia, latencia y congestión, y en cuanto a la percepción en el procesamiento de imágenes, nubes de puntos LIDAR o datos de videos. Estas limitaciones podrían superarse aprovechando las ventajas de cada enfoque. Al hacerlo, se ampliaría el alcance de las investigaciones y se obtendrían resultados más realistas, abordando aspectos fundamentales de la conducción autónoma, como el reconocimiento de objetos, detección de obstáculos, adaptación a diversas condiciones ambientales y

capacidad para comunicar el estado de la red vial. De esta manera, al tener información del entorno, el vehículo podría actuar en consecuencia a los *inputs* de dicha información, mientras transmite información a otros vehículos. La integración del factor perceptivo tendría un impacto significativo en el avance del desarrollo de vehículos autónomos, mejorando su seguridad y eficiencia.

En el campo de la investigación, se han publicado numerosos artículos utilizando las plataformas y herramientas mencionadas, pero pocos vinculan estas plataformas con otras que simulan de manera realista el entorno y aprovechen esta información para añadir percepción. Se han hecho estudios en el cual se presentan soluciones para utilizar las herramientas existentes para vehículos autónomos aprovechando la co-simulación o simulación cooperativa, uno de esos estudios [10] plantea el hecho que se ha mencionado anteriormente: el abordaje de los estudios en esta materia de manera separada (autonomía, percepción, comunicación). En este trabajo, se propone utilizar la co-simulación para combinar aspectos como la conectividad y la automatización, destacando que ambos términos son abordados de manera idealista en investigaciones actuales, donde se consideran resueltos los problemas de forma individual sin tratarlos conjuntamente. Este enfoque demuestra la viabilidad de una solución para mejorar las simulaciones de CAVs, empleando un *framework* gRPC (*Remote Procedure Call*) que permite acoplar el simulador de conducción autónoma CARLA con el simulador de redes vehiculares Veins. De esta forma, se logra simular la conectividad y la conducción en vehículos autónomos cooperativos de manera integrada.

Otro artículo [11] aborda el desafío de optimizar la comunicación y el intercambio de datos entre vehículos conectados y su entorno, para garantizar una operación eficiente y segura en la carretera. Dado que el enorme intercambio de datos puede generar redundancia y no incrementar el nivel de conocimiento del entorno, los autores proponen un enfoque basado en la visión para identificar vehículos conectados y optimizar el intercambio de mensajes CPM (*Collective Perception Messages*). De esta manera, se utiliza las características visibles de los vehículos percibidos, detectadas por sensores locales, para determinar su ID. Luego, fusiona los datos V2X con los datos detectados localmente, excluyendo vehículos previamente identificados que hayan intercambiado mensajes CAM en futuros CPM.

En [12] , los autores utilizan el simulador CARLA junto con SUMO y OMNeT++ para crear un marco de simulación denominado CARTERY, que integra tres tipos de simuladores: físico (LIDAR y cámaras), de red (transmisión de datos V2X) y de tráfico (imitación del tráfico real). El *framework* creado combina tres subsistemas, un simulador de red, un simulador de tráfico y un simulador basado en un motor de juego, en una sola plataforma. Se seleccionó CARLA como el componente responsable de simular fenómenos físicos y renderizar. El marco de simulación Artery V2X, que se construye sobre el marco OMNET++, se utiliza para simular las comunicaciones de red, específicamente, las comunicaciones V2X. Para el control y la coordinación de la simulación, se utiliza SUMO.

En otro artículo [13] se utiliza CARLA junto a SUMO para crear una plataforma integrada basada en el protocolo C-V2X (*Cellular-Vehicle-to-Everything*), que permite evaluar tanto el tráfico vial como la dinámica de los vehículos. Los autores implementan un esquema híbrido de programación multi-intersección para coordinar el movimiento de vehículos a través de múltiples intersecciones en una red de carreteras, empleando comunicaciones V2I (*Vehicle-to-Infrastructure*) y V2V (*Vehicle-to-Vehicle*) para mejorar la eficiencia del tráfico. El estudio se centra en la conducción autónoma asistida por redes de comunicación, complementando enfoques basados en la percepción del entorno. Se plantean dos escenarios típicos de conducción autónoma: el primero es un escenario de programación de vehículos que utiliza SUMO para la simulación, teniendo en cuenta la alta densidad de tráfico; el segundo es un escenario de conducción remota que emplea CARLA para lograr una percepción precisa del entorno y un control adecuado de los vehículos. Estos enfoques demuestran ventajas en términos de eficiencia del tráfico y tolerancia a fallos, ofreciendo una solución que integra la comunicación en el proceso.

Estos artículos destacan el potencial de combinar las simulaciones de red con las simulaciones de entornos reales para aprovechar la percepción que tienen los vehículos de su entorno y, al mismo tiempo, contar con la información que proporciona las comunicaciones entre vehículos o elementos en la red vial como por ejemplo una RSU (*Road-Side Unit*). Sin embargo, ninguno de ellos aborda el problema de la comunicación de manera efectiva entre los vehículos autónomos, sino que se centran en la implementación de esquemas para la planificación de rutas de los vehículos,

identificación de las características de estos para su identificación en el entorno, la redundancia en los mensajes enviados y en establecer marcos de simulación para simular de manera realista.

En los artículos se destaca la importancia de las simulaciones de acuerdo con el enfoque seguido en cada caso, destacando el uso de las co-simulaciones. Por un lado, simulaciones de red para las comunicaciones, por otro, simulaciones de entorno reales para aprovechar la percepción de los vehículos. Los métodos y conclusiones a las que se llegan en los artículos aportan una base importante en el campo del vehículo autónomo conectado y la percepción cooperativa.

Sin embargo, se percibe una ausencia o tratamiento superficial de aspectos cruciales como la densidad de vehículos y los parámetros de comunicación. Factores que son relevantes en las simulaciones de los CAVs y el intercambio de información entre ellos, tanto en enfoques orientados hacia el estudio de la percepción como aquellos más enfocados en el análisis de las comunicaciones. Adicionalmente, el uso de la co-simulación a menudo implica tiempo en el desarrollo de puentes para unificar diversos tipos de simuladores, lo que añade complejidad a la arquitectura dependiendo del objeto de estudio.

Derivado de lo anterior, resulta de gran importancia desarrollar estrategias que permitan estudiar la comunicación fiable entre los vehículos autónomos, ya que esto es fundamental para la coordinación y la toma de decisiones conjuntas en situaciones de tráfico complejas. Para lograrlo, se necesitará investigar enfoques que aborden aspectos tan importantes como la potencia de transmisión, la tasa de paquetes transmitidos, el número de subcanales empleados en la comunicación y la densidad de vehículos en la red vial.

2.2 PLATAFORMAS DE SIMULACIÓN

En el estudio de la conducción autónoma, los vehículos deben someterse a pruebas rigurosas y exhaustivas. Un aspecto clave es contar con un entorno que permita entrenar a los vehículos para que logren un control total de su entorno y puedan enfrentar diversos inconvenientes que puedan surgir. El principal desafío en el entrenamiento de sistemas de conducción autónoma radica en la infraestructura necesaria para lograr un entrenamiento eficaz. Las pruebas en entornos reales resultan peligrosas y difíciles de recrear. Para abordar este problema, se utilizan simuladores 3D realistas de alta fidelidad que permiten entrenar a los sistemas sin riesgos y a un costo significativamente menor de los que implican los preparativos para un entorno real. El simulador seleccionado debe ser capaz de recrear un entorno muy similar a la realidad. Este tipo de simuladores es fundamental para el desarrollo de soluciones basadas en percepción cooperativa y para el estudio de su impacto real en el desarrollo del vehículo autónomo conectado.

Las características principales que debe tener un buen simulador incluyen la capacidad de lograr una detección realista mediante sensores reales o simulados (cámara, GPS, LIDAR, etc.) y de interpretar de manera fiable los mensajes sensoriales proporcionados por estos sensores. También debe contar con una cartografía precisa que permita simular mapas realistas, ya sea cargando mapas de terceros o utilizando una biblioteca que contenga variedad de mapas. Los vehículos simulados deben tener configuraciones que permitan la planificación y trayectoria dentro de los mapas a través de una red de carreteras, como por ejemplo la que se muestra en la Figura 2.

El simulador también debe permitir el control total del mundo simulado y ofrecer una conexión bidireccional del estado de los vehículos involucrados en la simulación, permitiendo controlar velocidad, dirección, frenado y cambios de marchas. Para ello, deberá contar con sistemas de planificación y algoritmos de seguimiento de trayectorias, entre otros. En la Figura 3, podemos ver un ejemplo de planificación y generación de trayectoria para desplazarse de un punto A hasta un punto B.

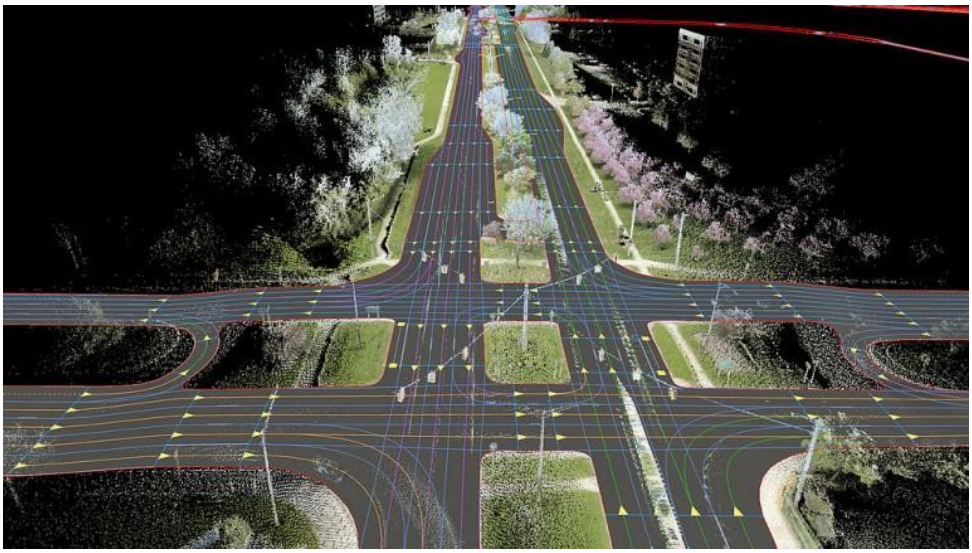


Figura 2. Red de carreteras.

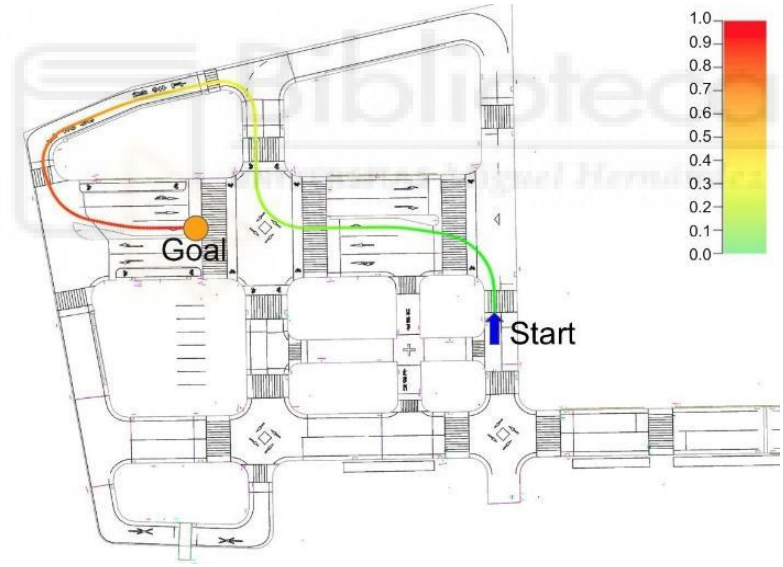


Figura 3. Ejemplo de planificación y generación de trayectoria.

Se puede afirmar que un buen simulador es aquel que satisface los requisitos de ser capaz de percibir objetos estáticos y dinámicos en un entorno realista, navegar a través de rutas planificadas y mantener un control total de la simulación de vehículos en un escenario de tráfico similar al mundo real. Además, debe permitir la recopilación de datos durante toda la simulación. Por último, debe contar con un buen soporte y mantenimiento, ser portátil, escalable y mantener actualizada tanto su documentación como las APIs.

Los simuladores más representativos en la actualidad son SVL (*LG Electronics America R&D Lab)* [14] y CARLA (UAB). SVL es un simulador multi-robot de código abierto que permite probar algoritmos para vehículos autónomos. Desarrollado por el centro de I+D de LG Electronics America, se basa en el motor de videojuegos Unity. SVL incluye sensores como cámaras de profundidad/segmentación, LIDAR, IMU, radar, odometría y GPS. Los entornos disponibles en este simulador cuentan con nueve mapas, siete tipos de vehículos y peatones, además de una tienda de complementos. Permite la configuración de escenarios a través de APIs de Python, como la colocación de objetos, movimiento de vehículos, obtención de datos de sensores y control del clima. Es compatible con las pilas más utilizadas en conducción autónoma: Apollo y Autoware. Aunque permite la edición de mapas, esta tarea no resulta sencilla de realizar.

A continuación, se detalla el hardware recomendado para ejecutar SVL:

- CPU de cuatro núcleos de al menos 4 GHz
- NVIDIA GTX 1080 (memoria de 8 GB) o superior
- Windows 10 (64 bits), Ubuntu 18.04 (64 bits) o Ubuntu 20.04 (64 bits)
    - En Windows se requiere tarjeta gráfica compatible con DirectX 11
    - En Linux una tarjeta gráfica compatible con Vulkan 1.1.
- Si se ejecuta Apollo o Autoware en la misma máquina se requiere una GPU de al menos 10 GB de memoria. En un sistema diferente (recomendado), se requiere una conexión una velocidad gigabit.

Por otro lado, CARLA se basa en el motor de videojuegos Unreal Engine 4 (UE4). Es un simulador de código abierto, modular y flexible diseñado para apoyar la formación y validación de sistemas ADAS (*Advanced Driver Assistance Systems*) y el entrenamiento de algoritmos en sistemas de conducción autónoma. Está desarrollado conjuntamente por Intel Labs, Toyota Research Institute (TRI), Computer Vision Center Barcelona (CVC) [15] y cuenta con la colaboración de NVIDIA.

Los sensores disponibles en CARLA incluyen LIDAR, cámaras (RGB, HDR, profundidad, segmentación semántica, DVS, de flujo óptico), GPS, odometría, RSS, GNSS, IMU, detección de invasión de carril y detector de obstáculos. La definición de escenarios en CARLA se realiza a través de una potente API utilizando Open Scenario [16] como norma estandarizada para describir escenarios de conducción. Cuenta con ocho escenarios (denominados *towns*), y con múltiples modelos de vehículos y peatones. También permite la edición de escenarios, aunque no es una tarea sencilla y se requiere experiencia al trabajar con múltiples capas.

CARLA es compatible con ROS y ROS2. ROS (*Robot Operating System*) es un marco de software flexible y de código abierto para la programación de robots y vehículos autónomos, que proporciona una amplia variedad de bibliotecas y herramientas para simplificar el desarrollo de aplicaciones robóticas. A través de un puente, es capaz de lograr comunicación con Autoware y Apollo, aunque la vinculación con Apollo presenta algunos problemas, ya que no lo soporta de forma nativa (la vinculación con Autoware sí es nativa).

Los requisitos de hardware que necesita CARLA para ejecutarse de forma local son:

- 170 GB de espacio en disco.
- CPU Intel i7 de 9ª a 11ª generación / Intel i9 de 9ª a 11ª generación / AMD Ryzen 7 / AMD Ryzen 9
- GPU NVIDIA RTX 2070 / NVIDIA RTX 2080 / NVIDIA RTX 3070 / NVIDIA RTX 3080
- Windows10 (64 bits), Ubuntu18.04 (64 bits)
- Si se ejecuta Apollo o Autoware en la misma máquina se requiere una GPU de al menos 10 GB de memoria. En un sistema diferente (recomendado), se requiere una conexión gigabit.
- Dos puertos TCP y buena conexión a Internet.

La Tabla 1 presenta una comparativa detallada entre los simuladores SVL y CARLA. En resumen, ambos simuladores cumplen una serie de requisitos, que incluyen el modelado de sensores, la simulación de condiciones meteorológicas, la planificación de rutas, el soporte para la dinámica de vehículos y la capacidad de generar un entorno virtual 3D. Tanto SVL como CARLA permiten la simulación de infraestructuras de tráfico, incluyendo semáforos, intersecciones, señales de stop y carriles. Además, ambos proveen soporte para objetos dinámicos en escenarios de tráfico, generación de datos de verificación en 2D y 3D, y la posibilidad de interfaz con otros programas, como ROS, Apollo y Autoware.

En cuanto a la escalabilidad, SVL y CARLA poseen una arquitectura de servidor-multicliente y ambos son de código abierto. Sin embargo, es importante señalar que el desarrollo de SVL se suspendió a partir del 01/01/2022. A pesar de esta suspensión, la última versión (V2021.3) sigue estando disponible y mantenida. Finalmente, tanto SVL como CARLA son portables, con soporte para Windows y Linux, y ofrecen una API flexible que facilita su integración y adaptación a distintos escenarios y necesidades.

En este trabajo de fin de grado opta por utilizar CARLA por varias razones. En primer lugar, desafortunadamente, SVL ha sido discontinuado y no se continuará con su desarrollo, lo que significa que no habrá soporte futuro; si se escogiera SVL, esta limitación sería importante si se quisiese utilizar este proyecto como punto de partida para proyectos futuros. En segundo lugar, CARLA cuenta con una potente API que permite controlar el mundo de forma completa y tiene una amplia comunidad enfocada en el desarrollo y estudio de la conducción autónoma. Además, cuenta también con una documentación muy completa [17] y en cada actualización mejoran y añaden funciones que permiten ampliar el desarrollo.

Requisitos	Simulador	
	SVL	**CARLA**
Percepción: Modelos de sensores	Sí	Sí
Percepción: Condiciones meteorológicas	Sí	Sí
Calibración de cámara	No	Sí
Planificación de ruta	Sí	Sí
Control de vehículo: Soporte para dinámica de vehículo	Sí	Sí
Entorno virtual 3D	Sí (Urbano)	Sí (Urbano)
Infraestructura de tráfico	Semáforos, intersecciones, señales de stop, carriles	Semáforos, intersecciones, señales de stop, carriles
Escenario de tráfico: Soporte de objetos dinámicos	Sí	Sí
2D/3D Ground truth	Sí	Sí
Interfaz para otros programas	Sí. Para ROS, Apollo y Autoware	Sí. Para ROS y Autoware
Escalabilidad: Arquitectura Servidor-Multicliente	Sí	Sí
Código abierto	Sí	Sí
Mantenimiento y Estabilidad	Latest V2021.3. Suspensión del desarrollo 01/01/2022	Sí
Portabilidad	Windows, Linux	Windows, Linux
API flexible	Sí	Sí

Tabla 1. Comparación CARLA - SVL.

3 SIMULADOR CARLA

CARLA, el simulador principal de este estudio, permite mediante distintas configuraciones, replicar con precisión el mundo real y establecer el entorno deseado para la ejecución de las simulaciones relevantes. En las siguientes secciones, se desglosarán los componentes más significativos de este simulador, los cuales son esenciales para el desarrollo de este trabajo.

3.1 SIMULACIÓN DE CONDUCCIÓN AUTÓNOMA

CARLA es una herramienta altamente personalizable que facilita, por ejemplo, el estudio de estrategias de conducción y el entrenamiento de algoritmos de percepción. La gestión de la simulación se lleva a cabo mediante una API y un servidor se encarga del mundo simulado. El servidor es responsable de todos los aspectos relacionados con la simulación, como el cálculo de la física, la representación de sensores, las actualizaciones del estado del mundo y de los actores involucrados. Por su parte, el cliente comprende un conjunto de módulos que controlan la lógica de los actores en la escena y determinan las condiciones bajo las cuales se ejecuta el mundo. Esto se consigue mediante la API de CARLA (disponible en Python y C++)[18][19] que actúa como intermediario entre el servidor y el cliente.

La Figura 4 ilustra el esquema cliente-servidor de un mundo controlado por múltiples clientes. Es importante mencionar que el uso de varios clientes puede generar problemas, ya que no todos los componentes están delegados al servidor. Por ejemplo, si varios clientes desean crear vehículos, existe el riesgo de que colisionen en el mundo debido a que cada cliente gestiona de manera diferente las órdenes de generación de sus actores.

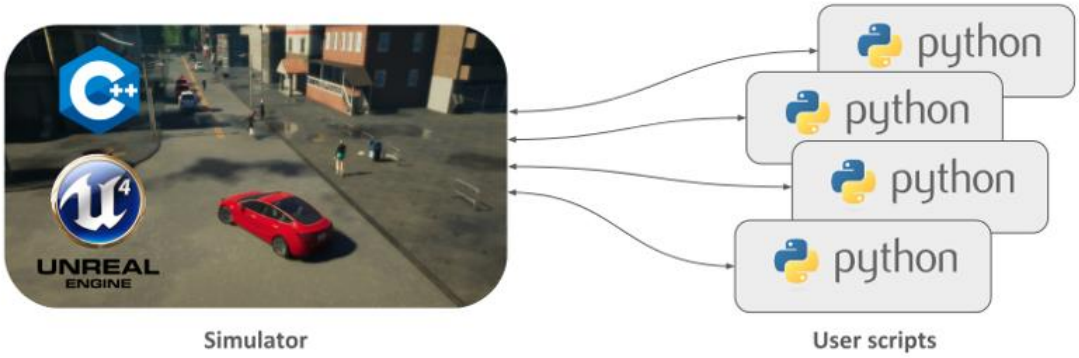


Figura 4. Esquema Cliente - Servidor en CARLA.

3.2 ARQUITECTURA

CARLA posee una arquitectura cliente-servidor escalable que se comunica mediante el protocolo TCP, como se muestra en la Figura 5. El cliente se conecta al servidor de CARLA, y este último, utilizando complementos y el motor gráfico Unreal Engine 4 (UE4), permite ejecutar simulaciones de alta fidelidad. UE4 alberga todos los activos y escenas que emplean los complementos de CARLA para ensamblar las ciudades y el comportamiento de los actores en la escena. Los complementos de CARLA para UE4 contienen todas las funcionalidades.

Desde el lado del cliente, se cuenta con un módulo de alto nivel que permite manejar el mundo simulado a través de la API en Python y scripting. Existe también un módulo de bajo nivel implementado en C++ que también cuenta con su API. Todas las funcionalidades se encuentran accesibles mediante estos módulos. La comunicación a través de la API de Python, por su simplicidad, es la opción utilizada en este trabajo. Por otro lado, el módulo en C++ permite realizar comunicaciones más complejas y de alto rendimiento, aunque su uso representa una mayor complejidad.

Conceptualmente, podemos entender CARLA como un mundo (servidor) y un cliente. Una vez que el cliente se conecta al servidor, es necesario crear un mundo en el cual el cliente pueda generar diferentes actores (vehículos, peatones, sensores, etc.).

Posteriormente, el cliente puede obtener información de forma continua y realizar cambios en los actores o en el mundo mediante comandos. El cliente incluye un gestor de

tráfico (*Traffic Manager* o TM) [20] implementado en C++, cuyo objetivo es recrear el tráfico urbano de manera realista, controlando los vehículos en modo de piloto automático presentes en el mundo.

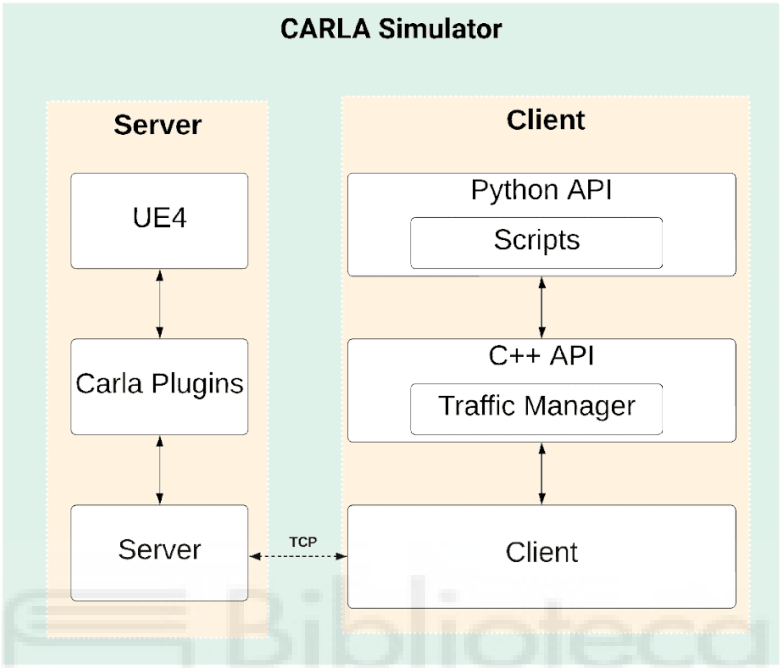


Figura 5 Arquitectura de CARLA.

3.3 API

La API de CARLA permite tener un control total del mundo simulado. Como se comentó anteriormente, en este trabajo se utiliza la API de Python que brinda múltiples funcionalidades que permiten gestionar la simulación. A través de la API, diferentes clases con sus respectivos métodos permiten gestionar el entorno de tráfico simulado, configurar comportamientos y lograr una interacción entre vehículos, peatones, sensores y otros elementos dentro de la simulación de CARLA. Esta combinación de características hace que sea una herramienta poderosa y relativamente fácil de usar para las simulaciones de tráfico urbano.

De este modo, los usuarios pueden ejercer cierto control sobre el flujo del tráfico al definir parámetros que permitan, exijan o promuevan comportamientos particulares.

Por ejemplo, es posible autorizar a los vehículos para que ignoren los límites de velocidad o se les obligue a realizar un cambio de carril en un momento específico. Este enfoque aporta realismo a las simulaciones, reproduciendo el comportamiento de los vehículos en situaciones reales. Cabe recordar que, los sistemas de conducción autónoma deben entrenarse en circunstancias poco comunes.

La Figura 6 ilustra el proceso de creación del entorno en el simulador CARLA, donde la API de Python juega un papel crucial en la configuración y conexión con el servidor. En este flujo, el usuario utiliza la API de Python como herramienta principal para configurar y establecer la conexión con el servidor de CARLA. Luego, el usuario ejecuta las instrucciones a través del cliente, el cual se conecta con el servidor utilizando las herramientas proporcionadas por la API. Finalmente, la simulación del mundo es generada en CARLA.

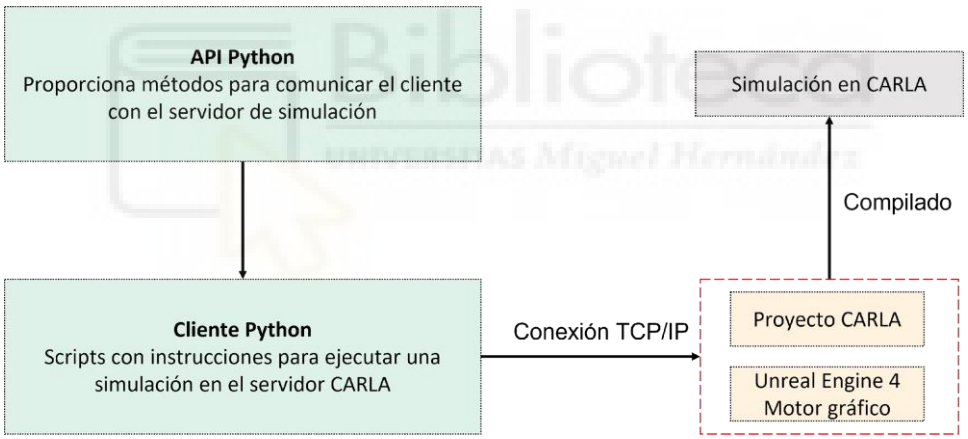


Figura 6 .Flujo desde la API al mundo simulado.

A continuación, se destacan algunas de las clases y elementos más relevantes dentro de la extensa API de CARLA. Aunque se requieren numerosas funciones y clases para simular el entorno y configurar el mundo de CARLA, las que se presentan a continuación son fundamentales y han sido necesarias para llevar a cabo este trabajo.

Es importante mencionar que estos componentes clave, en contraste con otros menos esenciales que no se abordan debido a su amplitud, facilitan la creación del entorno utilizado específicamente en este proyecto.

