Escola Politècnica Superior
d'Enginyeria de Vilanova i la Geltrú
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE GRADO

TÍTULO: Conducción autónoma y calibración de dispositivos en un entorno Smart City/F2C

AUTORES: RIVAS LÓPEZ, ADRIÁN; EL IDRISSI ACHAHBAR, ILYAS

FECHA: Octubre, 2019

APELLIDOS: RIVAS LÓPEZ **NOMBRE: ADRIÁN**

TITULACIÓN: INGENIERÍA INFORMÁTICA

PLAN: 2018/2019

DIRECTOR: SERGIO SÁNCHEZ

DEPARTAMENTO: ARQUITECTURA DE COMPUTADORES

APELLIDOS: EL IDRISSI ACHAHBAR **NOMBRE: ILYAS**

TITULACIÓN: INGENIERÍA INFORMÁTICA

PLAN: 2018/2019

DIRECTOR: SERGIO SÁNCHEZ

DEPARTAMENTO: ARQUITECTURA DE COMPUTADORES

CALIFICACIÓN DEL TFG

TRIBUNAL

PRESIDENTE **SECRETARIO** **VOCAL**

FECHA DE LECTURA:

Este proyecto tiene en cuenta aspectos medioambientales: ☐Sí ☐No

"Agradecer a nuestro tutor Sergi Sánchez
y nuestro cliente Alejandro Jurnet,
por la ayuda que nos han proporcionado
durante la creación de este proyecto.
También agradecer a nuestras familias
por su apoyo durante el desarrollo de este."

Agradecimientos

RESUMEN

Durante estos últimos años se ha podido ver un gran avance en el ámbito de la tecnología. El continuo crecimiento y auge de esta hace que esté presente cada vez con más frecuencia en tareas del día a día de cualquier habitante de una sociedad moderna.

Las ciudades son el lugar donde más impacto ha tenido este gran cambio, ya que es en estas donde la mayoría de sus ciudadanos realizan sus actividades diarias, como por ejemplo ir al trabajo, hacer la compra, etc. Todas estas actividades se ven afectadas directamente por la tecnología, y a partir de las diferentes necesidades que estas requieren, aparecen un gran número de servicios a disposición de los ciudadanos.

Este gran incremento tecnológico requiere que estos servicios sean cada vez más eficientes con la finalidad de poder servir al mayor número de personas posibles. Para hacer esto posible, se requiere de un gran número de recursos, que gestionados de la manera conveniente faciliten el desarrollo de tales servicios.

Basándonos en estas necesidades, aparece el concepto de Smart City. Una ciudad inteligente (Smart City en inglés) tiene como objetivo principal poder utilizar la tecnología de la información y comunicación (TIC) con la finalidad de mejorar el nivel de vida de los ciudadanos que habitan en esta.

Para el correcto desarrollo de estos servicios que una ciudad inteligente debe proporcionar, se ha de introducir el concepto de Fog to Cloud (F2C). El F2C pretende acercar las capacidades de las cuales dispone el Cloud a los usuarios, con la finalidad de obtener un tiempo de comunicación mucho más reducido que el que habría en una comunicación directa con el Cloud. Con este tiempo de comunicación tan reducido, se puede incrementar el número de servicios que puede proporcionar una ciudad inteligente.

En este proyecto, se pretende llevar a cabo el desarrollo de una conducción autónoma la cual utilice las diferentes características proporcionadas por F2C en un entorno de ciudad inteligente, es decir, un entorno donde el vehículo pueda realizar diferentes acciones en tiempo real dependiendo del estado de diferentes dispositivos de esta ciudad, como pueden ser semáforos, otros vehículos, etc.

A parte, se pretende desarrollar un sistema de calibración de los diferentes dispositivos de esta ciudad inteligente, con la finalidad de cerciorarnos de que el funcionamiento de estos es el correcto, de una manera sencilla y eficaz.

Palabras clave:

Smart City, Fog to Cloud, Cloud, servicio, dispositivo, ciudad inteligente

ABSTRACT

Over the last few years, there has been a breakthrough in the field of technology. The continued growth and boom of it makes it increasingly frequent in the day-to-day tasks of any inhabitant of a modern society.

Cities are the place where this big change has had the most impact, as it is in these where most of their citizens carry out their daily activities, such as going to work, go shopping, etc. All these activities are directly affected by technology and based on the different needs that these require, a large number of services appear available to citizens.

This great technological increase requires these services to be increasingly efficient in order to be able to serve as many people as possible. To make this possible, a large number of resources are required, which conveniently managed facilitate the development of such services.

Based on these needs, the concept of Smart City appears. A smart city has as its main objective to be able to use information and communication technology (TIC) in order to improve the standard of living of the citizens who live in it.

For the proper development of these services that a smart city must provide, the concept of Fog to Cloud (F2C) mus be introduced. F2C aims to bring the capabilities available on the Cloud to the users, in order to obtain a much shorter communication time than there would be in a direct communication with the Cloud. With this reduced communication time, we could increase the number of services that a smart city could provide.

In this project, it is intended to carry out the development of an autonomous driving which uses the different features provided by F2C in a smart city environment, which is an environment where the car can perform different actions in real-time depending on the state of different devices of the city, such as traffic lights, other vehicles, etc.

In addition, it is intended to deveop a calibration system of the different devices of this smart city, in order to make sure that the performance of these is correct, in a simple and effective way.

Keywords:

Smart City, Fog to Cloud, Cloud, service, device

APORTACIÓN INDIVIDUAL AL GRUPO

Adrián Rivas López

Ha participado activamente en el desarrollo global del proyecto.

Se ha encargado tanto del diseño como de la implementación de los diferentes servicios de conducción autónoma, así como de los de calibración.

También ha participado en el desarrollo de las nuevas pantallas del Front-End del administrador de la ciudad inteligente, aportando nuevas funcionalidades a este.

Finalmente, ha aportado en el diseño e implementación de los diferentes módulos que podemos encontrar en un agent, así como en la implementación de una aplicación cliente mediante la cual se pueden solicitar servicios directamente al agent del cliente.

Finalmente, ha preparado un conjunto de pruebas para poder verificar el correcto funcionamiento de los diferentes elementos desarrollados.

APORTACIÓN INDIVIDUAL AL GRUPO

Ilyas el Idrissi Achahbar

Ha participado activamente en el desarrollo global del proyecto.

Se ha encargado tanto del diseño como de la implementación de los diferentes servicios de conducción autónoma, así como de los de calibración.

También ha participado en el desarrollo de las nuevas pantallas del Front-End del administrador de la ciudad inteligente, aportando nuevas funcionalidades a este.

Además, ha aportado en el diseño e implementación de los diferentes módulos que podemos encontrar en un agent, así como en la implementación de una aplicación cliente mediante la cual se pueden solicitar servicios directamente al agent del cliente.

Finalmente, ha preparado un conjunto de pruebas para poder verificar el correcto funcionamiento de los diferentes elementos desarrollados.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

GLOSARIO DE ACRÓNIMOS Y TÉRMINOS

- **Agent:** Nodo capaz de intercambiar información con otros nodos que forman parte de la misma topología de red. El intercambio de estos mensajes generalmente es utilizado para comunicar datos de un servicio. A parte, un agent debe de tener capacidad de procesamiento de datos y algún tipo de inteligencia.
- **Cloud Agent:** Es un tipo de agent "especial". Se encuentra en el punto más alto de nuestra topología y su función principal es la de proporcionar datos a los agents que se encuentran por debajo suyo.
- **Cluster:** Subred formada por uno, o un conjunto de agents, encargados de ejecutar un servicio sobre el cual han recibido una petición.
- **Conducción autónoma:** Tipo de conducción, dividida en diferentes niveles, donde se tiende a añadir funcionalidades al vehículo para que este pueda realizar trayectos de forma autónoma, con la menor participación del conductor posible, llegando incluso a poder depender totalmente de este.
- **Contenedor docker:** Contenedor sobre el cual se pueden ejecutar programas. En nuestro proyecto los contenedores docker son utilizados para virtualizar nodos dentro de nuestra topología.
- **Front-End:** Parte visual de un sitio web con la cual pueden interactuar los usuarios, ya sea para obtener información como para realizar acciones.
- **F2C:** F2C (Fog-to-Cloud) es un tipo de arquitectura que utiliza nodos repartidos a lo largo de esta para realizar diferentes tareas de computación, almacenaje y comunicación entre ellos.
- **Leader:** Es un agent que tiene ciertas funcionalidades extras añadidas. Estas funcionalidades son, por ejemplo, la gestión de los agents que se han conectado a él, la creación un Cluster con los agents necesarios para ejecutar un servicio, etc.
- **IoT:** IoT (Internet of Things) es un concepto que se refiere a la interconexión digital de objetos cotidianos con internet. En nuestro proyecto, un IoT es un dispositivo capaz de generar y proporcionar un conjunto de datos, que pueden ser recogidos por nodos de la red para realizar diferentes acciones con estos.
- **Nodo:** Elemento que forma parte de la topología.
- **Tag RFID:** Tarjeta que dispone de la tecnología RFID, o identificación por radiofrecuencia (del inglés *Radio frequency Identification*), que en nuestro proyecto es utilizada para proporcionar diferentes posiciones dentro de nuestro Testbed.

- **Servicio:** Conjunto de acciones o actividades definidas y que tienen una finalidad concreta. En nuestro proyecto, un servicio puede ser solicitado por cualquier nodo de

la topología, siempre y cuando este servicio sea uno de los proporcionados por nuestra ciudad inteligente.

- **Smart city:** Smart city (ciudad inteligente) es un concepto que se refiere a un tipo de ciudad basada en la sostenibilidad, es decir, esta ha de ser capaz de responder adecuadamente a diferentes aspectos básicos como pueden ser necesidades de diferentes instituciones, empresas y los propios habitantes de la ciudad.
- **Testbed:** Testbed (banco de pruebas) es una plataforma que sirve para poder experimentar diferentes funcionalidades que forman parte de un gran proyecto, antes de llevar estas a un entorno real.
- **Topología:** Es la forma en que está diseñada la red. Dentro de esta topología se encuentran un conjunto de nodos capaz de comunicarse entre ellos, siguiendo una jerarquía determinada, la cual determina quien puede comunicarse con quien.

1. INTRODUCCIÓN

1.1 JUSTIFICACIÓN Y MOTIVACIÓN

La realización de este proyecto viene dada por el avance de las nuevas tecnologías en el ámbito de la conducción y el crecimiento de dispositivos interconectados que hay en las ciudades inteligentes.

El objetivo está en aprovechar dichos avances en el ámbito de la conducción para aportar un conjunto de beneficios en diferentes áreas de la conducción. Los principales beneficios aportados gracias a dichos avances son la seguridad, la comodidad y el bienestar de las personas que se encuentran en estos vehículos.

Por otra parte, también encontramos un gran interés general en el ámbito de la investigación y el desarrollo de ciudades que puedan proporcionar, de manera autosuficiente, un conjunto de funcionalidades y beneficios para los ciudadanos que residan en ellas. Estas ciudades son conocidas como ciudades inteligentes (*Smart Cities)*.

Así como con la conducción autónoma se introducen una serie de beneficios para el conjunto de personas que dispongan de ella, las Smart Cities gracias a la incorporación de variables inteligentes, sensores, etc., también aportan una gran variedad de beneficios para el conjunto de la sociedad. Entre los beneficios destacamos, la agilización de los servicios, la seguridad y la eficiencia energética.

Finalmente, para poder unificar el desarrollo de estos dos grandes conceptos y podernos beneficiar de las ventajas que nos proporcionan ambos, podemos disponer de un concepto, posiblemente no tan conocido el cual es el *Fog-to-Cloud (F2C)*, el cual como funcionalidad principal tiene la de poder comunicar diferentes componentes que estén dentro de su topología, de una forma eficiente y segura, con la finalidad de poder realizar diferentes acciones utilizando estos.

1.2 ORIGEN DEL PROYECTO

Actualmente el equipo de investigación CRAAX (Centre d'Arquitectures Avançades de Xarxes) [1], está formando parte en varios proyectos europeos, entre los cuales se encuentra uno llamado mF2C [2]. A raíz de este último surge la posibilidad de desarrollar

un sistema de conducción autónoma en un entorno de Smart City.

Para la realización de nuestro proyecto hemos aprovechado el proyecto de final de grado de alumnos de la UPC, “Diseño e Implementación de un Testbed para una SmartCity” [3], realizado también en el centro de investigación del CRAAX.

Al disponer de este Testbed, así como un conjunto de dispositivos interconectados, podremos llevar a cabo este proyecto, en un entorno de test, pero siempre teniendo en cuenta que la finalidad principal de este está enfocada a un entorno real donde podrán intervenir los elementos diseñados e implementados en nuestro proyecto.

1.3 OBJETIVOS

El objetivo principal de este proyecto es el diseño y desarrollo de un sistema de conducción autónoma capaz de utilizar la información proporcionada por diferentes dispositivos inteligentes de una Smart City para de esta manera añadir inteligencia al sistema de conducción, inteligencia que no existe en los sistemas de conducción autónoma actuales.

También se pretende diseñar y desarrollar un sistema de calibración de los diferentes dispositivos inteligentes de la Smart City con la finalidad de poder tener controlado en todo momento el estado de estos y asegurar su correcto funcionamiento dentro de la Smart City.

Finalmente, también se introduce como objetivo adicional de este proyecto el diseño e implementación de un agent a nivel de software, así como cada uno de sus módulos con la finalidad de poder crear una topología F2C capaz de ejecutar diferentes servicios.

Podemos dividir los objetivos de este proyecto en objetivos técnicos y objetivos académicos. A continuación, se detalla cada uno de estos.

1.3.1 Objetivos técnicos

Los objetivos técnicos son objetivos secundarios que nos van a ayudar a desarrollar los objetivos generales descritos anteriormente. Estos son los siguientes:

- Investigación sobre las diferentes áreas para tener en cuenta a la hora de llevar a cabo el desarrollo de este proyecto.
- Análisis de los requisitos del proyecto: detectar los requisitos y necesidades del proyecto.
- Desarrollo de diferentes servicios, tanto de conducción autónoma como de calibración de dispositivos.
- Diseño e implementación de diferentes pantallas del Front-End para poder visualizar datos de la topología en tiempo real.
- Realización de las pruebas de funcionamiento correspondientes para cada uno de los elementos diseñados e implementados.
- Redactado de la presente memoria y documentación técnica necesaria para poder utilizar todo lo que ha sido desarrollado.

1.3.2 Objetivos académicos

Los objetivos académicos de este proyecto están enfocados al tanto a nivel personal como profesional. Estos objetivos son los siguientes:

- Trabajar en conjunto con otros estudiantes que están realizando proyectos dentro del mismo ámbito de las Smart Cities.
- Aplicar conocimientos obtenidos en la universidad a la hora de llevar a cabo el desarrollo del proyecto, como pueden ser tanto la metodología *Brainstorming* como la metodología *Agile.*
- Lograr los objetivos planteados inicialmente para ofrecer el resultado final del proyecto al CRAAX y sus labores de investigación.
- Sentar las bases de una futura línea de trabajo en el ámbito de la conducción autónoma basada en F2C y la calibración de dispositivos en un entorno de Smart City.

1.4 ESTRUCTURA DEL CONTENIDO

El contenido del proyecto se encuentra dividido en diferentes capítulos, los cuales podemos encontrar brevemente resumidos a continuación:

- **Capítulo 1. Introducción:** En este primer capítulo se describe el problema el cual ha implicado la creación de este proyecto. También se plantean un conjunto de objetivos a alcanzar durante la realización de este.
- **Capítulo 2. Metodología y planificación:** Este capítulo describe las metodologías utilizadas a lo largo del proyecto para poder llevar a cabo el correcto desarrollo de este, así como un conjunto de datos informativos relacionados con el resumen temporal del proyecto.
- **Capítulo 3. Estado del arte:** Capítulo donde se explica la situación actual de los diferentes aspectos más importantes del proyecto. También se describen un conjunto de problemas que afrontan actualmente, así como trabajos ya realizados en esos ámbitos.
- **Capítulo 4. Análisis de requisitos:** El capítulo de análisis de requisitos se realiza un análisis de los requisitos del proyecto, así como las diferentes restricciones que presenta este para poder llevar a cabo el desarrollo de este.
- **Capítulo 5. Casos de uso:** En este capítulo se definen los diferentes casos de uso del proyecto.
- **Capítulo 6. Diseño:** En este capítulo se realiza un proceso de investigación y aprendizaje sobre ciertos conceptos que consideramos serán de gran ayuda a la hora de diseñar e implementar los diferentes aspectos que sean requeridos en el proyecto. Una vez obtenidos los conocimientos necesarios, se comienza con el diseño de diferentes elementos de nuestro proyecto.
- **Capítulo 7. Implementación:** El objetivo de este capítulo es, una vez concretado el diseño de todos los diferentes componentes que forman nuestro proyecto, detallar como se van a implementar cada uno de ellos.
- **Capítulo 8. Pruebas de funcionamiento:** Capítulo en el cual se muestran y explican el conjunto de pruebas que han sido realizadas a lo largo de nuestro proyecto con la finalidad de probar el correcto funcionamiento de cada uno de los diferentes elementos desarrollados.
- **Capítulo 9. Integración con otros proyectos:** Capítulo donde se explica como se ha realizado la integración de otros proyectos con el nuestro para que estos funcionen de manera conjunta correctamente.
- **Capítulo 10. Continuidad del proyecto:** Se definen un conjunto de posibles aspectos a trabajar con la finalidad de dar continuidad al proyecto realizado.
- **Capítulo 11. Presupuesto:** Capítulo en el que se muestran el conjunto de gastos requeridos para la realización de este proyecto.
- **Capítulo 12. Conclusiones:** En este capítulo se recogen las principales conclusiones obtenidas de nuestro proyecto a partir de los objetivos definidos al

inicio de este.

- **Capítulo 13. Referencias:** Recoge un listado con el conjunto de referencias citadas a lo largo del redactado de nuestro proyecto.
- **Capítulo 14. Bibliografía:** Recoge un conjunto de fuentes bibliográficas consultadas para la realización de nuestro proyecto.
- **Capítulo 15. Anexo:** En este capítulo se explican los diferentes manuales de instalación necesarios para poder poner en marcha el proyecto.

2. METODOLOGÍA Y PLANIFICACIÓN

2.1 METODOLOGÍA

A lo largo de este proyecto hemos utilizado varios tipos de metodologías, como pueden ser el *Brainstorming* y la metodología ágil conocida como *Kanban*.

El *Brainstorming* (lluvia de ideas), es una herramienta utilizada en grupos de trabajo para poder, a partir de un proyecto o problema determinado y generalmente bastante grande, aportar un conjunto de ideas las cuales puedan facilitar el inicio de este y también puedan ayudar a asignar una dirección sobre la cual se trabajará posteriormente.

Este tipo de metodología, que fue utilizado en la fase inicial del proyecto, nos sirvió para, a partir de un gran conjunto de ideas, escoger las que creímos eran las más indicadas para poder decidir los objetivos que queríamos alcanzar en este proyecto y también decidir el punto inicial del desarrollo de estos.

La metodología ágil *Kanban* tiene como objetivo principal poder gestionar de manera general cómo se van completando las tareas en las que han sido dividido el proyecto.

Esta metodología, que ya ha sido utilizada a lo largo de la carrera, fue útil a la hora de organizar el desarrollo del proyecto, dividiendo éste por *Sprints*, lo cual nos ayudó a conseguir poco a poco la idea principal del proyecto proporcionada por nuestro cliente, el CRAAX.

En nuestro proyecto, un Sprint, es un periodo de dos semanas, en las cuales debemos de asignarnos un conjunto de tareas a realizar. Una vez pasadas estas dos semanas, nos reunimos con nuestro cliente, y mediante una DEMO, que puede ser un PowerPoint, explicamos cómo ha transcurrido este periodo de dos semanas y que hemos conseguido.

Esta metodología es muy útil a la hora de llevar el seguimiento de un gran proyecto, ya que, si hay algo que no le acaba de gustar al cliente, este puede comunicarlo y, como los incrementos están muy controlados, estos cambios no deben de suponer un gran problema a la hora de desarrollarlos e introducirlos en el proyecto.

2.2 HERRAMIENTAS UTILIZADAS

Para llevar un control sobre el estado global del proyecto, así como el estado parcial de cada Sprint, se ha utilizado una aplicación web llamada Trello [4]. Esta aplicación permite de una manera visual y sencilla distribuir el conjunto de tareas a realizar dentro de un tablero para de esta manera poder hacer un seguimiento del estado actual del proyecto. Dentro de este tablero se pueden crear listas, en las cuales se reparten un conjunto de tarjetas, que básicamente son tareas que realizar. Dentro de cada tarjeta se puede añadir información en forma de: descripción, comentario, archivo adjunto, fecha de entrega, entre otras opciones. Esta aplicación también ha servido para poder comunicarnos tanto con el tutor del TFG, así como con el cliente del proyecto.

A continuación, se muestra una imagen en la cual se puede ver la estructura general de nuestro tablero utilizado para este proyecto:

Figura 1: Tablero Trello

Otra herramienta que ha sido utilizada a lo largo del proyecto ha sido GitHub [5]. GitHub es un sitio web el cual tiene como servicio principal poder almacenar y gestionar proyectos que son guardados en la nube en forma de repositorios. Estos proyectos normalmente son códigos creado por desarrolladores. GitHub también tiene un sistema de versiones el cual permite moverse entre diferentes estados del proyecto. Esto es muy interesante ya que permite acceder a cualquier versión anterior del proyecto en caso de haber tenido algún tipo de error en la implementación del código.

2.3 RESUMEN TEMPORAL DEL PROYECTO

En cuanto a la duración del proyecto y como ha estado repartido este, podemos ver en la siguiente tabla cuáles han sido los valores obtenidos para cada uno de los diferentes aspectos a tener en cuenta:

TEMPORIZACIÓN DEL PROYECTO	
Fecha de inicio	25/11/2018
Fecha de fin	22/10/2019
Duración de los sprints	2 semanas
Horas de trabajo por semana	15 - 25 horas/persona
Horas previstas	540 horas/persona
Horas finales	800 horas/persona

Cuadro 1: Tabla de temporización del proyecto

2.4 INFORME DE TIEMPOS DEL PROYECTO

Como hemos podido ver el proyecto se ha dividido en una serie de sprints. En nuestro caso el número de sprints que hemos realizado han sido 17. A continuación podemos ver, de una forma resumida, el objetivo principal a tratar para cada uno de los sprints de nuestro proyecto.

Sprint	**Resumen del sprint**
Sprint 1: 25/11/2018 a 13/12/2018	Investigación y creación del estado del arte.
Sprint 2: 13/12/2018 a 16/01/2019	Creación de las épicas de nuestro proyecto.
Sprint 3: 16/01/2019 a 30/01/2019	Primer contacto con los vehículos.
Sprint 4: 30/01/2019 a 13/02/2019	Calibración del vehículo y programación básica del movimiento de este.
Sprint 5: 13/02/2019 a 27/02/2019	Programación más avanzada del movimiento del

	vehículo y comienzo del diseño de la clase agent.
Sprint 6: 27/02/2019 a 13/03/2019	Primera fase de inclusión del agent en el vehículo juntamente con el uso del Front-End.
Sprint 7: 13/03/2019 a 27/03/2019	Interacción con nuevos elementos del Testbed y creación de un primer servicio básico.
Sprint 8: 27/03/2019 a 10/04/2019	Gestión de los nodos de la topología y reestructuración de los submódulos de un agent.
Sprint 9: 10/04/2019 a 02/05/2019	Introducción de nodos virtuales en la topología.
Sprint 10: 02/05/2019 a 15/05/2019	Unificación de agents y creación de la pantalla de visualización de nodos en el Front-End.
Sprint 11: 15/05/2019 a 29/05/2019	Refactorización de los submódulos del agent y creación de un catálogo de servicios.
Sprint 12: 29/05/2019 a 12/06/2019	Creación de la pantalla de visualización de servicios en el Front-End.
Sprint 13: 12/06/2019 a 03/07/2019	Creación de la pantalla de ejecución de servicios des del Front-End.
Sprint 14: 03/07/2019 a 17/07/2019	Creación de la pantalla de añadir servicios des del Front-End.
Sprint 15: 17/07/2019 a 04/09/2019	Mejoras en la implementación y redacción de la memoria del proyecto.
Sprint 16: 04/09/2019 a 18/09/2019	Redacción de la memoria del proyecto y creación de una aplicación cliente capaz de solicitar servicios sobre un agent.
Sprint 17: 18/09/2019 a 02/10/2019	Últimas pruebas de funcionamiento, mejora de la memoria y redacción del artículo.