3.3.1 CARLA ACTOR

CARLA considera a los actores como cualquier elemento que participa en la simulación o que tiene capacidad de movimiento. Esto abarca: peatones, vehículos, sensores y señales viales (incluyendo a los semáforos como parte de estos). La clase *carla.Actor* incluye variables de instancia y métodos importantes para interactuar con los actores en una simulación. Algunos métodos clave son:

- add_force(force): Aplica una fuerza en el centro de masa del actor.
- add_impulse(impulse): Aplica un impulso en el centro de masa del actor.
- add_torque(torque): Aplica un torque en el centro de masa del actor.
- add_angular_impulse(angular_impulse): Aplica un impulso angular en el centro de masa del actor.
- destroy(): Le dice al simulador que destruya este actor.
- get_transform(): Devuelve la transformación (ubicación y rotación) del actor.
- set_transform(transform): Teletransporta al actor a una transformación dada (ubicación y rotación).
- get_location(): Devuelve la ubicación del actor.
- set_location(location): Teletransporta al actor a una ubicación dada.
- get_velocity(): Devuelve el vector de velocidad del actor.
- set_target_velocity(velocity): Establece el vector de velocidad del actor.
- get_acceleration(): Devuelve el vector de aceleración 3D del actor.
- get_angular_velocity(): Devuelve el vector de velocidad angular del actor.
- set_enable_gravity(enabled): Habilita o deshabilita la gravedad para el actor.
- set_simulate_physics(enabled): Habilita o deshabilita la simulación de la física en este actor.

Estos componentes son fundamentales para controlar y manipular los actores en una simulación de CARLA. Los actores son creados en la simulación mediante la clase *carla.World* y requieren la creación de un objeto *carla.ActorBlueprint*.

3.3.2 CARLA BLUEPRINT-LIBRARY

La clase *carla.BlueprintLibrary* almacena los *blueprints*. Un *blueprint* es un concepto utilizado en CARLA para representar un conjunto de propiedades, atributos y configuraciones que definen un objeto específico, como por ejemplo un actor dentro de la simulación. En CARLA los *blueprints* son utilizados para crear actores como vehículos, peatones y otros elementos del mundo virtual. Esta clase tiene como objetivo proporcionar objetos *carla.ActorBlueprint*, que son esenciales para la creación de actores.

Métodos:

- filter(): Filtra una lista de blueprints que coinciden con el patrón comodín contra el id y las etiquetas de cada blueprint en la biblioteca, y devuelve el resultado como una nueva lista.
- find(): Retorna el *blueprint* correspondiente a ese identificador.

Métodos:

- getitem (): Retorna el *blueprint* almacenado en la posición dentro de la estructura de datos que los contiene.
- iter(): Itera sobre los objetos *carla.ActorBlueprint* almacenados en la biblioteca.
- len (): Retorna la cantidad de *blueprints* que comprende la biblioteca.
- str(): Convierte los identificadores de cada *blueprint* a *string*.

3.3.3 CARLA ACTOR-BLUEPRINT

Como se comentó antes, se dispone de una librería de planos para actores a la que se puede acceder mediante *carla.BlueprintLibrary*. Cada blueprint cuenta con un identificador y atributos, que pueden ser ajustables o no. El servidor crea automáticamente la biblioteca y se puede acceder a ella mediante la clase carla.World que representa el entorno de simulación. Esta clase actúa como un puente entre la librería de planos (*carla.BlueprintLibrary*) y la creación de actores. Los actores requieren un modelo basado en un plano específico para ser generados.

Este modelo guarda la información del plano en un objeto, junto con sus atributos y etiquetas de clasificación. Luego, el usuario puede ajustar algunos de los atributos y, finalmente, crear los actores utilizando la clase *carla.World*.

Variables de instancia:

- id (): El identificador del blueprint en la biblioteca. Ej.: *walker.pedestrian.0001*.
- tags (list(str)): Una lista de etiquetas que describen el *blueprint*. Ej.: ['*0001*', *'pedestrian'*, *'walker'*].

Métodos:

- has_attribute(id): Retorna *True* si el *blueprint* contiene el atributo con el id especificado.
- has_tag(tag): Retorna *True* si el *blueprint* contiene la etiqueta especificada.

  match_tags(wildcard_pattern): Retorna *True* si alguna de las etiquetas coincide con el patrón comodín.
- get_attribute(id): Retorna el atributo del actor con el id especificado, si existe.
- set_attribute(id,value): Si el atributo con el id especificado es modificable, cambia su valor al valor proporcionado.

Métodos:

- iter(): Permite iterar sobre los *carla.ActorAttribute* que tiene este *blueprint*.
- len(): Devuelve la cantidad de atributos de este *blueprint*.
- str(): Convierte a cadena de texto la información del *blueprint*.

3.3.4 CARLA WORLD

La clase *carla.World* en CARLA representa el mundo o entorno de la simulación. Proporciona una variedad de métodos y atributos que permiten interactuar con el entorno, controlar los actores, modificar el clima, etc.

Variable de instancia:

- id (int): ID del episodio asociado con este mundo.

  debug (*carla.DebugHelper*): Para crear formas de depuración.

Métodos:

- apply_settings(self, world_settings): Aplica configuraciones al mundo.
- spawn_actor(self, blueprint, transform, attach_to=None, attachment=Rigid): Crea y coloca un actor en el mundo.
- tick(self, seconds=10.0): Envía un *tick* en modo síncrono para avanzar en la simulación.
- get_actors(self, actor_ids=None): Recupera una lista de actores en el mundo.
- get_blueprint_library(self): Retorna una lista de *blueprints* disponibles para crear actores.
- get_map(self): Obtiene el mapa del mundo en formato *carla.Map*.
- get_snapshot(self): Retorna un *carla.WorldSnapshot*, un instante del estado del mundo.
- get_weather(self): Obtiene las condiciones meteorológicas actuales.
- set_weather(self, weather): Establece las condiciones meteorológicas.

3.3.5 CARLA ACTOR-LIST

La clase *carla.ActorList* contiene todos los actores presentes en la escena y permite acceder a ellos. La lista se crea y actualiza automáticamente por el servidor y se puede obtener mediante *carla.World*. Los métodos incluyen poder filtrar actores según patrones comodín en su variable *type_id* (que identifica el *blueprint* utilizado para generarlos) y buscar actores por su identificador. Además, proporciona métodos para iterar sobre los actores contenidos en la lista, obtener la cantidad de actores y representarlos en forma de cadena de texto. Algunos de los métodos que permiten trabajar con esta clase son:

Métodos:

- filter(wildcard_pattern): Este método filtra la lista de actores según el patrón comodín proporcionado y lo compara con la variable *type_id* de cada actor, que

identifica el *blueprint* utilizado para generarlos. Devuelve una lista de actores que coinciden con el patrón.

- find(actor_id): Este método busca un actor en la lista utilizando su identificador. Si el actor está presente, lo devuelve; de lo contrario, devuelve *None*.

Métodos

- getitem(pos): Devuelve el actor correspondiente a la posición 'pos' en la lista.
- iter(): Permite iterar sobre los actores *carla.Actor* contenidos en la lista.
- len(): Devuelve la cantidad de actores en la lista.
- str(): Convierte a cadena de texto las ID de cada actor en la lista.

3.3.6 CARLA ATTACHMENT-TYPE

La clase *carla.AttachmentType* define opciones de vinculación entre un actor y su actor padre, especialmente útil para sensores. Contiene tres tipos de vinculación:

- Rigid: Vinculación fija que sigue estrictamente la posición del actor padre, ideal para obtener datos precisos.
- SpringArm: Con esta vinculación el objeto (Objeto sensor) se expande o retrae según la posición del actor, recomendada para grabar videos con movimiento suave.
- SpringArmGhost: Similar a SpringArm, pero sin pruebas de colisión, permitiendo atravesar geometrías (atravesar paredes). Por ejemplo, con este tipo de vinculación un objeto cámara podría seguir al vehículo sin chocar con las paredes. Recomendada para videos con movimiento suave.

3.3.7 CARLA SENSOR-DATA

Clase base para todos los objetos que contienen datos generados por un sensor de carla. Estos objetos deben ser el argumento de la función que está escuchando dicho sensor, para poder trabajar con ellos. Cada uno de estos sensores requiere un tipo específico de

datos de sensor. A continuación, se muestra una lista de los sensores y sus datos correspondientes.

- Cámaras (RGB, profundidad y segmentación semántica): *carla.Image*.
- Detector de colisiones: *carla.CollisionEvent*.
- Sensor GNSS: *carla.GnssMeasurement*.
- Sensor IMU: *carla.IMUMeasurement*.
- Detector de invasión de carril: *carla.LaneInvasionEvent*.
- Sensor LIDAR: *carla.LidarMeasurement*.
- Detector de obstáculos: *carla.ObstacleDetectionEvent*.
- Sensor de radar: *carla.RadarMeasurement*.
- Sensor RSS: *carla.RssResponse*.
- Sensor LIDAR semántico: *carla.SemanticLidarMeasurement*.

Variables de instancia:

- frame (int): Número de fotogramas cuando se generaron los datos.
- timestamp (float - segundos): Tiempo de simulación cuando se generaron los datos.
- transform (carla.Transform): Transformación del sensor cuando se generaron los datos.

Método:

- str(self): Permite imprimir datos del objeto detectado y el actor principal.

3.3.8 CARLA SENSOR

Los sensores constituyen una categoría particular de actores, diversa y singular. Generalmente, se crean como elementos vinculados o adjuntos de un vehículo. Los sensores tienen como objetivo recopilar distintos tipos de datos a los que están "escuchando". Los datos que obtienen adoptan la forma de varias subclases derivadas de

carla.SensorData, según el tipo de sensor. La mayoría de los sensores se pueden clasificar en dos grupos: aquellos que capturan datos en cada *tick* (cámaras, nubes de puntos y ciertos sensores específicos) y aquellos que solo recolectan datos en circunstancias particulares (detectores de disparo). CARLA ofrece un conjunto específico de sensores cuyo modelo se puede localizar en *carla.BlueprintLibrary*.

Algunos de los métodos más importantes y sus funciones son los siguientes:

- is_listening(): Devuelve si el sensor está en estado de escucha.
- is_listening_gbuffer(gbuffer_id): Devuelve si el sensor está en estado de escucha para una textura GBuffer específica.
- listen(callback): La función que el sensor llamará cada vez que se reciba una nueva medición. Esta función necesita un argumento que contenga un objeto de tipo *carla.SensorData*.

3.3.9 CARLA TRAFFIC-MANAGER

Es una clase de C++ construida sobre la API de CARLA que simula tráfico urbano realista. Permite personalizar comportamientos de vehículos en modo automático, como cambios de carril, detección de colisiones, distancia a vehículos líderes, ignorar semáforos o señales de tráfico, y controlar la velocidad. También proporciona métodos para obtener información sobre acciones futuras y rutas, así como establecer rutas y parámetros específicos para los vehículos.

El módulo puede ser configurado en modos síncrono, híbrido y determinístico según las necesidades del usuario. Los métodos que se utilizan son los siguientes:

- auto_lane_change(): Activa o desactiva el comportamiento de cambio de carril para un vehículo.
- collision_detection(): Activa o desactiva la detección de colisiones entre un vehículo y otro actor específico.
- distance_to_leading_vehicle(): Establece la distancia mínima en metros que un vehículo debe mantener con otros vehículos.

- set_desired_speed(): Establece la velocidad deseada para un vehículo.
- set_global_distance_to_leading_vehicle(): Establece la distancia mínima en metros que todos los vehículos deben mantener entre sí.
- set_hybrid_physics_mode(): Activa o desactiva el modo de física híbrida, reduciendo el costo computacional al desactivar la física para vehículos lejanos.
- set_synchronous_mode(): Establece el modo síncrono para el *Traffic Manager*, controlando qué instancia de TM será la maestra en modo sincrónico.
- set_route(): Establece una lista de instrucciones de ruta para que un vehículo las siga mientras está controlado por el Traffic Manager.
- get_next_action(): Devuelve la siguiente opción de ruta y *waypoint* que un actor controlado por el *Traffic Manager* seguirá.
- set_random_device_seed(): Establece una semilla específica para el generador de números aleatorios del *Traffic Manager*, haciéndolo determinístico.

3.3.10 CARLA OBSTACLE-DETECTION-EVENT

carla.ObstacleDetectionEvent es la clase que define los datos de los obstáculos detectados si se utiliza el sensor para detección de obstáculos. El objeto proporciona información esencial sobre la ubicación del obstáculo detectado y la distancia entre el actor principal y el obstáculo.

Variables de instancia:

- actor (carla.Actor): El actor al que está conectado el sensor.
- other_actor (carla.Actor): El actor u objeto considerado como un obstáculo.
- distance (float - metros): Distancia entre el actor y el otro objeto/actor.

3.4 GESTOR DE TRÁFICO

El funcionamiento del gestor de tráfico o *Traffic Manager* (TM) se organiza en etapas, cada una con operaciones y objetivos independientes, con el propósito de mejorar la eficiencia computacional de la simulación. La Figura 7 presenta un esquema de los diferentes módulos que conforman la arquitectura del TM. A continuación, se describen los componentes más relevantes.

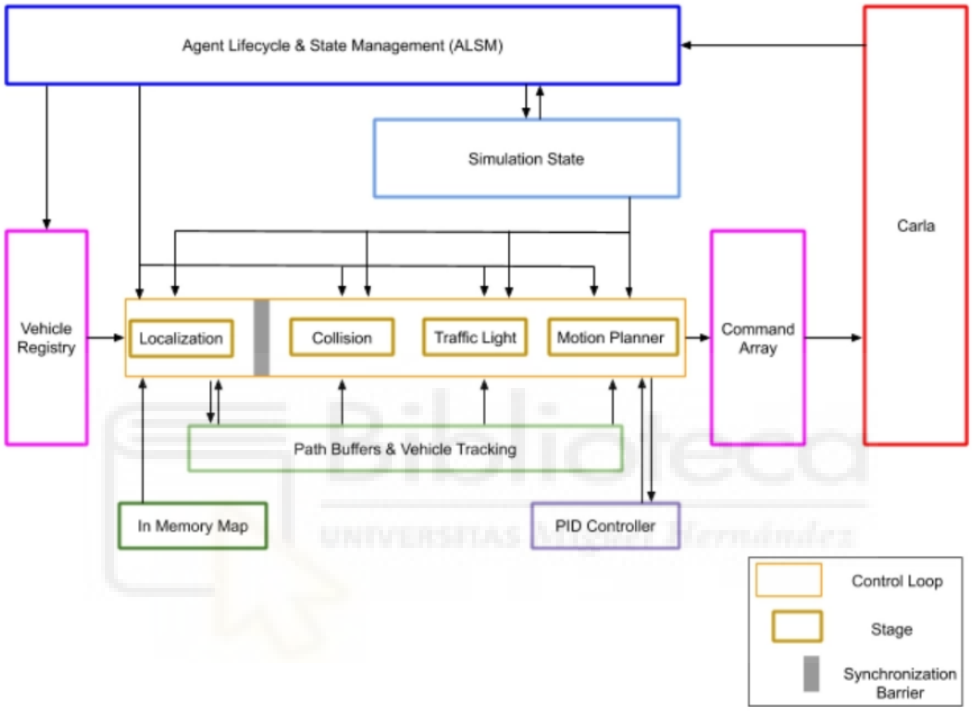


Figura 7. Diagrama de arquitectura del TM.

Almacenamiento y actualización del estado actual de la simulación:

- "*Agente Lifecycle & State Management*" (ALSM): se encarga de escanear el mundo y realizar un seguimiento de los vehículos y peatones en la escena eliminando los que ya no existan.
- "*Vehicle Registry*": contiene una serie de vehículos en piloto automático controlados por el TM y también una lista de peatones y vehículos que no están en piloto automático los cuales no están controlados por el TM.

- "*Simulation state*": es un almacenamiento en caché de la posición, la velocidad y la información adicional de todos los vehículos y peatones en la simulación.

Calcular el movimiento de cada vehículo del piloto automático:

El gestor de tráfico genera comandos para todos los vehículos en el registro de vehículos según el estado de simulación. El cálculo para cada vehículo se realiza por separado y se dividen en diferentes etapas. El lazo de control o *Control loop* se asegura de que todos los cálculos sean fiables y consistentes para crear barreras de sincronización entre las etapas, asegurándose de que ningún vehículo pasa a la siguiente etapa sin que finalicen los cálculos para todos los vehículos que se encuentran en esta etapa.

- Etapa de localización: las rutas que siguen los vehículos se crean de manera dinámica utilizando *waypoints* cercanos que son recopilados por los mapas de memoria *In-Memory Map*. Las direcciones en los cruces se eligen al azar y la ruta de cada vehículo es almacenada y mantenida por *Path Buffers & Vehicle Tracking* (PBVT).
- Etapa de colisión: los cuadros que delimitan se extienden sobre la trayectoria de cada vehículo para identificarlo y así sortear los posibles peligros de colisión.
- Etapa de semáforo: igual que la etapa de colisión, se localiza los peligros que pueden afectar a la ruta del vehículo debido a la influencia de los semáforos, señales de *stop* y prioridad en cruce.
- Etapa planificadora de movimiento: se calcula en función de la ruta definida. Se utiliza un controlador PID (Proporcional, Integral, Derivado) que determina cómo alcanzar los *waypoints* de destino.
- Etapa de luces del vehículo: las luces del vehículo se encienden o apagan dinámicamente en función de los factores ambientales simulados o el movimiento que realice el vehículo, por ejemplo, el intermitente de giro.

Aplicación de los comandos en la simulación:

Los comandos que se generan en el paso anterior se recopilan en la matriz de comandos *Command Array* y se envían al servidor de CARLA en un lote para que se apliquen en un mismo *frame* o instante del mundo.

Es importante mencionar que un administrador de tráfico está diseñado para funcionar correctamente en modo síncrono. El funcionamiento en modo síncrono de CARLA consiste en que el simulador o el mundo espera un *tick* del cliente para continuar con el siguiente paso de simulación. De esta manera, el cliente, desde la API, puede tratar, por ejemplo, información de sensores. De lo contrario, si se trabajara en modo asíncrono, podría perderse parte de la información debido a que los pasos de simulación avanzan sin tener control sobre el mundo. En el modo asíncrono, la simulación se ejecuta lo más rápido posible sin esperar al cliente, mientras que, en el modo síncrono, el cliente controla los pasos de simulación. Tanto el TM como el mundo deben configurarse en modo síncrono, como se muestra a continuación:

```
# Poner la simulación en modo de sincronización
init_settings = world.get_settings()
settings = world.get_settings()
settings.synchronous_mode = True
# After that, set the TM to sync mode
my_tm.set_synchronous_mode(True)
...

# Marque tick al mundo
world.apply_settings(init_settings)
world.tick()
```

Antes de continuar es importante aclarar cómo se gestiona el tiempo al hacer cada *tick* al mundo con la siguiente línea de código:

```
timestep = settings.fixed_delta_seconds = 0.05
```

Como se comentó, cada paso de la simulación, guiado por el servidor, es marcado por el cliente mediante un *tick* del mundo *(world tick*). Este procedimiento facilita obtener la posición exacta de la simulación al solicitar al servidor el *frame* del mundo.

En este contexto, la variable *time_step* se utiliza para establecer un paso de tiempo fijo para la simulación (*fixed_delta_seconds*). Este valor, que en este caso es de 0.05 segundos, determina la cantidad de tiempo en segundos que transcurre en la simulación entre cada *tick*. Esto implica que, si la simulación se ejecuta a su máxima capacidad, se tendrán 20 *frames* por segundos (o 20 fps) de simulación por cada segundo en tiempo real.

El registro de este paso de tiempo fijo es crucial para mantener una coherencia temporal en la simulación, y permite que los movimientos y eventos se simulen de manera precisa y coherente. Este parámetro juega un papel fundamental, especialmente cuando se simulan aspectos como la física de los vehículos, donde se requiere una alta tasa de actualización para garantizar una simulación precisa.

Cuando un cliente crea un TM, si este es el primero en conectarse a un puerto libre (si no se especifica, el 8000), se convierte en un TM Server. El TM Server dictará el comportamiento de todos los TM Clients, que son aquellos clientes que se conecten al mismo puerto posteriormente.

En el siguiente fragmento de código se puede ver la configuración de los TM, donde tm01 es un TM Server y tm02 es el TM Client:

```
tm01 = client01.get_trafficmanager() # tm01 --> tm01 (p=8000)
tm02 = client02.get_trafficmanager() # tm02() --> tm01 (p=8000)
```

3.5 ACTORES Y PLANOS

Los actores en CARLA son elementos que llevan a cabo acciones dentro del entorno simulado, y el comportamiento de cada uno puede afectar a otros actores. Entre estos actores se encuentran vehículos, peatones, sensores, señales de tráfico, semáforos e incluso el espectador (objeto que permite visualizar el mundo simulado) dentro de la simulación.

Como se mencionó anteriormente, para crear actores en el entorno, se necesitan planos, los cuales son diseños que facilitan al usuario la incorporación de nuevos actores a la simulación. Los planos incluyen modelos predefinidos con animaciones y atributos, algunos de los cuales pueden ser modificados, mientras que otros no. Por ejemplo, se puede alterar el color de un vehículo, ajustar el número de canales en un sensor LIDAR o variar la velocidad de un peatón al caminar, entre otras posibilidades.

Los planos o *blueprints* [22] se encuentran en una biblioteca que enumera los atributos que esta incluyen. En CARLA, existen dos generaciones distintas de planos: la primera abarca planos correspondientes a las versiones iniciales de CARLA, mientras que la segunda generación engloba planos actualizados con los modelos más recientes de vehículos y peatones. Se puede localizar el plano de un actor mediante identificadores o filtrando los resultados. Una vez obtenido el plano de un actor, es posible modificar aquellos atributos que sean ajustables. La Figura 8 ilustra un modelo de peatón disponible en el simulador, la Figura 9 muestra todos los diferentes tipos de peatones disponibles para su generación.

Figura 8. Actor peatón en CARLA.

Figura 9. Actores peatones de primera y segunda generación en CARLA.

3.6 CICLO DE VIDA DE LOS ACTORES

El ciclo de vida de los actores en CARLA se puede dividir en cuatro etapas principales:

1. Creación: Una vez obtenidos los planos de los actores, es esencial generarlos, o en términos técnicos, hacer un "spawn", en una ubicación y orientación específicas dentro del mundo simulado. Este proceso asegura que el lugar de aparición del actor en la simulación no provoque colisiones con la generación de otros actores. En caso de colisión, el actor no se generará.

2. Manipulación: Como se detalla en la sección 3.3.6, la clase *attachment.Type* permite vincular, asociar o acoplar actores a otros. Esta funcionalidad es particularmente útil cuando se desea, por ejemplo, unir sensores como una cámara a un vehículo. Al adjuntar un actor a otro, la posición de generación del primero será relativa al actor al que está acoplado. Todos los actores cuentan con métodos *get()* y *set()*, que facilitan la manipulación de estos en el mapa, como establecer u obtener su velocidad.

3. Interacción: Durante su "vida", los actores pueden interactuar con otros actores y con el mundo en general. Esto puede implicar colisiones, seguir rutas específicas o reaccionar a cambios en el entorno. La forma en que un actor interactúa está dictada tanto por sus propios atributos como por las reglas y condiciones establecidas en la simulación.

4. Destrucción: Finalmente, cuando ya no se necesita un actor, se debe eliminar del mundo. Esto se hace usando el método *destroy* del actor. Es importante destacar que, al culminar la simulación, los actores generados deben ser eliminados manualmente, ya que no se destruyen automáticamente. Este proceso de eliminación debe implementarse de manera explícita mediante scripting.

3.7 MAPAS EN CARLA

En CARLA, los mapas son mucho más que simples representaciones visuales del entorno; integran tanto el modelo tridimensional de una ciudad como su infraestructura vial. Esta estructura vial se basa en el estándar OpenDRIVE 1.4 [21], que establece las carreteras, carriles y cruces, y dicta el funcionamiento de la API de Python en CARLA. La API de Python actúa como un consultor de alto nivel, facilitando la navegación por calles y carreteras en la simulación.

Existen ocho escenarios preconfigurados en CARLA, denominados *Towns*. Cada uno presenta características únicas y distintivas que proporcionan una variedad de condiciones y desafíos para la simulación de la conducción autónoma. La Tabla 2. Características de los pueblos en CARLA. brinda detalles sobre las particularidades de cada uno de estos pueblos, mientras que la Figura 10 ofrece una representación gráfica de los mismos, que ofrece una visión más clara de la variedad de entornos de simulación disponibles en CARLA.

Pueblo	Característica
01	Un diseño básico de la ciudad que consta de "cruces en T".
02	Similar a Town01, pero de tamaño más reducido.
03	El pueblo más complejo, con cruce de 5 carriles, rotonda, desnivel, túnel, y más.
04	Un bucle infinito con una carretera y un pequeño pueblo.
05	Ciudad de cuadrícula con cruces y un puente. Tiene varios carriles por sentido.
06	Carreteras largas con numerosas entradas y salidas de carreteras.
07	Un entorno rural con caminos estrechos, hórreos y apenas semáforos.
08	Un entorno de ciudad con diferentes ambientes como una avenida o paseo marítimo, y texturas más realistas.

Tabla 2. Características de los pueblos en CARLA.

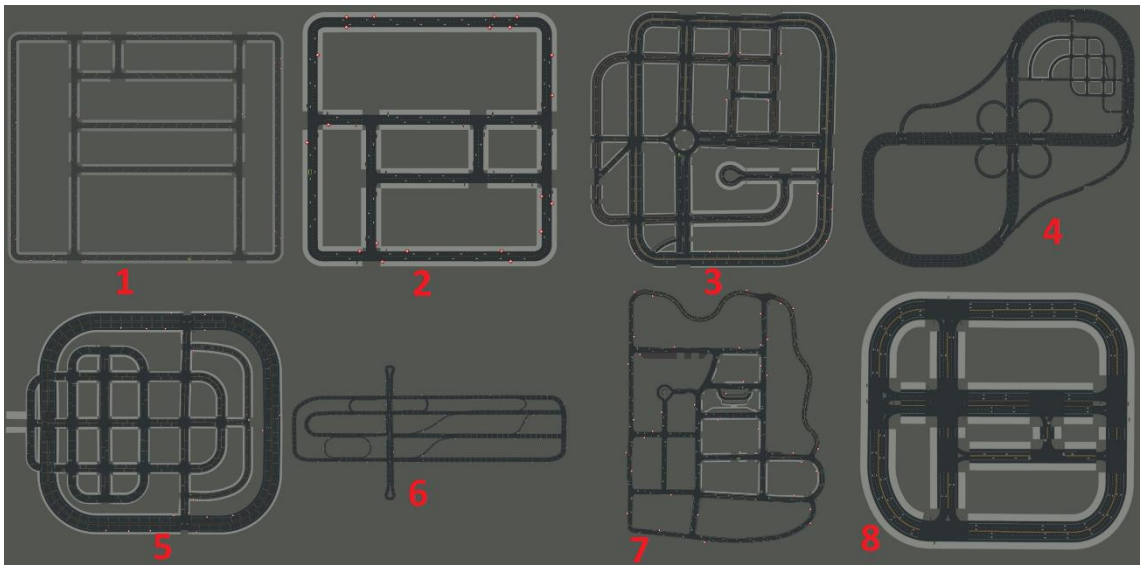


Figura 10. Pueblos disponibles en CARLA.

En cada uno de los pueblos, se dispone de dos tipos de mapas en los que se pueden activar o desactivar diferentes capas. Las capas son los objetos que conforman lo que se ve en el mundo simulado, como edificios, árboles, vehículos estacionados, farolas, paredes, etc. Estas capas se pueden activar o desactivar mediante la API, aunque siempre se mantiene un diseño mínimo que incluye carreteras, aceras, semáforos y señales de tráfico. La Figura 11 y Figura *12* muestran un ejemplo de un mapa con capas activadas y desactivadas, respectivamente.

Figura 11. Mapa con capas.

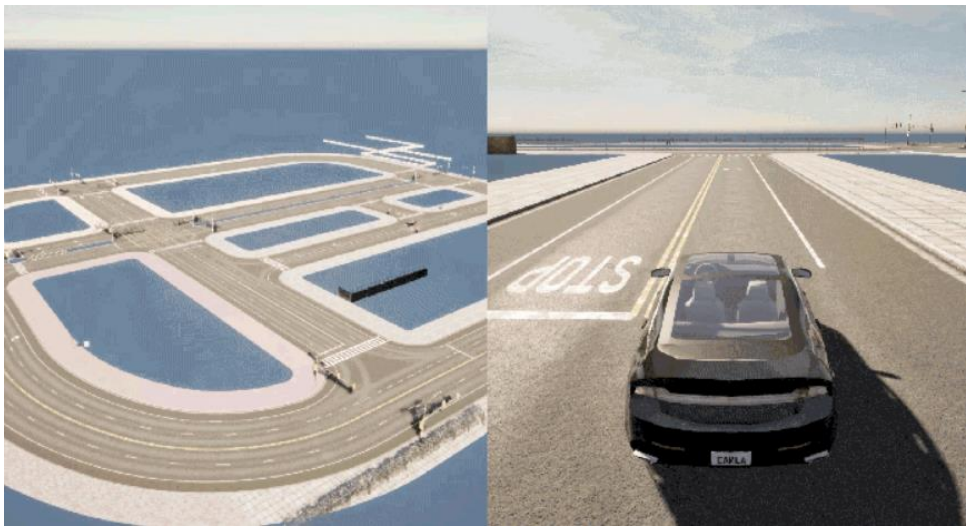


Figura 12. Mapa sin capas.

3.8 SENSORES DEL VEHÍCULO

Los sensores en CARLA son un tipo de actor que se encarga de recopilar información del entorno. Estos sensores desempeñan un papel fundamental en la detección del mundo en el que se desplaza el vehículo y, mediante técnicas de fusión de datos, contribuyen a la percepción en los vehículos autónomos, permitiendo la creación de mensajes (CAM, DENM) para la percepción cooperativa. En CARLA, existen diversos tipos de sensores, como cámaras y detectores. Los sensores disponibles en la versión 0.9.13 de CARLA incluyen:

- Detector de colisiones
- Cámara de profundidad
- Sensor GNSS
- Sensor IMU
- Detector de invasión de carril
- Sensor LIDAR
- Detector de obstáculos
- Sensor de radar
- Cámara RGB
- Sensor RSS

- Sensor LIDAR semántico
- Cámara de segmentación semántica
- Cámara de segmentación de instancias
- Cámara DVS
- Cámara de flujo óptico

Los sensores en CARLA están implementados dentro del motor gráfico que utiliza el simulador, el cual es Unreal Engine 4 (UE4). El funcionamiento del sensor se lleva a cabo dentro del marco de UE4 y transmite datos al cliente. Estos datos son recibidos a través de la API de Python, como se ilustra en la Figura 13. Este proceso permite una comunicación eficiente y fluida entre los sensores y el cliente, facilitando el análisis y la utilización de los datos recopilados por los sensores en tiempo real.

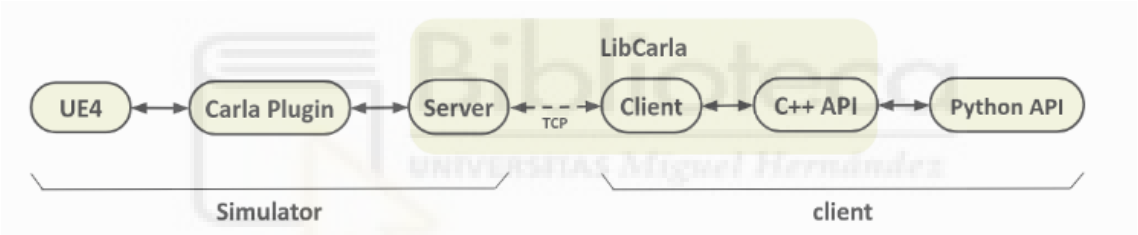


Figura 13. Esquema de comunicación de los sensores en CARLA.

Existen diversos tipos de sensores en CARLA. Algunos de ellos generan datos de forma continua cada vez que se actualiza el sensor, mientras que otros producen datos solo cuando ocurre un evento específico. Los datos son capturados en el marco de UE4 (Unreal Engine 4) y se serializan del lado del servidor para luego deserializarse del lado del cliente. La información obtenida representa un objeto con los datos generados por el sensor, que se puede acceder en la API de Python a través de métodos de escucha o *listen*, que capturan los eventos devueltos por el sensor con el objetivo de procesarlos y utilizar la información adquirida.

Un sensor puede asociarse a un vehículo para desplazarse junto a él y capturar información del entorno. Tal como se mencionó anteriormente, en la preparación del script se añadió un argumento --rgb_ego para seguir visualmente la trayectoria del

vehículo ego. Al proporcionar este parámetro de entrada, se asocia un sensor de tipo cámara RGB a un actor vehículo específico, como se muestra en la Figura 14. Generalmente, todos los sensores se adjuntan a un actor vehículo; por ejemplo, en la Figura 15, se vinculó un sensor LIDAR a un vehículo.

Figura 14. Cámara RGB asociada a vehículo

Figura 15. Nube de puntos de LIDAR asociado a un vehículo.

Los sensores en CARLA, como todo actor se encuentran en una biblioteca de planos (*blueprints*) accesible gracias a la exposición de los sensores implementados en UE4. Esto permite que, desde el lado del cliente (API de Python), podamos acceder a ellos y configurar los parámetros expuestos o atributos de entrada del sensor.

Es importante recordar que los planos contienen información de los actores, incluidos los sensores donde cada sensor cuenta con atributos de configuración que permiten personalizar su funcionamiento.

4 MODELADO, SIMULACIÓN Y CONECTIVIDAD V2X

La conectividad V2X es esencial para la percepción cooperativa y el mejoramiento de las capacidades de percepción de vehículos autónomos a través de sistemas de comunicaciones móviles e inalámbricas. Por lo tanto, el estudio de la percepción cooperativa requiere que las herramientas de simulación incorporen el modelado de conectividad. No obstante, las herramientas de modelado de entornos 3D y conducción autónoma carecen de dicha conectividad V2X.

En esta sección del trabajo, se presenta la implementación de capacidades de comunicación V2X en el simulador 3D virtual CARLA, y cómo se han creado diferentes escenarios para su evaluación. La integración de la comunicación V2X en CARLA es fundamental para estudiar las comunicaciones y la percepción cooperativa, lo que implica ampliar las funcionalidades y mejorar las capacidades de plataformas de simulación existentes.

La sección 4.1 detalla la implementación del escenario para ejecutar las simulaciones. En esta sección, se introduce la clase principal *world*, responsable de la inicialización y configuración de la simulación. Se comenta las partes implementadas más relevantes. También se destacan las modificaciones implementadas y los argumentos adicionales resultantes de la personalización del script *generate_traffic.py* para adaptar la aplicación a nuestras necesidades.

En la sección 4.2 se presentan los detalles de la implementación de los sensores utilizados en este trabajo, específicamente un detector de obstáculos y una cámara RGB. La elección del detector de obstáculos se debe a su capacidad para generar un evento cada vez que un obstáculo se detecta a una distancia específica, y la simplicidad que esto aporta para probar el proceso de comunicación entre vehículos en la simulación. También se

proporciona una descripción detallada de la asignación de la cámara y su configuración, que permite visualizar el recorrido de los vehículos y registrar información relevante.

La sección 4.3 presenta la arquitectura de alto nivel adoptada para implementar la conectividad V2X en CARLA, explicando su funcionamiento y conceptos clave. Esta arquitectura está diseñada para servir como base para futuros estudios en percepción cooperativa y comunicaciones vehiculares en simuladores de alta fidelidad.

La sección 4.4 describe los pasos que se llevaron a cabo para realizar la implementación de los escenarios, las descripciones de cada uno y la selección de los mapas para llevar a cabo las simulaciones, así como también los parámetros utilizados.

La sección 4.5 describe de forma general el funcionamiento del módulo implementado a través del script *v2x.py* que representa el módulo de conectividad V2X que, entre otras funciones, gestiona y controla la información de los vehículos en la simulación para establecer una comunicación V2X.

La sección 4.6 presenta el modelo analítico utilizado y muestra los pasos realizados para implementar un método de visualización en tiempo real de una gráfica PDR empírica basada en este modelo.

Por último, la sección 4.7 presenta el escenario de simulación considerado, y los resultados obtenidos integrando el módulo de comunicación V2X implementado.

4.1 IMPLEMENTACIÓN DE ESCENARIO

Para los propósitos de este trabajo, se llevó a cabo una modificación del script *generate_traffic.py* para adaptarlo a los requisitos específicos del proyecto. Esta modificación permite aprovechar las funcionalidades ya incorporadas y, además, añadir nuevas capacidades, como la comunicación.

Este script no solo mantiene la función de configurar un TM que controle los vehículos generados, sino también la creación de otros actores como sensores. Recordemos que el administrador de tráfico (TM) es el módulo que controla los vehículos durante la simulación en modo de piloto automático, de manera que se pueda simular un entorno realista que se asemeje al tráfico urbano real. Además, permite modificar, si es necesario, a través de la API de Python, el comportamiento de los vehículos en la simulación, como activar o desactivar las colisiones entre vehículos, forzar cambios de carril o establecer distancia mínima entre vehículos.

El escenario implementado en este trabajo, al modificar el *script generate_traffic.py*, se evita utilizar una configuración con múltiples administradores de tráfico en un tipo de arquitectura multi-cliente. Al decidir integrar todo en un único *script*, se simplifica la gestión de la simulación y se evita la necesidad de ejecutar múltiples *scripts* para controlar diferentes aspectos de la simulación, como la generación de tráfico o la gestión de la climatología. Además, al implementar un solo TM servidor, se garantiza un control centralizado sobre todos los vehículos y actores en la simulación.

El *script v2x.py* desarrollado además de contar con sus propios módulos que se detallarán más adelante, genera una instancia administradora de tráfico que gestionará todos los vehículos en piloto automático que se generen en el mundo para tener el control de todos a lo largo de la simulación, requisito importante para poder evaluar la interacción entre los vehículos.

La mayoría de los argumentos predefinidos en el *script generate_traffic.py* no se utilizan para el propósito de este trabajo, pero se decidió mantenerlos para futuros proyectos. Además de los argumentos propios del script original, se añadieron los siguientes parámetros con su respectiva lógica:

- *--time-seconds:* Este parámetro permite definir el tiempo total que durará la simulación, expresado en segundos. Al ajustar este valor, se determina cuánto tiempo se ejecutará la simulación, lo que facilita el análisis de diferentes escenarios y situaciones a lo largo del tiempo establecido.

- --*distance_detected*: Este parámetro permite definir la distancia máxima a la cual un vehículo puede identificar un obstáculo en su trayectoria.
- --*ego*: Establecimiento de un vehículo ego: Un vehículo ego se refiere a un actor prominente dentro de una simulación. Al configurar un vehículo como ego, se le asigna automáticamente una etiqueta denominada "Ego Vehicle" a uno de los N vehículos presentes en la simulación mediante código. Por ejemplo, si establecemos mediante parámetros que la simulación incluya diez vehículos, uno de ellos será designado como vehículo ego. Esto permite identificar fácilmente un vehículo específico entre los N vehículos agregados al entorno simulado y llevar a cabo acciones pertinentes.
- --*rgb_cam*: Visualización del vehículo en tiempo real. Se incorporó esta función con el objetivo de facilitar el seguimiento visual de los vehículos. Al activar esta característica, se vincula una cámara RGB al vehículo, la cual sigue su trayectoria a lo largo del mapa, permitiendo así una observación detallada de su desplazamiento.
- --*res*: Ajuste de la resolución de la ventana flotante para visualizar el vehículo ego. Es posible modificar el tamaño de la ventana donde se visualiza la cámara asociada al vehículo ego, permitiendo así adaptar la visualización según las preferencias o necesidades del usuario.
- --*pdr*: Activación de la visualización en tiempo real de la gráfica PDR (*Packet Delivery Ratio*). Al habilitar este parámetro, se puede observar en tiempo real la evolución de la curva PDR, la cual representa la probabilidad de recepción de paquetes entre dos vehículos en función de la distancia que los separa.
- --*pot-trans:* Este argumento permite establecer la potencia de transmisión en dBm utilizada en la comunicación V2X. La potencia de transmisión determinará el vector de PDR a emplear en la simulación.
- --*num-sub:* Este parámetro permite definir el número de subcanales a utilizar en la comunicación V2X. El número de subcanales establecido influirá en el vector PDR que se aplicará en la simulación.
- --*rate:* Este argumento permite establecer la tasa de paquetes en Hercios para la comunicación V2X. El valor de la tasa de paquetes determinará el vector PDR a utilizar en la simulación.

- *--beta:* Este parámetro define la densidad de vehículos en veh/m presente en el entorno simulado. El argumento beta influye en la PDR que se utilizará durante la simulación, ajustando así el desempeño de la comunicación entre vehículos en función de la densidad vehicular.

En resumen, al modificar el script *generate_traffic.py* y fusionarlo con la lógica de la tarea en *v2x.py*, se logra lo siguiente:

1. Se crea un único script que se encarga de la generación de tráfico, la creación de vehículos y peatones, y la ejecución de la lógica de la tarea.
2. Se evita la necesidad de ejecutar múltiples scripts y de conectar diferentes TM clientes al TM servidor para controlar diferentes aspectos de la simulación.
3. Se asegura un control centralizado de todos los actores en la simulación a través del TM servidor.
4. Se añade funcionalidad extra al crear un módulo de comunicación.

Este enfoque además de las funcionalidades añadidas permite una mayor simplicidad y un mejor control sobre la simulación, haciendo que sea más fácil de gestionar y de adaptar a las necesidades específicas del trabajo. También permite una mayor flexibilidad, ya que se pueden ajustar fácilmente los argumentos de entrada y modificar el script para adaptarlo a diferentes escenarios de simulación.

Todos los argumentos de configuración, en primer instancia, son utilizados por una función principal que es la encargada de inicializar todos los valores. Esta función forma parte de una clase llamada *World*, que sirve como controlador de un entorno de simulación en CARLA. La inicialización y configuración de la simulación recae en una función principal, que forma parte de la clase denominada *World*. Esta clase es la representación de un entorno de simulación, y actúa como el controlador principal de dicho entorno. Esta clase se sirve de una serie de argumentos que se procesan en su método de inicialización, para definir las propiedades y el comportamiento del mundo de la simulación.

Los argumentos de configuración son utilizados para establecer múltiples aspectos de la simulación, como las propiedades físicas del mundo, el comportamiento de los vehículos y peatones, y la configuración de la interfaz gráfica, entre otros. Esto permite adaptar la simulación a las necesidades específicas de cada situación, otorgando una gran flexibilidad al sistema.

La clase World, y en particular su método de inicialización se posiciona como el núcleo de la simulación. Todas las demás funciones y métodos que configuran y manipulan el entorno de la simulación se ramifican a partir de este núcleo. Sin embargo, dado el alcance y la profundidad de estos métodos, nos enfocaremos principalmente en el funcionamiento general de la clase y sus aspectos más relevantes, en lugar de profundizar en cada una de las funciones individuales que se desprenden de ella.

Una de las primeras acciones es que a una función llamada *get_pdr_v2v* se le pasa los argumentos proporcionados por el usuario que permiten seleccionar la PDR correcta para el uso en la simulación.

Posteriormente, se establece una conexión con el servidor de CARLA y se configuran varios parámetros en el gestor de tráfico del simulador. Dependiendo de los parámetros de configuración, el entorno de simulación puede operar en modo síncrono, que proporciona un control preciso sobre el tiempo de simulación, o en modo asíncrono, que permite que la simulación avance al ritmo de tiempo real.

El siguiente fragmento de código muestra la inicialización de la clase y cómo posteriormente se llama a la función *get_pdr_v2v* quien contiene todos las PDR del modelo:

```
class World(object):
    def __init__(self, args):

        # Obtener datos de curva PDR-----------------------------------------------
        read_pdr = get_pdr_v2v(float(args.beta), int(args.pot_trans),
        int(args.num_sub), int(args.rate))

       sensor_queue = Queue()
       obstacle_sensor = []
       vehicles_list = []
       walkers_list = []
       all_id = []
```

Además, se muestra cómo se declaran ciertas listas y una cola. Dichas listas se utilizan para guardar los peatones y vehículos generados, así como también su identificación. También se crea una cola que permite guardar información devuelta por el sensor de obstáculos.

También se configura el gestor de tráfico y los pasos de simulación. El siguiente fragmento de código muestra la creación de una instancia del gestor de tráfico, la configuración para que la simulación se ejecute en modo síncrono, y se establecen los pasos fijos de simulación en 0,05 segundos (20 fps). Este "paso de tiempo fijo" se puede ajustar en la configuración del mundo. En este caso, el simulador tomará veinte pasos (1/0.05) para recrear un segundo del mundo simulado. Con un paso de tiempo fijo, si se requiere recrear la simulación se tendrá más facilidades que con una configuración de tiempo variable ya que el servidor puede configurarse con el mismo paso de tiempo que se utilizó en la simulación original.

No está demás aclarar que también se puede ejecutar la simulación en modo asíncrono, hay que recordar que se mantiene parte de la configuración del script original. A continuación, se muestra un fragmento de código que muestra la configuración.

```
try:
            client = carla.Client(args.host, args.port)
            client.set_timeout(20.0)
            world = client.get_world()
            synchronous_master = False
            traffic_manager = client.get_trafficmanager(args.tm_port)

            if not args.asynch:
                traffic_manager.set_synchronous_mode(True)
                if not settings.synchronous_mode:
                    synchronous_master = True
                    settings.synchronous_mode = True
                    time_step = settings.fixed_delta_seconds = 0.05
                else:
                    synchronous_master = False
```

Una vez establecida la conexión con el servidor y configurado el gestor de tráfico, la función procede a crear los actores en el entorno de simulación, que pueden ser vehículos o peatones. La creación de estos actores se realiza mediante funciones auxiliares que toman como entrada los planos (*blueprints*) de los actores, que se obtienen utilizando los filtros y generaciones especificados en los argumentos de configuración. El usuario puede configurar, por ejemplo, qué tipo de modelo de vehículos se inician en la simulación, así como también el aspecto o tipo de peatones.

Las siguientes líneas de código muestran cómo se obtienen los *blueprints.*

```
blueprints = get_actor_blueprints(world, args.filterv, args.generationv)

blueprintsWalkers = get_actor_blueprints(world, args.filterw, args.generationw)
```

En este caso la primera línea corresponde a la obtención de los modelos para los vehículos y la siguiente para los peatones. Estos planos se pasarán a las funciones encargadas de generar los actores en el mundo.

```
spawn_vehicles(world, client, args, traffic_manager, blueprints, vehicles_list,
               SpawnActor, SetAutopilot, FutureActor, synchronous_master)

spawn_walkers(world, args, blueprintsWalkers, client, SpawnActor, walkers_list,
              all_id, synchronous_master)
```

Se puede observar que al llamar a las funciones se envía la configuración del mundo (*world*), los argumentos proporcionados por el usuario (*args*), los planos de los actores (*blueprints*), la lista para almacenar los actores generados entre otros parámetros, como indicar que los actores se muevan automáticamente por el mundo (*SetAutopilot*), referencias de la clase del actor (*SpawnActor*) e indicación de que se utiliza el modo síncrono (*synchronous_master*).

Posteriormente se llama a la función *ObstacleSensor*, que se encarga de asociar los sensores de obstáculos a los vehículos. Esta función toma como argumentos la lista de vehículos creados, una cola de sensores y el gestor de tráfico:

```
ObstacleSensor(args, world, vehicles_list, sensor_queue, traffic_manager,
               obstacle_sensor))
```

Finalmente, la función entra en un bucle de juego (*game_loop*), que constituye el núcleo de la simulación. Dentro de este bucle, se simula el comportamiento de los vehículos y peatones en el entorno, y se recogen datos sobre su desempeño. El bucle de juego puede funcionar con diferentes configuraciones dependiendo de los argumentos proporcionados, incluyendo la activación de una cámara RGB para capturar imágenes de la simulación y la habilitación de la recopilación de datos. La función recibe como parámetros los argumentos iniciales (*args*), los datos del mundo generado (*world*), una serie de parámetros (*camera, actor, camera_bp, camera_init_trans, all_vehicle_actors*) relacionados con la opción de asociar una cámara RGB a un vehículo ego, la pila FIFO (*sensor_queue*) y un valor booleano que indica si el *script* se ejecuta en modo síncrono (*synchronous_master*).

En el programa, se inician varios procesos en función de los argumentos seleccionados. Si se ha optado por utilizar tanto la cámara como la funcionalidad PDR en tiempo real, el código inicializa y configura la cámara y prepara la funcionalidad PDR. Una vez preparados estos componentes, se recopila una lista de todos los vehículos en la simulación y se pone en marcha el bucle principal del juego.

A continuación, se muestra un bloque del código que lanza el bucle de juego teniendo en cuenta las configuraciones del usuario.

```
if args.rgb_ego and args.pdr:
                all_vehicle_actors = world.get_actors(vehicles_list)
                camera, actor, camera_bp, camera_init_trans = CameraSensor(world,
                vehicles_list, args)
                game_loop(args, world, camera, actor, camera_bp, camera_init_trans,
                          obstacle_sensor, sensor_queue, synchronous_master,
                          all_vehicle_actors)
elif args.rgb_ego:
                all_vehicle_actors = world.get_actors(vehicles_list)
                camera, actor, camera_bp, camera_init_trans = CameraSensor(world,
                vehicles_list, args)
                game_loop(args, world, camera, actor, camera_bp, camera_init_trans,
                          obstacle_sensor, sensor_queue, synchronous_master,
                          all_vehicle_actors)
elif args.pdr:
                camera = None
                actor = None
                camera_bp = None
                camera_init_trans = None
                all_vehicle_actors = world.get_actors(vehicles_list)
                game_loop(args, world, camera, actor, camera_bp, camera_init_trans,
                          obstacle_sensor, sensor_queue, synchronous_master,
                          all_vehicle_actors)
else:
                camera = None
                actor = None
                camera_bp = None
                camera_init_trans = None
                all_vehicle_actors = world.get_actors(vehicles_list)
                game_loop(args, world, camera, actor, camera_bp, camera_init_trans,
                          obstacle_sensor, sensor_queue, synchronous_master,
                          all_vehicle_actors)
```

En el bloque de código anterior, se inician varios procesos en función de los argumentos seleccionados. Si se ha optado por utilizar tanto la cámara como la funcionalidad PDR en tiempo real, el código inicializa y configura la cámara y prepara la funcionalidad PDR. Una vez preparados estos componentes, se recopila una lista de todos los vehículos en la simulación y se pone en marcha el bucle principal del juego.

En caso de que sólo se haya elegido utilizar la cámara, sin la funcionalidad PDR, el código configura y activa la cámara, pero no realiza ninguna preparación relacionada con la funcionalidad PDR. Tras configurar la cámara, al igual que en el caso anterior, se obtiene una lista de los vehículos presentes en la simulación y se inicia el bucle principal del juego.

Por otro lado, si la elección fue activar la funcionalidad PDR, pero no la cámara, el código se centra únicamente en la preparación y activación de la funcionalidad PDR, sin realizar ninguna configuración de la cámara. Finalmente, se recopila una lista de los vehículos presentes en la simulación y se pone en marcha el bucle principal del juego.

Finalmente, en caso de que no se haya seleccionado el uso de ninguna de las dos funcionalidades, ni la cámara ni la funcionalidad PDR, el código omite estas preparaciones e inicializaciones y pasa directamente a recoger una lista de los vehículos en la simulación e iniciar el bucle principal del juego. Este último caso demuestra que la simulación puede funcionar con un conjunto mínimo de funcionalidades, sin necesidad de contar con una cámara o la funcionalidad de la PDR en tiempo real.

Antes de finalizar la simulación, la función realiza una serie de operaciones de limpieza para liberar los recursos utilizados durante la simulación, de esta manera se evita que quede memoria residual que genere problemas al lanzar las siguientes simulaciones. La limpieza consiste en eliminar todos los actores que intervienen en la simulación. Hay que recordar nuevamente que los actores incluyen a los vehículos, peatones y sensores que se han añadido al mundo.

Para ello los actores creados son destruidos, los sensores se desactivan y se cierran otros recursos como la cámara si el usuario ha decidido que estas sean utilizadas. El siguiente bloque de código muestra cómo se recorre mediante bucles los actores del mundo para ser eliminados.

```
finally:
            print('\nDestruyendo %d vehículos' % len(vehicles_list))
            client.apply_batch([carla.command.DestroyActor(x) for x in
            vehicles_list])
            for i in range(0, len(all_id), 2):
                all_actors[i].stop()
            for sensor in obstacle_sensor:
                sensor.destroy()
            if args.rgb_ego:
                # # Detener la cámara y salir de PyGame después de salir del bucle
                camera.stop()
                pygame.quit()
                dist_empirical.clear()
                pdr_empirical.clear()
            print('\nDestruyendo %d peatones' % len(walkers_list))
            client.apply_batch([carla.command.DestroyActor(x) for x in all_id])
```

Una vez eliminado los actores, se genera un archivo CSV con los datos recogidos durante la simulación, y se guarda un archivo de texto con la configuración utilizada, es decir, se guardan todos los parámetros configurados por el usuario. Estos archivos sirven como registro de la simulación, proporcionando datos valiosos para el análisis posterior. Los datos guardados en el archivo de texto son el nombre del mapa, el número de *frames*, el número de vehículos, el tiempo en segundos que dura la simulación y parámetros relevantes configurados por el usuario como por ejemplo los que determinan la PDR a utilizar: la densidad de vehículos, la potencia de transmisión, el número de subcanales y la tasa de paquetes o configuraciones generales como si se ha configurado el haya un vehículo ego o el uso de la cámara RGB. Se registra también en este archivo el porcentaje total de mensajes recibidos o no recibidos en la simulación por los vehículos.

El archivo CSV almacena los datos correspondientes a los vehículos tras la detección de un obstáculo. En dicho archivo, se registra el fotograma (*frame*) en que se ha producido la detección, el modelo del vehículo o el nombre de rol (si el usuario ha configurado el uso del vehículo ego, este se etiqueta como "Ego vehicle"), y el ID del agente que realizó la detección. Además, se documentan los mismos datos para los vehículos que son capaces de recibir el mensaje de detección proveniente de otro vehículo. En estos casos, se registra también la probabilidad PDR y si el vehículo puede recibir o no el mensaje.

En la sección 4.4 se proporciona un ejemplo de este archivo al describir el funcionamiento del módulo de conectividad V2X. El siguiente fragmento de código demuestra la implementación que permite la generación de estos archivos CSV y TXT.

```
# ------------------------------CREAR ARCHIVO CSV y TXT------------------------------
            df = pd.DataFrame(
                {'Nº frame': frame, 'Transmisor': transmisor, 'ID transmisor': id_t,
                 'Modelo Transmisor': modelo_t,
                 'Receptor': receptor, 'ID receptor': id_r, 'Modelo Receptor':
                  modelo_r, 'Distancia': dist,
                  'Probabilidad aleatoria': prob_aleatory, 'Probabilidad analítica':
                  prob_analytical, 'Recibe': recibe})
            try:
                # Guardar la configuración en un archivo de texto
                with open(config_file_path, "w") as file:
                    file.write("Mapa: {}".format(map_name))
                    file.write("Número de
                     frames:{}\n".format(world.get_snapshot().frame))
                    file.write("Número de vehículos: {}\n".format(number_vehicles))
                    file.write("\nCONFIGURACIÓN DEL PROGRAMA:\n")
                    file.write("-ts, --time-seconds: {}\n".format(args.time_seconds))
                    file.write("--ego: {}\n".format(args.ego))
                    file.write("--rgb_ego: {}\n".format(args.rgb_ego))
                    file.write("-pt, --pot-trans: {}\n".format(args.pot_trans))
                    file.write("-ns, --num-sub: {}\n".format(args.num_sub))
                    file.write("-r, --rate: {}\n".format(args.rate))
```

```
                    file.write("-beta, --beta: {}\n".format(args.beta))
                    file.write("\nPORCENTAJES:\n")
                    file.write("Porcentaje de mensajes recibidos:
                    {:.2f}%\n".format(porcentaje_mensajes_recibidos))
                    file.write("Porcentaje de mensajes no recibidos:
                    {:.2f}%\n".format(porcentaje_mensajes_no_recibidos))
                print("Se ha creado el archivo de configuración con éxito en:",
config_file_path)
```

Finalmente se detienen los hilos paralelos al programa principal y se da por finalizada la simulación:

```
# Esperar a que el hilo frame_monitor termine
os._exit(0)
frame_monitor.join()

# Detener los hilos
stop_event_frame_monitor.set()

# # Después de detener los hilos
# hilo_colector.join()
# hilo_pdr.join()
os._exit(0)
```

4.2 IMPLEMENTACIÓN DE SENSORES

Se implementaron dos tipos de sensores en este trabajo: Un detector de obstáculos y una cámara RGB. La elección del detector de obstáculo se debe a que genera un evento cada vez que un obstáculo se encuentra a una distancia determinada del actor al cual se le ha asociado el sensor. En comparación con otros sensores disponibles, se optó por este sensor por su simplicidad, ya que el objetivo era que, tras un evento generado por un vehículo (con el sensor asociado), éste pudiera comunicar a los demás vehículos la detección del evento.