Cuadro 2: Informe de tiempos del proyecto

2.5 RESUMEN DE LOS SPRINTS

2.5.1 Sprint 1

En el primer sprint, nuestro objetivo era realizar una investigación sobre los aspectos más importantes que nos encontraremos en nuestro proyecto. Estos aspectos son la conducción autónoma, las ciudades inteligentes y F2C. Una vez realizada esta investigación, escribimos el estado del arte, donde se explica cuál es la situación actual para cada uno de estos aspectos a investigar. Dentro de este estado del arte podemos encontrar varios apartados, donde se explican aspectos más concretos como pueden ser los problemas que tenemos actualmente con relación a estos temas, las ventajas que nos podrían proporcionar, así como también el conjunto de trabajos que ya han sido realizados en esos ámbitos y una conclusión donde damos nuestra opinión.

2.5.2 Sprint 2

Para el segundo sprint, el objetivo principal era el de crear una serie de épicas. Estas épicas son historias de usuario muy grandes, las cuales no se pueden realizar en un mismo sprint debido a su gran tamaño y al tiempo que conllevan. La creación de estas épicas es muy importante para poder llegar al objetivo final del proyecto sin problemas.

2.5.3 Sprint 3

En el tercer sprint de nuestro proyecto ya comenzamos a trabajar con los vehículos que forman parte del Testbed. Estos vehículos son utilizados para realizar pruebas a lo largo del proyecto, en un entorno de desarrollo y testeo como puede ser el Testbed. El objetivo principal de este sprint era poder mover el vehículo por todo el Testbed de forma manual, para comenzar a entender el funcionamiento de este y de esta manera poder realizar programas más complejos en sprints futuros que permitan añadir inteligencia al vehículo.

2.5.4 Sprint 4

En este sprint, una vez entendimos cómo funcionaba la mecánica del vehículo, comenzamos a desarrollar un programa que nos permitiera hacer circular el vehículo sin necesidad de tomar el control en ningún momento. Para conseguir esto, tuvimos que

calibrar dos de los sensores que pueden ser incorporados en el vehículo. Estos sensores eran el sensor de línea y el sensor de ultrasonido. El sensor de línea es el encargado de proporcionar información al vehículo indicando si este se encuentra encima de una línea blanca, o si por el contrario se ha salido de su carril y ya no tiene esta línea blanca debajo suyo. Con la información proporcionada por este sensor fuimos capaces de programar los giros y velocidades del vehículo asignando los valores que consideramos óptimos para que este realizara la mejor circulación posible. El sensor de ultrasonido nos sirvió para realizar la detección de obstáculos que el vehículo pudiese tener delante suyo. Con esta información podíamos detener o poner el vehículo en movimiento dependiendo de si había un obstáculo o no delante suyo.

2.5.5 Sprint 5

En el quinto sprint de nuestro proyecto, mejoramos los algoritmos de movimiento del vehículo y comenzamos a desarrollar uno de los aspectos más importantes, el agent. Este agent lo comenzamos a desarrollar como una clase Python [6] y en este punto del proyecto el funcionamiento principal de este era el de poder comunicarse con otros agents que estuvieran dentro de la misma red mediante mensajes TCP [7].

2.5.6 Sprint 6

Durante este sprint continuamos mejorando tanto la conducción autónoma del vehículo como la clase agent. Una vez desarrolladas las funcionalidades principales de esta clase (intercambio de mensajes con otros agents), decidimos introducir este agent dentro del propio vehículo. Esto nos permitiría añadir al vehículo un mecanismo de comunicación con otros elementos que también tuvieran integrada la clase agent y que se encontraran dentro de la misma red. Gracias a la incorporación de esta clase, ahora el vehículo podía comunicarse con el Front-End proporcionado por otro alumno que realizó el TFG con nombre “Panel Front-End para el control de una Smart City” [8] y de esta manera podíamos ver en tiempo real en que posición de la ciudad se encontraba nuestro vehículo.

2.5.7 Sprint 7

En este sprint, una vez conseguida la correcta comunicación entre diferentes agents que se encontraran dentro de la misma red, pasamos a desarrollar una nueva clase

llamada leader. Esta clase es una extensión de la clase agent, con la diferencia principal que contiene un conjunto de funcionalidades características de un leader, funcionalidades que un agent como tal no puede realizar. También implementamos nuestro primer servicio proporcionado por la ciudad inteligente. Este era un servicio de emergencia, donde se enviaba un vehículo (ambulancia) de un punto a otro, y una vez atendida la solicitud de emergencia esta volvía al hospital. Disponíamos de un leader, que controlaba un conjunto de semáforos, y de la ambulancia. Durante el recorrido mencionado anteriormente, conforme la ambulancia pasaba por un semáforo, y gracias a la comunicación entre agents, esta se comunicaba con el leader de los semáforos y modificaba el estado de estos para ponerlos en estado de emergencia (verde con led azul parpadeante). De esta manera pudimos ver como la comunicación entre diferentes nodos de la red se realizaba correctamente y también cómo podían interactuar entre ellos.

2.5.8 Sprint 8

En este sprint tuvimos que reestructurar el código, tanto del agent como del leader, para tener claramente diferenciados los submódulos de los cuales disponen estos. En este punto del proyecto, estos módulos eran los siguientes: RT (Runtime), SEX (Service Execution) y TRM (Topology and Resource Management). También introducimos una API, la cual nos permitía registrar diferentes nodos dentro de una base de datos y de esta manera poder ver quien se encontraba dentro de nuestra topología en todo momento. A parte, creamos otro servicio, el cual permitía, dados dos puntos de la ciudad, encontrar la ruta más corta para ir de uno al otro.

2.5.9 Sprint 9

Durante este sprint nos centramos en la virtualización de nodos para simular un escenario más real, donde el número de nodos será mucho mayor al número de nodos físicos de los cuales disponemos en el CRAAX. En este sprint los agent virtuales se registran en la base de datos y van cambiando sus atributos de forma indefinida con el objetivo de simular un entorno con un gran número de nodos.

2.5.10 Sprint 10

En este sprint unificamos los agents implementados y sus submódulos con los agents implementados por otros compañeros que también están haciendo su TFG juntamente con el CRAAX. Esta unificación tiene como objetivo principal que los agents puedan comunicarse sin problemas entre ellos, aunque las implementaciones hayan sido hechas por alumnos de diferentes proyectos. Todo esto es posible gracias a un conjunto de requisitos que están presentes tanto para el propio agent como sus submódulos. También añadimos una nueva pantalla al Front-End proporcionado, la cual nos permite ver en tiempo real los nodos que hay conectados en nuestra topología y que están activos, así como los nodos que alguna vez han sido conectados y que están desconectados.

2.5.11 Sprint 11

Para este sprint realizamos ciertas modificaciones en los submódulos del agent, con la finalidad de poder ejecutar cualquier servicio proporcionado por la ciudad inteligente, teniendo en cuenta que nodos de la topología pueden formar parte de la ejecución de ese servicio y que nodos no pueden formar parte. Esta selección de nodos depende de las características que presente el servicio solicitado y del estado de los diferentes nodos que se encuentren activos en la topología.

2.5.12 Sprint 12

En este sprint se añadió una nueva pantalla al Front-End del administrador de la ciudad. Esta pantalla tiene como objetivo principal mostrar todos los servicios que existan en el catálogo de servicios de la ciudad inteligente. Estos servicios son mostrados en una tabla donde, para cada servicio, se pueden observar todos los atributos que han sido asignados a este una vez creado.

2.5.13 Sprint 13

En este sprint añadimos una funcionalidad a la tabla de la pantalla de visualización de nodos de la topología. Esta funcionalidad permitía, una vez pulsado un nodo de la tabla, abrir una nueva pantalla donde poder ver toda la información del nodo seleccionado. En esta nueva pantalla también se muestran los diferentes servicios que el nodo seleccionado puede solicitar.

2.5.14 Sprint 14

Durante este sprint nos centramos en la creación de una pantalla del Front-End des de la cual se pueden añadir servicios al catálogo de servicios de la ciudad inteligente. Esta pantalla puede ser accedida por el administrador de la ciudad, que es el encargado de añadir servicios a esta.

2.5.15 Sprint 15

En este sprint se han realizado mejoras en la implementación de la conducción autónoma, así como mejoras también en cuanto a los diferentes servicios de calibración y las diferentes pantallas del Front-End. Estas mejoras se hicieron hasta finales de julio, ya que durante todo el agosto nos centramos en la redacción de la memoria del proyecto.

2.5.16 Sprint 16

Durante este sprint continuamos redactando la memoria, mientras que por otra parte implementamos una aplicación cliente. Esta aplicación nos iba a permitir solicitar servicios directamente sobre el módulo encargado de recibir estas incorporado dentro del agent. Esta aplicación también nos iba a permitir configurar un nodo y solicitar su registro dentro de la topología de la red.

2.5.17 Sprint 17

En este último sprint se han realizado pruebas de funcionamiento de todo lo que ha sido implementado a lo largo del desarrollo del proyecto. También se han hecho retoques en la memoria con tal de mejorar esta y finalmente se ha desarrollado el artículo que tiene como objetivo principal resumir el proyecto completo en un número de páginas mucho más reducido.

3. ESTADO DEL ARTE

3.1 INTRODUCCIÓN

En este apartado se definen el conjunto de conceptos a tener en cuenta para posteriormente poder llevar a cabo el desarrollo del proyecto.

3.1.1 Vehículos en la actualidad

Actualmente nos encontramos en una etapa en la que constantemente vemos cómo se están aplicando mejoras a los vehículos que tenemos disponibles en el día a día.

Estamos llegando a un punto en el que la informática está en un punto muy avanzado y gracias a la cual es posible realizar amplias mejoras en estos vehículos, mejoras que tienen que ver con el sistema de conducción de estos, la seguridad, confort, etc.

Actualmente, la mayoría de los vehículos modernos llevan incorporado un ordenador de abordo, del inglés On Board Diagnostics (OBD), el cual nos ayuda a conocer el estado tanto del motor, como de diferentes sensores que pueda llevar nuestro vehículo.

Al comienzo de la implantación de estos dispositivos, estos podían notificar simplemente ciertos diagnósticos mediante la iluminación de una luz, en cambio, con el paso de los años y con la aparición de nuevos modelos más sofisticados, se ha conseguido que el conductor, pueda ver a tiempo real, la mayoría de los sensores disponibles en su vehículo, lo cual puede ayudar a identificar mal funciones de este y actuar más rápidamente para solucionarlas.

La variedad de sensores que pueden ser monitorizados depende del vehículo, pero algunos de los más comunes son los siguientes:

- Velocidad del vehículo
- Temperatura ambiente
- Temperatura del motor
- Estado de los neumáticos
- Luz ambiental
- Niveles de líquidos
- Detección de colisiones
- Detección de cambio de carril

La información que podemos obtener a partir de estos sensores no es únicamente útil para el conductor, sino que, en modelos más modernos, el propio vehículo puede tomar ciertas decisiones dependiendo de la información proporcionada por estos.

Algunas de estas funciones automáticas pueden ser el encendido automático de las luces al detectar cierta oscuridad en el exterior También podemos ver que hay algunas acciones automáticas más complejas, las cuales han sido desarrolladas en estos últimos años, como pueden ser la asistencia de frenado si se detecta una posible colisión, aparcamiento automático, corrección de la dirección del volante en caso de detección de cambio de carril sin intermitentes, etc.
o la activación del limpiaparabrisas.

El incremento del número de acciones automáticas que pueden hacer los vehículos es debido a querer mejorar la seguridad y prevención de accidentes que ocurren diariamente, ya que la seguridad es una de las prioridades más importantes en relación a la creación de vehículos y uso de estos.

El siguiente avance que hemos podido observar en algunos vehículos de última generación, es la conducción autónoma. Diferentes marcas ya han comenzado la fabricación de modelos que disponen de modos de conducción autónoma, a parte de la conducción tradicional. Algunas de estas marcas son Ford, Hyundai, Audi, BMW y una de las más importantes actualmente, Tesla.

Esta última es una empresa que se centra en la producción de vehículos completamente eléctricos, con el objetivo principal, según dicen, de acelerar la transición del mundo hacia la energía sostenible.

3.1.2 Vehículos autónomos

Definimos un vehículo autónomo como un vehículo capaz de moverse de un punto a otro con muy poca o ninguna interacción humana. Esto es posible gracias a un conjunto de sensores que ayudan a percibir lo que estos tienen a su alrededor, permitiéndoles hacer las acciones correspondientes con tal de realizar una conducción tal y como la haría un humano actualmente.

El primer vehículo autónomo del cual tenemos constancia fue diseñado y presentado por Norman Bel Geddes [9], diseñador industrial y teatral que centró su trayectoria en el desarrollo del aerodinamismo. Este fue presentado en el año 1939, en la Exposición Universal de Nueva York y podía ser controlado por un circuito eléctrico instalado en el pavimento de la carretera sobre la cual circulaba.

Más tarde, a principios de los años 1980, Ernst Dickmanns [10] y su equipo realizaron una serie de modificaciones en una furgoneta de la marca Mercedes-Benz, como la instalación de cámaras y otros sensores. Con estas modificaciones consiguieron controlar la dirección del volante, el acelerador y el freno mediante comandos mandados a través de un ordenador en tiempo real, realizando evaluaciones de las imágenes captadas por las cámaras instaladas. Primero se realizaron experimentos en entornos sin tráfico por motivos de seguridad, más tarde en 1986, la furgoneta fue capaz de conducirse automáticamente y en el año 1987 esta fue capaz de alcanzar velocidades de 96 km/h.

Durante los siguientes años se siguieron realizando mejoras y entre 1994 y 1995, se consiguió que los vehículos autónomos circularán tanto por carreteras sin tráfico, como por carreteras con tráfico. También se consiguió que estos vehículos realizarán cambios de carril, tanto a la izquierda como a la derecha y que pudieran realizar adelantamientos entre ellos.

3.1.3 Conducción autónoma

Dentro de la conducción automática podemos distinguir diferentes niveles de autonomía [11]. Estos van del nivel 0 al nivel 5.

Nivel 0: Sin automatización en la conducción
Este nivel de conducción autónoma es simplemente un nivel donde la autonomía es 0, es decir, todas las acciones han de ser realizadas por el conductor del vehículo.

Nivel 1: Asistencia en la conducción
En este nivel nos encontramos con un vehículo que dispone de ciertas funciones que pueden ayudar a mejorar la experiencia de conducción al conductor. Algunas de estas funciones podrían ser las siguientes: mantenimiento del carril, control de velocidad adaptativo, etc. En este nivel, el control sobre el vehículo aún recae sobre el conductor.

Nivel 2: Automatización parcial

Este nivel se caracteriza porque el vehículo ya será capaz de realizar movimientos tanto laterales como longitudinales por sí mismo. También será capaz de realizar ciertas acciones que hasta ahora debían de ser realizadas por el conductor. Aún así, en este nivel el conductor sigue siendo imprescindible.

Nivel 3: Automatización condicionada

Al vehículo ahora tiene capacidad de, además de realizar todo lo descrito en niveles anteriores, detectar y reaccionar ante la presencia de objetos que se encuentren situados en su trayectoria. El vehículo podrá realizar acciones como frenar, cambiar de carril, etc. En este caso, aunque el nivel de autonomía sea mayor, se sigue precisando un conductor que en caso de necesidad pueda retomar el control del vehículo.

Nivel 4: Automatización elevada

Este nivel ya presenta un nivel de autonomía en el que no se precisa de un conductor para poder realizar trayectos. El vehículo es el que toma el control total y el cual, después de indicarle un destino, se encargará de transportar a los ocupantes hasta allí. En caso de encontrarnos en la situación que se produce un fallo en el sistema principal, el conductor dispondrá de las herramientas correspondientes para retomar el control del vehículo.

Nivel 5: Automatización completa

En este último nivel, el vehículo se encarga de hacer todo, incluso cuando se produzca un fallo en el sistema principal, este se encargará de solucionarlo mediante un sistema de respaldo. El vehículo no dispondrá de pedales, volante, mandos, etc. ya que el mismo se encargará de hacer todo.

Actualmente la mayoría de los fabricantes de vehículos cuentan con el nivel 2, aunque hay ciertas marcas que ya trabajan con el nivel 3 en algunos de sus modelos más avanzados. En cuanto a los niveles 4 y 5, actualmente no disponemos de ellos, pero se espera que en un futuro sean los niveles más comunes dentro del entorno de los vehículos y la conducción autónomos.

Los vehículos autónomos que hay actualmente en mercado, dependen de sus propios sensores para determinar cómo llegar de un punto A a un punto B. Pero ¿y si tuviéramos dispositivos capaces de avisar a los vehículos de un peligro o de por qué camino seguir para evitar grandes aglomeraciones?, todo esto se podría realizar gracias a la comunicación entre la ciudad y el vehículo.

3.1.4 Internet de las cosas

Para poder entender cómo se comunican vehículos con otros vehículos o con otros dispositivos de la carretera/ciudad, hablaremos de **IoT** (*Internet of Things*) o Internet de las Cosas.

El internet de las cosas es la forma en la cual los dispositivos están conectados entre sí, pudiendo enviar datos ya sea por cable o inalámbricamente sin la intervención de una persona.

El primer dispositivo del IoT trata del 1932 donde el científico y diplomático ruso *Pavel Schilling* [12], pudo conectar dos telégrafos en dos habitaciones distintas.

Actualmente en 2018, tenemos más de 11.000 millones de dispositivos interconectados. Todo lo que nos rodea está conectado sin ir más lejos un dispositivo que lleva todo el mundo siempre encima y que como se te olvide en casa, te vuelves loco. En efecto, me refiero al smartphone. Pero ¿Cómo puede un smartphone comunicarse con otro smartphone? ¿Y a un ordenador?, para que se puedan conectar entre ellos es necesario un medio llamado internet gracias a sus protocolos, puede un smartphone comunicarse con cualquier otra cosa que no sea un smartphone.

Cada vez hay más dispositivos inteligentes conectados, para 2020 se espera que haya más de 20.000 millones de dispositivos, en una industria donde estos dispositivos no solo se conectan, sino que, sin la necesidad de la intervención humana, puede tomar decisiones, desde por ejemplo un sensor que te detecta y activa la luz hasta una cinta de producción y una máquina que detecta si una pieza está correcta o no.

3.1.5 Smart City

El término de Smart City, también nombrado ciudad inteligente o sostenible, hace referencia a una ciudad capaz de auto mantenerse. Esto es debido a la gran información recogida por todos os IoT que se analizan minuciosamente para que sepan que decisiones tomar sin tener que depender de la intervención humana. Hay miles de IoT en estas ciudades, algunos de estos pueden ser semáforos que detectan la afluencia de los ciudadanos para determinar el tiempo de espera, un radar de coche, una red de termómetros que envían datos para saber la temperatura media de la ciudad.

Dichas ciudades siguen un modelo sostenible, ya que gracias a todas estas decisiones tomadas puedes conseguir agilizar el tráfico haciendo posible reducir las emisiones de CO2, también permite obtener una gran optimización en el subministro eléctrico.

3.2 PRESENTACIÓN DE LA PROBLEMÁTICA

Apartado en el que se explica, para cada uno de los conceptos mencionados anteriormente, cuáles son sus principales problemas en la actualidad.

3.2.1 Problemas de la conducción autónoma

Actualmente el problema principal de la conducción autónoma no es la limitación en cuanto la tecnología de la cual disponemos, es decir, software, hardware, etc., sino que esta limitación aparece cuando contemplamos aspectos políticos, jurídicos, de regulación, infraestructura y responsabilidad.

Otra de las problemáticas más difíciles de solventar es el de qué hacer en cuanto se produce un accidente. Como el conductor del vehículo no es un ser humano, sino que es el propio vehículo, ya no se puede señalar como responsable directo al conductor. En este caso hay que desarrollar un procedimiento para determinar quién ha sido el culpable de cierto accidente. Actualmente este problema aún no está reflejado en las leyes, por lo que habría que introducirlo en estas, a parte, también habría que realizar modificaciones en todos los seguros de estos vehículos.

Los coches autónomos están conectados a sistemas de navegación los cuales, por ejemplo, al entrar en un túnel, se pierde la conexión con dichos satélites o otro caso más típico, los atascos, si el coche va de un punto A a un punto B, todos los coches autónomos irían por el mismo camino produciendo atascos. Para poder solventar dichas problemáticas los vehículos deberían contar con sistema de comunicación con las infraestructuras o bien con otro vehículo. Pero para ello habría que introducir unos estándares para que todos los fabricantes lo siguieran.

La inteligencia artificial de la cual tendrían que disponer estos vehículos es muy probable que a pesar de poder funcionar en entornos donde la circulación sea normal, haya situaciones donde no sea posible esta circulación autónoma, por ejemplo, en entornos donde la ciudad esté sometida a un entorno caótico, como pueden ser horas punta etc.

3.2.2 Principales ventajas de los vehículos autónomos

Estos vehículos autónomos también nos podrían proporcionar una serie de ventajas en comparación con los vehículos actuales. Las ventajas principales son las siguientes:

- Seguridad
- Bienestar
- Tráfico
- Costes más bajos
- Aparcamientos

Seguridad

En cuanto a la seguridad, expertos en la seguridad en conducción aseguran [13] que una vez desarrollado al completo los vehículos autónomos la cantidad de accidentes de tráfico causados por errores humanos, como por ejemplo una reacción lenta, no respetar distancias de seguridad, etc., serán bastante reducidos.

Bienestar

Los vehículos autónomos también reducirían los costes laborales, podrían aliviar a los ocupantes de los vehículos de tener que estar todo el rato pendiente de la conducción y ese tiempo que se ahorran de estar conduciendo podrían aprovecharlo para realizar cualquier otra tarea que les apeteciera, o que necesitaran hacer. Las personas jóvenes, mayores o con discapacidad podrían disponer de un medio con el cual su habilidad para desplazarse sería mucho mayor que actualmente. A parte, el conductor del vehículo ya no deberá de estar en una posición “adelantada” para sujetar el volante, ya que este no existirá y eso hará posible que el interior de los vehículos sea más ergonómico.

Tráfico

En cuanto al tráfico también tendríamos varios avances como, por ejemplo, el límite de velocidad máximo podría ser aumentado, a la misma vez, la conducción sería más “estable” y se podría aumentar la capacidad de las carreteras. El número de congestiones producidas sería menor gracias a los aspectos anteriores y también a que el espacio de seguridad entre vehículos podría ser reducido ya que la precisión de los vehículos a la hora de realizar acciones sería muy buena.

Costes más bajos

Esta conducción más segura que nos ofrecerían los vehículos autónomos podría implicar una reducción en los precios de los seguros de vehículos. Para los vehículos que no sean eléctricos, las mejoras en el tráfico (menos congestión, tráfico más fluido, etc.) significarían una conducción más eficiente por lo tanto el consumo de combustible sería menor. Este ahorro de combustible también ocurriría en otras circunstancias como puede ser el cambio de marchas a velocidades incorrectas, aceleraciones y frenadas poco eficientes, etc.

Aparcamiento

Actualmente los vehículos que tenemos están en uso entre un 4-5% del tiempo, y el resto están aparcados. En cambio, con los vehículos autónomos podría haber un uso de estos más continuo, es decir, una vez estos hayan llegado a su destino, podrían ser usados de nuevo para ir a otro. Esto significaría que la necesidad de tener aparcamientos sería menor, lo cual implicaría que estos espacios para aparcar podrían ser "liberados" y podrían ser utilizados para construir parques, edificios, etc.

3.2.3 Problemas de las ciudades inteligentes

En cuanto a las ciudades inteligentes también tenemos bastantes problemas que solucionar en caso de que se quisiera implantar este modelo de ciudad. Sabemos que una ciudad inteligente necesita una enorme cantidad de datos para poder "funcionar". Estos datos son obtenidos con una serie de sensores, sensores que deben tener algún tipo de inteligencia. Esta inteligencia ha de permitirles ver, escuchar, "oler", etc. Los sensores podrían realizar funciones como, por ejemplo, medir la temperatura, detectar patrones de tráfico, detectar el tráfico generado por los peatones, analizar la calidad del aire que respiramos, etc.

El principal problema es el del nivel económico, ya que no cualquier ciudad puede invertir tanto capital para montar las ciudades inteligentes y es por este motivo por el que el crecimiento de estas tiene un ritmo tan lento.

El conjunto de sensores se estima que puede llegar a ser del orden de 1 trillón de sensores en el año 2020, lo cual nos lleva a uno de los problemas más importantes.

Este problema es el de la energía. Si se requiere obtener información fiable, la cual incluso puede llegar a salvar vidas, de cada uno de estos sensores, necesitaríamos una cantidad de energía bastante superior a la que disponemos hoy en día, la cual es insuficiente para alimentar este supuesto trillón de dispositivos o sensores.

Otro problema relacionado con la energía que alimenta a estos sensores es que esta alimentación debería de realizarse inalámbricamente, lo cual en estos momentos es imposible, a pesar de que existen empresas que ya están comenzando a trabajar en esto.