De esta forma, se evitó el tratamiento de información compleja y la carga computacional que pueden generar otros tipos de sensores. Esto resulta especialmente importante en nuestro caso, ya que se requiere la asociación de un número N de sensores a un número N de vehículos generados en la simulación. Las Tabla 3 Tabla 4 muestran los atributos de entrada y salida, respectivamente, para el detector de obstáculos. Estas tablas permiten conocer en detalle las características y capacidades del sensor, facilitando su configuración y uso en el contexto del proyecto.

Atributos de entrada	Descripción
distance	Distancia para detectar.
hit_radius	Radio de detección.
only_dynamics	Tener en cuenta objetos dinámicos
debug_linetrace	Radio de detección visible
sensor_tick	Segundos de simulación entre capturas del sensor

Tabla 3. Atributos de entrada de detector de obstáculos.

Atributos de salida	Descripción
frame	Número de frame cuando se realizó la medición
timestamp	Tiempo de simulación en segundos desde el inicio del episodio.
transform	Ubicación y rotación en coordenadas mundiales del sensor en el momento de la medición
actor	Actor padre que detectó el obstáculo
other_actor	Actor detectado como obstáculo
distance	Distancia de actor a other_actor

Tabla 4. Atributos de salida de detector de obstáculos.

En este mundo simulado, se generan N vehículos y N sensores asociados mediante un cliente en el script *v2x.py*. La API gestiona todos los objetos devueltos por los sensores al generar eventos y permite obtener información de los actores involucrados. Si conocemos el número de vehículos y sus posiciones en cada paso de simulación (*frame*), podemos calcular las distancias relativas entre ellos en todo momento. Así, mediante *scripting*, es posible realizar un estudio sobre comunicación V2X y percepción cooperativa.

El detector de obstáculos, unido al actor principal, detecta obstáculos usando una cápsula en UE4, similar a la detección de distancia segura con un área rectangular, para evaluar la proximidad en el entorno simulado, como se muestra en la Figura 16.

Figura 16. Detector de distancia segura.

Para la implementación del detector de obstáculos se creó una función *ObstacleSensor* que se encarga de inicializar y configurar un sensor de obstáculos para cada vehículo en la lista de vehículos *vehicles_list*. Primero, se extraen las características del sensor de la biblioteca de planos de CARLA. A continuación, se establecen una serie de atributos para el sensor, como el radio de detección (*hit_radius*), la detección solo de objetos dinámicos (*only_dynamics*) y la distancia a la que se detectarán los obstáculos (*distance_detected*).

Se usa un bucle para recorrer todos los vehículos en la simulación y se asocia el sensor de obstáculos a cada uno. Si el vehículo es el vehículo ego, se asigna un *role_name* de 'Ego Vehicle'. Para todos los demás vehículos, se asigna un *role_name* de 'Vehículo i', donde i es el número del vehículo en la simulación.

Por último, se agrega un oyente para cada sensor que llama a la función *sensor_callback* cuando se detecta un obstáculo. Cada sensor en CARLA devuelve información, y para procesarla, es necesario implementar métodos de *callback*. Estos métodos permiten tratar la información recibida. Por ejemplo, si deseamos obtener el objeto que devuelve un sensor tras una detección, debemos emplear una función lambda de escucha. Esta función recogerá y procesará los datos obtenidos por el sensor y los agregará a la cola de sensores (*sensor_queue*) para su posterior análisis.

Las siguientes líneas de código muestran cómo se realiza lo anteriormente comentado.

```
def ObstacleSensor(args, world, vehicles_list, sensor_queue, traffic_manager,
    obstacle_sensor):

    bp_lib = world.get_blueprint_library()
    obstacle_bp = bp_lib.find('sensor.other.obstacle')
    obstacle_bp.set_attribute('hit_radius', '0.5')
    obstacle_bp.set_attribute('only_dynamics', 'true')
    obstacle_bp.set_attribute('distance', str(args.distance_detected))
    ego = args.ego
    all_vehicle_actors = world.get_actors(vehicles_list)
    i = 0
    all_vehicle_actors = world.get_actors(vehicles_list)
    for actor in all_vehicle_actors:
        if ego:
            obstacle_bp.set_attribute('role_name', 'Ego Vehicle')
            ego = False
        else:
```

```
            obstacle_bp.set_attribute('role_name', 'Vehículo ' + str(i + 1))


        obstacle_sensor.append(world.spawn_actor(obstacle_bp, carla.Transform(),
        attach_to=actor))
        obstacle_sensor[i].listen(
            lambda data: sensor_callback(data, sensor_queue))
        i = i + 1
```

Otro sensor implementado es una cámara RGB. La asignación de la cámara depende de la configuración del usuario. Específicamente, si el usuario elige tener un vehículo ego, se le asignará una cámara exclusivamente a dicho vehículo. Esto permite visualizar el recorrido del vehículo ego a través del mapa. Además, tras la detección de un obstáculo, se registra información relevante sobre el vehículo ego y las detecciones que ha realizado durante la simulación.

Para poder hacer uso de la cámara se creó una función llamada *CameraSensor* que se encarga de asociarla a los vehículos. En la función *CameraSensor*, se inicializa y configura una cámara RGB para cada vehículo en la lista *vehicles_list*. Al igual que en la función del sensor de obstáculos, se extraen las características de la cámara de la biblioteca de planos de CARLA. Luego, se establece el tamaño de la imagen que generará la cámara (*image_size_x e image_size_y*) que el usuario puede escoger. Después, se usa un bucle para recorrer todos los vehículos en la simulación. Si el vehículo es el vehículo ego y se ha establecido el parámetro ego en *True*, se asocia la cámara con el vehículo ego. En caso contrario, se asocia la cámara con el primer vehículo de la lista. Finalmente retorna la información necesaria que necesita la función encargada de la ejecución de la simulación *game_loop* para mostrar la trayectoria del vehículo.

El código siguiente muestra la función *CameraSensor* que permite realizar las funciones anteriormente descritas:

```
def CameraSensor(world, vehicles_list, args):
    bp_lib = world.get_blueprint_library()
    camera_init_trans = carla.Transform(carla.Location(x=-5, z=3),
    carla.Rotation(pitch=-20))
    camera_bp = bp_lib.find('sensor.camera.rgb')
    args.width, args.height = [int(x) for x in args.res.split('x')]
    camera_bp.set_attribute('image_size_x', f'{args.width}')
    camera_bp.set_attribute('image_size_y', f'{args.height}')

    all_vehicle_actors = world.get_actors(vehicles_list)
    for actor in all_vehicle_actors:
        if args.ego:
            if (actor.attributes["role_name"] == 'Ego Vehicle'):
                camera = world.spawn_actor(camera_bp, camera_init_trans,
                attach_to=actor)
                break
        else:
            camera = world.spawn_actor(camera_bp, camera_init_trans, attach_to=actor)
            break

    return camera, actor, camera_bp, camera_init_trans
```

Al final de la función, se devuelven varias variables: la cámara creada, el vehículo al que se adjuntó la cámara, el plano de la cámara (*blueprint*) y la transformación inicial de la cámara (la posición relativa de la cámara al vehículo).

El *callback* de la cámara se declara de la siguiente manera*:*

```
camera.listen(lambda image: pygame_callback(image, renderObject))
```

La función *listen* en la cámara se llama cada vez que la cámara tiene una nueva imagen para procesar. Se le pasa una función lambda, que a su vez llama a *pygame_callback* con la imagen de la cámara y el objeto *renderObject* como argumentos.

La función *pygame_callback* toma los datos en bruto que la cámara genera, los procesa y los prepara para ser visualizados en la pantalla, facilitando la representación gráfica de las imágenes obtenidas en la simulación. De manera resumida, cada vez que se generen datos se remodela una imagen RGB en 2D que permite la visualización. El código siguiente muestra lo descrito.

```
def pygame_callback(data, obj):
    img = np.reshape(np.copy(data.raw_data), (data.height, data.width, 4))
    img = img[:, :, :3]
    img = img[:, :, ::-1]
    obj.surface = pygame.surfarray.make_surface(img.swapaxes(0, 1))
```

La Tabla 5 y Tabla *6. Atributos de salida de la cámara.* ofrecen una descripción detallada de los atributos de configuración y salida disponibles en CARLA para una cámara.

Atributo de entrada	Descripción
image_size_x	Ancho de la imagen en píxeles.
image_size_y	Altura de la imagen en píxeles.
fov	Campo de visión horizontal en grados
sensor_tick	Segundos de simulación entre capturas del sensor (*ticks*).

Tabla 5. Atributos básicos de la cámara.

Atributos de salida	Descripción
frame	Número de cuadro en el que se realizó la medición
timestamp	Ubicación y rotación en coordenadas mundiales del sensor en el momento de la medición
width	Ancho de la imagen en píxeles
height	Altura de la imagen en píxeles.
fov	Campo de visión horizontal en grados
raw_data	Matriz de píxeles BGRA de 32 bits.

Tabla 6. Atributos de salida de la cámara.

4.3 ARQUITECTURA DE COMUNICACIÓN V2X

Al iniciar el mundo con su correspondiente mapa en CARLA y conectarnos a él mediante un cliente utilizando la API de Python, obtenemos el control del mundo simulado y de todos los actores que participan en él. Tal como se comentó en secciones anteriores, a través de la API creamos actores, como vehículos que se desplazarán en piloto automático de manera autónoma por el mapa. Al agregar vehículos al mundo, también podemos incorporar otros actores, como sensores, que se adjuntan a los vehículos seleccionados.

En el entorno simulado, es necesario contar con una cantidad específica de vehículos equipados con sensores de obstáculos para generar eventos cuando detecten obstáculos.

Cada vez que un sensor de obstáculos genere un evento, se devolverá un objeto que contiene información sobre el obstáculo detectado y el vehículo que generó el evento, permitiendo así obtener información del actor al que está asociado el sensor (*actor*) y del actor detectado como obstáculo (*other_actor*).

Cuando un sensor en un vehículo detecta un obstáculo, debemos analizar qué vehículos reciben información sobre el evento captado por el vehículo generador del evento.

La Figura 17 muestra la arquitectura de la plataforma implementada. La plataforma cuenta con un servidor (Carla Server) que genera el mundo, la conexión con este servidor se realiza mediante un *script* de Python que funciona como cliente, implementando la lógica y gestionando el control del mundo que se simula.

En el cliente se crean los módulos más importantes que forman parte de este trabajo: como el módulo principal de *game loop*, quien contiene los *callbacks* de los sensores de cada vehículo y a través de *ticks* indica al servidor que ejecute el siguiente paso de la simulación. El módulo de conectividad V2X, que en función de la PDR se toma la decisión de qué vehículos tras una detección recibe o no mensajes del vehículo detector. Un módulo encargado de realizar la recolección de la información de qué vehículos reciben o no mensajes. Y el módulo encargado de mostrar las gráficas en tiempo real mientras se ejecuta la simulación, permitiendo la visualización de la PDR empírica que se tiene en el transcurso de la simulación.

Como se comentó, a través del *script*, se obtienen los datos de los distintos vehículos mediante sus sensores a través de *callbacks*. En base a esta información, el módulo de comunicaciones V2X se encarga de interconectar los diferentes vehículos.

Para determinar la conectividad entre vehículos, se utiliza un modelo de rendimiento analítico, el cual se verá con más detalles en la sección 4.6.1. El módulo V2X no solo se encarga de la lógica de decisión al detectar un obstáculo y decidir si otros vehículos reciben información o no, sino también de recopilar los datos generados por los sensores, almacenarlos y procesar la información. Esto puede incluir guardar un archivo CSV con los datos obtenidos, mostrar gráficos en determinados *frames* o visualizar una gráfica en tiempo real.

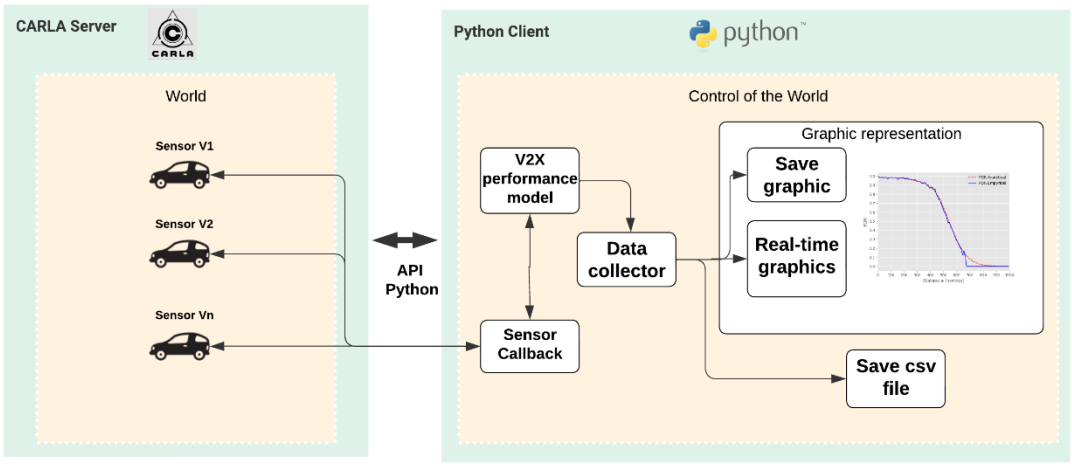


Figura 17. Arquitectura para la comunicación V2X.

4.4 ESCENARIOS

Es importante tener en cuenta que un escenario representa el mundo simulado y todos los elementos que intervienen en él. Al iniciar el simulador, se crea un mundo que contiene un mapa, y cada vez que se modifica el mapa, es necesario crear un nuevo mundo. Para llevar a cabo la implementación, es fundamental disponer de un mundo en el cual se encuentren todos los actores, así como de un mapa donde los actores puedan interactuar entre sí.

4.4.1 DESCRIPCIÓN Y SELECCIÓN DE MAPAS

Mediante la API de Python, es posible cargar uno de los ocho pueblos disponibles en el ecosistema de CARLA. La Figura 18 muestra el mapa completo *Town 10* que ya se exhibió parcialmente en la Figura 11 (sin capas) y Figura 12 (con capas).

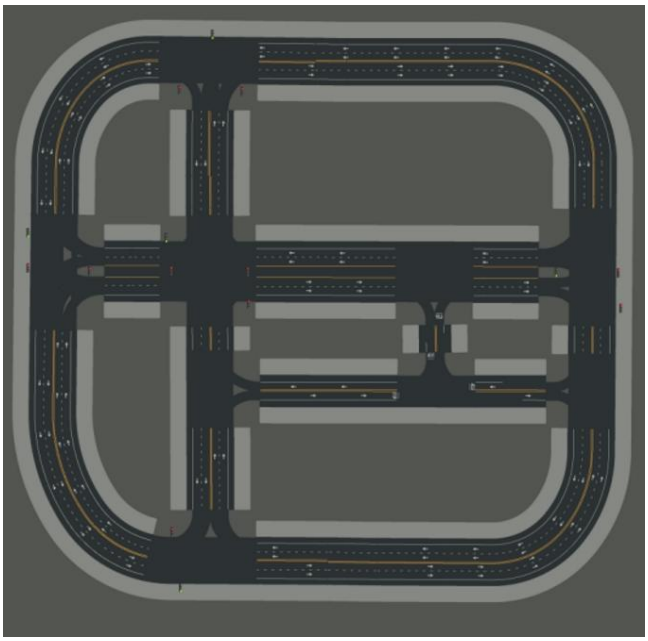


Figura 18. Mapa de Town 10 en CARLA.

En este trabajo de fin de grado, se implementaron dos tipos de escenarios. Por un lado, se utilizó *Town 2*, que es un mapa ligeramente más pequeño que *Town 10*, abarcando aproximadamente 250 metros cuadrados. Es un mapa sencillo que consta de cruces en T, tal como se muestra en la Figura 19. Este mapa se empleó para realizar las primeras pruebas en la creación del mundo y la generación de actores, ya que, al ser uno de los mapas más pequeños proporcionados por CARLA, permite que los eventos que puedan producirse durante la simulación se generen en un período de tiempo más corto. Además, ofrece mayor rapidez en el lanzamiento de la simulación, puesto que, al ser un mapa pequeño, se cargan sus respectivas capas y texturas de forma más rápida. *Town 2* sirvió para verificar que el script utilizado funcionara correctamente al generar los principales actores que intervienen en la simulación.

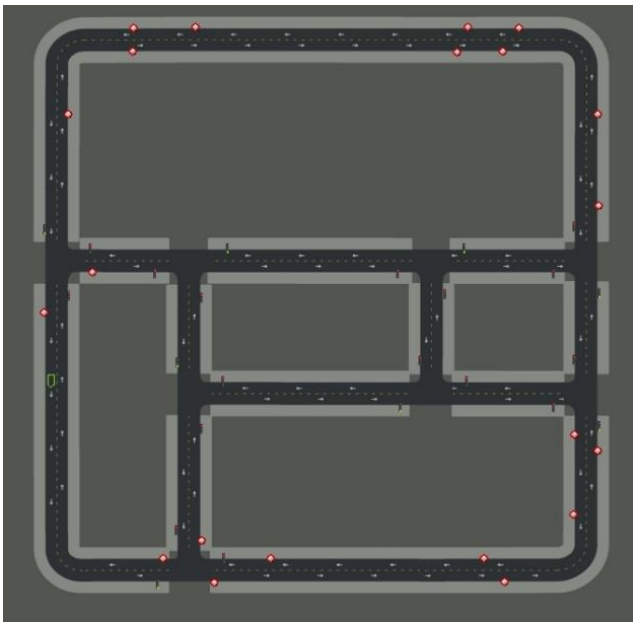


Figura 19. Mapa de Town 02 en CARLA.

La Figura 20 muestra una vista aérea de *Town 2* incluyendo la capa con edificios, vehículos, árboles, etc., y la Figura 21 muestra una perspectiva desde una de las intersecciones.

Figura 20. Vista aérea de "Town 2" en CARLA

Figura 21. Intersección de "Town 2" con todas sus capas.

CARLA ofrece la posibilidad de cargar mapas de gran tamaño. Aunque estos mapas amplios pueden proporcionar escenarios más variados y extensos para la simulación, para el propósito de esta investigación no fue necesario emplearlos. En lugar de ello, se optó por utilizar el mapa conocido como "Town 6". Este mapa se caracteriza por sus largas carreteras, tal como se puede apreciar en la Figura 22, y sus dimensiones se estiman en alrededor de 1200 x 600 metros.

Este mapa se utilizó específicamente para aquellas simulaciones en las que el tamaño de *Town 2* resultaba restrictivo y limitaba la obtención de resultados representativos. Este punto se aborda con mayor detalle en la sección 4.4, dedicada al módulo de comunicación V2X.

La Figura 23 ilustra la extensión total de *Town 6* a través de una vista aérea sin la capa de sombras, lo que proporciona una clara comprensión de la estructura del mapa. Por otro lado, la Figura 24 presenta una perspectiva diferente de este mapa, mostrando todas las capas y dando una visión más completa de su configuración.

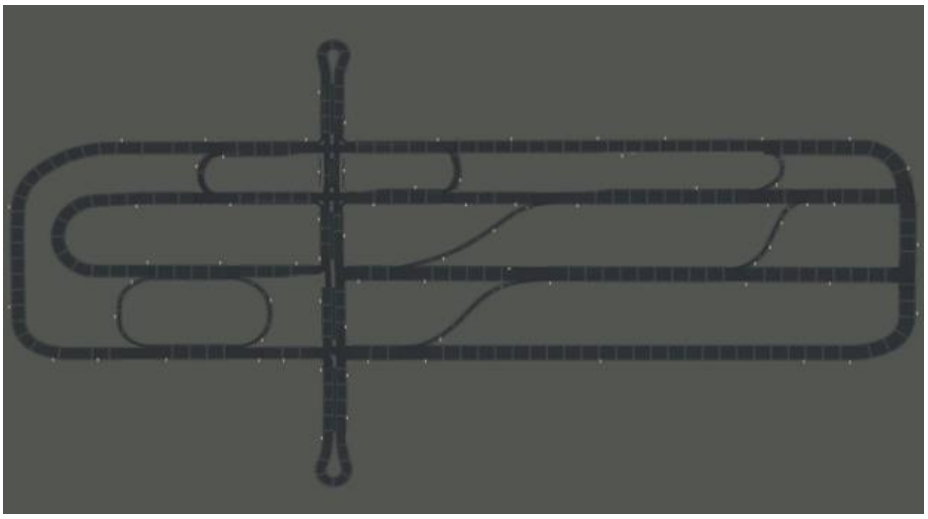


Figura 22. Mapa de “Town 6” en CARLA.

Figura 23. Vista aérea de "Town 6" con capa sin sombras.

Figura 24. "Town 6" desde otra perspectiva con todas sus capas.

4.4.2 DESCRIPCIÓN DE ESCENARIOS

Dado que la instalación de CARLA se realizó a través del código fuente, se emplea el editor Unreal Engine para el lanzamiento del simulador CARLA. Al iniciar el simulador se crea un mundo que, en este caso, contiene por defecto el mapa *Town 10*, como se muestra en la Figura 25. Desde el editor, se puede cambiar entre los diferentes mapas disponibles, siendo los escenarios con sufijo "*_opt*" aquellos que incluyen capas (Figura 26).

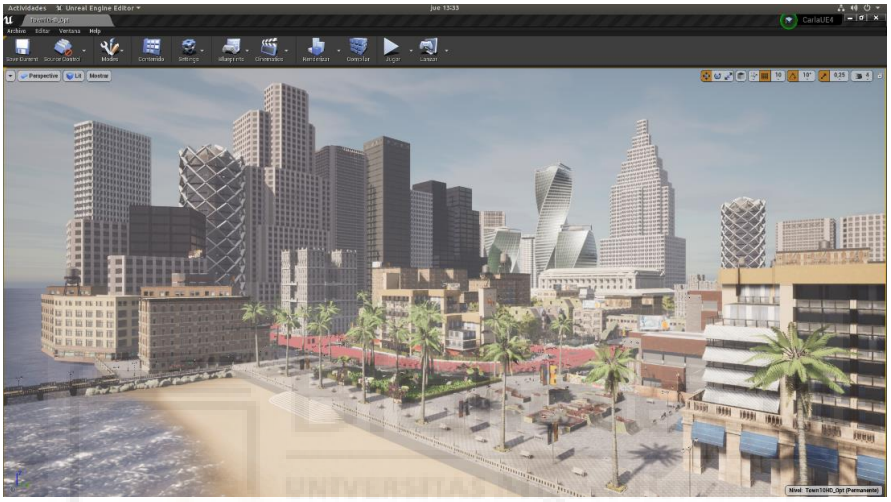


Figura 25. Editor Unreal Engine con el mapa "Town 10".

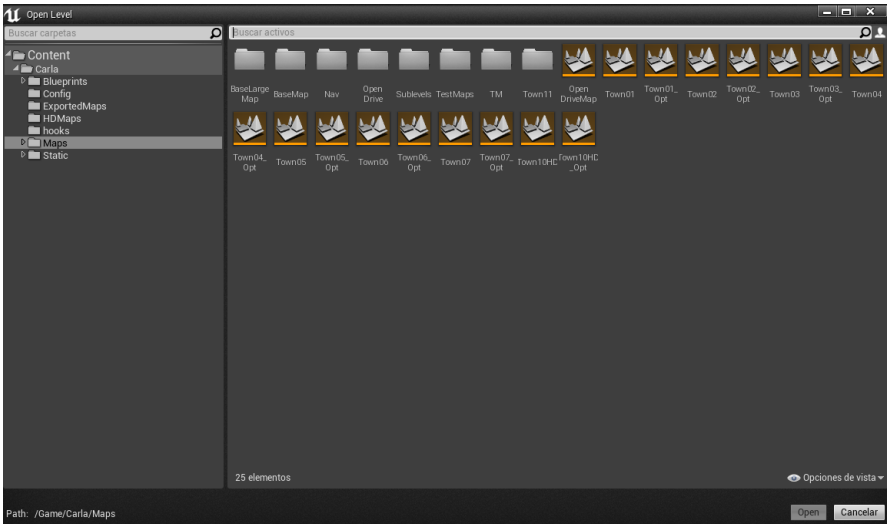


Figura 26. Mapas disponibles para cambiar desde el editor.

Una vez creado el mundo con su respectivo mapa, se pone en marcha un servidor que es responsable de la simulación. Mediante scripts en Python, nos conectamos al servidor en calidad de cliente, proporcionando los parámetros necesarios para su creación:

```
client = carla.Client(args.host, args.port)
```

En particular, se requiere la dirección IP como identificador del cliente y el puerto TCP para establecer la comunicación con el servidor. Los parámetros, como la IP y el puerto, pueden modificarse. Por defecto, CARLA utiliza la IP del host local y el puerto 2000 para establecer la conexión. Tras crear el cliente, se establece un tiempo de espera de dos segundos para limitar las operaciones de red y evitar que bloqueen al cliente:

```
client.set_timeout(2.0)
```

Para controlar el mundo, se obtienen los datos de este a través del cliente utilizando la siguiente instrucción:

```
world = client.get_world()
```

Una vez que se tienen los datos del mundo, podemos cambiar el mapa del mundo, por ejemplo, utilizando la siguiente sentencia:

Teniendo en cuenta que se creará un nuevo mundo con el mapa *Town02*, como se mencionó, estas sentencias no fueron necesarias en el script implementado, ya que el cambio de mapas se realiza desde el mismo editor.

Con el mundo y su mapa establecidos y un cliente que pueda gobernar el mundo, se pueden ejecutar comandos en consola que aprovechen la potente API de Python que proporciona CARLA. Esto permite realizar acciones como cambiar el clima del mundo y añadir actores que realicen acciones dentro de la simulación (vehículos y peatones).

Debido a la arquitectura cliente-servidor de CARLA, es posible conectarse al mismo mundo con diferentes clientes, lo que permite, por ejemplo, generar tráfico mediante un cliente y cambiar el clima con otro. El siguiente código es un ejemplo de ejecución de scripts mediante consola, donde en el terminal A se genera tráfico (vehículos que se mueven automáticamente por el mapa) y en el terminal B se cambia la climatología del mundo a medida que avanza la simulación. Cada script realizará el procedimiento previamente mencionado: creará un cliente y obtendrá datos del mundo para poder modificarlo.

```
# En el terminal A
cd PythonAPI/examples
python3 -m pip install -r requirements.txt
python3 generate_traffic.py

# En el terminal B
cd PythonAPI/examples
python3 dynamic_weather.py
```


En este trabajo, al utilizar *Town 2* y *Town 6*, se pueden aplicar los mismos comandos para cambiar la climatología de nuestro mundo. En este caso, los scripts se ejecutaron para que surtan efecto en el mundo donde se encuentra contenido el mapa *Town 2*, como se muestra en la Figura 27.

Figura 27.Vista de “Town 2” sin ejecutar los scripts.

Al ejecutar el script de generación de tráfico en un terminal, obtendremos un número de vehículos predeterminado por el script que se mueven de forma automática por el entorno. En la Figura 28, se puede observar un ejemplo de ello, donde se ha eliminado la capa de sombras para facilitar una mejor visualización.

Figura 28. Vista de "Town 2" con tráfico.

Si ejecutamos en otro terminal el script que cambia dinámicamente el clima, obtendremos una serie de escenas a lo largo del tiempo que simulan las posibles condiciones climáticas que se pueden presentar en la vida real. En la Figura 29, podemos ver un ambiente nocturno. En la Figura 30 y Figura *31*, se observa respectivamente un amanecer y un clima lluvioso.

Figura 29. Vista de "Town 2" de noche.

Figura 30. Vista de "Town 2" amaneciendo.

Figura 31. Vista de "Town 2" con clima lluvioso.

4.4.3 PARÁMETROS DE SIMULACIÓN

Como se mencionó en la sección anterior, se han utilizado dos mapas: *Town 2* y *Town 6*. En cada mundo se ejecutó el script *v2x.py*. Este script es una modificación del generador de tráfico proporcionado por CARLA (*generate_traffic.py*), lo que permite mantener los argumentos propios del script. Por defecto, el script original consta de los siguientes parámetros:

- --*host*: Establecer la IP del servidor.
- --*port*: Establecer número de puerto de escucha.
- --*number-of-vehicles*: Establecer el número de vehículos.

- --*number-of-walkers*: Establecer el número de peatones.
- --*safe*: Evita que se generen vehículos propensos a accidentes.
- --filterv: Filtro para establecer el modelo de vehículo en la simulación.
- --*generationv*: Establecer generación de vehículo, de primera generación (actores con características antiguas de CARLA) o segunda (actores con características nuevas de CARLA).
- --*filterw*: Filtro para establecer el tipo de peatón en la simulación.
- --*generationw*: Establecer generación de peatón, de primera generación (actores con características antiguas de CARLA) o segunda (actores con características nuevas de CARLA).
- --*tm-port*: Establecer número de puerto para un administrador de tráfico.
- --*asynch*: Activar el modo asíncrono.
- --*hybrid*: Activa el modo de física híbrida. Los vehículos en piloto automático que estén a un determinado radio de un vehículo ego deshabilitan sus cálculos físicos.
- --*seed*: Establece una semilla para usar el modo determinista de un administrador de tráfico para garantizar un mismo comportamiento en diferentes ejecuciones del script si se mantiene las mismas condiciones.
- --*seedw*: Establecer una semilla para la generación de peatones en el mapa.
- --*car-lights-on*: Activa la gestión automática de las luces del vehículo.
- --*no-rendering*: Activa el modo de no renderizado del mundo deshabilitando la presentación. Los gráficos no se calculan y los sensores basados en GPU devuelven resultados vacíos.

4.5 MÓDULO DE CONECTIVIDAD V2X

En esta sección se describe el módulo de conectividad V2X implementado, cuyo funcionamiento se basa en la obtención de información de los sensores asociados a cada vehículo utilizando la API de Python en CARLA. Este módulo recibe datos del mundo simulado a través de un objeto proporcionado como argumento, lo cual le permite acceder a la configuración de la simulación.

Adicionalmente, si alguno de los sensores de los vehículos involucrados en la simulación detecta un obstáculo, el módulo está equipado para recibir toda la información correspondiente de dichos vehículos, tanto el vehículo detectado como obstáculo como quién lo detectó. Esta información se recibe a través de otro objeto entregado como argumento, el cual contiene detalles acerca de los vehículos participantes en el evento. Al disponer de la información del mundo simulado y los datos proporcionados por los sensores de todos los vehículos en el mapa, el módulo tiene la capacidad de determinar cuáles vehículos pueden enviar o recibir información de los demás vehículos en la simulación.

En secciones previas, se ha discutido la generación del mundo, sus actores, la creación de sensores y su asociación con otros actores como vehículos. También se mencionó que, al generar un evento (detectar un obstáculo), se obtiene un objeto con información relativa al evento generado.

Antes de seguir con la sección es importante volver a repasar sobre el sensor de obstáculos. El detector de obstáculos debe implementar una función que pueda ser invocada mediante un método *callback* cada vez que el sensor genere un evento. En este caso, lo primero que se hace es configurar cada sensor para que esté en modo de escucha, de manera que, al generarse un evento, se llame a la función correspondiente:

```
obstacle_sensor[i].listen(lambda data: sensor_callback(data, sensor_queue))
```

Se añade a una pila FIFO (*First In - First Out*) el objeto devuelto por el sensor:

```
def sensor_callback(sensor_data, sensor_queue):
    # Añadir a la cola
    sensor_queue.put((sensor_data.frame ,sensor_data.actor.attributes['role_name'],
                      sensor_data.other_actor.attributes['role_name'],
                      sensor_data.actor.parent, sensor_data.other_actor))
```

En el código presentado previamente, se muestra que se añaden a una pila algunos atributos de salida del sensor, que contienen información sobre el objeto detectado, quién lo detectó y los nombres de rol en la simulación. La razón para almacenar los objetos retornados por el sensor en una pila radica en evitar la pérdida de información. A lo largo de la ejecución, como cada vehículo cuenta con un sensor asociado, en cada paso de simulación o *frame* un sensor puede detectar, lo que resulta en la obtención de N objetos provenientes de N sensores en N *frames*. Si no se guarda en la pila, existe el riesgo de que se avance a un siguiente paso de simulación sin haber almacenado la información proporcionada por los sensores. Almacenando los datos en una pila, se asegura que la información estará resguardada en el transcurso de la simulación.

Al iniciar el mundo, una vez realizados los controles pertinentes y tras obtener la información de configuración proporcionada por los argumentos, se invoca a la función *game_loo*p:

```
def game_loop(args, world, camera, actor, camera_bp, camera_init_trans,
              obstacle_sensor, sensor_queue, synchronous_master, all_vehicle_actors):
```

Al ingresar a esta función, se ejecuta un bucle de forma indefinida hasta que el programa se interrumpa o la simulación finalice.

Dentro de este bucle, se lleva a cabo la lógica del programa y se controlan los pasos de la simulación mediante llamadas a *world.tick(),* las cuales indican al simulador que puede avanzar al siguiente paso.

Además, en esta función se verifica constantemente si algún actor sensor ha devuelto algún objeto, es decir, si existe algún objeto en la pila FIFO.

Para no mostrar código en exceso, el siguiente fragmento ilustra la idea principal:

```
while True:
     world.tick()
     w_frame = world.get_snapshot().frame
     print("\nWorld's frame: %d" % w_frame)
     # Obstáculos detectados
     try:
         for _ in range(len(obstacle_sensor)):
                         s_frame = sensor_queue.get(True, 0.01)
                              v2x(world, args, s_frame)
     except Empty:
            print(“No hay información”)
```

Obsérvese que, en caso de no producirse una excepción, esto indicará que hay información en la pila y, por lo tanto, la variable *s_frame* contendrá un objeto con la información del sensor obtenido de dicha pila:

```
S_frame = sensor_queue.get(True, 0.01)
```

En la línea de código anterior se intenta obtener un elemento de la cola *sensor_queue*. Si hay un elemento disponible, se asigna a la variable *s_frame*. Si la cola está vacía, el programa esperará un máximo de 10 milisegundos para obtener un elemento antes de lanzar una excepción en caso de que no haya ningún elemento disponible en ese tiempo.

Cuando se detecta un obstáculo, invocamos la función V2X, pasándole como parámetros la información del mundo (*world*), los argumentos iniciales (*args*) y *s_frame*. Es importante recordar que, previamente, mostramos la función de llamada que se activa al detectar un obstáculo, en la cual añadimos a una cola la información relevante devuelta por el sensor:

```
sensor_queue.put((sensor_data.frame ,sensor_data.actor.attributes['role_name'],
                    sensor_data.other_actor.attributes['role_name'],
                    sensor_data.actor.parent, sensor_data.other_actor))
```

El objeto obtenido de la cola es una tupla que contiene parte de la información devuelta por el sensor que incluye información sobre el actor padre que tiene el sensor que ha detectado el obstáculo y el obstáculo detectado en cuestión con información sobre él. Como, por ejemplo, el modelo de vehículo (tanto del vehículo que detecta como el detectado), momento de detección (*frame*), distancia a la que se ha detectado, entre otros. Este objeto será el que pasamos al módulo de comunicación V2X. Dentro de la función o módulo V2X, inicialmente se obtiene una lista completa de los vehículos presentes en el mundo simulado. Posteriormente, se centra en el vehículo de interés (vehículo detector), que se designa como el quinto elemento de s_frame, calculando su distancia relativa a cada uno de los demás vehículos en la simulación. A medida que el programa recorre cada uno de estos vehículos, captura y almacena información en listas.

Esta información incluye datos como la distancia al vehículo de interés, el ID del vehículo, su modelo, entre otros.

La función V2X, como parte de su proceso de simulación, implementa una serie de pasos para calcular si un mensaje de un vehículo que ha detectado un obstáculo es recibido o no por los demás vehículos en la simulación. Este proceso utiliza un modelo analítico determinado a partir de la configuración de las comunicaciones proporcionada por el usuario, que incluye factores como la potencia de transmisión, el número de subcanales, la tasa de paquetes y la densidad de vehículos.

De manera simplificada, la tarea de decisión comienza con el proceso de calcular un índice basado en la distancia relativa (d) entre el vehículo detector y los demás vehículos en la simulación. Este índice se calcula dividiendo la distancia relativa por 10 y redondeándola hacia abajo, tal como se muestra a continuación:

```
index = math.floor(d / 10)
```

Posteriormente, se genera un número aleatorio entre 0 y 1:

```
p = random.random()
```

Este número aleatorio y el valor de PDR correspondiente al índice (de la distancia relativa) se añaden a las listas de probabilidades aleatorias y analíticas, respectivamente:

```
prob_aleatory.append(p)
prob_analytical.append(pdr[index])
```

Luego, la función compara el número de probabilidad aleatoria con la probabilidad de la PDR indexada a la distancia relativa calculada. Si el número aleatorio es mayor que la probabilidad indexada a la distancia relativa de la PDR analítica, se asume que el vehículo no recibe el mensaje y la distancia se agrega a la lista de distancias donde no se recibió el paquete:

```
if p > pdr[index]:
    recibe.append('NO')
    dist_no_ok.append(d)
```

Si el número aleatorio es menor o igual a la probabilidad indexada a la distancia relativa de la PDR analítica, se asume que el vehículo recibe el mensaje y la distancia se agrega a la lista de distancias donde se recibió el paquete:

```
else:
    recibe.append('SI')
    dist_ok.append(d)
```

Las distancias relativas en las listas *dist_ok* y *dist_no_ok* proporcionan información de la distribución de éxito o fracaso en la recepción de paquetes en función de la distancia relativa entre los vehículos en la simulación. Así, contribuyen a la generación de una

representación empírica de la tasa de entrega de paquetes (PDR) en la red de comunicación simulada. El uso de estas listas se verá con más detalles en la sección 4.6.2 que trata sobre la representación gráfica realizada en este trabajo. A través de este proceso, la función V2X simula la transmisión de mensajes, teniendo en cuenta factores realistas como la potencia de transmisión, la densidad de vehículos y la distancia entre vehículos.

Además, la función V2X gestiona la creación de nuevos hilos de ejecución, en concreto tres hilos: uno destinado a la recopilación de datos (*getData*), uno para la creación de gráficos estáticos (*static_graphic*) y otro para el cálculo y visualización en tiempo real de la PDR (*func_pdr*). De manera resumida, el funcionamiento básico de estas funciones son las siguientes:

- *getData:* se encarga de recopilar y procesar los datos de la PDR empírica, que es la probabilidad de recepción de paquetes basada en las simulaciones realizadas. En cada ciclo de ejecución, comprueba si hay datos empíricos disponibles. Si los hay, genera una gráfica estática con esos datos.

- *static_graphic:* se encarga de generar las gráficas estáticas de la PDR. Utiliza los datos recopilados por getData y traza la PDR analítica y empírica en la misma gráfica para una comparación visual. Las gráficas resultantes se guardan en archivos individuales.

- func_pdr: es responsable de generar y exhibir un gráfico animado que ilustra la evolución en tiempo real de la PDR conforme se recolectan más datos. Similar a *static_graphic*, esta función traza tanto el PDR analítico como el empírico en el mismo gráfico, pero en tiempo real.

Las explicaciones pertinentes con respecto al apartado de representación gráfica se verán con más detalles en la sección 4.6.2.

Para ejemplificar lo expuesto hasta ahora, se configura un escenario con 10 vehículos en el mapa *Town 2* donde se indica a partir de las configuraciones de usuario (argumento --*ego*) que uno de los vehículos sea etiquetado como ego (*Ego vehicle*) para analizar lo que sucede. Esto significa que de los 10 vehículos participantes en la simulación sólo a uno de ellos se le asociará un sensor de obstáculos.

A lo largo de la simulación, en un momento dado, el vehículo ego detecta un vehículo como obstáculo (cabe destacar que este obstáculo no tiene por qué ser otro actor de tipo vehículo, sino también un actor de tipo peatón).

Tras la detección, obsérvese la tupla de información que se envía a la función V2X en la Figura 32. En la figura, se puede ver cómo, en este caso, *s_frame* en el índice 0 contiene el número de *frame* 11.498. Instante del mundo en el que se ha realizado la detección.

En el índice 1, se observa la etiqueta de rol del vehículo, en este caso "*Ego vehicle"* (se configuró uno de los vehículos como ego). Este representa al actor que detectó el obstáculo (actor padre, atributo *Actor*). En el índice 2, "Vehículo 2" es otra etiqueta de rol que representa al actor obstáculo (atributo *other_actor*). En el índice 3, se encuentra la distancia a la que se ha detectado el obstáculo (3,98 metros). Finalmente, en los índices 4 y 5, se tienen objetos de tipo de *Actor* que incluyen, entre otras cosas, el ID y el modelo del vehículo del actor padre que ha detectado y el vehículo obstáculo.

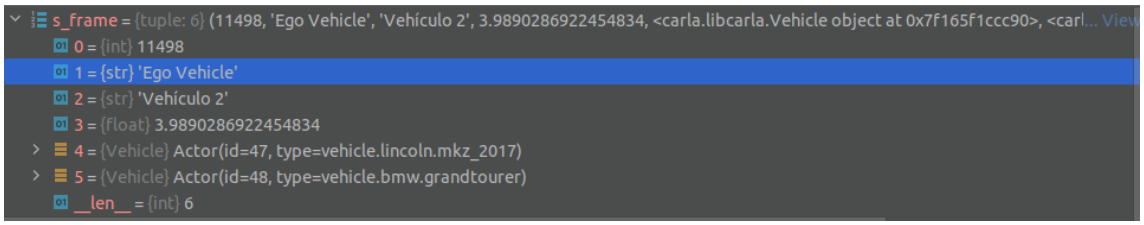


Figura 32. Tupla obtenida de la cola.

Al expandir los objetos de los dos últimos índices mencionados, podemos observar el contenido dentro de la tupla de un objeto de tipo *Actor* que corresponde al vehículo ego, como se muestra en la Figura 33 quien tiene como *role_name* "*Ego Vehicle*". Además, en la Figura 34 se presenta el objeto de tipo *Actor* correspondiente al actor que ha sido detectado como obstáculo que tiene como *role_name "Vehículo 2"*.

Figura 33. Contenido de objeto Actor de vehículo que detecta.

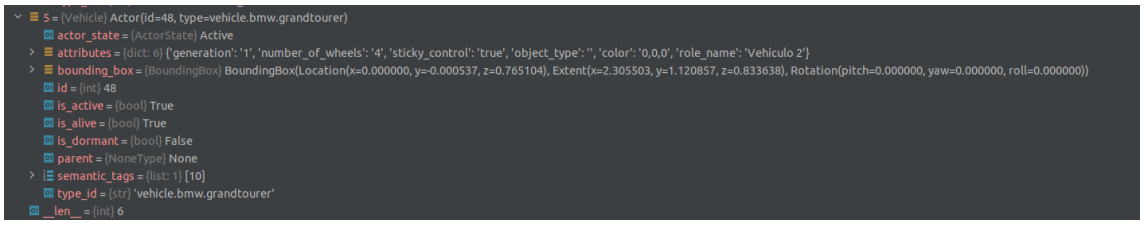


Figura 34. Contenido de objeto Actor de vehículo que ha sido detectado.

Como se mencionó anteriormente, tanto vehículos como sensores son actores en el mundo virtual. En la Figura 35, se puede apreciar un ejemplo del contenido de un objeto *Actor* correspondiente al sensor de obstáculos. Se observa la configuración inicial en sus atributos y cómo el sensor está adjunto, en este caso, al vehículo ego. El ejemplo de la figura corresponde a otra simulación; nótese que está asociado a un vehículo ego con ID 113 y un modelo ”Mercedes *Coupé*”. En contraste, si observamos la Figura 33 en la simulación que se está analizando, el sensor está asociado a un ego con ID 47 y modelo “*Lincoln MKZ*”.

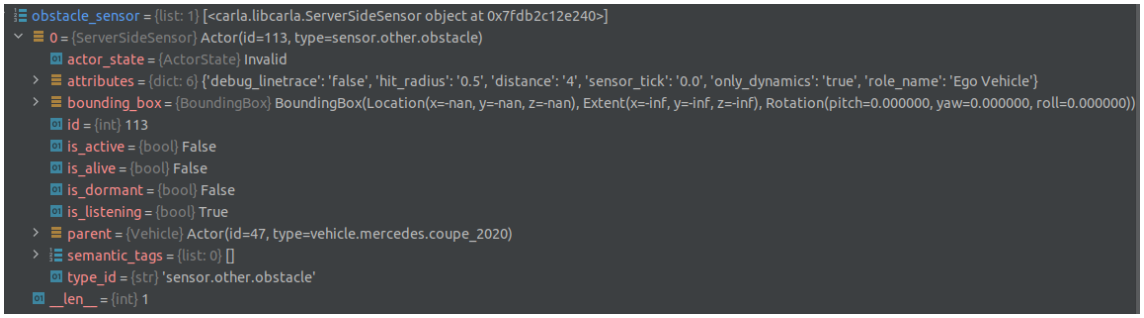


Figura 35. Contenido de objeto Actor de sensor.

Al invocar la función V2X, dentro de ella se recopila la información de todos los actores de tipo vehículo presentes en el mundo virtual:

```
vehicles = world.get_actors().filter('vehicle.*')
```

Adicionalmente, se determina la posición del vehículo que realizó la detección del obstáculo en el mundo:

```
t = s_frame[4].get_transform()
```

En *s_frame[4]* se halla la información del actor padre, que cuenta con el sensor asociado que genera el evento. La función *get_transform()* devuelve la transformación del actor (ubicación) que el cliente recibió durante el último *tick*. La Figura 36 muestra el contenido de la variable *"t"*, que contiene la información de la posición y orientación del vehículo que ha detectado el obstáculo.

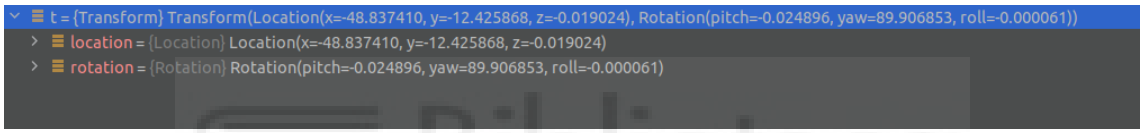


Figura 36. Coordenadas del vehículo que ha generado el evento.

Teniendo la información de la ubicación exacta del vehículo con el sensor asociado en el mundo, podemos calcular las distancias relativas de éste con respecto a los demás vehículos. Para ello, se define una función lambda *distance* que calcula la distancia euclidiana en 3D y se crea una lista *list_vehicles* que contiene tuplas con la distancia relativa y la información del vehículo correspondiente, excluyendo obviamente, el vehículo que detectó el obstáculo:

```
distance = lambda l: math.sqrt((l.x - t.location.x) ** 2 + (l.y - t.location.y) ** 2 + (l.z - t.location.z) ** 2)
list_vehicles = [(distance(x.get_location()), x) for x in vehicles if x.id != s_frame[4].id]
```

De esta manera, la lista list_vehicles contiene todos los vehículos junto con sus distancias relativas respecto al vehículo detector. El contenido de esta lista se puede apreciar en la

Figura 37, donde se muestran las distancias relativas del vehículo con respecto a los otros nueve vehículos presentes en el mundo.

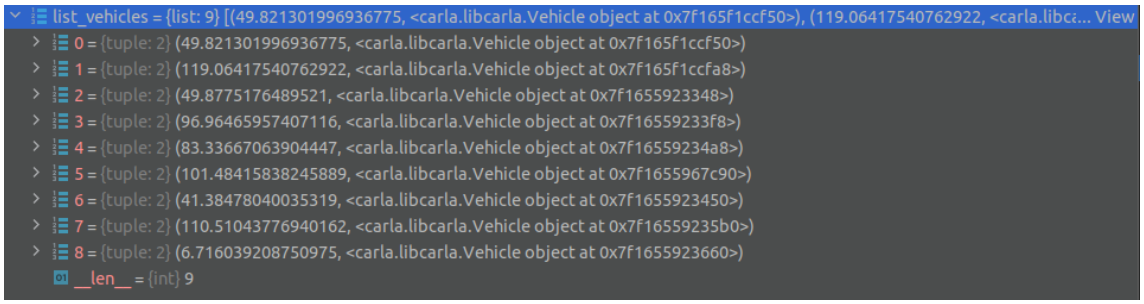


Figura 37. Lista de vehículos con sus respectivas distancias relativas al vehículo detector.

Con toda esta información, queda por determinar si el vehículo en cuestión podría comunicar que su sensor asociado ha generado un evento, es decir, que ha detectado un obstáculo. Para ello, como se comentó se emplea un modelo de rendimiento analítico en el que se toma un vector de probabilidades basado en los parámetros de entrada del script. De este modo, se obtienen datos de una PDR cuyos detalles se abordarán en la sección 4.6 sobre el modelado de comunicaciones V2X.

En la Figura 38 se presenta el caso del "Vehículo 4", que se encuentra a una distancia relativa del vehículo ego de 41,38 metros. Para esta distancia relativa, se obtiene un índice de 4 en la posición del vector de probabilidad (como se muestra en la Figura 39), que corresponde a una probabilidad aproximada del 98%.

El número aleatorio generado es cerca del 94% y el valor analítico es del 98%, por lo que se concluye que el "Vehículo 4" recibe el evento generado por el "*Ego vehicle*" al detectar al "Vehículo 2" en el entorno.

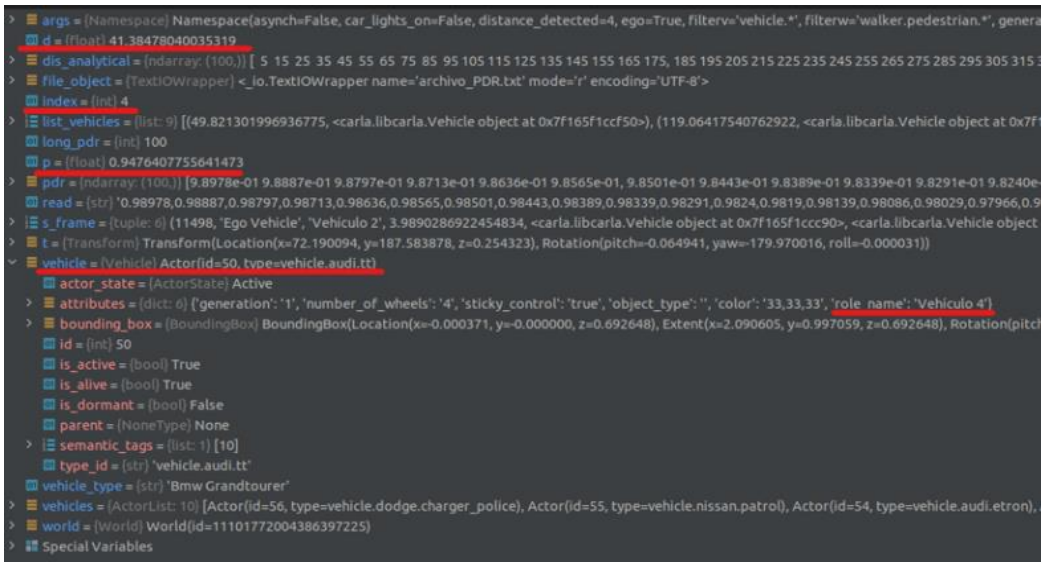


Figura 38. Muestra de los valores obtenidos.

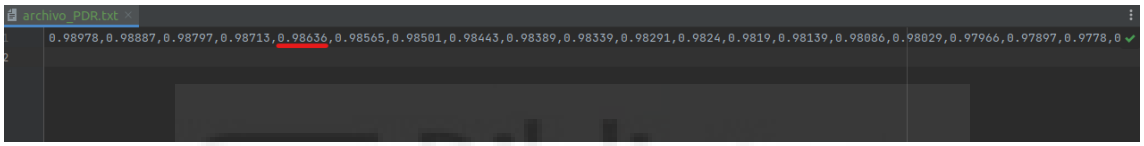


Figura 39. Valores del vector de probabilidades del archivo PDR.

La función V2X almacena diferentes tipos de información en listas con el objetivo de recuperarlas y analizarlas mediante un archivo al finalizar la ejecución del programa. La simulación continuará hasta que se establezca un tiempo de finalización previamente configurado por el usuario o hasta que se interrumpa el programa desde la consola.

Al concluir la simulación, se crea un archivo "out_data.csv" en el que se depositan de manera ordenada los datos de las listas.

Los resultados de esta simulación pueden apreciarse en la Tabla 7. Esta tabla muestra la información correspondiente al *frame* número 11.498, punto en el que el vehículo ego detectó un obstáculo.

En ella también se muestran los demás vehículos que participaron en la simulación, así como las distancias a las que se encontraban del vehículo ego en el momento en que su sensor asociado generó un evento. Cabe destacar que, en este caso, todos los vehículos

recibieron correctamente la información a través de la comunicación V2X, lo que se indica con un "SÍ".

Nº	Frame	TX	ID TX	Mod. TX	RX	ID RX	Mod. RX	d (m)	Prob(%).	Recibe
0	11498	Ego Vehicle	47	Lincoln Mkz 2017	Vehículo 2	48	Bmw Grandtourer	6,7160	97,26190	SI
1	11498	Ego Vehicle	47	Lincoln Mkz 2017	Vehículo 4	50	Audi Tt	41,3848	94,76410	SI
2	11498	Ego Vehicle	47	Lincoln Mkz 2017	Vehículo 10	56	Dodge Charger Police	49,8213	31,06550	SI
3	11498	Ego Vehicle	47	Lincoln Mkz 2017	Vehículo 8	54	Audi Etron	49,8775	55,04610	SI
4	11498	Ego Vehicle	47	Lincoln Mkz 2017	Vehículo 6	52	Dodge Charger 2020	83,3367	53,07360	SI
5	11498	Ego Vehicle	47	Lincoln Mkz 2017	Vehículo 7	53	Dodge Charger Police ...	96,9647	48,81760	SI
6	11498	Ego Vehicle	47	Lincoln Mkz 2017	Vehículo 5	51	Seat Xen	101,4840	71,38740	SI
7	11498	Ego Vehicle	47	Lincoln Mkz 2017	Vehículo 3	49	Jeep Wrangler Rubicon	110,5100	51,19720	SI
8	11498	Ego Vehicle	47	Lincoln Mkz 2017	Vehículo 9	55	Nissan Patrol	119,0640	92,56420	SI

Tabla 7. Muestra del archivo out_data.csv.

Que todos los vehículos reciban información del vehículo ego que ha detectado el evento no es una coincidencia. Como se mencionó anteriormente , el mapa *Town 2* es pequeño, de aproximadamente 200 metros cuadrados.

Las distancias relativas de los vehículos que participan en la simulación en relación con el vehículo ego no indexan en el vector de la PDR probabilidades que resulten en la imposibilidad de recibir un paquete al compararlas con la probabilidad obtenida de manera aleatoria. En otras palabras, los vehículos en este mapa no están (ni pueden estar) lo suficientemente lejos para provocar una pérdida de paquetes.

Anteriormente se abordó la utilidad de utilizar *Town 6* como escenario de pruebas, con el objetivo de obtener resultados que difieran de los generados en *Town 2*. Este último, debido a su tamaño reducido, limitaba la diversidad de los resultados que podían lograrse en las simulaciones.

La Figura 40 muestra un diagrama de flujo que resume de forma concisa todos los aspectos tratados en este trabajo con respecto al módulo de conectividad V2X, proporcionando una visión integral del funcionamiento a alto nivel.

Como se puede ver, al iniciar la simulación se asignan inicialmente todos los parámetros de configuración de entrada, los cuales se pasan mediante los argumentos para establecer los valores que indica el usuario y así poder inicializar la simulación.

Con todos los parámetros e información necesaria a disposición, se crean los actores (vehículos, peatones, sensores, etc.) que intervendrán en el mundo simulado. En esta etapa, a cada uno de los N vehículos se les asocia un sensor de obstáculos y una cámara (en el caso de que no se indique configurar un vehículo ego). Tras configurar todos los actores, se inicia una función (bucle de juego) que permite que el tiempo simulado avance. Mediante la emisión de *ticks*, esta función ordena al servidor de CARLA que ejecute el siguiente paso de la simulación, en este punto, también se registra el número de *frame*.

El mundo continúa su curso, es decir, el bucle de juego sigue en ejecución, a menos que se presente una interrupción, como una orden de detención del usuario (ctrl+c), el fin del tiempo de simulación especificado por los argumentos, o algún problema que desencadene una excepción. En cualquiera de estos casos, se detendrá la simulación y se eliminarán eficientemente todos los actores participantes.

Si no se produce ninguna excepción, la simulación continúa indefinidamente. En cada paso de la simulación, se verifica si algún sensor dentro del mundo ha generado un evento, lo cual indica que uno o más vehículos tienen un obstáculo delante.