Suponiendo que tenemos todo este conjunto de dispositivos funcionando y alimentados de forma inalámbrica, surgen una serie de preguntas que actualmente son difíciles de responder. Algunas de estas preguntas podrían ser:

- ¿Cómo podemos disminuir la cantidad de tráfico en las horas punta?
- ¿Cómo reducimos la cantidad de polución tanto en interior de edificios como en el exterior de estos?
- ¿De dónde viene esta polución y cómo podemos evitarla?

Estas preguntas, entre otras, son algunas de las cuales pensamos que actualmente no pueden ser respondidas, o no pueden tener alguna solución directa.

3.2.4 Principales ventajas de las ciudades inteligentes

Las Smart City también nos proporcionan ventajas sobre las ciudades tradicionales. Las principales ventajas son:

- Seguridad
- Eficiencia
- Salud

Seguridad

Gracias a todos los dispositivos de vigilancia como pueden ser, cámaras, sensores de movimiento, carreteras inteligentes, hacen que dichas ciudades puedan estar en continua vigilancia y monitorización los 365 días al año, evitando incidentes como, por ejemplo, actos vandálicos, robos, accidentes.

Eficiencia

El ahorro energético en las ciudades inteligentes es debido a la gran optimización de la energía gracias a la optimización de las redes eléctricas y en redes de abastecimiento de agua.

Pero no solo de la propia ciudad sino también en la carretera, haciendo que el transito sea continuo sin atascos, permite que los vehículos gasten menos combustible.

Salud

Estas ciudades permiten la creación y cuidado de una gran cantidad de plantas capaces de absorber grandes niveles de CO2, que en cantidades no controladas es perjudicial para la salud de las personas. Gracias a la eficiencia obtenida por las ciudades inteligentes, hace que estos niveles aún bajen haciendo que haya un ambiente más puro y menos tóxico.

3.3 TRABAJOS REALIZADOS

En este apartado se detallan algunos de los trabajos más relevantes que han sido realizados en los ámbitos de conducción autónoma y smart cities.

3.3.1 Transporte público autónomo en Europa

Actualmente existen diversas iniciativas que cuentan con el apoyo y financiación de la Unión Europea para poder introducir el transporte público autónomo en las grandes ciudades.

Uno de estos proyectos más importantes es CityMobil2 [14]. Este proyecto busca implantar autobuses sin conductores con la intención de mejorar el transporte público urbano.

CityMobil2 fue creado por la compañía francesa Robosoft [15] y dispone de vehículos capaces de alcanzar los 50 km/h. El número de ocupantes que pueden viajar en estos autobuses es de 10, lo cual nos hace una idea del reducido tamaño que actualmente tienen estos.

Estos disponen de GPS y un sistema de láseres, mediante los cuales permiten saber su ubicación y también determinar su movimiento. A parte también tienen unos sensores ultrasónicos que son utilizados para detectar posibles obstáculos. A pesar de ello, estos

vehículos han de circular por un carril especial, lo que nos indica que aún han de realizarse ciertas mejoras para implantarlos completamente en la ciudad.

Estos autobuses son utilizados en Trikala, Grecia, donde hay 6 de ellos circulando y monitorizados de forma remota. Hasta el momento no ha habido accidentes, lo cual significa que este tipo de conducción ha sido un éxito.

3.3.2 Estados Unidos y los vehículos autónomos

Actualmente Estados Unidos es el país donde circulan más coches autónomos, más en concreto, en la ciudad de Mountain View, en California, ciudad donde se encuentra la sede central de Google.

A pesar de que la mayoría de estos vehículos realizan una circulación exitosa, hace un tiempo ocurrió el primer accidente de un coche autónomo de Google. En este accidente el coche impactó con un autobús al intentar incorporarse en el carril donde se encontraba este.

Con el objetivo de introducir los vehículos autónomos en las calles, el Estado de California ha puesto en marcha una nueva ley la cual regula la circulación de dichos vehículos autónomos. Esta responsabilidad será de un nuevo departamento llamado Departamento de Vehículos Automotores (DMV). A parte, el Departamento de Transporte de Estados Unidos ha sacado a la luz un plan nacional dedicado al desarrollo de vehículos autónomos.

3.3.3 Los camiones inteligentes

Algunos de los avances que nos pueden proporcionar los vehículos autónomos forman parte del campo de la logística. Una iniciativa holandesa, llamada European Truck Platooning Challenge [16], propone que es posible que haya una flota entera de camiones en circulación de forma autónoma sin ningún problema.

En este reto formaron parte diferentes marcas de camiones importantes, entre las cuales estaban, DAF Trucks, Iveco, Scania, Volvo, etc. Esta flota de camiones salió de diferentes puntos de ciudades europeas y finalizaron el trayecto en el Puerto de Rotterdam.

A lo largo del trayecto, que fue de unos 2.000 km, no hubo ningún accidente, lo cual reforzó la propuesta del propio reto de que es posible que en el ámbito de la logística se utilicen vehículos autónomos sin ningún problema en un futuro.

3.3.4 Las ciudades inteligentes

Uno de los trabajos realizados sobre las ciudades inteligentes es el de Huawei en Longgang, Shenzhen (China) [17], dónde se ha hecho una ciudad inteligente adaptada a la necesidad de los ciudadanos de la zona utilizando IoT, cloud computing, Big Data, y otras TIC. Con una inversión de casi 79 millones de USD (casi 70 millones de €) y que cubre una superficie de casi 18 mil metros cuadrados.

Esta idea de ciudad inteligente está dividida en tres partes con si se tratara de un sistema nervioso:

Centro de desarrollo en la nube (cerebro)
Dónde van todos los datos de la gente y dispositivos, para saber que ocurre en cada lugar y en cada momento, para ayudar al gobierno inteligente o simplemente para ofrecer mejores rutas turísticas a los visitantes.

Banda ancha (sistema nervioso central)
inalámbrica y por cable, cada dispositivo que tiene integrado el sistema IoT o liteOs, envía todos los datos que van recibiendo las personas, los sensores de temperatura, las cámaras de tráfico, etc.

Plataforma (terminaciones nerviosas)
Una plataforma abierta para que otras compañías puedan crear dispositivos compatibles con su sistema operativo.

3.4 APLICACIONES

Actualmente los principales campos en los que se podría instalar o utilizar vehículos autónomos son:

- Transporte de mercancía
- Transporte de personas

En el caso de las ciudades inteligentes:

- Gobierno inteligente
- Ciudad segura

3.4.1 Transporte de mercancía

Actualmente el transporte de mercancía es una de las áreas en las que los vehículos autónomos podrían suponer un gran avance. Ya existen varias empresas que están trabajando en el desarrollo de camiones los cuales pueden realizar trayectos sin la necesidad de que haya un conductor dentro del vehículo.

Algunas de las empresas que están trabajando en la creación de estos camiones autónomos son las siguientes:

Daimler

Daimler [18] es una de las primeras empresas que se han introducido en el campo de la conducción autónoma dirigida al transporte de mercancía. Esta empresa comenzó a realizar pruebas en el año 2014 y se está centrando en una série de campos, entre los cuales están el “Truck platooning” y la utilización de un conductor dentro del vehículo para solamente ciertas situaciones como pueden ser entradas y salidas de autovías y autopistas, para tener más seguridad.

Waymo

Waymo [19] comenzó a testear sus camiones hace pocos años en California y Arizona. En marzo, despegaron camiones en Atlanta con el fin de poder distribuir carga a distintos centros de datos de Google. Cada uno de estos camiones está equipado con un sistema radar para poder moverse y también consta de una persona en caso de que ocurra una emergencia.

Tesla

Tesla [20], una de las empresas más relevantes del momento, lanzó su primer camión autónomo en noviembre de 2017. El objetivo de esta empresa es tener los camiones listos para realizar transportes en el año 2019. El sistema de conducción de estos camiones se centrará en una conducción igual a la de los vehículos que fabrican actualmente, en los cuales el vehículo puede controlar el acelerador, freno y volante,

pero con una persona siempre al volante en caso de que surja una emergencia. El objetivo principal de Tesla es el de ofrecer unas características como las de "Truck platooning", donde una flota de camiones autónomos pueda ser capaz de seguir a un "camión líder", que sería el único camión de la flota conducido por una persona.

Embark

Embark [21] fue fundada en el año 2016 en San Francisco. Esta empresa tiene como objetivo principal que los camioneros tengan que pasar menos tiempo al volante, lo cual les podría permitir realizar más repartos diariamente. Están trabajando principalmente en la automatización de la conducción en las autopistas o autovías, lugar donde estos pasan la mayoría del tiempo. Una vez el camión tenga que salir de estas vías el conductor de este será el que tome el control del vehículo. Las pruebas de esta empresa se están realizando en El Paso, Texas y Palm Springs, California.

3.4.2 Transporte de personas

En cuanto al transporte de personas actualmente también hay varias empresas que están trabajando para poder incorporar los vehículos autónomos en su entorno de trabajo.

El principal objetivo que buscan estas empresas es el de poder realizar trayectos de un sitio a otro donde las únicas personas que estén dentro del vehículo sean pasajeros, cada uno con su destino correspondiente.

Algunas empresas como Uber [22] o Lyft [23] ya han comenzado a realizar pruebas de transporte de personas con vehículos autónomos, aunque por el momento estas pruebas aún deben de hacerse con un conductor que siempre esté vigilando el correcto funcionamiento del vehículo y pueda tomar el control en caso de que haya algún problema.

Una de las ventajas que podría tener la implantación de este sistema de transporte es que estos vehículos podrían estar en continuo funcionamiento lo cual aumentaría la rentabilidad de las empresas que utilicen este medio.

3.4.3 Gobierno inteligente

Gracias a tener una ciudad inteligente, comporta un gran manejo datos, los cuales se pueden utilizar para hacer simulación de posibles escenarios futuros y por lo tanto ayudar a elegir un mejor escenario para esa ciudad. Un mejor escenario incluye un tipo de persona al mando, por lo que el partido propuesto como ganador sería aquel que a partir de los datos sería lo que necesita la ciudad. Haciendo que la transición entre gobiernos sea más suave.

3.4.4 Ciudad segura

Gracias a las mejoras tecnológicas, podemos evitar que nos suplanten la identidad, robo de datos o bien evitar estafas, ya que, se integran medidas de seguridad para que dichos ciberdelincuentes no puedan hacernos daño.

Aunque no nos referimos a mejores en seguridad del ámbito de ciberseguridad sino también que gracias a datos recopilados podrían detectarse escenarios de riesgo y poderlos evitar y en el caso de que no, poder avisar a emergencias de inmediato, reduciendo así el número de accidentes.

4. ANÁLISIS DE REQUISITOS

En este apartado se pretende detallar el conjunto de requisitos necesarios para poder llevar a cabo el desarrollo de los objetivos generales del proyecto.

4.1 ESCENARIO DEL PROYECTO

Primero de todo, y antes de comenzar a definir los diferentes requisitos, vamos a contextualizar el proyecto con tal de saber en el punto en que nos encontramos.

4.1.1 CRAAX

Como ya se ha mencionado anteriormente, el CRAAX es un grupo de investigación formado por profesores e investigadores del Departamento de Arquitectura de Computadores (DAC) [24], así como por investigadores del Hospital Clínic de Barcelona. También forman parte de este un conjunto de estudiantes de doctorado (PhD) y otros estudiantes en el ámbito de las ciencias de computación e ingenierías de telecomunicaciones. Los principales objetivos del CRAAX son, una investigación de gran calidad y la formación de estudiantes, con la finalidad de poder convertirlos en profesionales altamente calificados.

Hasta el día de hoy, los miembros del CRAAX, están participando de forma activa en diferentes proyectos de investigación a nivel nacional e internacional. También están trabajando en el desarrollo de tecnologías, principalmente en los ámbitos del transporte, salud, hogares y ciudades inteligentes.

En el contexto de este proyecto, el CRAAX actúa principalmente como cliente, el cual es quien solicita la realización de este y es el interesado en ver los resultados obtenidos una vez finalice este.

4.1.2 MF2C

El mF2C es un proyecto europeo, en el cual forma parte actualmente el equipo del CRAAX, proyecto que se encuentra en su fase final.

En la actualidad, la conexión entre diferentes dispositivos se realiza a través de una nube, la cual suele estar ubicada a una gran distancia de estos dispositivos. Teniendo en cuenta esta gran distancia, el tiempo de comunicación es muy lento, cuando la

finalidad es implementar soluciones de *Internet of Things* en tiempo real. MF2C pretende incorporar una capa intermedia con el nombre de Fog entre los dispositivos y la nube, la cual tendría la capacidad de realizar computación y procesamiento de datos. Esta nueva capa sería propia de las ciudades, lo cual permitiría que estos tiempos de comunicación se redujeran drásticamente, hasta el punto de ser prácticamente inmediatos.

Además, actualmente Fog y Cloud se encuentran en áreas separadas. MF2C pretende unificar estas dos áreas con la finalidad de obtener un rendimiento mayor. Esto se podría conseguir interconectando diversos dispositivos IoT, como por ejemplo teléfonos móviles, ordenadores portátiles, computadoras de los vehículos, entre otros, y utilizando la capacidad de computación que estos ofrecen y que en muchas ocasiones no se utiliza.

A continuación, se puede observar una imagen con un diagrama que representa la unificación del Fog Computing y el Cloud Computing propuesta por el equipo de mF2C:

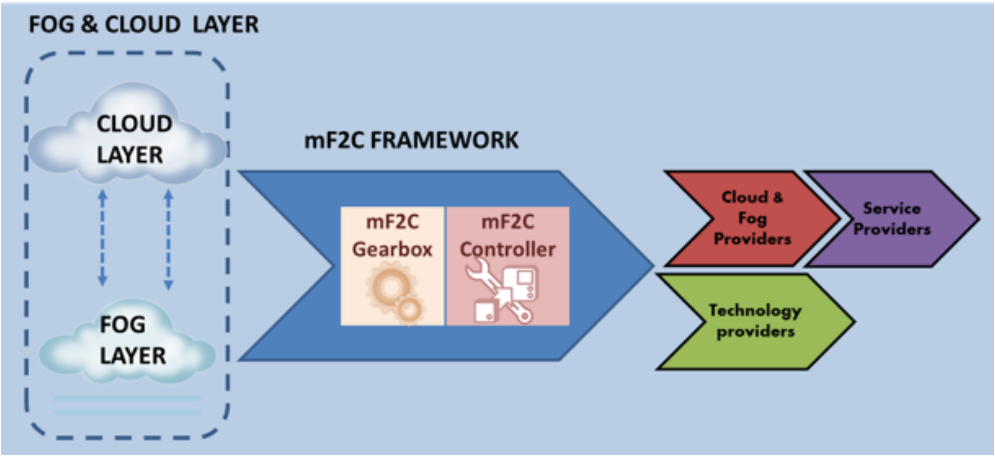

Figura 2: Unificación de Fog Computing y Cloud Computing

4.1.3 Testbed

Las dimensiones del Testbed son de 15 metros cuadrados. Dentro de este podemos encontrar un total de 5 calles, 3 de ellas son verticales y las otras 2 horizontales. Este es travesado horizontalmente por un puente levadizo y también cuenta con una serie de edificios que representan, un hospital, una estación de bomberos y el centro de tecnología *Neàpolis*, edificio situado en Vilanova i la Geltrú donde se encuentra el laboratorio del CRAAX.

Este Testbed ha sido diseñado e implementado por otro equipo que ha realizado un TFG con nombre “Diseño e Implementación de un Testbed para una SmartCity”. Para la realización de este proyecto no se ha definido ningún objetivo tal como desarrollar o implementar ningún elemento del Testbed, pero si que se han utilizado los diferentes elementos de los cuales dispone este con la finalidad de aportar en el desarrollo de nuestro proyecto.

4.1.3.1 Elementos del Testbed

En cuanto a los elementos que podemos encontrar en el Testbed, podemos distribuir estos en las siguientes categorías: elementos inteligentes y elementos no inteligentes.

Elementos inteligentes

Definimos como elementos inteligentes todos aquellos elementos que tienen capacidad de computación y procesamiento de datos.

Como elementos inteligentes podemos encontrar los siguientes:

- Vehículo
- Gestor de los semáforos
- Gestor de las farolas
- Gestor del tráfico
- Gestor del parking

Consideramos estos elementos anteriores como inteligentes gracias a la incorporación de un componente muy importante, la Raspberry Pi [25].

Una Raspberry Pi es una placa computadora (SBC) de bajo coste con capacidad de computación y procesamiento de datos. Es gracias a esta capacidad de computación y procesamiento de datos que podemos considerar como inteligentes los elementos nombrados anteriormente.

A parte, estos elementos también disponen de un componente importante llamado Arduino [26]. Un Arduino es una placa con entradas y salidas, tanto analógicas como digitales, en un entorno de desarrollo que está basado en el lenguaje de programación Processing [27].

Los Arduinos son utilizados para conectar diferentes dispositivos, que generalmente actuaran como sensores, a estos elementos, para de esta manera hacerlos capaces de generar o recopilar información útil que puede ser utilizada posteriormente.

Elementos no inteligentes

Definimos como elementos no inteligentes todos aquellos elementos que no tienen capacidad de computación ni procesamiento de datos.

Como elementos no inteligentes podemos encontrar los siguientes:

- TAG
- Farola
- Semáforo
- Barrera
- Pista
- Acera
- Contenedor
- Edificio

Figura 3: Testbed

4.1.4 Front-End

El Front-End es la parte gráfica de una plataforma desarrollada e implementada por un estudiante que ha realizado el TFG nombrado “Panel Front-End para el control de una Smart City”. Este dispone de una interfaz donde el usuario es capaz de interactuar con los diferentes elementos que la componen.
En el contexto de este proyecto, el Front-End será utilizado para visualizar el estado de diferentes componentes del Testbed en la interfaz que este nos proporciona en tiempo real. Los elementos que se visualizarán serán tanto elementos físicos como elementos virtuales.

A continuación, se puede ver una imagen con la interfaz del mapa del Testbed que nos proporciona el Front-End:

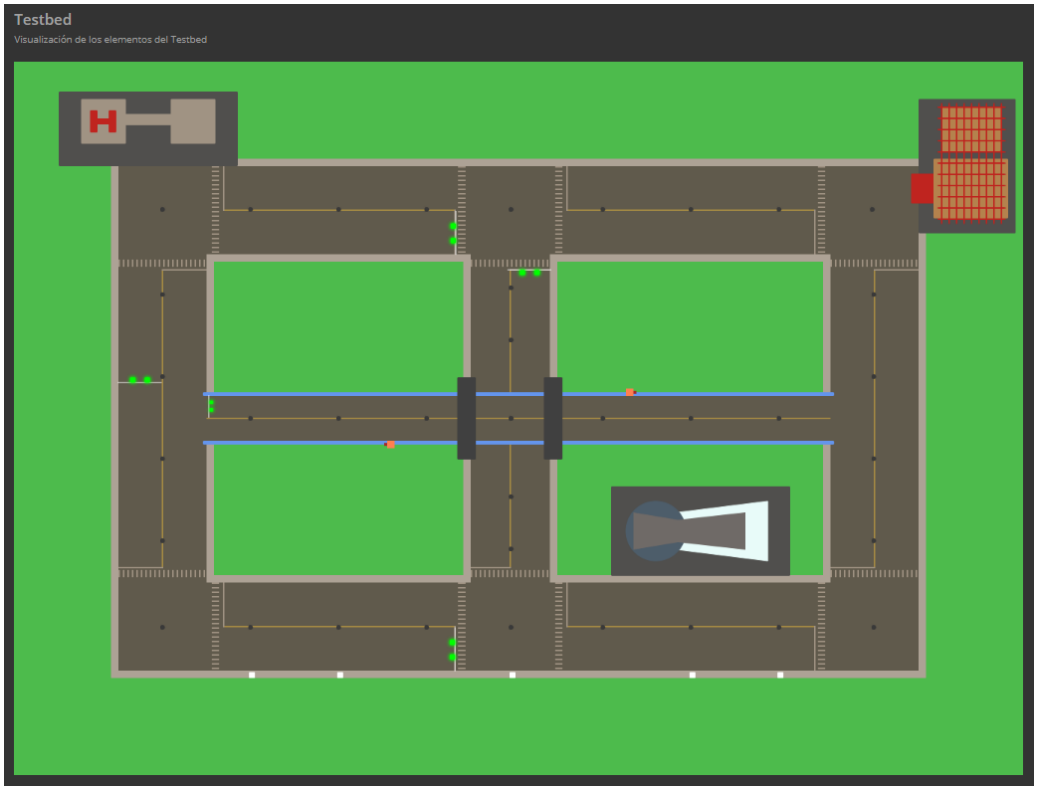

Figura 4: Mapa del Testbed (Front-End)

4.1.5 Agent

Un agent es un componente que permite añadir inteligencia a ciertos elementos de una ciudad inteligente. A parte, también aporta una capacidad de computación, que puede ser utilizada para ejecutar servicios de la ciudad inteligente.

La estructura del agent está dividida en cuatro módulos. Estos módulos son los siguientes:

- Módulo API
- Módulo Runtime
- Módulo Service Execution
- Módulo Topology Resource Management

Módulo API

Este módulo es el encargado de comunicar el agent con otros agents. A parte de comunicar los agents, este módulo también es el encargado de realizar conexiones con la base de datos topológica. Este módulo puede ser utilizado para añadir nuevos nodos a la topología, modificar información de nodos ya existentes, eliminar nodos de la base de datos o simplemente obtener datos de estos.

Módulo Runtime

El módulo Runtime es el encargado de descargar los códigos de los servicios a ejecutar, realizar la ejecución de estos y finalmente devolver el resultado.

Módulo Service Execution

Este módulo tiene la función de gestionar los servicios solicitados al propio agent. Es el encargado de comprobar las diferentes características de los servicios y dependiendo de estas realiza unas acciones u otras.

Módulo Topology Resource Management

Este módulo es el encargado de gestionar conexiones entre agents de la topología, así como de mantener actualizado el estado de esta en tiempo real.

4.2 REQUISITOS FUNCIONALES

Este apartado tiene como objetivo identificar los diferentes requisitos necesarios para poder llevar a cabo el desarrollo del proyecto. Estos requisitos son identificados a partir de diferentes necesidades del cliente que harán que la solución obtenida se adapte completamente a estas.

A continuación, podemos ver el listado de requisitos funcionales de cada uno de los diferentes apartados a desarrollar en este proyecto:

4.2.1 Agent

Los diferentes requisitos que ha de ser capaz de cumplir un agent son los siguientes:

- Tiene que ser capaz de comunicarse con otros agents de la misma topología de red.
- Ha de ser capaz de poder comunicarse con otros elementos que se encuentren dentro de la misma topología de red.
- La estructura de un agent tiene que estar formada por diferentes módulos, definidos por el cliente, cada uno de ellos independiente de los demás.
- Debe ser capaz de gestionar servicios, es decir, poder recibir, ejecutar y solicitar estos.

4.2.2 Panel de administración (Front-End)

El panel de administración debe de cumplir los siguientes requisitos:

- Ha de permitir la visualización de todos los nodos que se encuentren registrados en la base de datos topológica.
- Debe poder visualizar todos los servicios del catálogo de servicios de la ciudad inteligente.
- Ha de contemplar la visualización detallada de un nodo específico de la topología. A demás de ver esta información detallada, también debe de poder mostrar el conjunto de servicios que el nodo seleccionado puede solicitar.
- Ha de permitir añadir nuevos servicios al catálogo de servicios de la ciudad inteligente, siguiendo un formato definido.
- Debe poder solicitar la ejecución de cualquier servicio del catálogo de servicios al nodo cloud agent, para que posteriormente este gestione el servicio solicitado de la manera correspondiente.

4.2.3 Aplicación cliente de un agent

La aplicación del cliente debe cumplir con las siguientes características y funcionalidades:

- Debe de poder configurar un agent des de cero, es decir, asignar a este todos los parámetros necesarios para que pueda ser registrado correctamente en la base de datos topológica.
- Ha de poder mostrar un listado de los servicios que el agent puede solicitar.
- Ha de permitir la solicitud de cualquier servicio que se encuentre en el listado de servicios mostrado.
- Debe de poder mostrar el resultado de la ejecución de cualquier servicio que haya sido solicitado.

5. DISEÑO

En el apartado de diseño se detalla de forma precisa el conjunto de características que ha de tener cada uno de los elementos a implementar para que estos cumplan sus funcionalidades correctamente.

Los diferentes elementos que han sido diseñados son los siguientes:

- Topología de la red
- Conducción autónoma del vehículo
- Agent
- Servicio
- Nuevas funcionalidades del Front-End
- Aplicación cliente de un agent

5.1 TOPOLOGÍA DE LA RED

El diseño de la topología de la red está basado en la arquitectura Fog-to-cloud. Dicha topología está compuesta por varios nodos repartidos en diferentes niveles o capas. Estas capas son las siguientes:

- Capa Cloud
- Capa Fog
- Capa Edge

La **capa Cloud** es la capa más alta de la topología. Esta capa está formada por uno o más servidores que ofrecen diferentes servicios a los dispositivos que se encuentran en las capas situadas bajo esta. En nuestro proyecto se ha utilizado para almacenar el Front-End, la base de datos topológica, el catálogo de serviciosy los códigos de dichos servicios.

La **capa Fog** es una capa en la cual podemos encontrar nodos, los cuales pueden tener como rol agent o leader. La funcionalidad principal de esta capa es la de poder intercambiar información entre estos nodos de forma directa y casi inmediata.

La **capa Edge** es una capa en la cual se encuentran todos los dispositivos IoT que podemos encontrar en nuestra topología. Estos están relacionados directamente con su respectivo agent/leader y sirven para generar o recopilar información que puede ser útil para su agent/leader. Esta información puede ser, por ejemplo, la temperatura

ambiental, nivel de humedad, intensidad de la luz, etc.

A continuación, se puede ver un diagrama donde se representa cada uno de los niveles anteriormente nombrados y como se relacionan entre ellos:

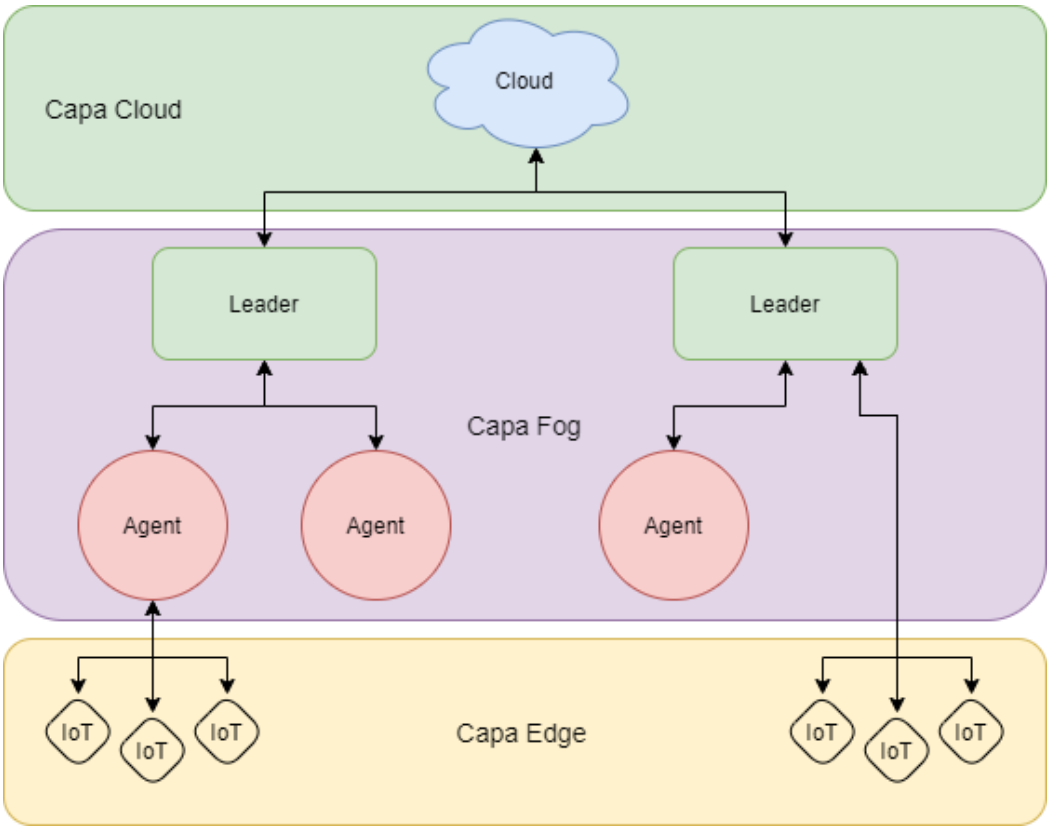

Figura 5: Niveles de la topología de la red

5.2 CONDUCCIÓN AUTÓNOMA DEL VEHÍCULO

Para poder llevar a cabo una conducción autónoma de los vehículos ha sido necesario cumplir una serie de requisitos. Estos requisitos son los siguientes:

- Disponer de diferentes sensores instalados en los vehículos
  - Lector de línea (Sensor fotovoltaico)
  - Lector de RFID
  - Sensor de ultrasonido
- Disponer de la topología anteriormente nombrada
- Disponer de la implementación completa y funcional de un agent

Una vez hemos cumplido los requisitos anteriores hemos pasado al diseño de la conducción de nuestros vehículos. El sistema de conducción de estos está dividido en

diferentes módulos, estructurados de la siguiente manera:

- car_movement.py
- sensors.py
- line_follower.py
- decision_maker.py

car_movement.py

Este módulo es el encargado de controlar la parte mecánica del vehículo, es decir, el motor y el direccionamiento de las ruedas. Dispone de un conjunto de funciones capaces de hacer girar las ruedas y de poner el motor en movimiento, tanto para ir hacia delante como para ir hacia atrás o detenerlo por completo.

sensors.py

El módulo de sensores es un módulo que dispone de diferentes funciones capaces de proporcionar información sobre cada uno de los diferentes sensores que están instalados en nuestros vehículos. La información que podemos obtener de estos es la siguiente:

- Lector de línea: El lector de línea es un sensor que capta una serie de valores los cuales indican si el vehículo se encuentra sobre una línea blanca, o no. Esta línea es utilizada para que el vehículo sea capaz de moverse dentro de los carriles de la ciudad.

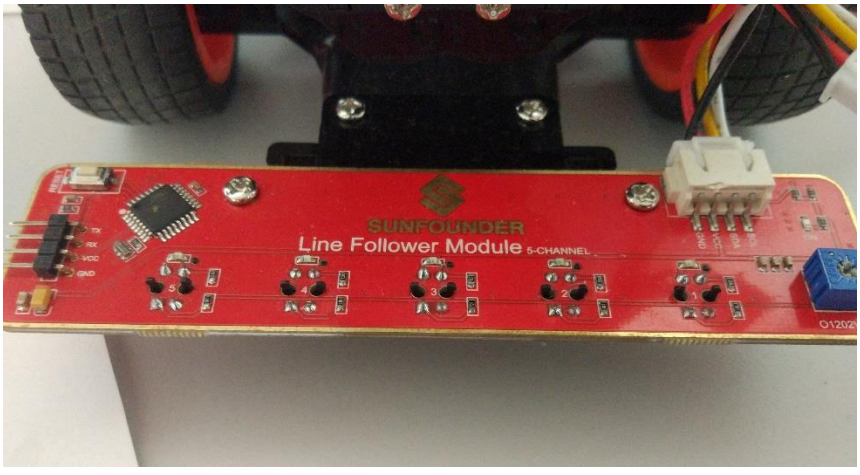

Figura 6 : Line Follower Module

- Lector de RFID: El lector de RFID es utilizado para poder tener constancia de en qué posición de la ciudad se encuentra el vehículo en todo momento. Esto se hace mediante la lectura de tags RFID situados bajo la ciudad, con la

finalidad de que los vehículos puedan leer estos y actualizar su localización en tiempo real.

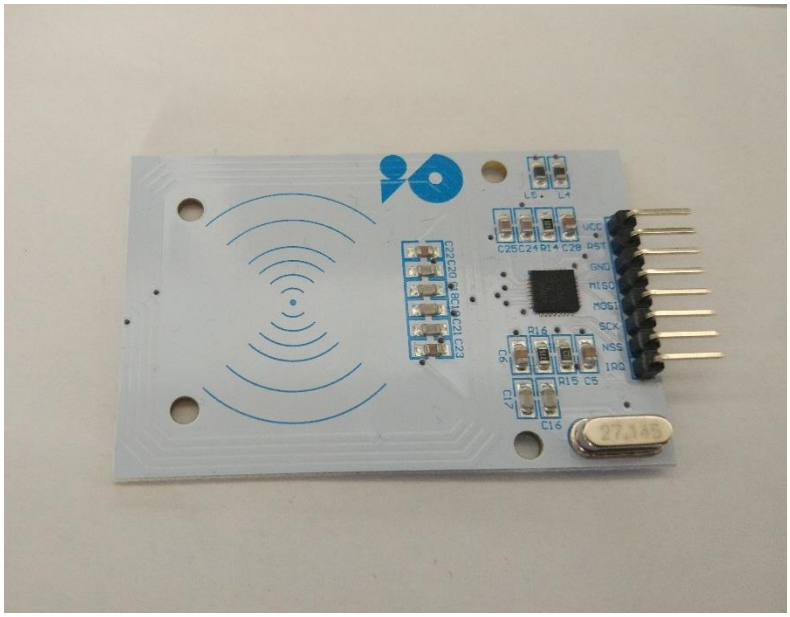

Figura 7: Rfid read and write module

- Sensor de ultrasonido: Este sensor está situado en la parte frontal del vehículo. Su función es la de proporcionar la distancia de cualquier objecto situado delante del propio vehículo para poder detectar obstáculos y realizar las acciones correspondientes en el caso de detectar alguno.

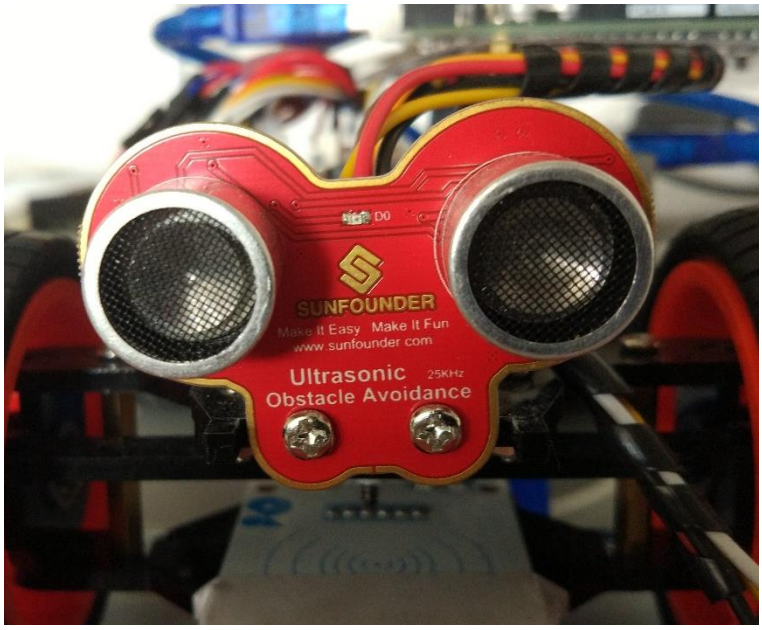

Figura 8: Ultrasonic Obstacle Avoidance Module

line_follower.py

Este módulo es el encargado de recoger información del lector de línea del módulo de sensors.py. Con la información obtenida, este módulo se encarga de actualizar el ángulo de giro de las ruedas del vehículo para que de esta manera este pueda circular siempre por su carril correspondiente.

decision_maker.py

Este último módulo es el más importante de todos. Este recoge información de los dos sensores restantes y realiza ciertas acciones dependiendo de esta. Estas acciones pueden ser detener el vehículo por completo, detenerlo para ponerlo posteriormente en circulación, o ponerlo a circular directamente. Pero la funcionalidad más importante que tiene este módulo es la de poder comunicarse con otros agents que tienen el control de otros dispositivos necesarios para poder realizar una conducción autónoma óptima. Estos agents con los que se ha de comunicar el vehículo, generalmente son agents que controlan, semáforos, farolas y el estado del tráfico de la ciudad.

A continuación, se puede observar un diagrama en el cual se muestran las diferentes conexiones que existen entre los módulos capaces de realizar acciones sobre un vehículo autónomo:

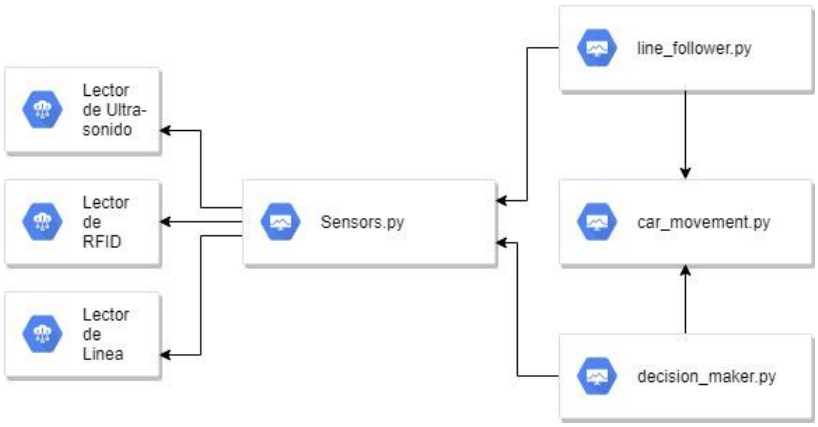

Figura 9: Diagrama de los módulos de conducción autónoma

5.3 AGENT

Como hemos visto anteriormente en los requisitos de un agent, este cuenta con diferentes módulos, cada uno encargado de realizar una función específica.

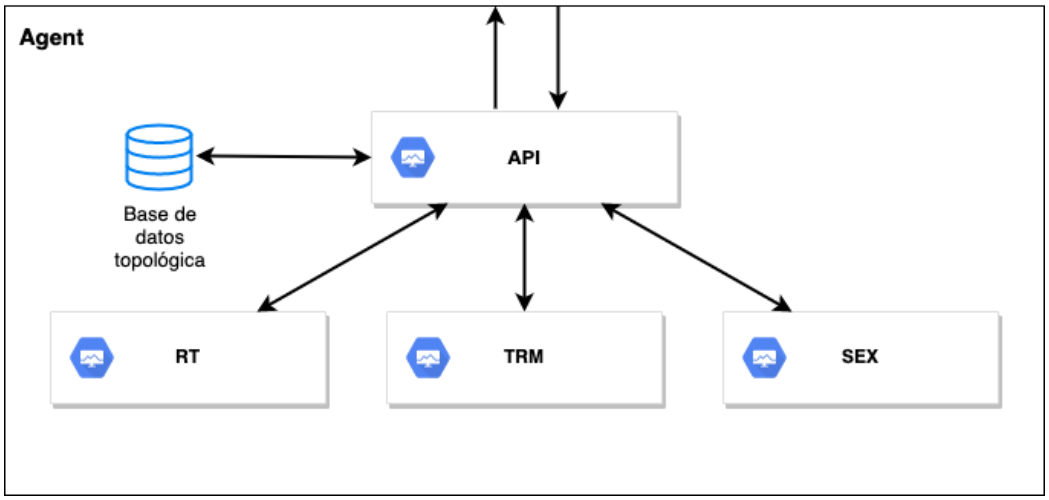

Figura 10: Estructura de un agent

A continuación, se muestra de forma detallada el diseño de cada uno de estos módulos:

5.3.1 Módulo API

Este módulo tiene como funcionalidad principal poder recibir peticiones HTTP [28] de otros elementos que le sean visibles a nivel de red, así como también poder realizar peticiones HTTP a otros elementos.

Esta API ha de ser capaz de resolver un conjunto de peticiones necesarias para la correcta comunicación entre agents. Dichas peticiones son aquellas que hemos definido nosotros (explicadas más adelante).

Los diferentes tipos de peticiones que puede gestionar la API son los siguientes:

- **GET:** Las peticiones GET son utilizadas para recuperar recursos.
- **POST:** Las peticiones POST son utilizadas para crear recursos.
- **PUT:** Las peticiones PUT son utilizadas para modificar valores de recursos.
- **DELETE:** Las peticiones DELETE son utilizadas para eliminar recursos.

5.3.2 Módulo Runtime

Este módulo se encarga de la ejecución de servicios. Para hacerlo debe descargar los codigos a ejecutar, analizar los parametros de entrada para recogerlos desde el codigo, y una vez se ha hecho todo esto, ejecuta el codigo del servicio para finalmente devolver el resultado de este.

5.3.3 Módulo Service Execution

Este módulo tiene la función de gestionar los servicios solicitados al propio agent. Es el encargado de comprobar las diferentes características de los servicios y dependiendo de estas realiza unas acciones u otras.
Para los servicios complejos, este módulo se encarga de resolver dichas dependencias para que se ejecute correctamente el servicio solicitado.

5.3.4 Módulo Topology Resource Management

Este módulo es el encargado de gestionar conexiones entre agents de la topología, así como de mantener actualizado el estado de esta en tiempo real.

5.4 SERVICIO

Una vez disponemos del diseño de todos los anteriores elementos, pasamos a diseñar los diferentes servicios que podrán ser ejecutados en nuestra ciudad inteligente. Estos servicios serán almacenados en un catálogo de servicios, el cual será accesible por todo habitante que se haya dado de alta en la ciudad inteligente.

En cuanto al diseño de estos servicios se ha definido un conjunto de parámetro para tener en cuenta. Estos parámetros tienen un papel muy importante a la hora de solicitar cualquier servicio.

En nuestro proyecto, el catálogo de servicios es representado como una base de datos no relacional. Esta base de datos ha sido creada utilizando la herramienta MongoDB [29].

MongoDB es una herramienta que nos permite crear bases de datos que contienen colecciones, dentro de las cuales se pueden crear, modificar, obtener o eliminar documentos. Los documentos son entradas en la base de datos con un conjunto de campos o parámetros que sirven para añadir información a estos documentos. En nuestro proyecto, cada uno de estos documentos es un servicio.

Para la creación de nuestros servicios se ha decidido que los parámetros necesarios sean los siguientes:

Identificador (id)	Nombre del servicio. Este campo es utilizado dentro de la colección para identificar un servicio.
Descripción (description)	Descripción del servicio utilizada para ayudar al usuario a entender el funcionamiento de este.
Versión de Python (pr-language)	Versión de Python necesaria para ejecutar el servicio. En nuestro caso las versiones de Python disponibles son “python2” y “python3”.
CPU (cpu)	Porcentaje de CPU que supuestamente será ocupado durante la ejecución del servicio.
Memoria (memory)	Porcentaje de memoria RAM que supuestamente será consumido durante la ejecución del servicio.
Permanente (permanent)	Campo que indica si un servicio se ejecutará de forma permanente o si por el contrario tendrá un fin.
Servicios de los que depende (depending-services)	Listado de identificadores de servicios de los cuales depende el servicio.
IoT (iots)	Conjunto de IoT necesarios para poder ejecutar el servicio.
Parámetros de entrada (params)	Conjunto de parámetros necesarios para ejecutar el servicio.
Programa del servicio (service-programs)	Fichero que contiene el código del servicio a ejecutar.
Programas de los que depende (depending-programs)	Fichero o ficheros de los cuales depende el servicio para poder ser ejecutado correctamente.
Dirección de descarga (download_host)	Dirección dónde se encuentran los archivos para ser descargados
Puerto de descarga (download_port)	Puerto dónde se encuentran los archivos para ser descargados

5.5 NUEVAS FUNCIONALIDADES DEL FRONT-END

Para el desarrollo de este proyecto, la ayuda que nos aporta el Front-End proporcionado es muy importante. Aun así, requerimos de ciertas funcionalidades que este no tiene implementadas.

Estas funcionalidades que han tenido que ser diseñadas e implementadas son las siguientes:

- Pantalla de visualización de agents
- Pantalla de visualización del catálogo de servicios de la ciudad inteligente
- Pantalla de visualización de la información de un agent
- Pantalla para añadir servicios al catálogo de servicios de la ciudad inteligente

Pantalla de visualización de agents

Pantalla que contiene un listado con todos los agents que se encuentran dentro de la topología de red de la ciudad inteligente. La pantalla en cuestión cuenta con un filtro para poder filtrar en la tabla. Para cada agent dentro de esta lista se pueden visualizar los siguientes parámetros:

nodeID	Identificador de un agent
myIP	Dirección IP del propio agent
leaderIP	Dirección IP del leader del agent
port	Puerto utilizado para realizar comunicaciones entre agents dentro de un Cluster, por defecto, el 5000
IoT	Listado de IoT's que tiene el agent
broadcastIP	Dirección IP de broadcast
cpu	Porcentaje de CPU de la cual dispone el agent expresado en GHz (gigahertz)
ram	Porcentaje de RAM de la cual dispone el agent expresado en GB (Gigabyte)
status	Parámetro almacenado en la base de datos que indica si un agent se encuentra activo o no
device	Parámetro que sirve para indicar que tipo de elemento es el agent (coche, ambulancia, etc.)
role	Parámetro que sirve para indicar de que tipo es el agent (agent, leader, cloud agent, etc.)

Cuadro 3: Parámetros de un agent

A continuación, podemos ver el diseño de la pantalla de visualización de los agents:

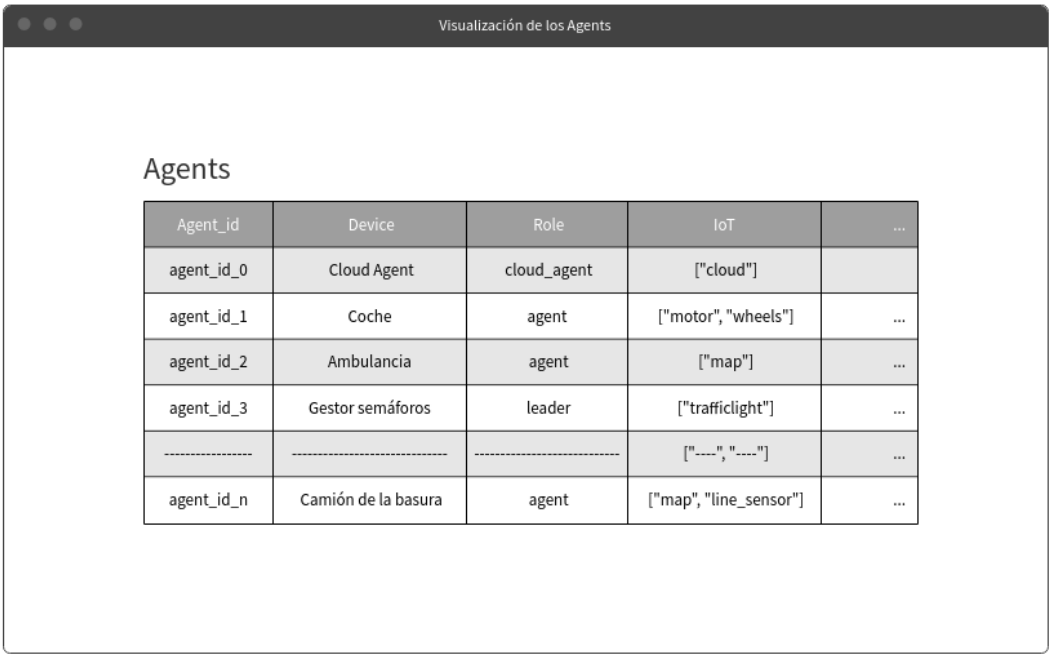

Figura 11: Diseño de la pantalla de visualización de los agents

Pantalla de visualización del catálogo de servicios de la ciudad inteligente

Pantalla que contiene un listado con el conjunto de servicios que se encuentran disponibles dentro de la ciudad inteligente. En esta pantalla habrá un enlace sobre la tabla para poder añadir más servicios en el caso de haya la necesidad de crearlos. Para cada servicio dentro de la lista se pueden observar los siguientes parámetros:

ID	Identificador del servicio
Descripción	Descripción del servicio utilizada para que los usuarios de la ciudad inteligente sepan qué servicio están ejecutando
Código	Nombre del fichero que contiene el código del servicio
IoT	Listado de IoT's necesarios para poder ejecutar el servicio
Parámetros	Parámetros de entrada necesarios para poder ejecutar el servicio
Versión de Python	Versión de Python con la que se ejecuta el código del servicio
Dependencias	Listado de identificadores de servicios de los cuales depende el servicio para poder ser ejecutado
Servicio permanente	Indica si el servicio se va a ejecutar de forma permanente (nunca acaba la ejecución) o si por el contrario se ejecuta de forma definida (su ejecución tiene fin)

A continuación, podemos el diseño de la pantalla de visualización del catálogo de servicios de la ciudad inteligente:

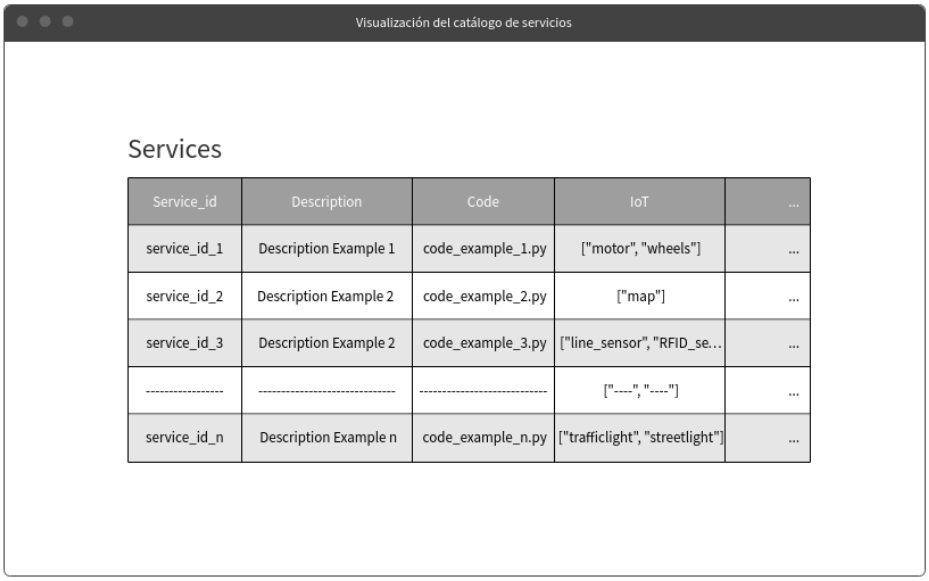

Figura 12: Diseño de la pantalla de visualización del catálogo de servicios

Pantalla de visualización de la información de un agent

En la pantalla de visualización de la información de un agent podemos ver todos los campos que tiene un agent cuando este es creado. Estos son mostrados en forma de listado y se encuentran situados a la parte izquierda de la pantalla.