Si se da esta condición, los objetos devueltos por el sensor se almacenan en una cola, tal como se mencionó en esta sección.

En la función V2X se recopilan los datos de los vehículos participantes en el mundo. Esto permite, por ejemplo, identificar qué vehículos están involucrados en caso de detección de un obstáculo. En esta función, en función de la configuración obtenida de los parámetros seleccionados por el usuario (como la potencia de transmisión, el número de subcanales, la tasa de paquetes y la densidad de vehículos), se determina una PDR analítica. En este punto también se calcula la distancia relativa entre el vehículo que ha detectado el obstáculo (VD) y los demás vehículos del mapa (OV).

Con dicha distancia relativa, se selecciona una distancia correspondiente al vector de distancias analíticas de la PDR. Dado que a cada distancia de la PDR le corresponde una probabilidad, se utiliza la probabilidad analítica seleccionada para compararla con una probabilidad aleatoria, como se explicó anteriormente. Así se decide si los vehículos

reciben o no el mensaje. Finalmente, se recopila y almacena toda la información, independientemente de si los vehículos reciben o no mensajes, para su posterior registro en un archivo CSV.

Es importante señalar que el diagrama de flujo presentado se refiere al hilo principal del programa, el cual ejecuta el mundo y, tras detectar un obstáculo, invoca al módulo V2X. Los hilos que se encargan de la representación gráfica y recopilación de datos se ejecutan de forma paralela y no se muestran en el diagrama para evitar complicaciones en la visualización de la función principal del módulo V2X.

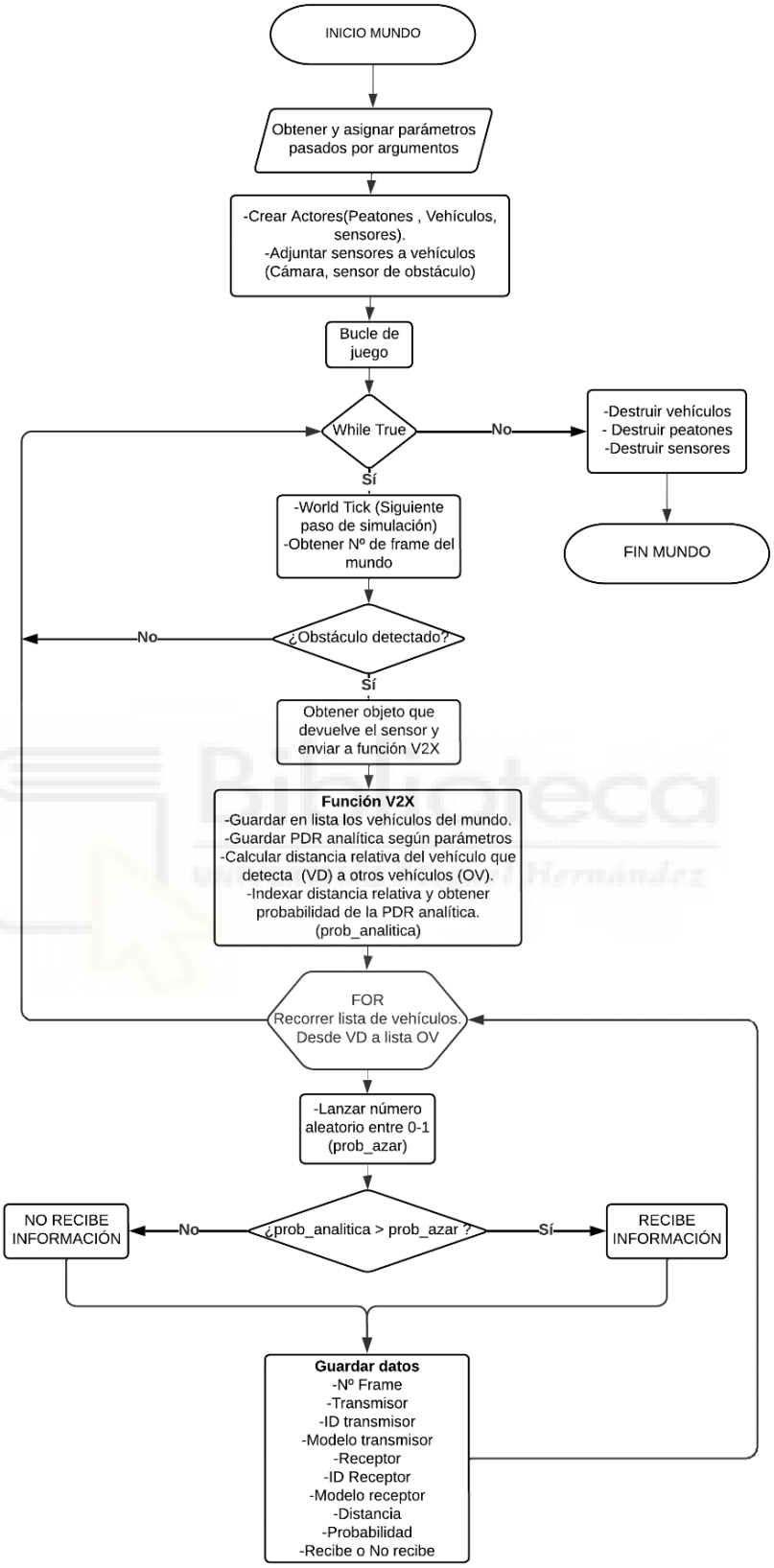


Figura 40. Diagrama de flujo.

4.6 MODELADO DE LAS COMUNICACIONES V2X

En la sección anterior, analizamos cómo se utiliza un vector de probabilidades para determinar si un vehículo, al identificar un obstáculo, puede transmitir información a otros N vehículos en la simulación. Este vector de probabilidades se fundamenta en un modelo de rendimiento analítico perteneciente al estándar de comunicación LTE-V2X [23]. Este estándar es una versión evolucionada del LTE, diseñada específicamente para las comunicaciones vehiculares en modo *sidelink*, esto es, facilitar la comunicación directa entre vehículos sin la necesidad de emplear infraestructura celular. En esta sección, se discute el modelo de rendimiento analítico que sirve de base para las simulaciones del presente trabajo, y desempeña un papel esencial en la determinación de las probabilidades de transmisión en función de las distancias entre vehículos.

4.6.1 MODELO DE RENDIMIENTO ANALÍTICO

La conectividad V2X se modela en la plataforma implementada mediante un modelo de rendimiento analítico. Este modelo cuantifica la probabilidad de recepción de paquetes (o PDR) en función de la distancia entre transmisor y receptor. La curva de probabilidades PDR depende de los parámetros de transmisión y aspectos relacionados con el escenario, como la densidad de tráfico.

El vector de probabilidades PDR, que se emplea durante la ejecución del script *v2x.py*, varía en función de los parámetros proporcionados por el usuario. Esta variabilidad facilita la creación de una PDR que se modela en concordancia con dichos parámetros.

El modelo considera no solo la distancia entre transmisor y receptor para su validación, sino también un conjunto de factores como densidades de tráfico, errores de transmisión y de potencia, modulación, codificación, y errores provenientes de diversas alteraciones en la propagación. En resumen, el modelo analítico cuantifica la probabilidad de

recepción entre transmisor y receptor basándose en la distancia y los parámetros mencionados.

Es importante mencionar que, para lograr coherencia entre las simulaciones en CARLA y aquellas realizadas para generar el modelo, se ha adaptado el código fuente principal responsable de la generación de vehículos. Aunque es factible configurar un determinado número de vehículos a través de argumentos, las simulaciones cobran mayor relevancia cuando se aplica este modelo analítico. Por tanto, teniendo en cuenta las distancias de las carreteras de los dos mapas seleccionados (*Town 2* y *Town 6*), y de acuerdo con la densidad de vehículos escogida, se generan una cantidad específica de vehículos en estos mapas, la cual depende de sus dimensiones.

De esta forma, las simulaciones en CARLA mantienen consistencia con las simulaciones realizadas para obtener los vectores de probabilidad del modelo.

El motivo principal de aplicar el modelo analítico es para evitar el elevado coste computacional y de tiempo asociado a la simulación de escenarios que tomen en cuenta los parámetros antes descritos. En nuestro caso, gracias al modelo, podemos predecir la recepción o no de paquetes utilizando las probabilidades derivadas de este modelo analítico con parámetros predefinidos, utilizando un simulador de alto rendimiento sin utilizar otros simuladores.

A continuación, se explica los parámetros más representativos que se tomaron en cuenta para generar los vectores de probabilidades del modelo:

1. Ancho de banda del canal: Se refiere al ancho de banda asignado para la norma de comunicación LTE-V2X. Aunque LTE-V2X permite anchos de banda de 10

MHz y 20 MHz, los modelos solo han sido validados para 10 MHz en nuestra plataforma, por lo que este pará metro se mantiene constante.

2. Potencia de transmisión: Este parámetro representa la potencia irradiada por los transmisores inalámbricos LTE-V2X. Según las regulaciones establecidas por la SAE (*Society of Automotive Engineers*)[24], la potencia máxima permitida en EE. UU. es de 20 dBm. En contraste, la ETSI (*European Telecommunications Standards Institute*)[25] permite una potencia máxima de 23 dBm en la Unión Europea.

3. Frecuencia (o tasa) de transmisión de mensajes: Este parámetro indica el número de mensajes transmitidos por cada vehículo por segundo. LTE-V2X es especialmente eficiente para la transmisión de 10 o 20 mensajes por segundo. Dado que los Mensajes Básicos de Seguridad (BSM) se generan a una frecuencia predefinida de 10 Hz, esta es la configuración recomendada si se desea modelar dichos mensajes.

4. Tamaño del mensaje: El tamaño del mensaje se establece en 190 bytes. Este tamaño es consistente con el de los BSM y mensajes similares.

5. Número de subcanales: El ancho de banda se puede dividir en 2 o 4 subcanales y el MCS (*Modulation and Coding Scheme*) se ajusta al número de subcanales cambiando la tasa de datos dinámicamente. MCS determina la velocidad de transmisión de datos al seleccionar la combinación adecuada de esquema de modulación y tasa de codificación para adaptarse a las condiciones actuales de la red. Cuanto mayor es el MCS, menor es la fiabilidad de la transmisión, pero se transmiten más mensajes en el mismo ancho de banda ya que se utilizan modulaciones más complejas, sin embargo, este tipo de modulaciones son

susceptibles a errores de transmisión debido a interferencias o ruidos en el canal de comunicación.

En la Figura 41, se presenta un ejemplo de una curva PDR de cien puntos para distancias que varían entre 5 y 955 metros. Esta curva se generó mediante el modelo analítico, considerando una densidad de tráfico (β) de 60 vehículos por kilómetro. Se asume que los vehículos transmiten 10 mensajes por segundo (*rate*), cada uno de 190 bytes, con una potencia de transmisión de 23 dBm, y teniendo en cuenta un total de 4 subcanales en el canal de 10 MHz del LTE-V2X empleado.

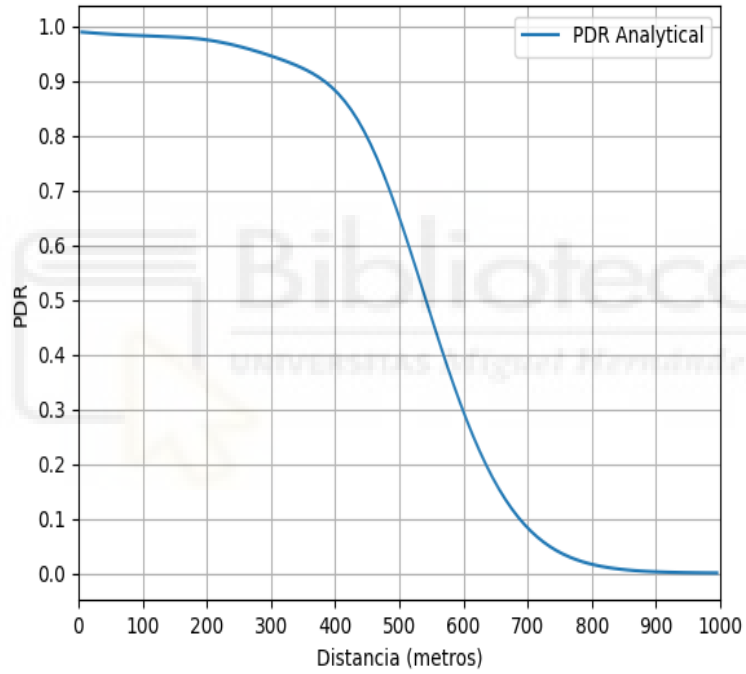
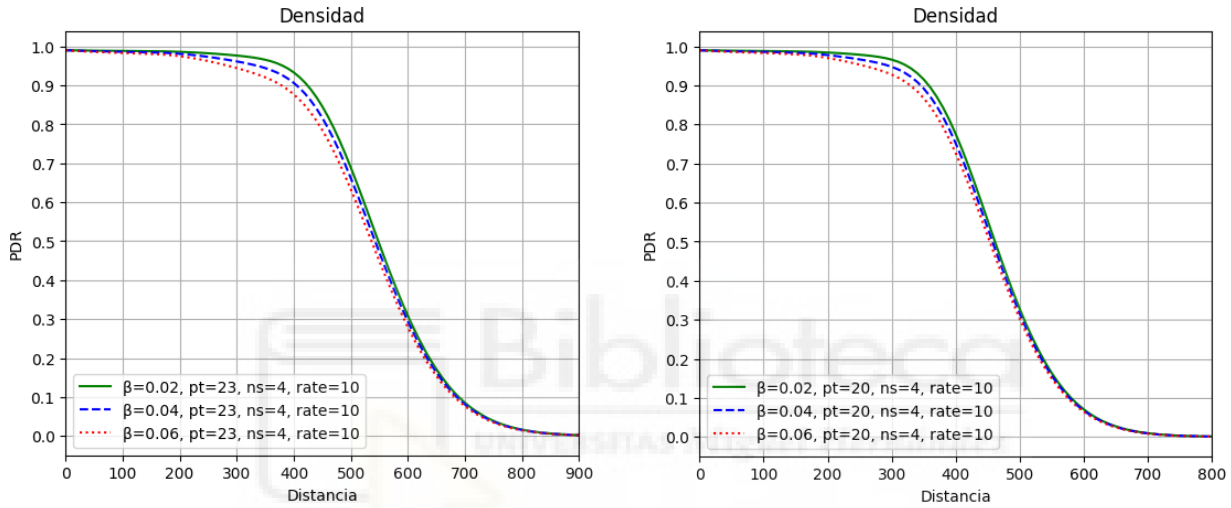


Figura 41. PDR analítica del modelo.

Para las simulaciones ejecutadas, se emplearon diversas PDR que definen la probabilidad de entrega basándose en factores como la densidad de vehículos, la tasa de paquetes, el número de subcanales y la potencia de transmisión. A fin de generar distintas curvas en función de estos parámetros, se establecieron valores predefinidos en las simulaciones para densidades de vehículos bajas (β=0,02), medias (β=0,04) y altas (β=0,06). Se consideraron también números de subcanales bajos (*ns*=2) y altos (*ns*=4), tasas de paquetes bajas (*rate*=10) y altas (*rate*=20), así como potencias de transmisión bajas (*pt*=20) y altas (*pt*=23).

Para analizar la influencia de estos parámetros en la simulación, se establecen inicialmente valores fijos de potencia baja (*pt=20*), un número alto de subcanales (*ns=4*), una densidad media de vehículos (β=0,04) y una tasa de paquetes baja (*rate=10*).

Como primer análisis, se muestra el efecto de variar la densidad de vehículos, probando con densidades bajas (β=0,02), medias (β=0,04) y altas (β=0,06), manteniendo constantes los demás parámetros fijos. Posteriormente se aumenta a una potencia alta (*pt=23*), para visualizar el efecto de este aumento en el análisis de la densidad vehicular.



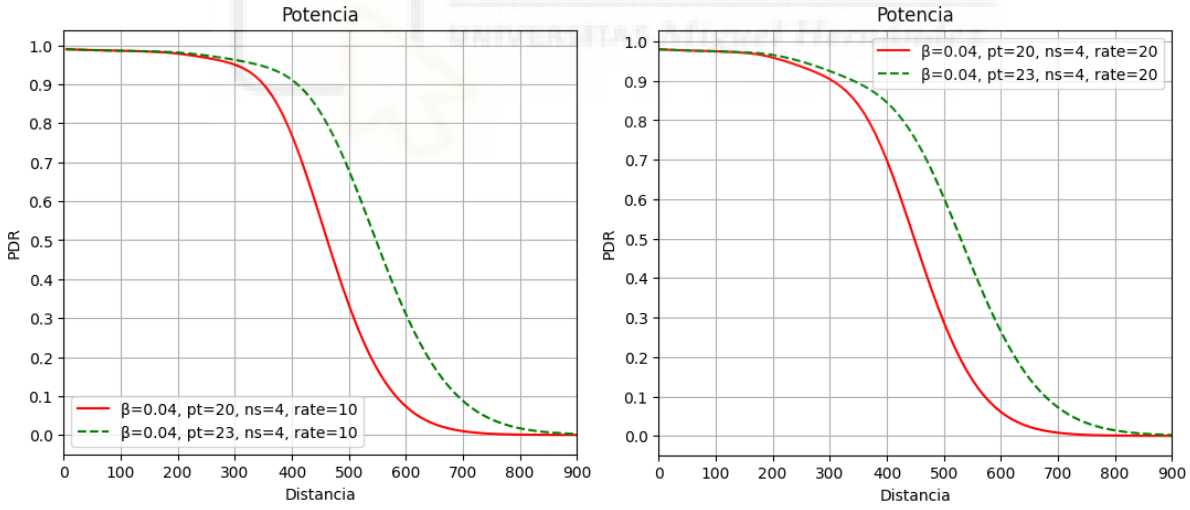
(a) Potencia a 20dBm (b) Potencia a 23 dBm

Figura 42. Variación de densidad de vehículos.

La Figura *42* (a) presenta las diversas curvas resultantes de la variación en la densidad de vehículos. Se observa que, a menor densidad de vehículos, representada por la curva de color verde, se obtiene un rendimiento superior en términos de comunicación. En un escenario con baja densidad de vehículos, se favorece un mayor alcance de la comunicación y una mayor probabilidad de transmisión exitosa de paquetes.

Por otro lado, la densidad de vehículos media, ilustrada por la curva azul, presenta un alcance ligeramente disminuido en comparación con la densidad baja. Además, en un escenario de alta densidad vehicular, que se representa con la línea punteada de color rojo, se evidencia la menor capacidad de alcance. Esta alta densidad supone una reducción en la probabilidad de éxito en la entrega de paquetes.

La Figura *42* (b) exhibe la misma configuración con una excepción: la potencia se incrementa de 20 dBm a 23 dBm. Se observa el mismo efecto relacionado con la densidad de vehículos; es decir, a menor densidad de vehículos, se obtiene una mayor probabilidad de éxito en la tasa de rendimiento y en el alcance de la comunicación. Con el aumento de potencia, se aprecia una mejora considerable en las curvas, extendiéndose estas hasta una distancia de aproximadamente 100 metros. Por otro lado, se analiza el efecto de la potencia, variando entre potencias bajas (*pt=20*) y altas (*pt=23*). En este análisis, los demás parámetros se mantienen constantes, y luego se incrementa la tasa de paquetes de una configuración baja (*rate=10*) a una alta (*rate=20*) para evaluar cómo influye la variación de este parámetro.



(a) Tasa de transmisión a 10 Hz (b) Tasa de transmisión a 20 Hz

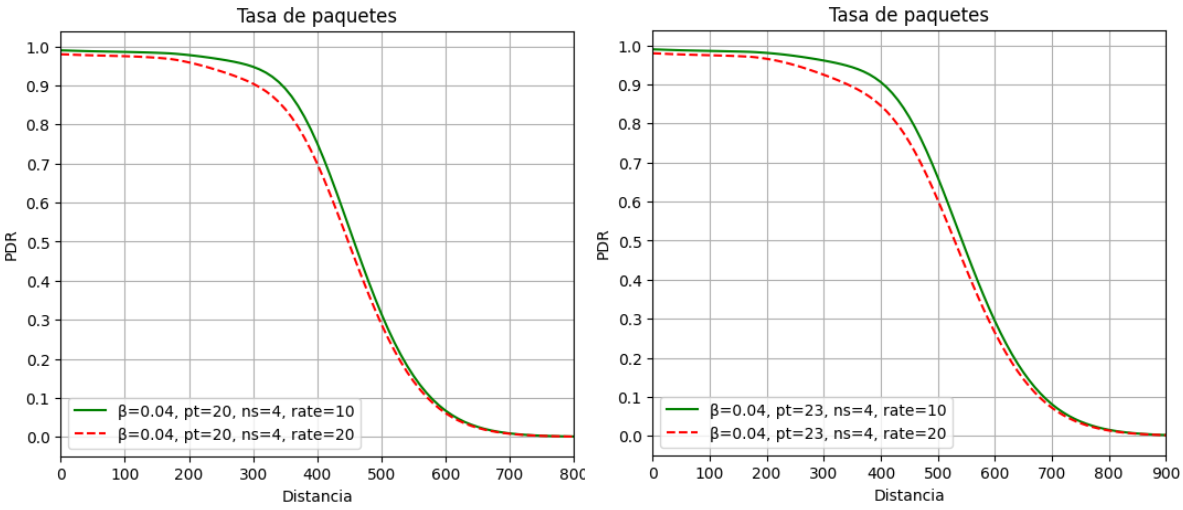
Figura 43. Variación de la potencia de vehículos a 20 Hz.

La Figura 43 (a) muestra el impacto de la potencia de transmisión en la tasa de entrega de paquetes (PDR). Con 23 dBm (línea punteada verde), se alcanza más distancia y mayor éxito en la entrega de paquetes que con 20 dBm (línea roja).

La Figura 43 (a) muestra el impacto de la potencia de transmisión en la tasa de entrega de paquetes (PDR). Con 23 dBm (línea punteada verde), se alcanza más distancia y mayor éxito en la entrega de paquetes que con 20 dBm (línea roja).

La Figura 43 (b) muestra las curvas resultantes al analizar la potencia al incrementar la tasa de paquetes a 20 Hz. Se observa una leve disminución en las curvas comparado con una tasa de paquetes de 10 Hz. Sin embargo, la tasa de paquetes no tiene un impacto tan significativo como la potencia, la cual sí tiene un efecto notable en las curvas.

A continuación, se analiza específicamente el efecto de la variación de la tasa de paquetes manteniendo fijos los demás parámetros. En este análisis, se varían las tasas de paquetes entre bajas (*rate=10*) y altas (*rate=20*) manteniendo los demás parámetros constantes. Posteriormente, se incrementa la potencia para observar su impacto frente a la tasa de paquetes.



(a) Potencia a 20 dBm

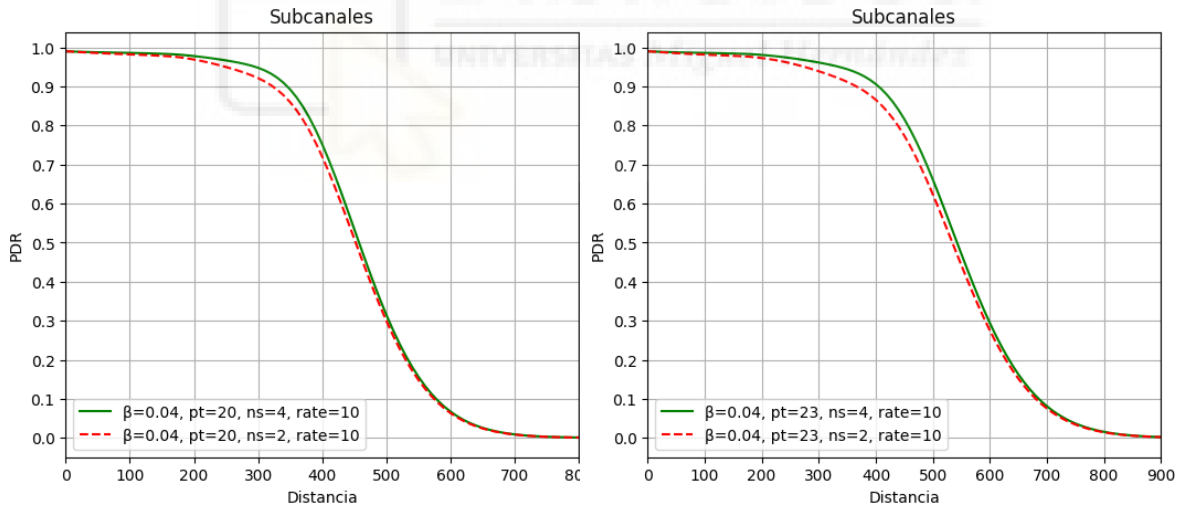
(b) Potencia a 23 dBm

Figura 44. Variación de la tasa de paquetes.

La Figura 44 (a) Figura *44*muestra que una menor tasa de paquetes, representada por la línea verde, resulta en una mejora en la tasa de éxito en la entrega de paquetes. Por otro lado, una tasa de paquetes alta, representada por la línea roja, conduce a un rendimiento inferior.

La Figura *44* (b) representa las curvas resultantes al incrementar la potencia a 23 dBm. Se observa un ligero mejor rendimiento con tasas de paquetes bajas, y un rendimiento menor con tasas altas. Este gráfico reafirma el impacto significativo de la potencia: se logra un mayor alcance que con una potencia de 20 dBm, mejorando las curvas para ambas configuraciones de tasas de paquetes.

Analizando también el efecto del número de subcanales, se comparan los resultados obtenidos con un número bajo de subcanales (*ns=2*) y un número alto de subcanales (*ns=4*). Asimismo, se incrementa la potencia en este escenario.



(a) Potencia a 20 dBm (b) Potencia a 23 dBm

Figura 45. Variación del número de subcanales.

La Figura 45 (a) presenta las curvas al variar el número de subcanales. Es notable que un número alto de subcanales (*ns=4*), representado por la línea verde, mejora ligeramente la probabilidad de éxito en la entrega de paquetes. En contraste, un menor número de subcanales (*ns=2*) resulta en una tasa de éxito inferior.

La Figura 45 (b) muestra que, al aumentar la potencia, sigue existiendo una diferencia ligera entre las dos curvas que representan la variación de los canales altos y bajos. No obstante, la potencia tiene un impacto significativo en la probabilidad y el alcance de ambas configuraciones.

Los gráficos mostrados previamente evidencian que un incremento en la potencia de transmisión conduce a una mejora en la tasa de entrega de paquetes (PDR), ampliando así el alcance de las comunicaciones. Sin embargo, este alcance es afectado por diversos factores como la densidad de vehículos, la tasa de paquetes y el número de subcanales.

En particular, cuando la densidad de vehículos es menor, el rendimiento de la comunicación mejora, permitiendo un mayor alcance y una mayor probabilidad de éxito en la transmisión de paquetes. Por otro lado, la potencia de transmisión juega un papel crítico. Al aumentar la potencia de 20 dBm a 23 dBm, se mejora de forma notable el rendimiento general, incrementando consecuentemente tanto el alcance de la comunicación como las tasas de éxito en la entrega de paquetes.

Además, la tasa de paquetes también afecta el rendimiento. En situaciones donde la tasa de paquetes es menor, se observa una mejora en la tasa de éxito de la entrega de paquetes. Por el contrario, una tasa de paquetes alta da lugar a un rendimiento inferior. Este fenómeno tiene lógica ya que, si se produce una alta tasa de paquetes, la cantidad de datos que se intenta transmitir puede superar la capacidad de la red. Esta sobrecarga puede provocar pérdida de paquetes o retrasos significativos, resultando en una disminución en la tasa de éxito de la entrega de paquetes.

Finalmente, el número de subcanales tiene un impacto ligero pero notable en la probabilidad de éxito de la entrega de paquetes. Un mayor número de subcanales parece mejorar esta probabilidad, mientras que un número menor da lugar a una tasa de éxito inferior.

Tras el análisis, se observa que varios parámetros afectan de forma significativa al rendimiento de las comunicaciones en las curvas del modelo. De todos ellos, la potencia de transmisión parece ser el más determinante, dado que su incremento mejora el rendimiento general de manera más notable, independientemente de las demás variables.

En los modelos analíticos de LTE-V2X, se destaca que incrementar la frecuencia de transmisión de mensajes a 20 Hz conduce a una disminución en la PDR debido a la presencia de interferencias más intensas. Sin embargo, este fenómeno puede ser mitigado con el uso de cuatro subcanales, que contribuye a un aumento en la PDR. Esta mejora se atribuye a una disminución en el esquema de modulación y codificación (MCS), lo que posibilita transmisiones más robustas y eficaces.