También podemos observar un listado de servicios que pueden ser solicitados para que ese agent los ejecute. Estos servicios se encuentran situados en la parte derecha de la pantalla. Para cada servicio dentro de este listado se pueden observar los campos necesarios para poder solicitar la ejecución de este, asegurándonos que siempre que se solicite un servicio este sea solicitado conteniendo todos los parámetros que son requeridos.

Para acceder a esta pantalla, es necesario que el agent se encuentre activo, de lo contrario, no se podría acceder.

A continuación, podemos ver el diseño de pantalla de visualización de la información de un agent:

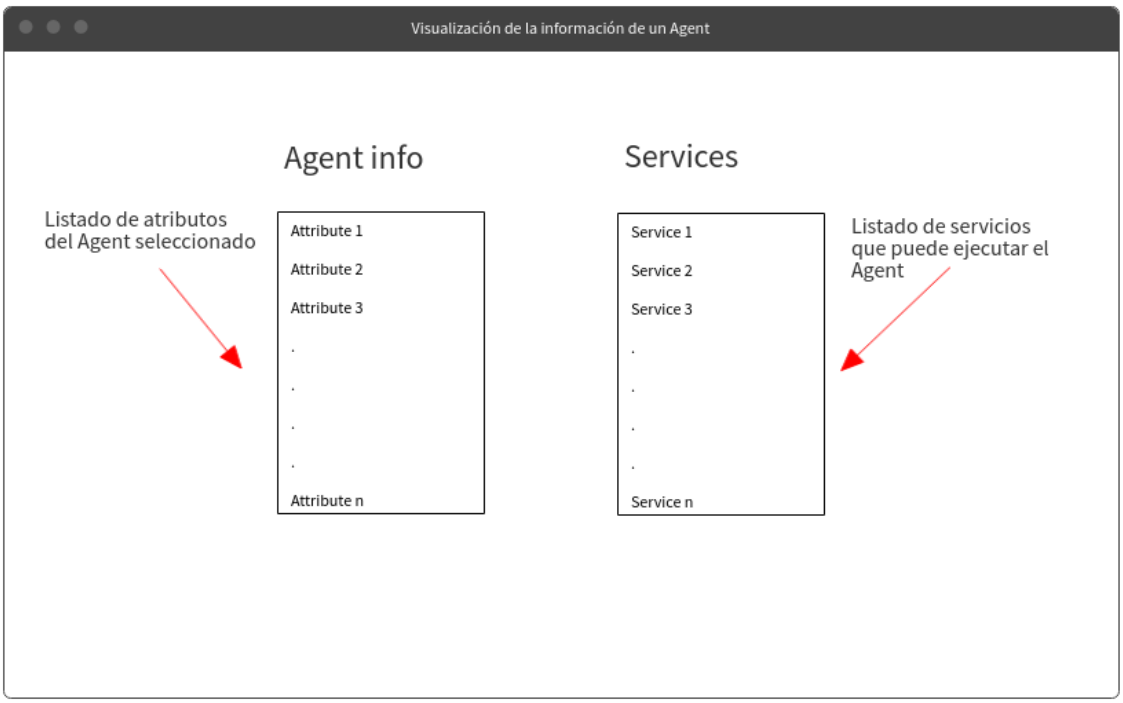

Figura 13: Diseño de la pantalla de visualización de la información de un agent

Pantalla para añadir servicios al catálogo de servicios de la ciudad inteligente

En esta última pantalla podremos añadir servicios nuevos al catálogo de servicios. Para ello se han añadido una serie de campos que el usuario tendrá que rellenar para poder crear un nuevo servicio. Estos campos son obligatorios, ya que todos los servicios que contenga el catálogo de servicios han de disponer de ellos. En caso de que algún servicio no disponga de alguno de los campos obligatorios este no podría ser creado para ser añadido al catálogo.

En la pantalla se puede ver que para cada atributo que ha de tener un servicio, existe un campo a rellenar. Algunos de estos campos son introducciones de texto, otros son listas de donde se pueden escoger diferentes opciones, etc.

Finalmente, una vez se hayan rellenado todos los campos correctamente, mediante la pulsación de un botón se podrá solicitar que se añada este nuevo servicio creado al catálogo de servicios de la ciudad inteligente.

El diseño de esta pantalla ha sido realizado de la siguiente manera:

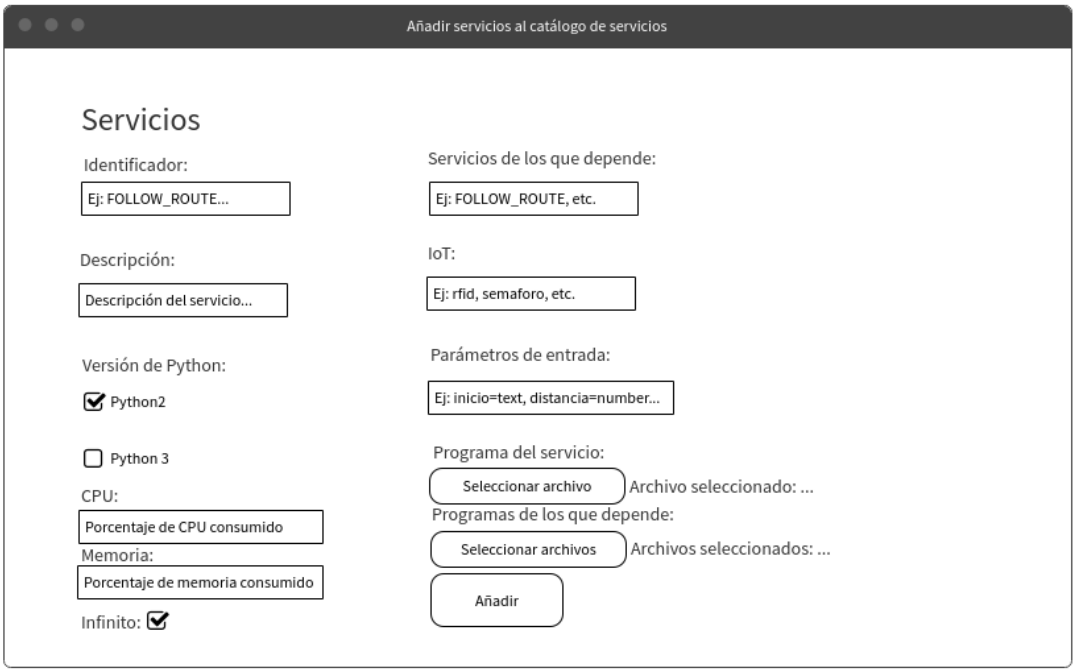

Figura 14: Diseño de la pantalla para añadir servicios al catálogo de servicios

5.6 APLICACIÓN CLIENTE DE UN AGENT

Como hemos mencionado anteriormente, la aplicación cliente de un agent tiene como objetivo principal poder solicitar la ejecución de diferentes servicios disponibles para ese agent. Pero previamente a poder solicitar estos servicios, debe de registrar el agent a la topología de red.

Esta aplicación dispone de un primer proceso donde, si esta es ejecutada por primera vez, permite al cliente configurar el agent. En esta configuración se asigna un tipo de "device" al agent, y también se le asigna un "role". El parámetro device sirve identificar fácilmente que elemento de la ciudad inteligente será el agent. El parámetro role sirve para indicar si el nuevo agent registrado hará la función de agent o de leader.

Una vez ya se ha configurado el agent, se pasa al segundo proceso, donde se muestra un listado con todos los servicios que puede solicitar el propio agent. Sobre esta lista, se puede solicitar el que quiera para solicitar la ejecución de este y posteriormente obtener el resultado de la ejecución.

6. CASOS DE USO

Un caso de uso (CU), es la descripción de una funcionalidad de nuestro sistema en la cual intervienen uno o varios actores. Para ello primero definiremos los actores y definiremos los casos de uso para finalmente mostrar los casos de uso a ejecutar por cada uno de los actores.

6.1 CASOS DE USO DE LA CONDUCCIÓN AUTÓNOMA

En este apartado se explican los diferentes casos de uso que podemos encontrar en la conducción autónoma.

6.1.1 Actores

Los actores son aquellos que pueden realizar acciones sobre nuestro sistema y que pueden interactuar sobre otros elementos de este. Para la conducción autónoma disponemos de los siguientes actores:

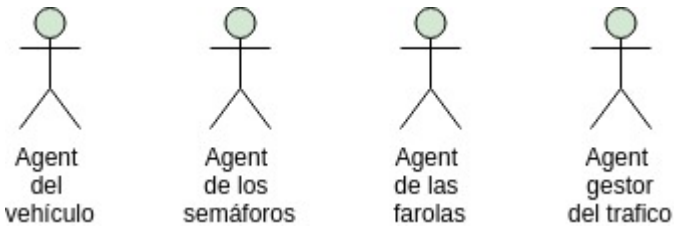

Figura 15: Actores de los casos de uso de la conducción autónoma

6.1.2 Descripción de los casos de uso

A continuación, se muestran de forma detallada los casos de uso de la conducción autónoma.

<table>
<tr><th>CU01</th><th colspan="2">Interacción con un semáforo</th></tr>
<tr><td>Descripción</td><td colspan="2">Diferentes acciones que un vehículo puede realizar cuando este llega a un semáforo</td></tr>
<tr><td>Actores</td><td colspan="2">Agent del vehículo y agent de los semáforos</td></tr>
<tr><td>Precondiciónes</td><td colspan="2">• Debe de existir una conexión directa entre el agent del vehículo y el agent de los semáforos.
• El vehículo debe de haber recibido la ruta a seguir
• El agent del vehículo ha recibido las posiciones de los semáforos</td></tr>
<tr><td>Secuencia normal</td><td>PASO</td><td>ACCIÓN</td></tr>
<tr><td></td><td>1</td><td>El vehículo lee un RFID relacionado con la posición de un semáforo</td></tr>
<tr><td></td><td>2</td><td>El agent del vehículo solicita al agent de los semáforos el estado del semáforo</td></tr>
<tr><td></td><td>3</td><td>El agent de los semáforos recibe la solicitud del estado del semáforo</td></tr>
<tr><td></td><td>4</td><td>El agent de los semáforos responde con el estado de este</td></tr>
<tr><td></td><td>5</td><td>Si la respuesta indica que el semáforo se encuentra en verde, el vehículo puede seguir circulando</td></tr>
<tr><td>Postcondiciones</td><td colspan="2">En caso de que el vehículo esté realizando un servicio de emergencia y solicite el cambio de estado del semáforo, este debe de volver a su estado normal una vez el vehículo ya haya pasado la intersección</td></tr>
<tr><td>Excepciones</td><td>EX1</td><td>ACCIÓN</td></tr>
<tr><td></td><td>2</td><td>Si el vehículo está realizando un servicio de emergencia, el agent del vehículo solicita el cambio del semáforo al estado de emergencia</td></tr>
<tr><td></td><td>3</td><td>El agent de los semáforos recibe la solicitud de cambio de estado del semáforo</td></tr>
<tr><td></td><td>4</td><td>El agent de los semáforos cambia el estado del semáforo a emergencia</td></tr>
<tr><td></td><td>EX2</td><td>ACCIÓN</td></tr>
<tr><td></td><td>5</td><td>Si la respuesta indica que el semáforo se encuentra en rojo o amarillo, el vehículo se detiene</td></tr>
<tr><td></td><td>6</td><td>Vuelve al paso 2</td></tr>
<tr><td></td><td>4</td><td>El agent de los semáforos cambia el estado del semáforo a emergencia</td></tr>
<tr><td>Comentarios</td><td colspan="2">Sin comentarios</td></tr>
</table>

Cuadro 4: Tabla del caso de uso "Interacción con un semáforo"

CU02	**Interacción con una farola**
Descripción	Diferentes acciones que un vehículo puede realizar cuando este llega a una farola
Actores	Agent del vehículo y agent de las farolas
Precondiciónes	• Debe de existir una conexión directa entre el agent del vehículo y el agent de las farolas. • El vehículo debe de haber recibido la ruta a seguir • El agent del vehículo ha recibido las posiciones de las farolas
Secuencia normal	**PASO** / **ACCIÓN** **1** / El vehículo lee un RFID relacionado con la posición de una farola **2** / El agent del vehículo solicita al agent de las farolas el cambio de estado de esta **3** / El agent de las farolas recibe la solicitud del cambio de estado esta **4** / El agent de las farolas modifica el estado para encender la/las farolas correspondientes
Postcondiciones	Una vez haya pasado el vehículo la farola debe de volverse a apagar
Excepciones	*Sin excepción*
Comentarios	*Sin comentarios*

Cuadro 5: Tabla del caso de uso "Interacción con una farola"

CU03	**Posibilidad de avanzar**
Descripción	Diferentes acciones que un vehículo puede realizar cuando este llega a un RFID
Actores	Agent del vehículo y agent gestor del tráfico
Precondiciónes	• Debe de existir una conexión directa entre el agent del vehículo y el agent gestor del tráfico. • El vehículo debe de haber recibido la ruta a seguir
Secuencia normal	**PASO** / **ACCIÓN** **1** / El vehículo lee un RFID **2** / El agent del vehículo solicita al agent gestor del tráfico si puede avanzar **3** / El agent gestor del tráfico recibe la solicitud **4** / El agent gestor del tráfico revisa que no haya nadie en esa posición y en el caso de que esté libre, reserva la siguiente para que este avance **5** / El agent gestor del tráfico le devuelve que puede avanzar **6** / El vehículo recibe la respuesta del gestor del tráfico y avanza
Postcondiciones	En caso de que el vehículo haya avanzado el gestor del trafico libera la posición donde estaba anteriormente
Excepciones	**EX1** / **ACCIÓN** **4** / El agent gestor del tráfico revisa que no haya nadie en esa posición y en el caso de que esté ocupada, devuelve al

		agent del vehículo que la posición esta ocupada
	5	El vehículo recibe la respuesta y se detiene
	6	Vuelve al paso 2
Comentarios	*Sin comentarios*	

Cuadro 6: Tabla del caso de uso "Posibilidad de avanzar"

6.1.3 Diagramas de los casos de uso

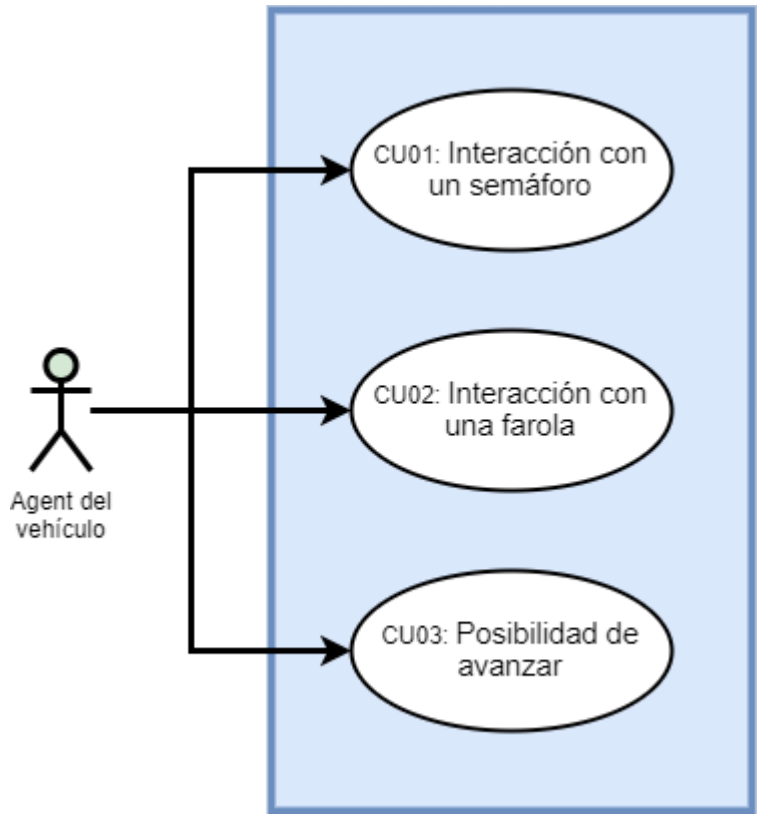

Figura 16: Diagrama de los casos de uso del agent de un vehículo

6.2 CASOS DE USO ADICIONALES

A continuación, se explican los diferentes casos de uso adicionales a la conducción autónoma con la finalidad de facilitar al usuario tanto la configuración de agents, creación de servicios y la propia ejecución de estos.

6.2.1 Actores

En cuanto a los casos de uso adicionales a la conducción autónoma, disponemos de los siguientes actores: el **administrador**, un **usuario** y el **agent**.

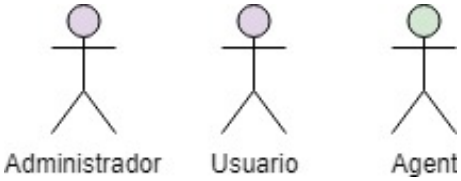

Figura 17: Actores de nuestro sistema

El Administrador y el Usuario personas físicas y el agent parte del sistema, de ahí que los primeros sean de color lila y el último de color verde. El administrador también puede ejecutar todo lo del usuario, pero no viceversa.

6.2.2 Descripción de los casos de uso

En este apartado se describe de forma detallada cada uno de los casos de uso de nuestro sistema.

CU04	**Configuración del agent**
Descripción	Un usuario configura un agent para poder usarlo posteriormente.
Actores	Usuario
Precondiciónes	El agent no debe estar previamente configurado
Secuencia normal	**PASO** / **ACCIÓN** **1** Arrancar el device.py **2** Indicar la IP del leader (si es el leader, la del cloud_agent) **3** Indica el tipo de Device (ambulancia, vehículo, gestor de semáforos…) **4** Indicamos el Rol del Agent, es decir si es agent o leader.
Postcondiciones	Si el leader/cloud agent, está conectado se registrará a él.
Excepciones	*Sin Excepciones*
Comentarios	Una vez configurado no se podrá modificar el agent.

Cuadro 7: Tabla del caso de uso "Configuración del agent"

CU05	**Registro del Agent**
Descripción	El Agent se registra a otro Agent leader/cloud Agent
Actores	Agent
Precondiciones	Que el Agent esté previamente configurado
Secuencia normal	**PASO** / **ACCIÓN** **1** El agent manda una petición de registro al a su leader **2** Una vez envía esta petición, el leader lo registra a la base de datos asignándole un ID único **3** El leader devuelve el ID al agent **4** El agent se guarda el leader
Postcondiciones	*Sin postcondición*
Excepciones	**EX1** / **ACCIÓN** **2** Una vez envía esta petición, este no está conectado **3** Vuelve al paso 1
Comentarios	Una vez se ha registrado ya no podrá volverse a registrar en la topología

Cuadro 8: Tabla del caso de uso "Registro de agent"

CU06	**Añadir servicio**
Descripción	El usuario Administrador añade un servicio al catalogo de servicios
Actores	Administrado
Precondiciones	• Si el servicio tiene dependencia a otros servicios, tenerlos añadidos previamente. • Tener el Front-End arrancado
Secuencia normal	**PASO** / **ACCIÓN** **1** El usuario administrador accede al Front-End y se va al enlace de servicios **2** Desde esta pantalla y sobre la tabla hay un botón el cuál pulsamos **3** Nos redirige a la pantalla de añadir servicio **4** Añadimos todos los datos requeridos **5** Ahora nos lleva de vuelta a la pantalla de servicios, donde aparece el nuevo servicio
Postcondiciones	*Sin postcondiciones*
Excepciones	*Sin excepciones*
Comentarios	Una vez se ha añadido ya no podrá modificarse el servicio

Cuadro 9: Tabla del caso de uso "Añadir servicio"

CU07	**Ejecución del servicio desde aplicación cliente**
Descripción	El usuario pide ejecutar un servicio de los que puede ejecutar al agent desde la aplicación de terminal.
Actores	Usuario, Agent
Precondiciones	Tener el agent/leader configurado, registrado Tener el leader de dicho agent/leader conectado al leader/cloud agent
Secuencia normal	**PASO** / **ACCIÓN** **1** El usuario arranca el device.py **2** Una vez arrancado, pide al leader la lista de servicios que puede ejecutar **3** Nos aparece el listado de servicios que puede ejecutar **4** Seleccionamos el servicio **5** Informamos los campos de entrada **6** Le damos a ejecutar **7** Se envía la petición al leader(en el caso de ser leader, a él mismo) **8** Comprueba que el servicio pedido lo puede ejecutar **9** Entonces le envía a ejecutar el servicio **10** Devuelve el resultado al leader **11** El leader le devuelve la información al agent que lo ha solicitado
Postcondiciones	*Sin postcondiciones*
Excepciones	**EX1** / **ACCIÓN** **2** Si este es leader se la pide a si mismo **3** Pasamos al paso 3

	EX2	**ACCIÓN**
	8	En el caso de que haya dependencia, no lo podrá ejecutar hasta que estas estén resueltas y por cada dependencia, volverá al paso 7
	EX3	**ACCIÓN**
	8	El servicio o alguna de las dependencias no se pueden ejecutar
	9	Montaremos el mensaje de que no se ha podido ejecutar el servicio porque nadie lo puede ejecutar y pasamos al paso 10
Comentarios	Si un servicio es infinito, una vez se está ejecutando, no se reiniciará	

Cuadro 10: Tabla del caso de uso "Ejecución del servicio des de la aplicación cliente"

CU08	**Ejecución del servicio des del Front-End**	
Descripción	El administrador desde el Front-End, pedirá la ejecución del servicio	
Actores	Administrador, Agent	
Precondiciones	Tener el cloud agent conectado	
Secuencia normal	**PASO**	**ACCIÓN**
	1	El usuario administrador accede al Front-End y se va al enlace de agents
	2	Desde esta pantalla podemos filtrar por el agent que queremos
	3	Clicamos en la lista sobre el agent que queremos que ejecute un servicio
	4	Nos aparece la información del Agent, los servicios y los servicios de calibración
	5	Seleccionamos el servicio a ejecutar
	6	Informamos los campos de entrada
	7	Le damos a ejecutar
	8	Se envía la petición al cloud agent
	9	El cloud agent busca al leader que lo puede ejecutar o que tiene el agent que lo puede ejecutar.
	10	Se envía la petición al leader encontrado
	11	Comprueba quien puede ejecutar el servicio(si él o sus agents)
	12	Entonces le envía a ejecutar el servicio
	13	Al leader le llega el resultado de la operación
	14	Se lo pasa al cloud agente
	15	El cloud agent envía la respuesta al Front-End
Postcondiciones	*Sin postcondiciones*	
Excepciones	**EX1**	**ACCIÓN**
	11	En el caso de que haya dependencia, no lo podrá ejecutar hasta que estas estén resueltas y por cada dependencia,

		hará el paso 11 y 12
	EX2	**ACCIÓN**
	11	El servicio o alguna de las dependendencias no se pueden ejecutar
	12	Montaremos el mensaje de que no se ha podido ejecutar el servicio porque nadie lo puede ejecutar y pasamos al paso 14
Comentarios	Si un servicio es infinito, una vez se está ejecutando, no se reiniciará	

Cuadro 11: Tabla del caso de uso "Ejecución del servicio des del Front-End"

6.2.3 Diagramas de los casos de uso

A partir de los casos de uso definidos anteriormente, podremos ver de forma esquemática qué casos de uso puede ejecutar cada uno de los actores:

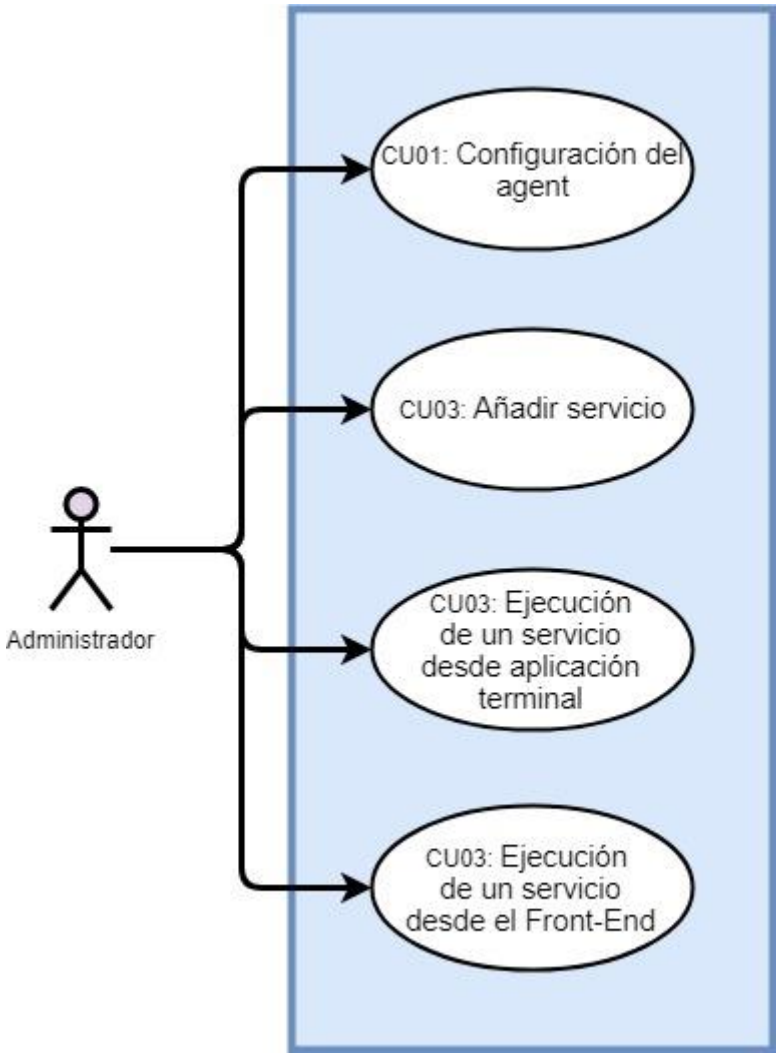

Figura 18: Diagrama de los casos de uso del Administrador

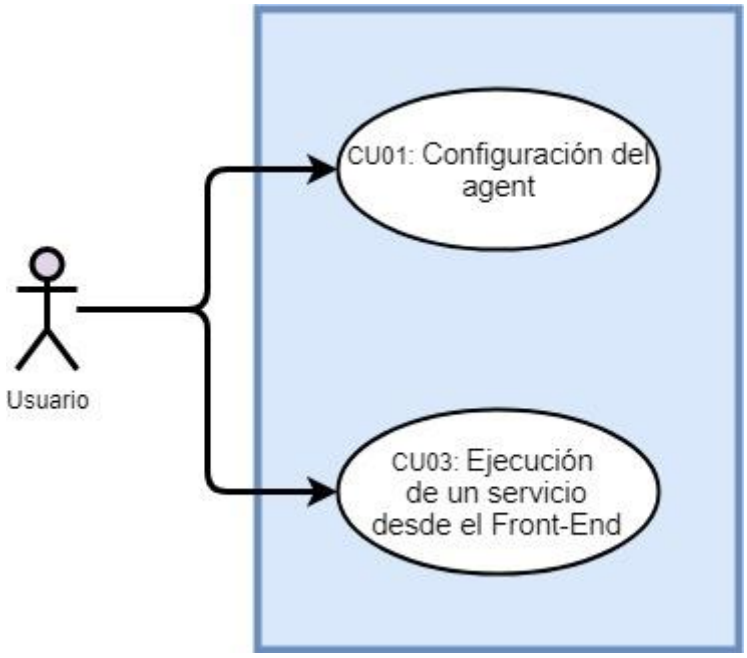

Figura 19: Diagrama de los casos de uso del usuario

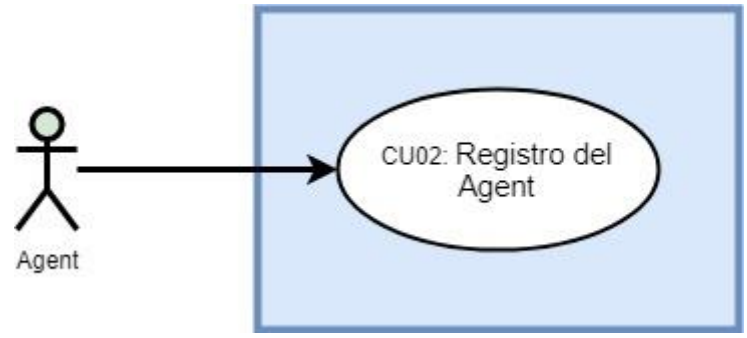

Figura 20: Diagrama de los casos de uso del agent

7. IMPLEMENTACIÓN

Una vez se ha acabado con el diseño de cada uno de los elementos necesarios, pasamos a la implementación de estos. Esta implementación se ha realizado sobre los mismos elementos nombrados anteriormente en el diseño, que son los siguientes:

- Topología de la red
- Conducción autónoma del vehículo
- Agent
- Servicio
- Nuevas funcionalidades del Front-End
- Aplicación cliente de un agent

7.1 TOPOLOGÍA DE LA RED

Para implementar la topología de la red ha sido necesario tener muy bien definida la estructura de esta. Tal y como hemos podido ver en el apartado del diseño, disponemos de cuatro capas o niveles.
Estas capas o niveles son creados mediante la asignación de diferentes roles a los nodos que forman parte de la topología de la red. Como se ha mencionado anteriormente los diferentes roles que puede tener un nodo de la topología son los siguientes: agent, leader y cloud_agent.

Para crear esta topología, primero de todo se ha de crear un nodo que tenga como rol cloud_agent. Este nodo es el que se encuentra en la capa más superior de la topología.

Posteriormente se ha de crear, por lo menos un leader, el cual tendrá como leader propio el nodo cloud_agent creado anteriormente. Y una vez tenemos el leader, podemos tener agents que tienen como leader, el leader que cuelga del cloud_agent. De esta manera, ya tenemos la segunda capa (Capa Fog) y la tercera (Capa Edge).

Después, ya se pueden crear, tanto leaders como agents, teniendo en cuenta las restricciones de conexiones entre nodos vistas en el diagrama del apartado “5.1 Topología”. Esto implica que todo leader que sea creado ha de tener como propio leader el cloud_agent y todo agent creado ha de tener como propio leader algún leader de la topología, sin poder ser este el cloud_agent.

De esta manera, mediante la asignación de roles a los nodos que vayan entrando en la ciudad inteligente, se puede crear la topología de red diseñada anteriormente.

7.2 CONDUCCIÓN AUTÓNOMA DEL VEHÍCULO

En la implementación de la conducción autónoma del vehículo, disponemos de cuatro módulos que, mediante la información recibida por sus sensores, analiza e interpreta para hacer posible este tipo de conducción.

A parte de estos módulos, también hemos aprovechado de otros códigos que han sido proporcionados por los desarrolladores de los vehículos utilizados en este proyecto. Los vehículos que han sido utilizados en este proyecto son los siguientes:

- PiCar-S
- PiCar-V

Los códigos de los desarrolladores han sido utilizados para ayudarnos a entender el funcionamiento de los diferentes sensores proporcionados, y de esta manera poder implementar nuestros propios módulos.

A continuación, se detalla el funcionamiento de cada uno de los cuatro módulos desarrollados para poder realizar la conducción autónoma.

Módulo car_movement.py

Como se ha explicado anteriormente, este módulo es encargado de controlar la parte mecánica del vehículo, es decir, el motor y el direccionamiento de las ruedas.

Este control de la parte mecánica del vehículo ha sido realizado mediante la programación de diferentes funciones que se encuentran dentro de una Clase Python llamada “CarMovement”.

A continuación, podemos ver la estructura de este módulo, y algunas de las funciones más utilizadas en la conducción autónoma del vehículo.

```
from picar import front_wheels, back_wheels
import random
import picar
import time


class CarMovement:

    DEFAULT_SPEED = 0
    DEFAULT_ANGLE = 90
    TURNING_DEGREES = 5
    SPEED_INCREMENT = 10
    MIN_SPEED = -80
    MAX_SPEED = 80
    MIN_ANGLE = 60
    MAX_ANGLE = 120
```

Figura 21: Declaración de la clase y sus atributos

En la figura anterior podemos ver que previamente a declarar la clase “CarMovement” hay una serie de líneas que importan diferentes módulos necesarios para poder programar las funciones que se encuentran dentro de ella. Podemos ver que uno de los módulos proporcionados por los desarrolladores de los vehículos es “picar”. Este módulo nos permite acceder tanto a las ruedas delanteras como a las ruedas traseras de un vehículo.

Más abajo, se declara la clase de la siguiente manera: **class CarMovement**, y finalmente, ya dentro de la clase se declaran una serie de atributos que son necesarios para poder llevar a cabo un buen control de la parte mecánica del vehículo, como pueden ser:

- **DEFAULT_SPEED:** Velocidad inicial a la cual se encuentra el motor
- **DEFAULT_ANGLE:** Angulo de giro inicial de las ruedas del vehículo
- **TURNING_DEGREES:** Incremento en grados del ángulo de giro de las ruedas
- **SPEED_INCREMENT:** Incremento de la velocidad del motor
- **MIN_SPEED:** Valor que indica cual puede ser la velocidad mínima del vehículo
- **MAX_SPEED:** Valor que indica cual puede ser la velocidad máxima del vehículo
- **MIN_ANGLE:** Valor que indica cual puede ser el ángulo mínimo de giro de las ruedas hacia la izquierda.
- **MAX_ANGLE:** Valor que indica cual puede ser el ángulo máximo de giro de las ruedas hacia la derecha.

Una vez definidos estos atributos, que actúan como valores límites tanto de la velocidad del motor, como del ángulo de giro de las ruedas, veremos dos ejemplos de funciones que han sido programadas para poder mover el vehículo y para poder realizar giros.

A continuación, podemos ver el código de la función encargada de actualizar la velocidad del vehículo:

```
def update_speed(self):
    if not self.is_stopped:
        if self.motor_speed < self.MIN_SPEED:
            self.motor_speed = self.MIN_SPEED
        elif self.motor_speed > self.MAX_SPEED:
            self.motor_speed = self.MAX_SPEED

        if self.motor_speed < 0:
            self.bw.speed = -self.motor_speed
            self.bw.backward()
        else:
            self.bw.speed = self.motor_speed
            self.bw.forward()
```

Figura 22: Función encargada de actualizar la velocidad de un vehículo

Observamos que al llamar a esta función solo se actuara si el vehículo se encuentra en movimiento. Esto ha sido hecho de esta manera ya que, si un vehículo se encuentra parado, por cualquier motivo, este no debería de estar moviendo las ruedas ya que no sería necesario.

Seguidamente, se realiza un control sobre la velocidad mínima y máxima del vehículo, así como el de los ángulos de giro de las ruedas. Este control se realiza de la siguiente manera:

1. Si la nueva velocidad del motor es menor a la mínima posible, se asigna la velocidad mínima posible definida en los atributos de la clase.
2. Si la nueva velocidad del motor es mayor a la máxima posible, se asigna la velocidad máxima posible definida en los atributos de la clase.
3. Si la velocidad resultante después de actualizar esta es menor que 0, la dirección de movimiento de motor se actualiza para que las ruedas se muevan hacia atrás.
4. Si la velocidad resultante después de actualizar esta es mayor que 0, la dirección del movimiento del motor se actualiza para que las ruedas se muevan hacia delante.
5. Si la velocidad resultante después de actualizar esta es igual a 0, se detiene el motor para que las ruedas no se muevan.

Siguiendo las comprobaciones anteriores aseguramos el correcto funcionamiento del motor del vehículo para que este pueda moverse correctamente.
A continuación, podemos ver la función encargada de actualizar el ángulo de giro de las ruedas del vehículo:

```
def update_angle(self, calibration=False):
    if calibration:
        if self.angle < self.MIN_ANGLE:
            self.angle = self.MIN_ANGLE
        elif self.angle > self.MAX_ANGLE:
            self.angle = self.MAX_ANGLE
        self.fw.turn(self.angle)
    elif not self.is_stopped and self.motor_speed != 0:
        if self.angle < self.MIN_ANGLE:
            self.angle = self.MIN_ANGLE
        elif self.angle > self.MAX_ANGLE:
            self.angle = self.MAX_ANGLE
        self.fw.turn(self.angle)
```

Figura 23: Función encargada de actualizar el ángulo de giro de las ruedas

Como podemos ver, esta función *está dividida en dos partes, una que será ejecutada si el vehículo se está calibrando y otra que se ejecutará si este no se está calibrando.*

Vemos qué en cualquiera de los casos, se comprueba lo siguiente:

1. Si el nuevo ángulo es menor que el ángulo mínimo posible, se asigna el valor mínimo posible al ángulo de giro de las ruedas. Esta comprobación es utilizada para controlar el ángulo de giro hacia la izquierda.
2. Si el nuevo ángulo es mayor que el ángulo máximo posible, se asigna el valor máximo posible al ángulo de giro de las ruedas. Esta comprobación es utilizada para controlar el ángulo de giro hacia la derecha.

Mediante estas comprobaciones nos aseguramos de que los ángulos de giro de las ruedas del vehículo siempre se encuentren dentro de los límites definidos anteriormente para evitar posibles problemas.

Módulo sensors.py

Dentro del módulo de sensores nos encontramos con una clase Python llamada Sensors la cual está compuesta por diferentes métodos encargados de obtener datos de los diferentes sensores que se encuentran instalados en el vehículo.

En nuestro caso, los diferentes sensores de los cuales queremos obtener información para de esta manera poder efectuar una conducción autónoma correcta son: sensor de línea, sensor de ultrasonido y lector de RFID.

Para poder obtener la información correspondiente de cada sensor, se han creado tres métodos principales, que son los siguientes:

```
def read_digital_line(self, line_references):
        lt = self.read_analog_line()
        digital_list = []
        if self.valid_data_line(lt):
            for i in range(0, 5):
                if lt[i] < line_references[i]:
                    digital_list.append(0)
                elif lt[i] > line_references[i]:
                    digital_list.append(1)
                else:
                    digital_list.append(-1)
        else:
            digital_list = [0,0,0,0,0]
        return digital_list
```

Figura 24: Función encargada de obtener información del sensor de línea

En la figura anterior podemos ver la función que es utilizada para obtener los datos generados por el sensor de línea que se ha instalado en el vehículo. Esta función utiliza dos funciones externas que son necesarias para obtener los datos del sensor y que de esta manera el vehículo pueda utilizar estos para poder circular correctamente.

Una de las funciones externas que podemos observar en el código es "read_analog_line". Esta función lee a través de los cinco sensores que tiene el lector de línea y devuelve una serie de valores para cada uno de estos sensores.

La segunda función externa es "valid_data_line". Esta se encarga de validar los datos obtenidos de la función anterior. En caso de que los datos sean validados se continua con la comprobación de estos. Si por el contrario estos no son válidos, se devuelve como resultado de la lectura una lista de cinco posiciones donde a cada posición se asigna el valor 0, indicando que no se ha detectado línea. Esta lista sería representada de la siguiente manera: [0, 0, 0, 0, 0].

Cada una de estas posiciones representa un sensor de los cinco que lleva incorporados el sensor de línea.

En el caso de que todos los sensores hayan podido leer información y, por lo tanto, esta haya sido validada correctamente mediante la función anterior, se pasa a un proceso de asignación de valores 1 o 0. Este proceso se realiza de la siguiente manera:

1. Se crea una nueva lista llamada "digital_list" la cual tendrá almacenados los diferentes valores obtenidos para cada uno de los cinco sensores.
2. Por cada posición de la lista obtenida con la función "read_analog_line" se comprueba:
    a. Si el valor de la posición es menor a un valor referencia se asigna un 0 a la misma posición de la nueva lista creada.
    b. Si el valor de la posición es mayor a un valor referencia se asigna un 1 a la misma posición de la nueva lista creada.
3. Una vez comprobadas las cinco posiciones de la lista se devuelve como resultado la nueva lista generada.

El valor referencia anteriormente nombrado es un valor almacenado en un fichero dentro del vehículo una vez este ha sido calibrado. Este fichero contiene cinco valores,

separados por comas, donde cada uno de estos es utilizado como referencia para diferenciar si el sensor ha leído blanco, o negro, para cada una de las cinco posiciones de las cuales dispone el sensor de línea.

La calibración del sensor de línea de un vehículo se realiza mediante la ejecución de un servicio del catálogo de servicios que hemos creado, con la finalidad de que esta calibración pueda ser realizada de forma sencilla para todos y cada uno de los vehículos de los cuales disponemos y bajo diferentes condiciones de luz.

A continuación, podemos ver la función encargada de obtener los datos del sensor de ultrasonido, el cual mide la distancia de cualquier objeto situado frente a este.

```
def read_distance(self):
        distance = self.UA.get_distance()
        while distance == -1:
            distance = self.UA.get_distance()
        return distance
```

Figura 25: Función encargada de obtener información del sensor de ultrasonido

Como podemos ver esta función es bastante simple. Primero de todo obtenemos el valor de la distancia (expresado en cm) gracias a la función “UA.get_distance()”, la cual ha sido proporcionada a través del módulo “Ultrasonic_Avoidance.py”, módulo que ha sido desarrollado por el equipo de SunFounder y contiene un conjunto de métodos que permiten obtener la distancia generada por el sensor de ultrasonido. Seguidamente comprobamos que el valor de la distancia obtenido sea correcto, es decir, que sea diferente a “-1”. En el caso de que el valor obtenido sea correcto, se devuelve tal valor, en caso contrario, se sigue comprobando hasta obtener uno correcto. De esta manera, nos aseguramos de que los valores devueltos siempre sean valores superiores o igual a 0, lo cual nos asegura un correcto funcionamiento del resto del código de la conducción continua.

Finalmente, para obtener los datos generados por el lector de RFID, contamos con la función “read_RFID”, que podemos ver a continuación.

```
def read_RFID(self):
        c = ""
        text = ""
        while c != '\n':
            if self.arduino.inWaiting() > 0:
                c = self.arduino.read(1)
                text += c
        return text.strip()
```

Figura 26: Función encargada de obtener información del sensor de RFID

Como podemos observar, esta función obtiene los RFID leídos a través de un Arduino, ya que el lector de RFID está conectado a este. Esta función está constantemente esperando a un carácter diferente a '\n', carácter que indica que no se ha leído nada. Una vez se encuentra un carácter diferente a este, comienza a almacenar los siguientes caracteres leídos uno tras otro hasta volver a leer el mismo carácter (\n). Finalmente devuelve la cadena de caracteres generada, cadena que representa el valor del tag RFID que ha sido leído por el sensor.

Módulo line_follower.py

En el módulo "line_follower" nos encontramos con una clase encargada de, una vez obtenidos los valores generados por el sensor de línea, actualizar el valor del ángulo de giro de las ruedas para que de esta manera el vehículo siempre siga la línea y no se salga de esta.

Dentro de este módulo podemos encontrar una de las funciones más importantes de la conducción autónoma, la cual se llama "follow_line". Como bien indica su nombre, esta función es la encargada de hacer que el vehículo se encuentre, en todo momento, sobre la línea que ha de seguir, para desplazarse de un punto a otro.

```python
def follow_line(self):
        while True:
            digital_list = self.sensors.read_digital_line(self.references)
            turning_direction = self.car.get_turning_direction()
            car_stopped = self.car.is_car_stopped()

            if turning_direction == "left":
                if digital_list[0] or digital_list[1]:
                    digital_list[2] = digital_list[3] = 0
                digital_list[4] = 0
            elif turning_direction == "right":
                if digital_list[3] or digital_list[4]:
                    digital_list[2] = digital_list[1] = 0
                digital_list[0] = 0
            elif turning_direction == "straight":
                digital_list[0] = digital_list[4] = 0
            elif turning_direction == "emergency_stop" or turning_direction == "stop":
                digital_list = [0, 0, 0, 0, 0]
            else:
                time.sleep(0.01)

            if digital_list[0] == 1:
                self.car.set_speed_level(5)
                self.car.set_angle(60)
            if digital_list[1] == 1:
                self.car.set_speed_level(7)
                self.car.set_angle(75)
            if digital_list[2] == 1:
                self.car.set_speed_level(9)
                self.car.set_angle(90)
            if digital_list[3] == 1:
                self.car.set_speed_level(7)
                self.car.set_angle(105)
            if digital_list[4] == 1:
                self.car.set_speed_level(5)
                self.car.set_angle(120)
```

Figura 27: Implementación de la función follow_line

Como podemos ver, la función está dividida en tres partes.

En la primera de ellas, se declaran tres variables diferentes, las cuales son: “digital_list”, “turning_direction” y “car_stopped”.

La variable “digital_list”, es una variable donde se guarda la lista de valores de lectura del sensor de línea. Como hemos explicado anteriormente es una lista con cinco posiciones y en cada posición puede haber el valor 0 o 1, donde el valor 0 indica que un sensor del lector de línea ha leído que no hay línea, y 1 indica que un sensor del lector de línea ha leído que sí que hay línea.

La variable “turning_direction” almacena un valor guardado dentro de la clase “CarMovement”, explicada anteriormente, que indica si el vehículo esta realizando algún tipo de acción en una intersección, o si este está siguiendo la línea de forma normal. Las diferentes acciones que puede realizar un vehículo en una intersección pueden ser las siguientes: “turn_left”, “turn_right”, “go_straight”, “stop” y “emergency_stop”.

La segunda parte de esta función es una comprobación de la variable "turning_direction". Nos encontramos con cuatro posibles resultados de esta comprobación, que son los siguientes:

- Un primer caso donde la variable "turning_direction" tiene el valor "turn_left". En este caso se está indicando que el vehículo se encuentra en una intersección, y que este va a realizar un giro hacia la izquierda. En este caso, para que el vehículo pueda realizar este giro correctamente, se anula el sensor de línea situado más hacia la derecha para no crear ningún conflicto a la hora de actualizar el ángulo de giro de las ruedas.
- Un segundo caso donde la variable "turning_direction" tiene el valor "turn_right". Este caso es el contrario al anterior. En este caso el vehículo se encuentra en una intersección y este va a girar hacia la derecha. De la misma manera se anula el sensor de línea situado más hacia la derecha para evitar cualquier tipo de conflicto.
- Un tercer caso donde la variable "turning_direction" tiene el valor "go_straight". En este caso, el vehículo se encuentra en una intersección, pero no va a realizar ningún giro, sino que va a seguir recto. En este caso, se anulan los sensores de los extremos izquierdo y derecho, para que de esta manera el vehículo no realice ningún giro brusco y pueda seguir la línea correctamente.
- Un cuarto caso, donde la variable "turning_direction" tiene como valor "stop" o "emergency_stop". En este último caso se ponen a 0 las cinco posiciones del vector "digital_list", para que el ángulo de giro de las ruedas no sea actualizado posteriormente.

En la tercera parte de la función se comprueban las cinco posiciones de la lista "digital_list" resultante después de haber realizado las modificaciones anteriores sobre esta. En este caso nos encontramos con las siguientes posibles situaciones:

- digital_list[0] == 1: Este caso indica que el sensor de línea situado más hacia la izquierda ha detectado que debajo suyo hay línea. En este caso, y al ser el sensor más alejado hacia la izquierda, debemos de actualizar el ángulo de giro de las ruedas al valor mínimo de giro posible. Este valor lo podemos encontrar dentro de la clase "CarMovement", que tal y como hemos explicado anteriormente guarda los valores mínimos y máximos, entre otros, de los ángulos de giro de las ruedas. El ángulo de giro de las ruedas asignado en este caso es de 60º.

- digital_list[1] == 1: En este caso, el sensor de línea situado en la segunda posición comenzando por la izquierda ha detectado que debajo suyo hay línea. En este caso, seguimos teniendo la necesidad de girar hacia la izquierda, pero en este caso, no de una manera tan exagerada. El ángulo de giro de las ruedas asignado en este caso es de 75º.
- digital_list[2] == 1: En este caso, el sensor que ha detectado que se encuentra sobre la línea es el sensor central. El ángulo de giro de las ruedas asignado en este caso es de 90º, ya que lo que queremos es que el vehículo siga circulando hacia delante sin realizar ningún tipo de giro.
- digital_list[3] == 1: Este caso es el opuesto al segundo. Se ha detectado que la línea se encuentra debajo del segundo sensor de línea comenzando por la derecha, por lo tanto, el vehículo ha de girar hacia ese lado. En este caso el ángulo de giro de las ruedas asignado es de 105º.
- digital_list[4] == 1: El último caso es el opuesto al primero. En este caso, se ha detectado que la línea se encuentra situada en el sensor situado más hacia la derecha, por lo tanto, se debe de girar hacia ese lado, con un ángulo mayor al del caso anterior. En este caso el ángulo de giro de las ruedas asignado es de 120º.