En resumen, la eficacia y el alcance de las comunicaciones en LTE-V2X están influenciados por un balance entre la potencia de transmisión, la frecuencia de los mensajes y el número de subcanales. En las curvas de PDR mostradas anteriormente podemos apreciar que la teoría en la que se basada el modelo es consecuentes con las gráficas obtenidas.

4.6.2 REPRESENTACIÓN GRÁFICA

La representación de la PDR en tiempo de simulación requiere un recordatorio importante: cuando un vehículo detecta un obstáculo en la función V2X, se producen cálculos significativos de distancias relativas y decisiones basadas en el modelo analítico acerca de la recepción de un mensaje. Sin embargo, la acción no se limita a eso; también se realizan importantes registros de datos. Como se comentó, existen varias funciones, incluyendo *getData*, la cual se encarga de la recolección de estos datos.

Si un vehículo es apto para recibir un paquete, su distancia relativa al transmisor se registra. Del mismo modo, si un vehículo no logra recibir un paquete, se almacena igualmente su distancia relativa. Estos registros de distancia se almacenan en dos

variables: *dist_ok* y *dist_no_ok*. Cada entrada en *dist_ok* puede interpretarse como la distancia a la que un vehículo ha recibido con éxito un paquete.

En contraposición, cada entrada en *dist_no_ok* refleja la distancia a la que un vehículo ha fallado en la recepción de un paquete.

En estas variables se recogen todas las distancias relativas durante el transcurso de la simulación donde la recepción de paquetes fue o no exitosa. A partir de estas variables, se puede deducir la PDR, que es la relación entre paquetes entregados y el total de paquetes enviados. De este modo, *dist_ok* y *dist_no_ok* brindan una colección de distancias relativas que denotan paquetes que han sido o no enviados, facilitando la obtención de una PDR empírica basada en los datos de la simulación.

Para calcular esta PDR empírica, se itera a través de las distancias relativas almacenadas utilizando un bucle *for*.

```
for d in dis_analytical:
```

Donde *dis_analytical* es el vector distancias:

```
dis_analytical = np.arange(5, 1000, 10)  # Distancia analítica
```

Iterando este vector con las siguientes instrucciones:

```
lim_inf = d
lim_sup = d + 10
time.sleep(0.05)
pack_ok = len(np.nonzero(np.logical_and(dist_ok < lim_sup, dist_ok >= lim_inf))[0])
pack_no_ok = len(np.nonzero(np.logical_and(dist_no_ok < lim_sup, dist_no_ok >= lim_inf))[0])
```

Esta secuencia de código se utiliza para identificar los paquetes que se reciben en los rangos de distancias del vector (5, 15, 25, 35...995).

Esta información se utiliza para determinar las probabilidades que nos permitirán calcular la PDR de la simulación, o empírica.

En la primera iteración del bucle, *lim_inf* es igual a 5 y *lim_sup* es igual a 15. Por tanto, *pack_ok* contendrá aquellos valores de distancias relativas que se clasificaron como *dist_ok* dentro de ese rango de distancias. Por ejemplo, si nuestra lista de distancias en metros es dist_ok = [3.3, 4.6, 6.3, 14.6, 20.7], en la iteración donde d=5 (es decir, el rango de búsqueda es [5,15]), sólo las distancias 6.3 y 14.6 caen dentro de este rango. Por lo tanto, *pack_ok* sería 2, lo que indica que hubo dos paquetes que se recibieron correctamente en ese rango de distancias. De manera similar, se calcula *pack_no_ok*.

Luego se calcula la PDR empírica de la siguiente manera:

```
pdr_emp = pack_ok / (pack_ok + pack_no_ok)

dist_empirical.append(d)

pdr_empirical.insert(d, pdr_emp)
```

La variable *pdr_emp* almacena una probabilidad correspondiente al rango de distancias evaluado. Al finalizar la ejecución del bucle, tendremos cien puntos de probabilidad para cien puntos de distancia, suficiente información para poder trazar la PDR de la simulación. La distancia que se evalúa se agrega a una lista denominada *dist_empirical* y los valores de la PDR obtenida se inserta en una lista *pdr_empirical*.

Es importante señalar que, durante una simulación, todos los vehículos pueden generar un evento a través de su sensor de obstáculos, lo que significa que los datos generados por todos los vehículos que participan en la simulación deben ser almacenados. Si el número de vehículos es elevado, esto resultará en grandes cantidades de datos, y las listas *dist_ok* y *dist_no_ok* pueden llegar a contener bastantes datos. Estos datos son fundamentales para calcular las probabilidades requeridas para determinar la PDR de la simulación, cuyo objetivo principal es proporcionar una representación gráfica de los

resultados obtenidos. En este sentido, la calidad de esta representación gráfica dependerá en gran medida del volumen de datos recopilados.

4.7 SIMULACIONES

Esta sección representa una de las partes más importantes de este trabajo de fin de grado. Aquí, se combina la teoría y la práctica, tal como se describen en las secciones anteriores, para la configuración y realización de la simulación. Esta sección se subdivide en dos áreas principales: la primera es una subsección dedicada a la configuración y selección de parámetros que se utilizarán en la simulación, y la segunda es una subsección en la que se presentan los resultados obtenidos de la simulación, basándose en las condiciones y parámetros previamente seleccionados.

4.7.1 CONFIGURACIÓN Y SELECCIÓN DE PARÁMETROS

Anteriormente se trató el tema de los argumentos necesarios para el desarrollo de la simulación y el modelo analítico que se utiliza y, como estos conforman la base para la simulación que realiza. Tal como se mencionó, la elección de una PDR en particular estará determinada por los argumentos proporcionados por el usuario. Como se pudo observar, existen múltiples curvas que se generan al fijar un parámetro y variar los demás. Para mantener esta sección concisa, se opta por una configuración específica. Con esta configuración, basada en parámetros como la densidad de vehículos, el número de subcanales, la tasa de paquetes y la potencia de transmisión, se prestará atención a las curvas de PDR que brinden distancias de transmisión máximas y mínimas.

Se realizan cuatro simulaciones manteniendo la potencia constante mientras se varían el número de subcanales y la tasa de paquetes. Las simulaciones se realizan con una potencia de transmisión fija de 23 dBm, mientras que los subcanales y la tasa de paquetes se varían para densidades bajas de vehículos (20 veh/km), y altas (60 veh/km). El objetivo es obtener las PDR empíricas, en función de estos parámetros y la densidad de vehículos.

La configuración descrita anteriormente se eligió con el fin de reducir la necesidad de simular una excesiva cantidad de curvas. Mediante la simulación con esta configuración específica, se obtienen resultados representativos que ilustran los resultados que se

alcanzarían si se realizaran simulaciones para todas las curvas producidas al mantener constante un parámetro y variar los restantes.

Los parámetros que se establecen para llevar a cabo la simulación en los dos mapas son los siguientes:

- ***--time-seconds:*** 240 (segundos)
- ***--distance_detected*:** 4 (metros)
- ***--ego*:** False
- ***--rgb_cam*:** True
- ***--res*:** 720x720 (píxel)
- ***--pdr*:** True
- ***--pot-trans:*** 23 (dBm)
- ***--num-sub****:* 2/4
- ***--rate:*** 10 / 20 (Hz)
- ***--beta****: 0,02 / 0,04 / 0,06 (vehículo/metro)*

Como se mencionó anteriormente, se realizaron cuatro simulaciones utilizando diferentes combinaciones de parámetros. Dichas configuraciones producen las curvas PDR y establecen distancias máximas y mínimas de transmisión. La configuración para la distancia máxima incluye una potencia de 23 dBm, una densidad de vehículos (β) de 0.02 veh/m, 4 subcanales y una tasa de paquetes de 10 Hz. Por otro lado, la configuración para la distancia mínima cuenta con una potencia de 23 dBm, una densidad de vehículos (β) de 0.06 veh/m, 2 subcanales y una tasa de paquetes de 20 Hz. Las curvas PDR que arrojan las combinaciones descritas se pueden apreciar en la Figura 46.

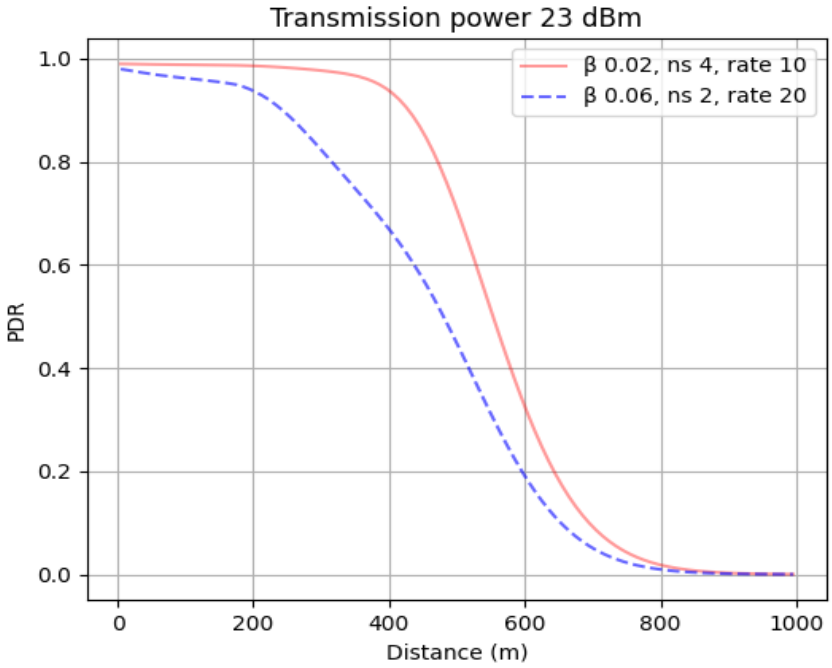


Figura 46. PDR de referencia para realizar las simulaciones.

4.7.2 DESARROLLO DE LA SIMULACIÓN

La simulación comienza en el mapa "Town 2" con la configuración siguiente: se establece un tiempo de duración de 240 segundos, se desactiva la opción de que haya un vehículo ego (todos los vehículos tienen un sensor asociado y pueden detectar obstáculos) y se activa la cámara RGB. Además, se establece una potencia de transmisión de 23 dBm, se configuran 4 subcanales, se fija una tasa de paquetes de 10 Hz y se ajusta la densidad de vehículos a 0.02 veh/m. La Tabla 8 muestra de manera resumida lo anterior. En este caso, se simula para la obtención de la curva PDR más favorable (en color rojo) que se muestra en la Figura 46.

Parámetro	Valor
Tipo de mapa	Town 02
Longitud de carretera	1.457 (m)
Número de frames	4.800
Número de vehículos	89
ts, --time-seconds	240 (s)
--ego	False
--rgb_ego	True
-pt, --pot-trans	23 (dBm)
-ns, --num-sub	4
-r, --rate	20 (Hz)
-beta, --beta	0.02 (veh/m)

Tabla 8. Parámetros de simulación 1.

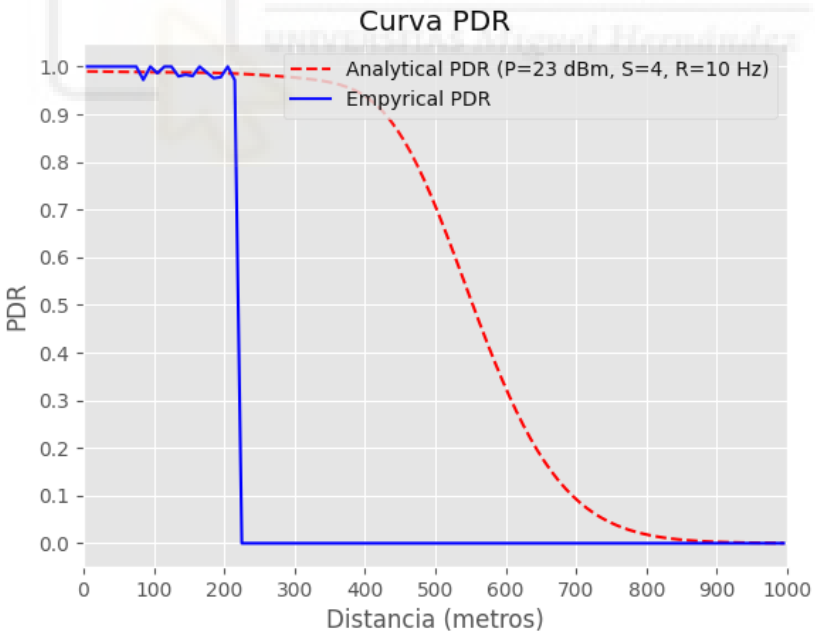


Figura 47. 1ª gráfica de PDR obtenida para el mapa "Town 2" con una densidad de veh/m baja.

La Figura 47 muestra la representación de la PDR del modelo analítico en rojo, contrastando con la curva empírica en azul. Esta última está relacionada con el cálculo inicial de probabilidades para las distancias relativas entre el vehículo que detecta un obstáculo y los demás vehículos en el mapa, registrado en el *frame* 313 (15,65 segundos). Se destaca que la curva PDR empírica no sigue la tendencia de la analítica, lo cual es coherente si se considera que la gráfica se genera cercana al *frame* donde se lleva a cabo la primera detección, ya sea de un vehículo o más. En este contexto, el número total de vehículos simulados alcanza los 30, una densidad baja.

Cabe resaltar que, en el marco de este trabajo, no sólo se muestra la gráfica en tiempo real, sino que también se producen y almacenan gráficas estáticas. Al representar y almacenar una gráfica, se registra el momento correspondiente en el mundo simulado. Cada paso de la simulación, guiado por el servidor, es señalado por el cliente a través de un *tick* del mundo (*tick world*). Esto permite obtener la posición exacta en la que se encuentra la simulación al solicitar al servidor el *frame* del mundo y con él el tiempo en el mundo real.

Cuando se almacena la gráfica, se incluye el *frame* para estimar de forma aproximada el tiempo simulado en el que se efectúa la representación gráfica de los datos.

Es importante tener en cuenta que el número de vehículos se establece automáticamente en función de la longitud de la carretera, para mantener la coherencia con la densidad de vehículos del modelo. Tras la primera activación del módulo V2X por una detección, las listas *dist_ok* y *dist_no_ok* comienzan a acumular una cantidad limitada de información, específicamente correspondiente al *frame* o instante en el que se genera el evento inicial.

En concreto, para este caso la primera lista albergará 29 elementos. Esta cantidad representa las distancias relativas desde el vehículo que detectó un obstáculo hasta cada uno de los restantes 29 vehículos presentes en la simulación. Es importante señalar que en este cálculo no se incluye la distancia del vehículo que realiza la detección a sí mismo. Esta información es la que se registra en el *frame* inicial tras la detección.

Las fluctuaciones en la curva empírica se deben precisamente a esta circunstancia, atribuibles a la falta de información sobre las distancias relativas en este instante inicial.

Conforme se menciona en la sección 4.6.2 de representación gráfica, estas listas son fundamentales para calcular la PDR. Por consiguiente, las probabilidades se determinan en función de los mensajes recibidos o no, y dependen de estas listas. De este modo, conforme se reúnan más datos en las listas, la curva PDR se acercará a la curva PDR analítica. En síntesis, a medida que se disponga de más datos para el cálculo de las probabilidades, mayor será la similitud entre ambas curvas.

La Figura 48 muestra una tercera gráfica obtenida tras un tiempo adicional de simulación, específicamente en el *frame* 496 (24,8 segundos), evidenciando así la evolución comparativa con respecto a la Figura 51.

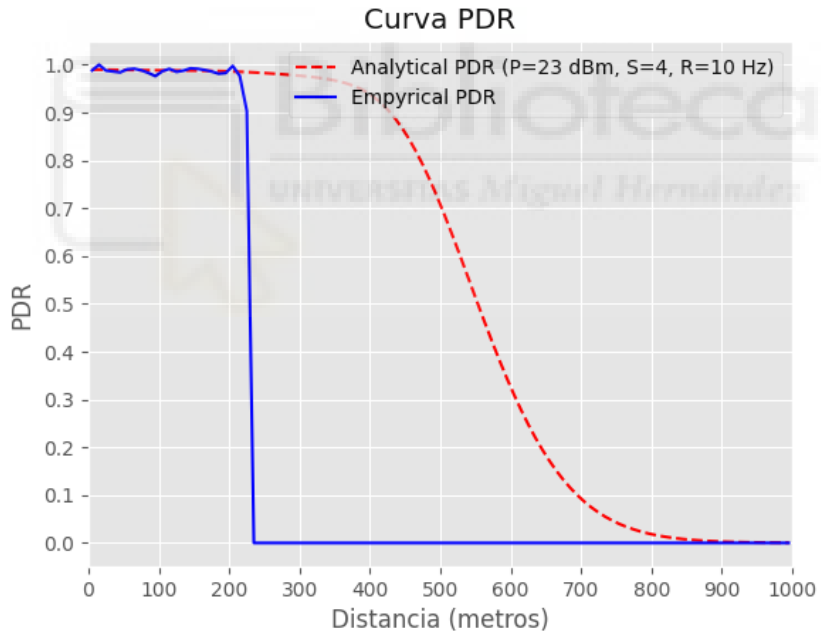


Figura 48. 3ª gráfica de PDR obtenida para el mapa "Town 2" con una densidad de veh/m baja.

Por otra parte, la Figura 49 muestra una aproximación correspondiente a un período de simulación más extenso, precisamente en el *frame* 4.779 (238,95 segundos), última gráfica obtenida para esta simulación. Hay que recordar que el tiempo de simulación está configurado en 240 segundos (4.800 *frames*). En este último caso, es notable la tendencia

marcada que adopta la curva de la PDR empírica hacia la PDR analítica, fundamentada en el modelo.

En este último caso, es notable la tendencia marcada que adopta la curva de la PDR empírica hacia la PDR analítica, fundamentada en el modelo.

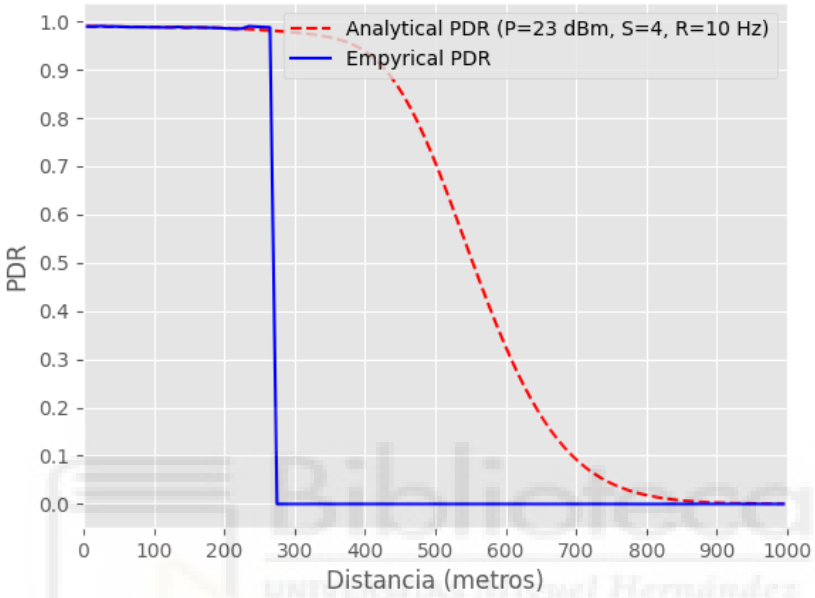


Figura 49. 21ª gráfica de PDR obtenida para el mapa "Town 2" con una densidad de veh/m baja.

Es relevante comentar sobre la pronunciada caída que se observa en las gráficas, que es consecuencia del mapa utilizado para la simulación. Se había mencionado anteriormente que se optó por *Town 6* debido a que *Town 2* restringía la visualización de algunos resultados. En el caso de *Town 02*, la gráfica refleja la distancia máxima en la que un vehículo puede o no recibir un paquete con una probabilidad cercana al 98%, a una distancia aproximada de 250 metros, que coincide con la dimensión de este mapa. *Town 2* restringía la representación de cómo se desempeñaría la curva de PDR empírica para distancias superiores a 250 metros. Por este motivo, se decidió utilizar *Town 6*.

Las simulaciones pueden realizarse sin mostrar nada gráficamente, guardando los resultados para su posterior visualización. Sin embargo, se hicieron algunas adaptaciones en el script *v2x.py* para poder ver todo el proceso de simulación en tiempo real, como, por

ejemplo, observar un vehículo en movimiento en el entorno simulado. Estas adaptaciones permiten visualizar tanto un vehículo ego, si se elige esta opción, como cualquier vehículo dentro del mundo a través de una cámara que muestra al actor en acción. La visualización del actor se puede cambiar presionando la tecla *SHIFT*.

Los datos generados a lo largo de la simulación, además de ser almacenados en un archivo CSV, se muestran en consola durante el transcurso de la simulación. Estos datos también se utilizan para mostrar gráficamente lo que está sucediendo en la simulación.

En la Figura 50 se muestra cómo se visualiza el entorno. Se puede apreciar una parte de la consola en la que se imprime información relevante tras la detección, el gráfico en tiempo real y la cámara RGB que sigue al vehículo. De la información de la consola en la figura, se observa que en el fotograma 1.564 (78,2 segundos), el vehículo 7, que es un *Tesla Model 3*, detecta a una distancia de 3 metros al vehículo 14, un *Mini Cooper*, tal como se puede ver en la cámara RGB asociada al Tesla.

Cabe aclarar que la posición de la cámara asociada al vehículo se determina con el objetivo de obtener una mejor visualización del entorno. Por lo general, las cámaras se sitúan en relación con la estructura del vehículo.

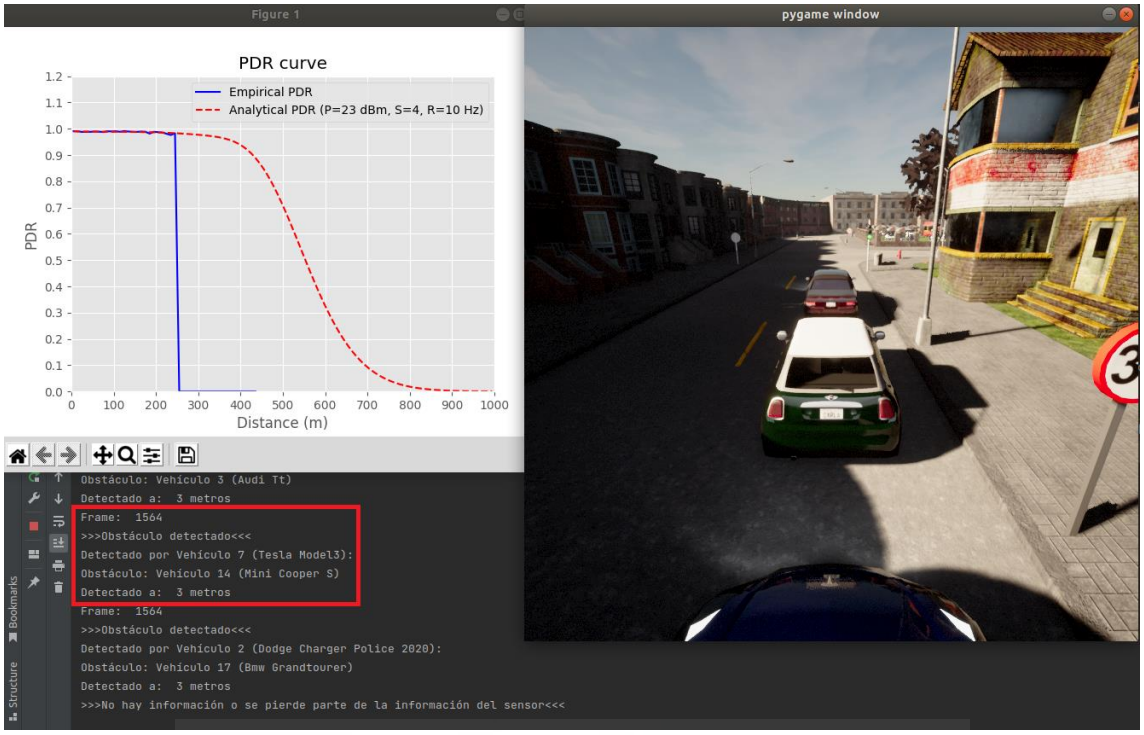


Figura 50. Vista gráfica del entorno de simulación en Town 2.

Una vez concluida la simulación y en base a esta curva, se observó que se enviaron 611.465 mensajes en total. De estos, se logró recibir el 98,82%, lo que significa que el 1,18% de los mensajes enviados por los vehículos no fue recibido.

Se plantea ahora el mismo escenario, utilizando el mismo mapa. En este caso se mantiene la potencia de transmisión a 23 dBm, se reduce el número de subcanales a 2 y se aumenta tanto la densidad de tráfico a 60 veh/m como la tasa de paquetes a 20 Hz. La Tabla 9 muestra en resumen los parámetros configurados para esta simulación. Para este caso obtendremos la curva PDR más desfavorable que se muestra en la Figura 50.

Parámetro	Valor
Tipo de mapa	Town 02
Longitud de carretera	1.457 (m)
Número de frames	4.800
Número de vehículos	89
ts, --time-seconds	240 (s)
--ego	False
--rgb_ego	True
-pt, --pot-trans	23 (dBm)
-ns, --num-sub	2
-r, --rate	20 (Hz)
-beta, --beta	0.06 (veh/m)

Tabla 9. Parámetros de simulación 2

Esta combinación de parámetros de configuración hace que se tome la PDR que menor distancia de comunicación presenta con una potencia alta. La Figura 51 muestra la primera gráfica obtenida tras la primera detección, repitiéndose nuevamente lo explicado anteriormente: la inconsistencia de la representación de la curva empírica debida a la falta de datos de distancias relativas. Esta primera figura es tomada en el *frame* 292 (14,6 segundos).

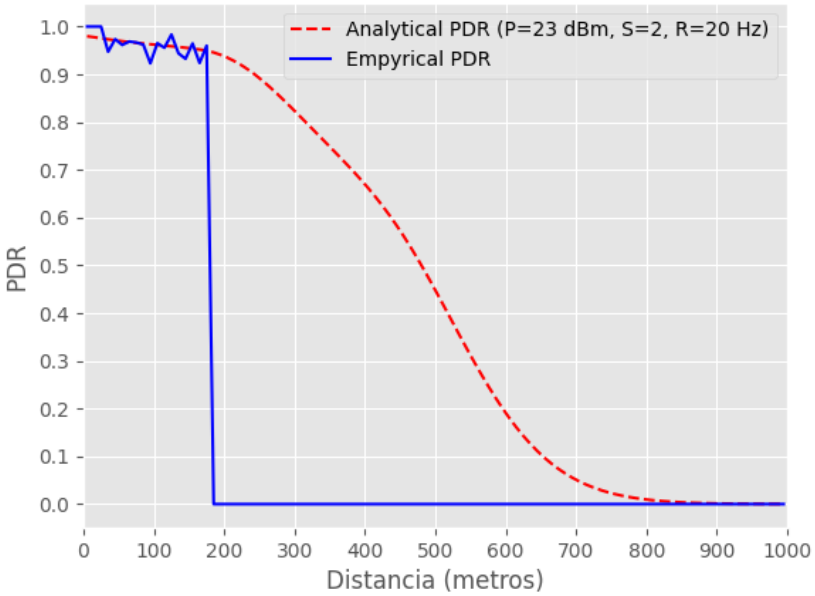


Figura 51. 1ª gráfica de PDR obtenida para el mapa "Town 2" con una densidad de veh/m alta.

Una vez transcurrido cierto tiempo desde el inicio de la simulación, observamos en la Figura 52 correspondiente al *frame* 2407 (120,35 segundos), la vigésima séptima imagen registrada. Se puede apreciar una clara diferencia respecto a la imagen inicial. En este caso, la curva del PDR empírica sigue la misma línea que la analítica.