Para poder entender un poco mejor cómo funciona la asignación de los ángulos de giro de las ruedas, a continuación, se puede observar un diagrama con los diferentes ángulos nombrados en los casos anteriores:

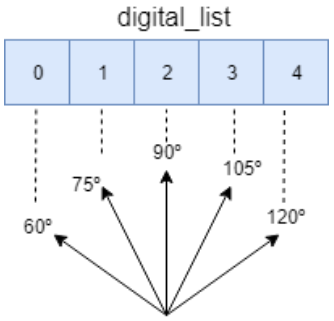

Figura 28: Relación entre "digital_list" y los ángulos de giros correspondiente

Módulo decision_maker.py

Como último módulo de la conducción autónoma tenemos el módulo “decision_maker.py”. Este es el módulo más importante, ya que dispone de diferentes funciones encargadas de tomar decisiones, dependiendo de una serie de datos obtenidos.

Las funciones utilizadas para tomar posibles decisiones son las siguientes:

- check_final
- check_next_rfid
- check_traffic_lights
- check_distance
- check_route
- check_streetlights

A continuación, se explican detalladamente cada una de las funciones listadas, para entender cómo y porque se toman las diferentes decisiones de conducción, dependiendo de un conjunto de datos a tener en cuenta.

Función check_final

La función check_final es la encargada de comprobar si el vehículo ha llegado al destino solicitado. En caso de que el vehículo haya llegado a su destino la función se encarga de detenerlo.

Es una función básica, pero a la vez necesaria ya que es la encargada de detener el programa principal una vez se ha cumplido la condición anterior y de esta manera se puede notificar cuando la ejecución del servicio de conducción solicitado ha sido finalizada.

Función check_next_rfid

Esta función es la encargada de comprobar si la siguiente posición a la cual ha de moverse un vehículo está libre o no. En caso de estar libre, el vehículo sería libre de moverse y por el contrario no se le permitiría.

Para realizar esta comprobación es necesario disponer de un sistema de comunicación con el agent encargado de gestionar el tráfico de la ciudad. En nuestro caso, el sistema de comunicación es mediante la creación de un cluster. Tal y como se ha mencionado anteriormente, el cluster está formado por diferentes agents, y tiene como finalidad poder intercambiar mensajes y datos con un tiempo de comunicación muy reducido, casi inmediato.

Función check_traffic_lights

La función check_traffic_lights tiene dos funcionalidades diferentes, en función de que servicio de conducción se esté ejecutando.

En caso de que el servicio de conducción sea un servicio de emergencia, es decir, uno en el que el vehículo sea una ambulancia, por ejemplo, o un camión de bomberos, la función realizará ciertas acciones. Por el contrario, si el servicio no es de emergencia, como puede ser el servicio de basuras, o un servicio de conducción entre dos puntos, la función realizará otras acciones.

En el caso de que el servicio en ejecución sea de emergencias, la función se encarga de, una vez el vehículo se aproxima a un semáforo, se solicita al agent encargado de los semáforos que ponga este en estado de emergencia. Esta comunicación tiene que ser muy rápida, casi inmediata, por lo que como hemos visto anteriormente, también se utiliza el cluster creado a la hora de solicitar un servicio para intercambiar estos mensajes.

En el caso de que el servicio en ejecución no sea de emergencias, una vez el vehículo se aproxima a un semáforo, mediante el cluster mencionado anteriormente, se intercambian una serie de mensajes con el agent encargado de los semáforos, solicitando el estado actual del semáforo. En nuestro proyecto, los posibles estados de un semáforo son: “green”, “red” y “yellow”. Una vez obtenido el estado del semáforo, si este se encuentra en “red” o “yellow”, la función se encarga de detener el vehículo. Si por el contrario se obtiene como estado “green”, se pone el vehículo en circulación.

Función check_distance

Esta función es la encargada de, mediante la información generada por el sensor de ultrasonido, detener el vehículo si la distancia de cualquier objeto que se encuentre delante del vehículo es menor o igual a una serie de valores predefinidos en este módulo.

Los valores predefinidos son los siguientes:

```
STOP_DISTANCES = {
        0: 20,
        1: 10,
        2: 12,
        3: 13,
        4: 14,
        5: 15,
        6: 16,
        7: 17,
        8: 18,
        9: 19,
        10: 20
```

Figura 29: Relación entre la velocidad y la distancia

Como podemos ver, disponemos de un diccionario de valores llamado "STOP_DISTANCES". Este diccionario tiene el formato de {clave: valor}, donde la clave es el primer valor definido y el valor el segundo.

En nuestro diccionario, la clave representa el nivel de velocidad actual del vehículo, teniendo como nivel de velocidad mínima el 0, es decir, el vehículo está detenido, y como nivel de velocidad máxima el 10.

Los diferentes valores que encontramos, cada uno de ellos relacionado con una clave, indican la distancia en centímetros, a la cual el vehículo se tiene que detener. Observando el diccionario anterior, si un vehículo está circulando con un nivel de velocidad 5, este debería de detenerse si, mediante el sensor de ultrasonido se obtiene una distancia de 15 centímetros o menos respecto a un objeto que tenga situado delante suyo.
Gracias a este diccionario se pueden definir una serie de relaciones velocidad-distancia, mediante las cuales, se podrá detener el vehículo de la forma más segura posible dependiendo de la velocidad de este.

Función check_route
La principal función de este método es comprobar, en todo momento, si el vehículo ha de realizar alguna acción en algún punto específico de la ruta que tiene asignada.

Esta comprobación se hace utilizado una variable de tipo diccionario, llamada "route_actions". Este diccionario contiene un conjunto de valores RFID, cada uno de ellos relacionado con un tipo de acción. Las acciones que se pueden encontrar en este diccionario son las mismas que son utilizadas en la clase "LineFollower". Estas son: "go_straight", "turn_left", "turn_right", "stop" y "emergency_stop".

Una vez el vehículo pasa por encima de algún RFID que se encuentra dentro del diccionario "route_actions", se comprueba cual es la acción relacionada con ese RFID y esta acción es asignada a la variable "turning_direction" del módulo "CarMovement", también visto anteriormente, mediante diferentes funciones que se encuentran dentro de este último módulo.

Si el RFID leído no forma parte de ningún RFID del diccionario, se asigna el valor " ", es decir, una palabra vacía, a la variable "turning_direction", para indicar que el vehículo no está realizando ninguna acción específica.

Gracias a esta función, que actualiza la variable "turning_direction", la clase

"LineFollower" puede actuar correctamente sobre los valores leídos por el sensor de línea para de esta manera hacer girar el vehículo hacia donde es deseado en lugares de la ciudad donde hay intersecciones.

Función check_streetlights

Esta última función es la encargada de encender las farolas de la ciudad conforme un vehículo se aproxima a estas. Para hacer esto, también se utiliza el cluster nombrado anteriormente, ya que nos interesa que el intercambio de mensajes sea lo más rápido posible.

Una vez el vehículo se aproxima a una farola, se envía un mensaje al agent encargado de gestionar estas, para que encienda las diferentes farolas correspondientes.

De esta manera, nos aseguramos qué las farolas solo se encuentren encendidas cuando realmente sea necesario, es decir, cuando haya algún vehículo cercano a estas.

7.3 AGENT

A continuación, una vez visto el diseño de cada uno de los módulos que forman un agent, se detalla la implementación de cada uno de estos.

7.3.1 Módulo API

Para implementar el módulo de comunicación entre agents llamado API, se ha utilizado **CherryPy** [30].

CherryPy es un entorno de trabajo web orientado a objetos y basado en el lenguaje de programación Python. Este permite desarrollar aplicaciones web de la misma manera en la que se crearía cualquier otro programa basado en Python.

De esta manera, se consigue como resultado un código fuente más reducido, así como el desarrollo de este en un periodo de tiempo menor.

El módulo API se puede dividir en las siguientes partes:

- Inicialización del módulo.
- Configuración de ciertos parámetros del módulo.
- Implementación de las funciones REST.
- Implementación de las funciones propias.

7.3.1.1 Inicialización del módulo

En la parte de inicialización del módulo se declaran un conjunto de variables que serán utilizadas posteriormente en el resto de las funciones de este. Estas variables son las siguientes:

- **host:** Dirección IP utilizada para acceder a la API.
- **port:** Puerto utilizado para acceder a la API.
- **leader_url:** Dirección URL del leader.
- **agent:** Información del agent sobre el que se va a ejecutar la API.

7.3.1.2 Configuración de ciertos parámetros del módulo

La configuración del módulo se ha realizado mediante la programación de las siguientes funciones:

Función OPTIONS

En este método se configuran las cabeceras del módulo CherryPy. En nuestro caso las diferentes cabeceras sobre las cuales se ha realizado algún tipo de configuración son las siguientes:

- **Access-Control-Allow-Headers:** Esta cabecera es utilizada en las respuestas a las peticiones para indicar que tipo de cabeceras pueden ser utilizadas en la propia petición. Hay que tener en cuenta, que hay cabeceras que siempre son permitidas, las cuales no hay que indicar en este campo.
- **Access-Control-Allow-Origin:** Esta cabecera sirve para indicar desde que orígenes se pueden realizar peticiones a la API.
- **Access-Control-Allow-Methods:** Indica que métodos pueden ser utilizados como peticiones a la API. Los diferentes métodos permitidos en este campo son los siguientes: GET, PUT, POST, DELETE y HEAD.
- **Content-Type:** Sirve para indicar que tipo de datos se permiten tanto en el cuerpo de las peticiones como en el de las respuestas.

A continuación, se puede ver la implementación de esta función, donde se muestran los diferentes valores que se han asignado a las cabeceras anteriormente nombradas:

```
def OPTIONS(self, a=None, b=None):
        cherrypy.response.headers['Access-Control-Allow-Headers'] = 'Access-Control-Allow-Origin, Content-Type'
        cherrypy.response.headers['Access-Control-Allow-Origin'] = '*'
        cherrypy.response.headers['Content-Type'] = '*'
        possible_methods = ('PUT', 'DELETE')
        methods = [http_method for http_method in possible_methods
                   if hasattr(self, http_method)]
        cherrypy.response.headers['Access-Control-Allow-Methods'] = ','.join(methods)
```

Figura 30: Implementación de la función OPTIONS

Función start

Esta función es llamada una vez se ha creado un objeto API. Dentro de esta encontramos una variable de configuración llamada "conf", en la cual se definen diferentes parámetros, que hacen posible que toda la configuración definida en la función anterior se pueda aplicar correctamente.

A continuación de la declaración de esta variable, se configuran el host y el puerto que serán utilizados para servir el API y de esta manera, poder acceder des de fuera a esta. Estos se definen en las siguientes variables:

- **cherrypy.server.socket_host:** Variable utilizada para asignar un host a la API.
- **cherrypy.server.socket_port:** Variable utilizada para asignar un puerto a la API.

Finalmente, se llama a la función **cherrypy.quickstart()**, la cual utilizando unos parámetros de entrada ejecuta la API, para que de esta manera ya se pueda acceder al agent desde otros elementos de la misma red y este pueda también realizar peticiones al exterior.

La implementación de esta función es la siguiente:

```
def start(self, silent_access=False):
    conf = {
        '/': {
            'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
            'tools.sessions.on': True,
            'tools.response_headers.on': True,
            'tools.response_headers.headers': [
                ('Access-Control-Allow-Origin', '*'),
                ('Content-Type', '*')
            ]
        }
    }
    cherrypy.config.update({'log.screen': not silent_access})
    cherrypy.server.socket_host = self.host
    cherrypy.server.socket_port = self.port
    cherrypy.quickstart(API(self.agent), '/', conf)
```

Figura 31: Implementación de la función start

7.3.1.3 Implementación de las funciones REST

Las funciones REST son aquellas que nos proporciona CherryPy por defecto. Estas pueden ser accedidas gracias a la configuración realizada en la función start, específicamente en la línea siguiente:
request.dispatch: cherrypy.dispatch.MethodDispatcher()

Gracias a esta configuración, podemos acceder a las funciones REST, que son las siguientes:

GET
Esta función es llamada cada vez que alguien realiza una petición GET sobre la API. Dentro de esta, dependiendo del valor que nos llegue en el parámetro **obj**, explicado anteriormente, se ejecutará una función u otra.
La implementación de esta función es la siguiente:

```
def GET(self, obj=None, id=None):
        if obj == "agent":
            return self.return_data(self.get_agents(id))
        elif obj == "service":
            return self.return_data(self.get_service(id))
        elif obj == "alive":
            return "Alive".encode()
```

Figura 32: Implementación de la función GET

Como se puede observar, dependiendo del valor del parámetro **obj**, se llamará a función **get_agents**, encargada de obtener la información de un agent o a la función **get_service**, encargada de obtener la información de un servicio. También es posible que simplemente se devuelva la palabra “Alive”, para indicar que el agent al que se le ha realizado la petición se encuentra activo.
Para poder realizarle dichas peticiones al agent se ha definido el siguiente formato:

http://AGENT_IP:8000/obj/id

En dicha petición tenemos tres atributos:

- **AGENT_IP**: Aquí indicamos la IP del agent a hacer la petición.
- **obj**: Utilizado para definir sobre que recurso se está realizando la petición GET. Los diferentes recursos sobre los cuales se pueden realizar peticiones GET son los agents, los servicios y para saber si un agent está activo
- id: Utilizado para actuar sobre un recurso en específico. En el caso de que este parámetro no estuviera presente en la URL de la petición realizada, se actuaría sobre todos los recursos

POST

Esta función es llamada cada vez que alguien realiza una petición POST a la API. En este caso tal y como se ha explicado anteriormente, dependiendo del parámetro **action** se ejecutará una función u otra.

A continuación, se muestra la implementación de esta función:

```
@cherrypy.tools.json_in()
    def POST(self, action=None):
        info = cherrypy.request.json
        result = ""
        if action == "register_agent":
            result = self.register_agent(info)
        elif action == "request_service":
            result = self.request_service(info)
        elif action == "execute_service":
            result = self.execute_service(info)
        elif action == "response_service":
            result = self.response_service(info)
        return self.return_data(result)
```

Figura 33: Implementación de la función POST

Como podemos ver, dependiendo del valor obtenido en el parámetro **action**, se ejecutará una función u otra. Finalmente, una vez obtenido el valor resultante de la función que se haya ejecutado, este será devuelto a quien haya realizado la petición.

A parte, podemos observar como encima del nombre de la función se encuentra una línea con el siguiente contenido: **cherrypy.tools.json_in()**

Esta línea es utilizada para que en esta función se pueda recuperar información proporcionada en el cuerpo de la petición. Esta información puede ser recuperada utilizando la siguiente variable proporcionada por CherryPy:
info = cherrypy.request.json

En este caso, la información proporcionada puede ser información de un agent o de un servicio.

Para poder realizarle dichas peticiones al agent se ha definido el siguiente formato:

http://AGENT_IP:8000/action

Este parámetro sirve para identificar qué tipo de acción se quiere realizar una vez recibida la petición. Los diferentes valores posibles para el parámetro **action**, son los siguientes:

- **register_agent:** Si se recibe una petición POST con este parámetro, se está solicitando el registro de un nuevo agent dentro de la topología de la red.

- **request_service:** Si se recibe una petición POST con este parámetro, se está solicitando un servicio al agent que ha recibido esta.
- **execute_service:** Si se recibe una petición POST con este parámetro, se está solicitando la ejecución de un servicio al agent que ha recibido esta.
- **response_service:** Si se recibe una petición POST con este parámetro, se está recibiendo el resultado de la ejecución de un servicio solicitado por el agent que ha recibido esta petición.

PUT

Esta función es llamada cada vez que llegue una petición PUT a la API. Para esta función, tal y como se ha explicado antes, solo se puede realizar una acción, que es la de actualizar la información de un agent en la base de datos topológica.

La implementación de esta función es la siguiente:

```
@cherrypy.tools.json_in()
    def PUT(self, action=None):
        info = cherrypy.request.json
        if action == "update_agent":
            self.update_agent(info)
```

Figura 34: Implementación de la función PUT

De la misma manera que en la función anterior, en esta también podemos recuperar información proporcionada en la propia petición. Esta información son los datos del agent que queremos actualizar.

Para poder realizarle dichas peticiones al agent se ha definido el siguiente formato:

http://AGENT_IP:8000/action

Las peticiones PUT, de la misma manera que con las peticiones POST, requieren el parámetro **action**, utilizado para identificar qué tipo de acción se quiere realizar.

En nuestra API, solo hay un tipo de petición PUT:

- **update_agent:** Esta acción sirve para actualizar la información de un agent específico en la base de datos topológica.

DELETE

Finalmente, esta función será ejecutada cada vez que alguien realice una petición DELETE a la API. Como en la función anterior, en este caso solo se puede realizar una acción, que es la eliminación de los datos de un agent de la base de datos topológica.

La implementación de esta función es la siguiente:

```
def DELETE(self, action=None):
        info = cherrypy.request.json
        if action == "delete_agent":
            self.delete_agent(info)
```

Figura 35: Implementación de la función DELETE

Para poder realizarle dichas peticiones al agent se ha definido el siguiente formato:

http://AGENT_IP:8000/action

La acción que se puede realizar con las peticiones DELETE es la siguiente:

- **delete_agent:** Esta acción sirve para eliminar la información de un agent específico de la base de datos topológica.

7.3.1.4 Implementación de las funciones propias

En cuanto a las funciones propias, podemos encontrar las diferentes funciones que han sido llamadas dentro de las funciones REST. Son funciones auxiliares que permiten obtener, crear, modificar y eliminar datos de la base de datos topológica.

Para poder realizar estos accesos a la base de datos topológica se ha utilizado una librería llamada “PyMongo”. Esta librería contiene diferentes herramientas que permiten trabajar fácilmente con bases de datos MongoDB.

La siguiente línea de código nos permite conectarnos a una base de datos MongoDB:

```
client = pymongo.MongoClient(self.agent.node_info["ipDB"], self.agent.node_info["portDB"])
```

Figura 36: Conexión a una base de datos MongoDB

Vemos como mediante la variable **self.agent.node_info[“ipDB”]** podemos obtener una IP y mediante la variable **self.agent.node_info[“portDB”]** obtenemos un puerto, los cuales son utilizados como parámetros para establecer la conexión remota con la base de datos.

Para obtener acceso a la colección de nodos de la topología y al catálogo de servicios de la ciudad inteligente lo podemos hacer de la siguiente manera:

```
self.agent_collection = client.globalDB.nodes
self.service_catalog = client.globalDB.service_catalog
```

Figura 37: Acceso a las colecciones de la base de datos

La primera línea guarda en la variable **self.agent_collection** la colección que contiene los nodos de la base de datos topológica. La segunda línea guarda en la variable **self.service_catalog** el catálogo de servicios de la ciudad inteligente.

7.3.2 Módulo Runtime

La funcionalidad principal del módulo Runtime es la de ejecutar servicios. Dentro de este módulo podemos destacar cuatro funciones principales, que son las siguientes:

- execute_service
- prepare_params
- get_remote_file
- execute_code

Función execute_service

Esta es la función principal del módulo. Se encarga de, dado en servicio, realizar todas las comprobaciones y acciones necesarias para poder ejecutar este. Este proceso es el siguiente:

1. Primero de todo, se comprueba si el agent ya tiene el código del servicio solicitado. En caso de no tenerlo, este se lo descarga ejecutando la función **get_code**, que se explicará posteriormente.
2. Una vez el agent dispone del código principal del servicio, se comprueba si este tiene dependencias a otros ficheros, por ejemplo, módulos necesarios para poder ejecutar el programa principal, y si tiene códigos dependientes estos se descargan de la misma manera.
3. En caso de que el servicio a ejecutar no sea permanente, es decir, no tenga fin, se realizan las siguientes acciones:
    a. Se ejecuta la función **execute_code** y una vez haya acabado la ejecución se guarda el resultado de esta en una variable.
    b. Si el resultado de la ejecución no tiene ningún error se devuelve como resultado un diccionario donde el valor de la clave “status” es “success”.

   c. Si el resultado de la ejecución presenta algún error, se devuelve como resultado un diccionario donde el valor de la clave “status” es “error”.

4. En caso de que el servicio a ejecutar no tenga fin, es decir, que sea permanente, se realizan las siguientes acciones:
   a. Se recuperan todos los parámetros necesarios para ejecutar el servicio.
   b. Se crea un Thread que llama a la función **execute_code** para que el servicio sea ejecutado pero el agent pueda seguir recibiendo nuevas solicitudes de ejecución de servicios.
   c. Se devuelve como resultado del servicio “success”, indicando que este ha podido ser ejecutado. Esta devolución de estado es necesaria para poder ejecutar las dependencias de un servicio permanente.

Función prepare_params

Esta función se encarga de convertir todos los parámetros pasados al servicio en el tipo de variable correspondiente, es decir, lista, diccionario, numero, string, etc.

Gracias a esta conversión se puede ejecutar el código con todos los parámetros necesarios correctamente.

Función get_code

Con esta función se puede obtener el código de un servicio a partir de una URL determinada. El formato de esta URL es el siguiente:
http://download_host:download_port/download/code

Se puede observar como para formar esta URL se requieren tres variables, que son las siguientes:

- **download_host:** IP del host sobre la que se tiene que solicitar la descarga del código del servicio a ejecutar.
- **download_port:** Puerto del host necesario para que la solicitud de descarga pueda ser enviada correctamente.
- **code:** Nombre del fichero que contiene el código del servicio a ejecutar.

Función execute_code

Función encargada de, dada la versión de Python necesaria, el código del servicio y los parámetros, ejecutar el código del servicio proporcionado. La función utilizada para ejecutar los servicios es la siguiente:
output = subprocess.getoutput(python_version+"./codes/"+code+" "+params)

Podemos ver como se ejecuta la función **subprocess.getoutput()**, a la cual se le pasa como parámetro la versión de python, el directorio donde se encuentran los códigos de los servicios, el nombre del fichero a ejecutar y los parámetros, en caso de que este tenga. De esta manera se puede ejecutar cualquier servicio que se solicite al agent.

7.3.3 Módulo Service Execution

Tal y como hemos explicado anteriormente, el módulo de Service Execution se encarga de gestionar los servicios que son solicitados al agent. Este, en caso de poder, comprueba los diferentes atributos del servicio solicitado con tal de realizar las acciones pertinentes.

A continuación, se detalla el comportamiento de las funciones más importantes de este módulo:

Función request_service

Esta función es llamada cada vez que el agent recibe una solicitud de servicio. El comportamiento de esta función es el siguiente:

1. El agent que recibe la solicitud tiene como rol asignado “agent”:
    a. El agent añade su IP a los datos del servicio y solicita este a su leader.
2. El agent que recibe la solicitud tiene como rol asignado “leader”:
    a. Si el servicio tiene dependencias, el leader recupera de la base de datos la información de cada servicio dependiente y se lo solicita a si mismo.
    b. Comprueba si quien ha solicitado el servicio puede ejecutarlo. Si es el caso, le solicita le ejecución de este.
    c. Si no puede ejecutar el servicio quien lo ha solicitado, el leader comprueba si el mismo puede solicitar, si puede se manda un mensaje de ejecución a si mismo.
    d. Si el propio leader tampoco puede ejecutar el servicio busca alguno de sus agents que si que pueda. En caso de encontrar uno le solicita le ejecución del servicio a este, si por el contrario no encuentra ninguno, le solicita el servicio a su leader, es decir al cloud agent.
3. El agent que recibe la solicitud tiene como rol asignado “cloud_leader”:
    a. El cloud agent se encarga de buscar alguien de la topología que pueda ejecutar el servicio.

   b. Si encuentra a alguien:
      i. Si es agent, le solicita al leader del agent el servicio, para que este gestione la solicitud y finalmente esta llegue al agent destinado.
      ii. Si es leader, le solicita directamente la ejecución del servicio a este.
4. Finalmente, si nadie puede ejecutar el servicio en el momento en que este es solicitado, se devuelve como resultado de la ejecución del servicio "unattended", indicando que este no ha podido ser atendido.

Función delegate_service

Esta función es la encargada de enviar una solicitud de ejecución de servicio a la API del agent que ha de ejecutar el este. También se encarga de devolver como resultado de la ejecución "unattended" en caso de no poder ejecutar todas las dependencias de un servicio solicitado.

Función find_agent_to_execute

El objetivo de esta función es, dadas las características de un servicio solicitado, encontrar una agent activo que pueda ejecutarlo.