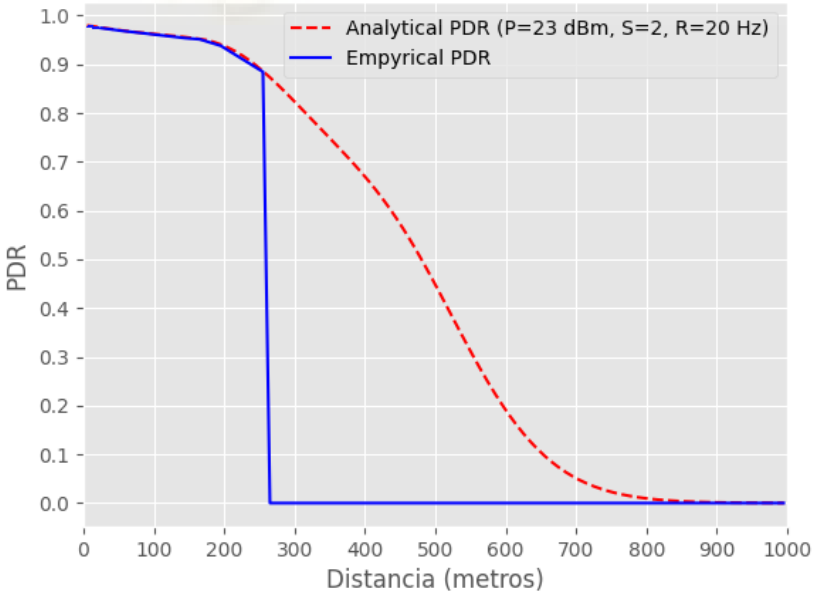


Figura 52. 27ª gráfica de PDR obtenida para el mapa "Town 2" con una densidad de veh/m alta.

Al finalizar la simulación, se comprobó que se transmitieron un total de 1.048.576 mensajes. De estos, se logró entregar el 95.78%, lo que implica que el 4.22% de los mensajes enviados por los vehículos no se recibieron.

Se simula ahora en el mapa Town *6*. Se lanzan las mismas simulaciones de antes, pero en el mapa que permite una visualización completa de la PDR debido a su tamaño. La Tabla 10 muestra la configuración.

Parámetro	Valor
Tipo de mapa	Town 06
Longitud de carretera	5.061 (m)
Número de frames	4.800
Número de vehículos	101
ts, --time-seconds	240 (s)
--ego	False
--rgb_ego	True
-pt, --pot-trans	23 (dBm)
-ns, --num-sub	4
-r, --rate	10 (Hz)
-beta, --beta	0.02 (veh/m)

Tabla 10. Parámetros de simulación 3.

La Figura 53 tomada en el *frame* 463 (23,15 segundos) presenta la primera gráfica registrada tras las detecciones iniciales. Se puede apreciar que la curva de PDR empírica tiende a seguir la analítica, aunque con variaciones significativas. Como se mencionó anteriormente, estas variaciones se deben a la escasez de datos.

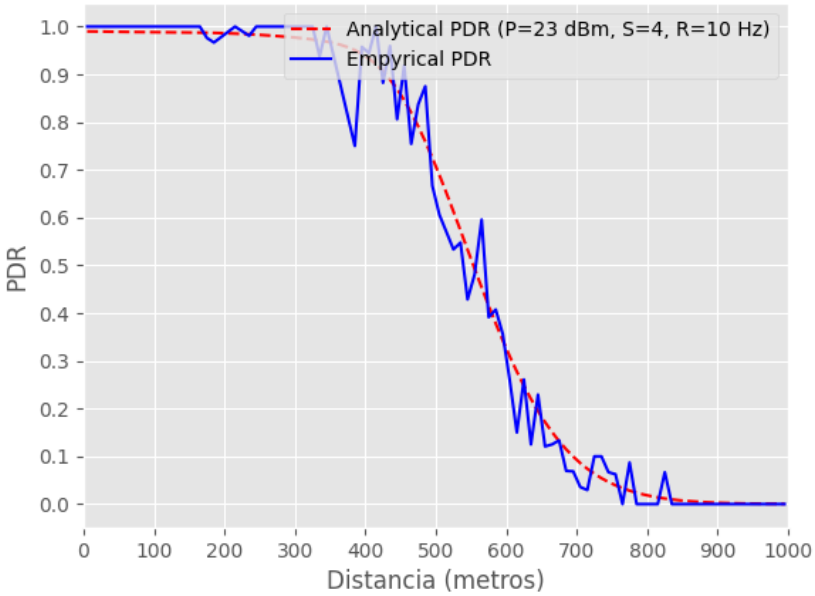


Figura 53. 1ª gráfica de PDR obtenida para el mapa "Town 6" con una densidad de veh/m baja.

La Figura 54 muestra el escenario de simulación, similar al presentado anteriormente, pero situado en este caso en el mapa Town 6. En el fotograma 1.152 (57,6 segundos), se puede apreciar cómo el vehículo 12, un *Toyota Prius*, detecta al vehículo 46, un *Nissan Patrol*, a una distancia de 3 metros.

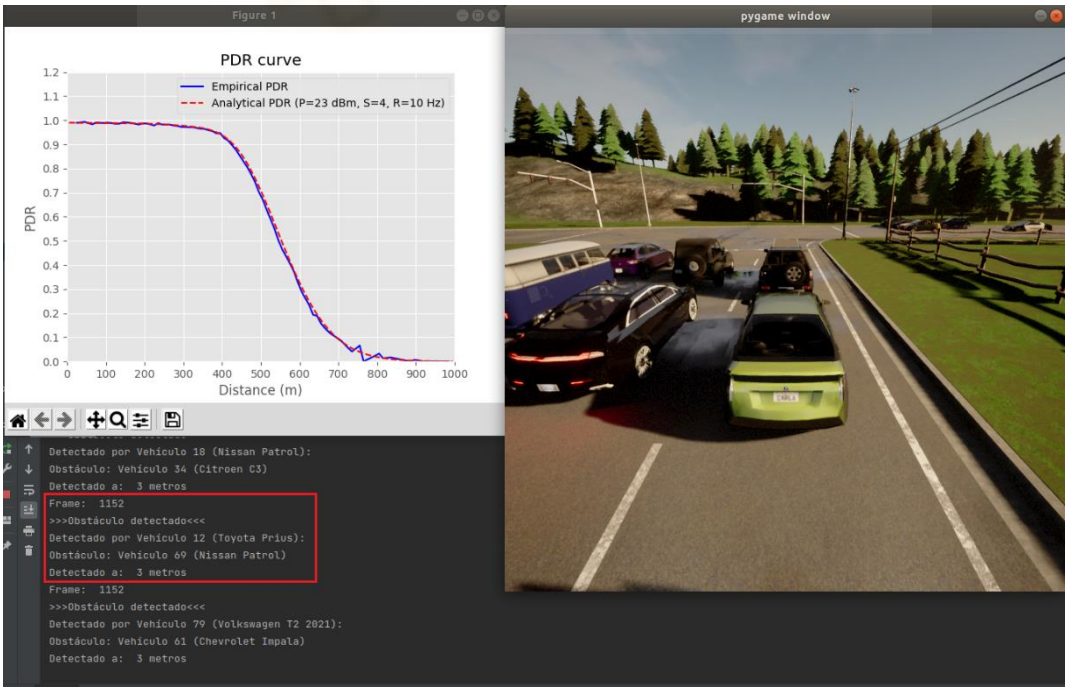


Figura 54. Vista gráfica del entorno de simulación en Town 6.

En la Figura 55, correspondiente al *frame* 2.260 (113 segundos), se puede apreciar que la curva empírica se asemeja bastante a la analítica. Esta correspondencia se registra en la vigésimo tercera gráfica.

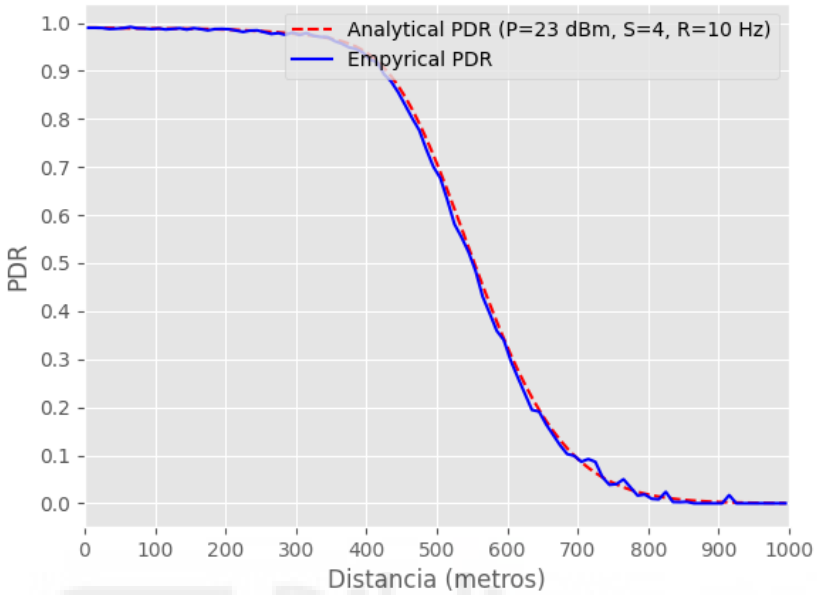


Figura 55. 23ª gráfica de PDR obtenida para el mapa "Town 6" con una densidad de veh/m baja.

Una vez concluida la simulación, se comprobó que se enviaron en total 483.901 mensajes. De este conjunto, se logró recibir el 81.39%, lo que indica que el 18.61% de los mensajes transmitidos por los vehículos que detectaron un obstáculo no fue recibido.

Se ejecuta nuevamente la simulación, pero esta vez reduciendo el número de subcanales a 2, aumentando la tasa de paquetes a 20 Hz y utilizando una densidad de vehículos alta de 60 veh/m. Los detalles de esta configuración se presentan en la Tabla 11.

Parámetro	Valor
Tipo de mapa	Town 06
Longitud de carretera	5.061 (m)
Número de frames	4.800
Número de vehículos	304
ts, --time-seconds	240 (s)
--ego	False
--rgb_ego	True
-pt, --pot-trans	23 (dBm)
-ns, --num-sub	2
-r, --rate	20 (Hz)
-beta, --beta	0.06 (veh/m)

Tabla 11. Parámetros de simulación 4.

La Figura 56 muestra la primera gráfica que se obtiene tras las primeras detecciones, tomada en el *frame* 260 (13 segundos) se observa que sigue el mismo patrón explicado anteriormente.

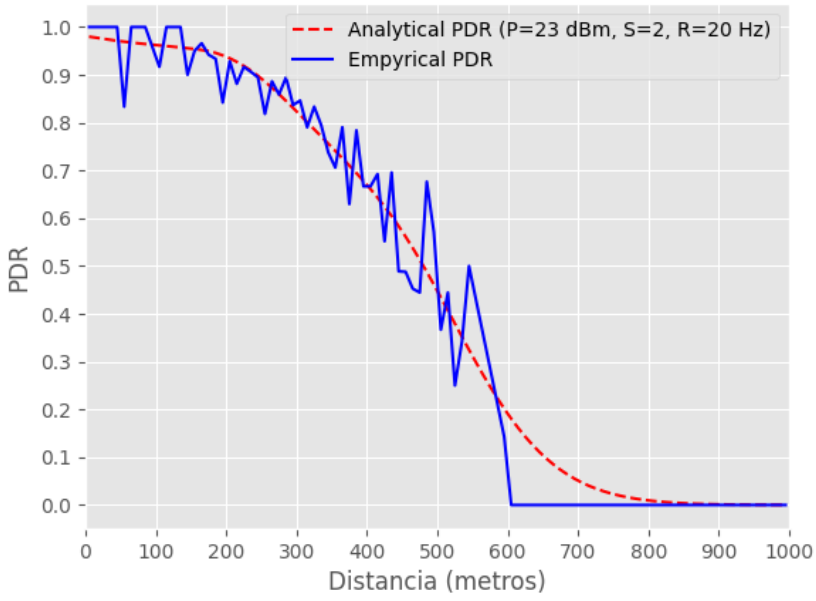


Figura 56. 1ª gráfica de PDR obtenida para el mapa "Town 6" con una densidad de veh/m alta.

La Figura 57 muestra la trigésima gráfica que se guarda tomada en el *frame* 3.125 (156,25 segundos), donde se observa que la PDR empírica toma una curva que sigue a la PDR analítica. Se aprecia como el mapa *Town 6* nos proporciona una representación visual adecuada debido a su tamaño, lo que nos permite apreciar cómo la curva de la PDR empírica sigue de cerca la curva analítica basada en el modelo a lo largo de la distancia evaluada. Como es de esperar, a medida que se realizan más aproximaciones o, dicho de otra manera, se disponga de más datos para calcular qué paquetes se reciben o no en los distintos rangos de distancias evaluados, la curva de la PDR empírica se acercará aún más a la curva analítica.

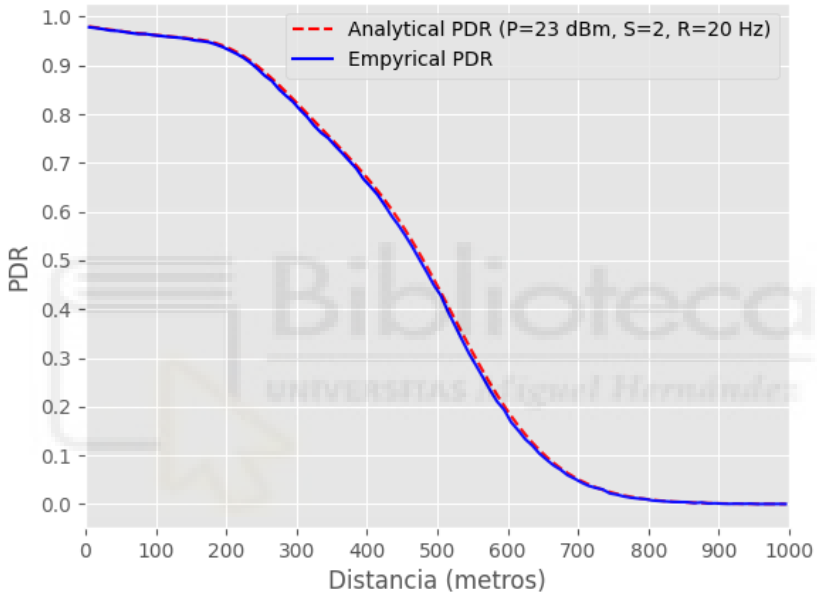


Figura 57. 30º gráfica de PDR obtenida para el mapa "Town 6" con una densidad de veh/m alta.

En la última simulación, se transmitieron un total de 1,048,576 mensajes. De estos, se recibió el 61.76%, lo que significa que el 38.24% de los mensajes enviados no se recibieron.

En el análisis de las cuatro simulaciones, es evidente cómo la tasa de entrega de paquetes está estrechamente vinculada con los parámetros de lanzamiento. Por ejemplo, en la primera simulación, se empleó una potencia de transmisión de 23 dBm, se utilizó 4 subcanales, se estableció una tasa de paquetes de 20 Hz, y se fijó una densidad de vehículos (β) de 0.02 veh/m.

Como resultado, se logró el rendimiento más notable con respecto a la tasa de éxito en la entrega de mensajes de todas las simulaciones. Esta eficacia puede ser atribuida a la baja densidad de vehículos en un mapa relativamente pequeño como Town 02, complementado con una configuración que, en este contexto particular, optimiza el alcance de las comunicaciones.

En la segunda simulación, se mantuvieron los mismos parámetros, excepto en la densidad de vehículos, que se incrementó a 0.06 veh/m, y en el número de subcanales, que se redujo a 2. A pesar de estos cambios, se logró una tasa de recepción de mensajes de 95,78% de un total de 1.048.576 mensajes enviados, lo que evidencia que el incremento en la densidad vehicular y la reducción en el número de subcanales pueden impactar en el rendimiento de las comunicaciones, sin necesariamente comprometer de manera significativa la tasa de éxito en la entrega de mensajes.

En la tercera simulación, que se llevó a cabo en un mapa de mayor extensión como *Town 6* y que mantuvo la configuración de la primera simulación, se observó una disminución en la tasa de éxito a 81,39% del total de 483.901 mensajes enviados. Este cambio se puede atribuir al incremento en el número de vehículos (101 vehículos para una baja densidad) distribuidos en un mapa con una mayor extensión de carretera (5.061 metros), a pesar de mantener la cantidad máxima de subcanales. Esto sugiere que, al incrementar la distancia y añadir más vehículos a la red, se introduce mayor complejidad en la comunicación entre vehículos y se aumenta la probabilidad de errores en la recepción de los mensajes.

Finalmente, en la cuarta simulación, la tasa de recepción de mensajes disminuyó hasta el 61,76% de un total de 1,048,576 mensajes enviados. Este descenso puede ser atribuido a la alta densidad de vehículos (304 para una alta densidad) y a la reducción en el número de subcanales, ambos factores que podrían aumentar la probabilidad de colisiones en las transmisiones de mensajes.

Los resultados indican que factores como la densidad de vehículos, el tamaño del mapa, el número de subcanales, la potencia, la tasa de mensajes y la densidad de vehículos, pueden tener un impacto significativo en la eficacia de las comunicaciones vehiculares.

Estos no son hallazgos aleatorios, sino resultados que están en línea con las predicciones del modelo analítico que se ha aplicado.

En esta sección, se han presentado principalmente los resultados obtenidos y las gráficas, siguiendo una configuración específica. Esto se ha hecho de esta manera para demostrar que es posible obtener una PDR empírica que sigue la PDR analítica del modelo, y que los resultados son coherentes con lo que se esperaría según el modelo utilizado. Los resultados de estas simulaciones se pueden ver en la Tabla 12 y Tabla 13 donde se muestran los resultados para los mapas *Town 2* y *Town 6* respectivamente con los mejores resultados (en color verde) y peores (en color rojo) resultados obtenidos en cada escenario.

Mapa	Nº frames	Nº vehículos	t (s)	Potencia (dBm)	Nº Sub	Tasa (Hz)	Beta	Mensajes recibidos (%)	Mensajes no recibidos (%)
Town02	4800	89	240	20	2	20	0.06	95.27	4.73
Town02	4800	89	240	20	4	20	0.06	96.58	3.42
Town02	4800	30	240	23	2	10	0.02	98.66	1.34
Town02	4800	30	240	23	4	10	0.02	98.74	1.26
Town02	4800	30	240	20	4	10	0.02	98.73	1,27
Town02	4800	89	240	20	2	20	0.06	95.39	4.61
Town02	**4800**	**30**	**240**	**23**	**4**	**10**	**0.02**	**98.82**	**1.18**
Town02	4800	89	240	23	2	20	0.06	95.78	4.22
Town02	4800	89	240	20	2	10	0.06	97.07	2.93
Town02	**4800**	**89**	**240**	**20**	**2**	**20**	**0.06**	**95.10**	**4.90**
Town02	4800	30	240	23	4	10	0.02	98.80	1.20
Town02	4800	30	240	23	4	20	0.02	97.76	2.24

Tabla 12. Resultados de simulaciones en Town 2.

Mapa	Nº frames	Nº vehículos	t (s)	Potencia (dBm)	Nº sub	Tasa (Hz)	Beta	Mensajes recibidos (%)	Mensajes no recibidos (%)
Town06	4800	304	240	20	2	20	0.06	57.79	42.21
Town06	4800	304	240	20	4	20	0.06	60.25	39.75
Town06	4800	101	240	23	2	10	0.02	74.65	25.35
Town06	4800	101	240	23	4	10	0.02	68.99	31.01
Town06	4800	304	240	20	2	20	0.06	58.03	41.97
Town06	4800	101	240	20	4	10	0.02	69.38	30.62
Town06	**4800**	**101**	**240**	**23**	**4**	**10**	**0.02**	**81.39**	**18.61**
Town06	4800	304	240	23	2	20	0.06	61.76	38.24
Town06	4800	304	240	20	2	10	0.06	61.46	38.54
Town06	**4800**	**304**	**240**	**20**	**2**	**20**	**0.06**	**53.79**	**46.21**
Town06	4800	101	240	23	4	10	0.02	74.79	25.21
Town06	4800	101	240	23	4	20	0.02	73.26	26.74

Tabla 13. Resultados de simulaciones en Town 6.

En conclusión, los resultados obtenidos y el modelo LTE-V2X empleado nos permiten realizar las siguientes observaciones. En primer lugar, la densidad de vehículos juega un papel importante en el porcentaje de mensajes recibidos. Al examinar las tablas, es evidente que los escenarios con menor densidad de vehículos (89 y 30 vehículos) presentan una tasa de éxito en la entrega de mensajes superior a la de los escenarios con mayor densidad de vehículos (101 y 304 vehículos).

Esta observación es coherente con el modelo, ya que sugiere que un entorno con un mayor número de vehículos puede generar más interferencias en la comunicación, lo que a su vez puede resultar en una menor tasa de entrega de paquetes.

En segundo lugar, la potencia de transmisión también tiene un impacto significativo en el porcentaje de éxito en la entrega de mensajes. En términos generales, las simulaciones realizadas con una potencia de transmisión de 23 dBm presentan un porcentaje de mensajes recibidos ligeramente superior al de las simulaciones realizadas con una potencia de transmisión de 20 dBm. Esta observación también es coherente con el modelo, ya que un aumento en la potencia de transmisión puede incrementar el alcance de las comunicaciones, lo que a su vez puede resultar en una mayor tasa de éxito en la entrega de mensajes.

En tercer lugar, el número de subcanales parece tener un impacto significativo en la tasa de éxito en la entrega de mensajes. En general, las simulaciones configuradas con 4 subcanales presentan una tasa de éxito mayor que las simulaciones realizadas con 2 subcanales. Esta observación es coherente con el modelo, ya que sugiere que un mayor número de subcanales puede permitir una mayor capacidad de transmisión, lo que a su vez puede resultar en una mayor tasa de éxito en la entrega de mensajes.

Por último, la frecuencia de transmisión de mensajes parece tener un ligero impacto en la tasa de éxito en la entrega de mensajes. Los escenarios simulados a 10 Hz y 20 Hz presentan porcentajes de mensajes recibidos similares, aunque se puede apreciar que las simulaciones lanzadas con un *rate* de 20 Hz reducen ligeramente la PDR. Esto sugiere que puede existir un equilibrio entre la capacidad de transmisión proporcionada por el número de subcanales y la frecuencia de transmisión de mensajes.

Estas observaciones son coherentes con el modelo LTE-V2X, que sugiere que el aumento de la potencia de transmisión puede aumentar la PDR y, por lo tanto, el alcance de las comunicaciones. Además, el modelo indica que el aumento de la frecuencia de transmisión de mensajes a 20 Hz, sumada a un aumento en la densidad de vehículos puede disminuir ligeramente la PDR debido a la generación de mayores interferencias.

Por otro lado, el uso de 4 subcanales puede aumentar la PDR, ya que disminuye el MCS, permitiendo transmisiones más robustas. Estos resultados son consistentes con las observaciones realizadas a partir de las simulaciones, lo que refuerza la validez del modelo utilizado.

5 CONCLUSIONES

Este Trabajo de Fin de Grado ha abordado un estudio detallado de la simulación de conectividad vehicular, conocida como V2X, en el marco de la conducción autónoma. Se ha profundizado en el estado del arte, abarcando tanto simuladores de tráfico y de red, como las plataformas líderes en simulación virtual para vehículos autónomos.

El foco principal del proyecto ha sido la implementación de conectividad V2X sobre el simulador CARLA, una plataforma de simulación de conducción autónoma. Para ello, en primer lugar, se han examinado a fondo sus diversos componentes y funcionalidades, desde su arquitectura general hasta su API. Se ha resaltado la versatilidad de los módulos que componen el simulador, incluyendo el gestor de tráfico, la creación y gestión de actores y planos, el ciclo de vida de estos y la implementación de mapas y escenarios.

El trabajo principal realizado es el desarrollo de un módulo específico para simular la conectividad V2X y se han modelado las comunicaciones. Este modelado incluye un modelo de rendimiento analítico y representaciones gráficas, tanto estáticas como en tiempo real.

Este modelo ha permitido analizar la *Packet Delivery Ratio* (PDR) en función de la distancia entre el transmisor y el receptor, considerando los diferentes tipos de errores que pueden ocurrir en las transmisiones en C-V2X Modo 4. Los resultados de las simulaciones demostraron que el aumento de potencia, el número de subcanales y la utilización de una tasa de paquetes acorde a la densidad vehicular pueden mejorar la tasa de éxito. En última instancia, la solución parece residir en encontrar la combinación correcta de estos factores para cada tipo de escenario.

Este proyecto evidencia la utilidad de utilizar un modelo analítico en las simulaciones realistas para el estudio de la percepción y las comunicaciones en el contexto de la conducción autónoma. Se ha propuesto un enfoque que prescinde de la intervención de un simulador de comunicaciones, simplificando así el proceso de análisis y estudio.

En conclusión, este Trabajo de Fin de Grado aporta al campo de las simulaciones de conectividad y percepción V2X, proporcionando herramientas y métodos para analizar y estudiar las comunicaciones en una plataforma realista. Aunque este es un campo emergente y queda mucho por explorar, se espera que este trabajo sirva como base para futuros proyectos que exploten los simuladores de entornos reales, abriendo así diversas líneas de investigación futuras. Entre ellas, sería interesante incorporar más sensores, especialmente aquellos que permiten detectar la forma del entorno, como los LIDAR, y fusionar los datos de estos sensores para obtener una información de campo precisa que permita tener en cuenta el entorno en las simulaciones de comunicaciones. Además, sería relevante incorporar actores que proporcionen información adicional para la percepción y comunicación V2X, como las *Roadside Units* (RSU's), para mejorar la conectividad de los vehículos y facilitar la comunicación entre ellos (V2V), entre vehículos e infraestructuras (V2I), y entre vehículos y cualquier otro elemento del sistema de transporte (V2X). Aunque muchos de estos temas han quedado fuera de este trabajo debido a su extensión, este proyecto abre las puertas para poder estudiar y tratar los temas mencionados.

BIBLIOGRAFÍA

[1] "VSIM: Multimodal Traffic Simulation Software". Online: https://www.myptv.com/en/mobility-software/ptv-vissim

[2] "AIMSUN: Simulation and AI for future mobility". Online: https://www.aimsun.com/

[3] "SUMO: Simulation of Urban Mobility". Online: https://www.eclipse.org/sumo/

[4] "TRACI: Traffic Control Interface". Online: https://sumo.dlr.de/docs/TraCI.html

[5] "CARLA: Car Learning to Act". Online: https://carla.org/

[6] "NS-3: Network Simulator". Online: https://www.nsnam.org/

[7] "OMNeT++: Discrete Event Simulator". Online: https://omnetpp.org/

[8] "Artery: V2X Simulation Framework". Online: http://artery.v2x-research.eu/

[9] "Veins: The open-source network simulation framework". Online: http://veins.car2x.org/

[10] Hardes, T., Sommer, C., & Dressler, F. (2023). A Co-Simulation Framework for the Evaluation of V2X Applications. Recuperado de https://www.cms-labs.org/bib/hardes2023cosimulation/hardes2023cosimulation.pdf

[11] Masuda, H., El Marai, O., Tsukada, M., Taleb, T., & Esaki, H. (2023). Feature-Based Vehicle Identification Framework for Optimization of Collective Perception Messages in Vehicular Networks. Recuperado de https://ieeexplore.ieee.org/document/9910433

[12] Anagnostopoulos, C., Koulamas, C., Lalos, A., & Stylios, C. (2021). Open-Source Integrated Simulation Framework for Cooperative Autonomous Vehicles. Recuperado de https://ieeexplore.ieee.org/document/9797115

[13] Fu, S., Zhang, W., & Jiang, Z. (2023). A Network-Level Connected Autonomous Driving Evaluation Platform Implementing C-V2X Technology. School of Communication & Information Engineering, Shanghai University. Recuperado de https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9459566

[14] SVL. Online: https://www.svlsimulator.com/

[15] Computer Vision Center. Online: http://www.cvc.uab.es/

[16] OpenSCENARIO. Online: https://www.asam.net/standards/detail/openscenario/

[17] CARLA documentation. Online: https://carla.readthedocs.io/en/latest/

[18] CARLA Python API. Online: https://carla.readthedocs.io/en/latest/python_api/

[19] CARLA Doxygen documentation. Online:http://carla.org/Doxygen/html/index.html

[20] CARLA Traffic Manager. Online: https://carla.readthedocs.io/en/latest/adv_traffic_manager/

[21] ASAM OpenDrive. Online: https://www.asam.net/standards/detail/opendrive/

[22] CARLA Blueprint Library. Online: https://carla.readthedocs.io/en/latest/bp_library/

[23] M. Gonzalez-Martín, M. Sepulcre, R. Molina-Masegosa and J. Gozalvez, "Analytical Models of the Performance of C-V2X Mode 4 Vehicular Communications", *IEEE Transactions on Vehicular Technology*, vol. 68, no. 2, pp. 1155-1166, Feb. 2019

[24] SAE. Online: https://www.sae.org/

[25] ETSI. Online: https://www.etsi.org/