Para encontrar este agent se hace una comparación de los IoT que tiene un agent y los IoT que tiene el servicio solicitado.

Si el agent tiene todos los IoT que tiene el servicio, este puede ejecutarlo. Si por el contrario, el agent no tiene algun IoT que el servicio sí que tiene, ese agent no es válido para ejecutar el servicio solicitado.

7.3.4 Módulo Topology Resource Management

Para cumplir con las funcionalidades básicas que tiene este módulo, explicadas anteriormente, disponemos de dos funciones principales que nos ayudan a mantener tanto la estructura topológica como la información de los nodos que forman parte de esta en tiempo real. Estas funciones son las siguientes:

Función check_alive_agents

Esta función es la encargada de comprobar, para cada uno de los agents que han sido registrados a un leader, si estos siguen activos o no. Esta función es necesaria para, una vez hecha esta comprobación, actualizar el estado de los agents en cuestión en la base de datos topológica.

Función change_agent_status

Esta función tiene como objetivo cambiar el estado de un agent dentro de la base de datos topológica. Este cambio de estado permite posteriormente saber sobre que agents activos se pueden realizar peticiones de servicio y sobre que agents no activos, no se deben de realizar estas peticiones.

7.4 SERVICIO

Una vez ya se ha decidido el formato que han de tener los servicios, como hemos visto en el apartado “5.2.4 Servicios”, podemos pasar a implementar un conjunto de estos. En este proyecto los diferentes servicios que han sido programados son los siguientes:

Servicio de emergencia

La funcionalidad de este servicio es la de, mediante una conducción autónoma, enviar una ambulancia a la posición indicada donde ha habido un accidente. La ambulancia seleccionada será la que se encuentre disponible y más cercana al lugar del accidente, para de esta manera tener el menor tiempo de respuesta posible.

Mientras la ambulancia realiza tanto la ruta de ida, como la de vuelta, esta interactuará con diferentes elementos de la ciudad, como pueden ser semáforos, farolas y otros vehículos. En cuanto a los semáforos, cuando esta se aproxime a uno, solicitará un cambio de estado para que este se ponga en estado de emergencia. En cuanto a las farolas, cuando la ambulancia se aproxime a una solicitará un cambio de estado sobre esta para que se encienda con la potencia correspondiente. Finalmente, con el resto de los vehículos, siempre se respetará una distancia de seguridad para evitar posibles accidentes.

Como parámetros necesarios para solicitar el servicio tenemos los siguientes:

- **Final:** Posición a la cual se ha de dirigir la ambulancia para solventar la emergencia.

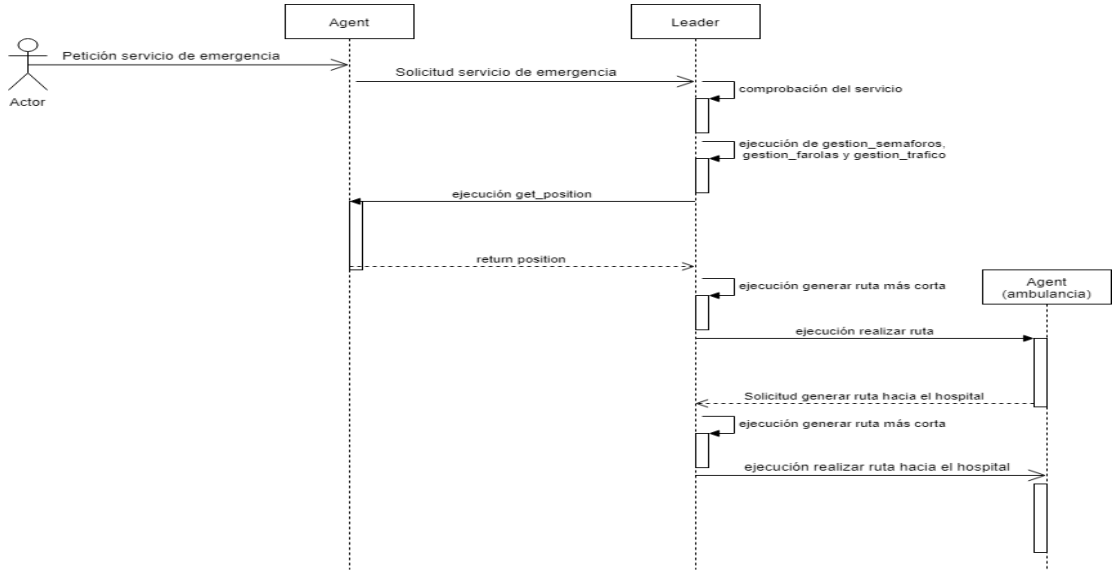

Figura 38: Diagrama de comunicación del servicio de emergencia

Servicio de basuras

El servicio de basuras es un servicio de conducción autónoma continua. La finalidad de este es la de simular un escenario donde tenemos un camión de la basura realizando, de forma indefinida, diferentes rutas entre los diferentes puntos de recogida de basura de la ciudad.

Para este servicio no se requiere ningún parámetro de entrada, ya que todos los datos necesarios para la ejecución de este se encuentran guardados en una base de datos, accesible por cualquier leader que puede proporcionarle estos al agent.

Mientras se esté realizando el servicio, el vehículo interactuará con los mismos elementos mencionados en el servicio anterior. Con los semáforos, cada vez que el vehículo se aproxime a uno, este solicitara su estado al leader gestor de los semáforos. Dependiendo del estado que obtenga como respuesta este se aturará o continuará la conducción. En cuanto a las farolas y los vehículos, realizará las mismas acciones nombradas en el servicio anterior.

El diagrama de comunicación resultante para este servicio es el siguiente:

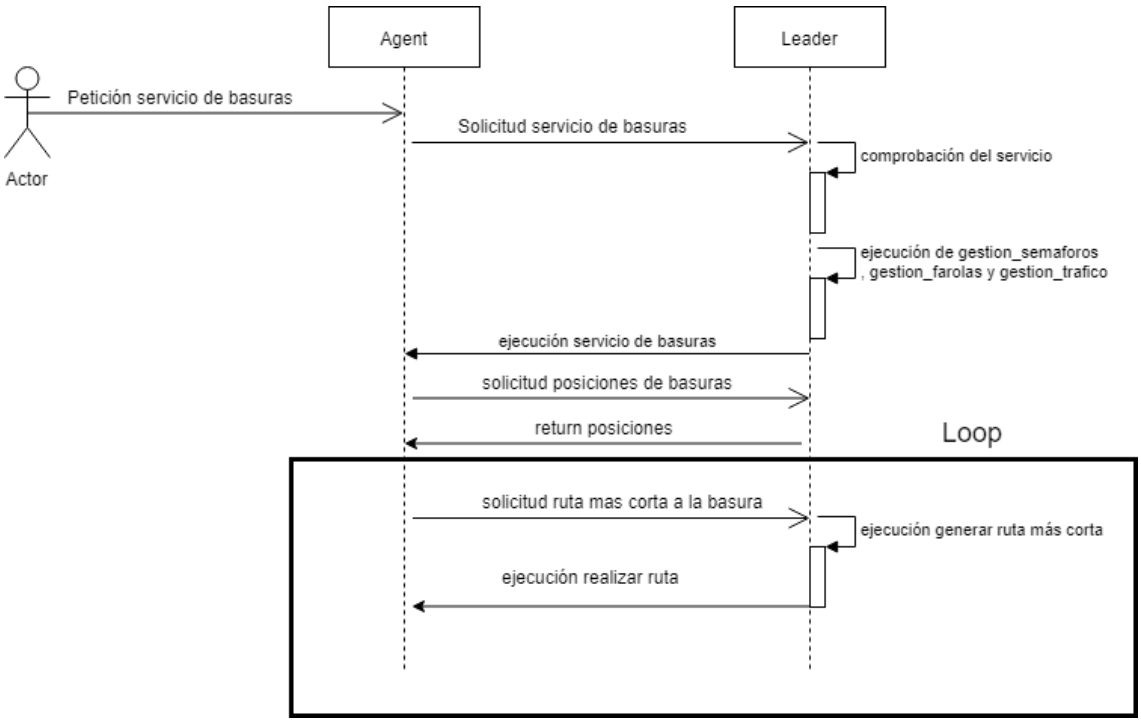

Figura 39: Diagrama de comunicación del servicio de basuras

Servicio de conducción hacia un punto

La finalidad de este servicio es la de hacer circular un vehículo des de su ubicación actual hacia otro punto en concreto de forma autónoma. Este servicio es utilizado en la resta de servicios de conducción autónoma, como, por ejemplo, los dos anteriores servicios.

Como parámetros de entrada necesarios para la ejecución de este servicio tenemos los siguientes:

- **Final:** Posición hacia la cual queremos hacer circular el vehículo.

Servicio de generación de rutas óptimas

Este servicio es el encargado de generar, dados dos puntos de la ciudad, la ruta óptima. Esta ruta es generada gracias a una base de datos de la cual disponen los leaders de la ciudad. Esta base de datos está basada en “Sqlite” [31].

Sqlite es una librería basada en el lenguaje de programación “C”, la cual implementa un motor de base de datos SQL, que tiene como características principales las siguientes: es pequeño, rápido, tiene una gran fiabilidad y dispone de un gran conjunto de funcionalidades.

En esta base de datos podemos encontrar 9 tablas diferentes, de las cuales 2 son necesarias para poder crear estas rutas óptimas.

La primera tabla se llama “rfid_connections”. Esta tabla almacena dos campos, uno llamado “current” y otro llamado “next”. Ambos campos son valores de tags RFID de la ciudad y la combinación de estos representa que desde el tag “current” hasta el tag “next” existe una comunicación directa, por lo que estos tags podrían ser utilizados para generar parte de una ruta.

La segunda tabla “rfid_turns” también almacena dos campos. El primer campo llamado “section”, tiene como valor la combinación de dos tags RFID. El segundo campo llamado “action”, tiene como valor las posibles acciones que se pueden realizar en una intersección. Los diferentes valores que encontramos en el campo “action” en esta base de datos son los siguientes: turn_left, turn_right y go_straight.

Una vez se genera una ruta, se obtiene como resultado una lista de RFID llamada “rfid_route”, que es la que deberá de seguir el vehículo para llegar al destino, y también se obtiene un diccionario llamado “rfid_actions”, que contiene, para cada intersección que se encuentre en la ruta, una acción a realizar.

Para calcular la lista “rfid_route”, se utiliza una función que, a partir de un grafo de conexiones, obtenido a partir de la tabla “rfid_connections” de la base de datos, busca el camino más corto entre dos puntos. De esta manera nos aseguramos de que la ruta que obtendremos siempre será la más corta.

A continuación, se puede ver la implementación de la función encargada de generar la ruta más corta para poder obtener la lista “rfid_route”:

```python
def shortest_path(graph, start, end, path=[]):
    if start in path:
        return None
    path = path + [start]
    if start == end:
        return path
    if not start in graph.keys():
        return None
    shortest = None
    for node in graph[start]:
        if node not in path:
            new_path = shortest_path(graph, node, end, path)
            if new_path:
                if not shortest or len(new_path) < len(shortest):
                    shortest = new_path
    return shortest
```

Figura 40: Implementación de la función shortest_path

Para calcular el diccionario “rfid_actions”, se utiliza una función que, a partir de la ruta anteriormente obtenida, y de los valores obtenidos de la tabla “rfid_turns” de la base de datos genera este.

A continuación, podemos observar la implementación de la función encargada de generar el diccionario de acciones:

```python
def car_route(route, end):
    route_actions = {}
    rfid_route = []
    for i in range(len(route)):
        rfid_route.append(card_id[route[i]])
        if(i+1 < len(route)):
            section = route[i] + route[i+1]
            if section in rfid_turns:
                route_actions[card_id[route[i]]] = rfid_turns[section]
                if route[i] in rfid_intersections:
                    rfid_route.append(card_id[rfid_intersections[route[i]]])
    route_actions[card_id[end]] = "stop"
    return route_actions, rfid_route
```

Figura 41: Implementación de la función car_route

Servicio de calibración del motor del vehículo

La finalidad de este servicio es comprobar que el motor y las ruedas un vehículo funcionan correctamente.

Para realizar esto, primero de todo se giran las ruedas del vehículo hacia la izquierda hasta llegar al ángulo mínimo de giro. Después las ruedas vuelven a colocarse en su estado inicial y finalmente se giran hacia la derecha hasta llegar al ángulo máximo de giro.

Una vez finalizado este proceso de calibración de las ruedas, estas se ponen rectas y el vehículo realiza un movimiento hacia delante y hacia atrás, para comprobar el buen funcionamiento del motor.

Este servicio no necesita ningún parámetro de entrada para poder ser solicitado.

Servicio de calibración del sensor de ultrasonido del vehículo

Este servicio tiene como finalidad comprobar el correcto funcionamiento del sensor de ultrasonido instalado en un vehículo.

Para comprobar esto, una vez ejecutado el servicio, el vehículo espera encontrarse un obstáculo delante suyo. Una vez detectado este mediante la información recogida por el sensor de ultrasonido, el vehículo se mueve hacia atrás para que el obstáculo no impacte contra él y se vuelve a detener. De esta manera se puede comprobar el correcto funcionamiento del sensor de ultrasonido de un vehículo.

Este servicio no necesita ningún parámetro de entrada para poder ser ejecutado.

Servicio de calibración del sensor de línea del vehículo

El objetivo de este servicio es poder calibrar el sensor de línea del vehículo. Esta calibración servirá para que el vehículo pueda seguir las líneas de la ciudad y de esta manera poder circular de forma autónoma.

La calibración del sensor se realiza de la siguiente manera:

1. Las ruedas del vehículo giran hacia el ángulo mínimo de giro, es decir, lo máximo

posible hacia la izquierda.

2. En este momento el sensor de línea se pone a leer valores del color negro durante 5 segundos.
3. Una vez pasan estos 5 segundos, el programa de calibración guarda en una variable una media de todos los valores leídos.
4. Las ruedas del vehículo giran hacia la derecha hasta llegar al ángulo máximo de giro.
5. En este momento el sensor de línea se pone a leer valores del color blanco, es decir, la línea que sigue el vehículo cuando está conduciendo de forma autónoma. Esta lectura también dura 5 segundos.
6. Una vez pasan estos 5 segundos, el programa guarda en una nueva variable la media de todos los valores leídos.
7. Finalmente, el programa escribe en un fichero llamado “calibration.config” los valores leídos, para que cuando algún servicio de conducción autónoma sea ejecutado, el vehículo sepa que valores ha de tomar como referencia y de esta manera pueda seguir correctamente las líneas.

Este servicio no requiere ningún parámetro de entrada para poder ser solicitado.

7.5 NUEVAS FUNCIONALIDADES DEL FRONT-END

Tal y como se ha explicado anteriormente en el apartado de diseño de las nuevas funcionalidades del Front-End, se han diseñado tres nuevas pantallas para la visualización de diferentes datos, las cuales son:

- Pantalla de visualización de agents
- Pantalla de visualización del catálogo de servicios de la ciudad inteligente
- Pantalla de visualización de la información de un agent
- Pantalla para añadir servicios al catálogo de servicios de la ciudad inteligente

La implementación de cada una de estas pantallas se ha hecho siguiendo el formato del TFG nombrado “Panel Front-End para el control de una Smart City”, para que de esta manera no haya ningún problema con el resto de las pantallas, ya implementadas, del Front-End.

A continuación, se explica de forma detallada como han sido implementadas cada una

de estas pantallas, así como el conjunto de características que estas tienen.

Pantalla de visualización de agents

Como hemos explicado anteriormente, esta pantalla permite visualizar todos los nodos que hayan sido registrados dentro de la topología de la red.

Para la implementación de esta pantalla se han creado dos ficheros diferentes, uno basado en el lenguaje de programación "Jade" [32] y otro basado en el lenguaje de programación "Javascript" [33]. Estos ficheros son los siguientes: "calibration.pug" y "calibration.js".

Dentro del fichero "calibration.pug" podemos encontrar diferentes apartados, entre los más importantes los siguientes:

- **block_content:** En este bloque se define cual será el contenido de la página web que se va a mostrar.
- **append_scripts:** En este bloque se definen una serie de scripts que serán llamados una vez sea cargada la página web en un navegador.

Dentro del apartado "block_content", se pueden diferenciar dos partes diferentes. Una primera parte donde se definen una serie de características de diseño de la página web, para que estas tengan un aspecto similar a las que ya habían sido implementadas y una segunda parte donde se define el conjunto de elementos que serán mostrados en la página web.

A continuación, se puede ver una imagen donde se muestra un fragmento de código, donde se declaran los diferentes elementos que tendrá la página web:

```
div.page-body
        h3= Filtro
        details(open)
            summary Filtro
            form(id="filtro", style="display:inline")
                div.input(style="display: inline")
                    label NodeID
                    input(type="number", name="nodeID", id="nodeID", style="width: 102px")
                div.input(style="display: inline")
                    label Device
                    input(type="text", name="device", id="device", style = "width: 102px")
                div.input(style="display: inline")
                    label Role
                    select(name="role", id="role")
                        option(value = "", selected='selected')
                        option(value = "agent") Agent
                        option(value = "cloudagent") Cloud Agent
                        option(value = "leader") Leader
                div.input(style="display: inline")
                    label Zone
                    select(name="zone", id="zone")
                        option(value = "", selected='selected')
                        option(value = "A") Zone A
                        option(value = "B") Zone B
                        option(value = "PB") Parking B
                div.input(style="display: inline")
                    label Status
                    select(name="status", id="status")
                        option(value = "", selected='selected') All
                        option(value = "1") Active
                        option(value = "0") Inactive
                div.actions(style="display: inline")
                    input(type="submit", onclick="filter()", value="filtrar", style = "float: right")

        div.table-container
            table(id = "taula", cellpadding="10")
                thead

                tbody
```

Figura 42: Fragmento del fichero calibration.pug

Como se puede observar en la imagen, dentro del apartado “div.page-body” es donde se define todo el contenido que tendrá la página web.

Los dos elementos principales de esta pantalla son, el filtro y la tabla de nodos.

El filtro, nos sirve para poder decidir que contenido queremos mostrar en la tabla, por ejemplo, solo mostrar nodos que sean agents, solo mostrar nodos que sean leaders, etc. Este elemento es de gran utilidad cuando el número de nodos registrados en la topología es muy grande y se quiere obtener información sobre uno o un conjunto de nodos específicos.

Los campos que pueden ser utilizados para filtrar son los siguientes:

- nodeID
- device
- role
- status

En cuanto a la tabla, podemos observar que en la imagen la declaración de esta es muy simple. Esta declaración se encuentra dentro del apartado “div.table-container” y se realiza de la siguiente manera:

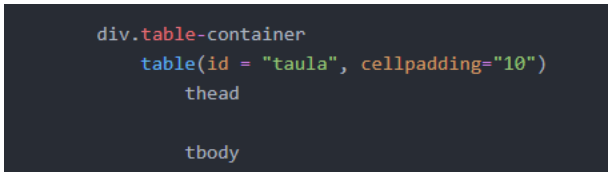

Figura 43: Declaración de la tabla de nodos

Podemos ver como se crea un objeto “table”, dentro del cual hay dos atributos “thead” y “tbody”. Estos atributos no tienen ningún tipo de información, ya que esta se genera en otro fichero llamado “calibration.js”.

Esta es una característica bastante interesante del lenguaje de programación Pug, ya que te permite crear elementos que serán insertados en la pantalla del navegador, y a partir de scripts creados en otros ficheros añadir información a estos.

A continuación, pasamos a explicar el fichero “calibration.js”. El contenido de este fichero son un conjunto de funciones que serán ejecutadas una vez se haya cargado la pantalla de visualización de agents en un navegador. Entre las funciones que se encuentran destacamos las siguientes:

- get_data_from_DB
- build_header
- build_body

La función “get_data_from_DB” es la encargada de obtener la información de los nodos que forman parte de la topología de red. Para obtener esta información se realiza una petición HTTP a un host, en nuestro caso el cloud, y una vez obtenida esta se llama a la función “build_body”, pasando como parámetro los datos obtenidos.

```
function get_data_from_DB(){
  var url = 'http://'+hostname+':8080/get_topoDB'
  var params = ""
  params += addFilter("nodeID", filterNodeID);
  params += addFilter("device", filterDevice);
  params += addFilter("role", filterRole);
  params += addFilter("zone", filterZone);
  params += addFilter("status", filterStatus);

  if(params != ""){
    params = params.slice(0, -1);
    params = "selec={"+ params + "}";
  }
  $.get(url, params, function(data){
    data.splice(data.length-2,2)
      data = data.join()
      data = JSON.parse(data)
      build_body(data)
  });
}
```

Figura 44: Estructura de la función get_data_from_DB

La función “build_body”, es la encargada de, a partir de un conjunto de datos, rellenar la tabla declarada en el fichero “calibration.pug”.

```
function build_body(agents) {
  var body = ""
  $.each(agents, function( index, agent ) {
    if(agent["nodeID"]){
      id_agents[agent["nodeID"]] = agent
    }
    if(agent["status"] != null){
      if(agent["status"] == "0"){
        body += "<tr class='status-KO'>" ;
      }
      else {
        body += "<tr>" ;
      }
      $.each(columns, function( index, column ) {
        if (agent[column] != null){
          body += "<td>" + agent[column] + "</td>";
        }
        else{
          body += "<td/>"
        }
      });
      body += "</tr>"
    }
  })
  table.find('tbody').empty();
  table.find('tbody').append(body);
}
```

Figura 45: Estructura de la función build_body

Como podemos observar, los datos obtenidos a partir de la función anterior se almacenan en la variable “agents”, que es una lista. Para mostrar estos datos, se hace un recorrido por toda la lista y se van añadiendo los datos a una variable llamada “body”. Una vez se ha acabado de recorrer esta, mediante la función

“table.find(‘tbody’).append(body)”, se añaden a la tabla todos los datos que han sido guardados en la variable “body”.

Para que la visualización de la información de los nodos sea en tiempo real, se realizan llamadas periódicamente a la función “get_data_from_DB”. Este periodo de tiempo entre llamadas a la función es de 2 segundos.

Finalmente, la función “build_header” es la encargada de construir el encabezado de la tabla, a partir de una lista llamada “columns”, que contiene todas las columnas a mostrar dentro de la tabla.

La estructura de dicha función la podemos ver a continuación:

```
var columns = ["nodeID", "device", "role", "zone", "myIP", "leaderIP", "port", "broadcast"]

function build_header() {
  var head = "<tr>";
  $.each(columns, function( index, column ) {
    head += "<th>" + column + "</th>";
  });
  head += "</tr>";
  table.find('thead').append(head);
}
```

Figura 46: Estructura de la función build_header

A parte de las funciones descritas anteriormente, esta pantalla tiene la funcionalidad de, pulsando sobre un agent de la lista de agents, abrir una nueva pantalla con la información de este agent. Esta pantalla será explicada en el apartado “Pantalla de visualización de la información de un agent” posteriormente.

A continuación, podemos ver la pantalla de visualización de los agents una vez implementada:

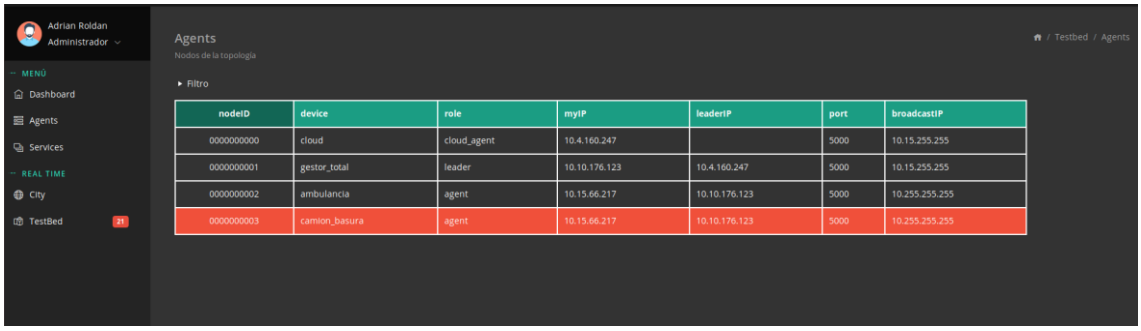

Figura 47: Pantalla de visualización de agents

Pantalla de visualización del catálogo de servicios de la ciudad inteligente

Esta pantalla ha sido implementada de la misma manera que la pantalla anterior, ya que la información que es necesario mostrar también de muestra en forma de lista y esta se recoge de la misma base de datos.

Como diferencia principal sobre la pantalla anterior, en esta disponemos la posibilidad de, mediante un botón, abrir la pantalla para añadir servicios al catálogo de servicios. De esta manera, podemos ver todos los servicios que tenemos disponibles y si queremos, podemos añadir uno nuevo fácilmente.

A continuación, podemos ver la pantalla del catálogo de servicios una vez implementada:

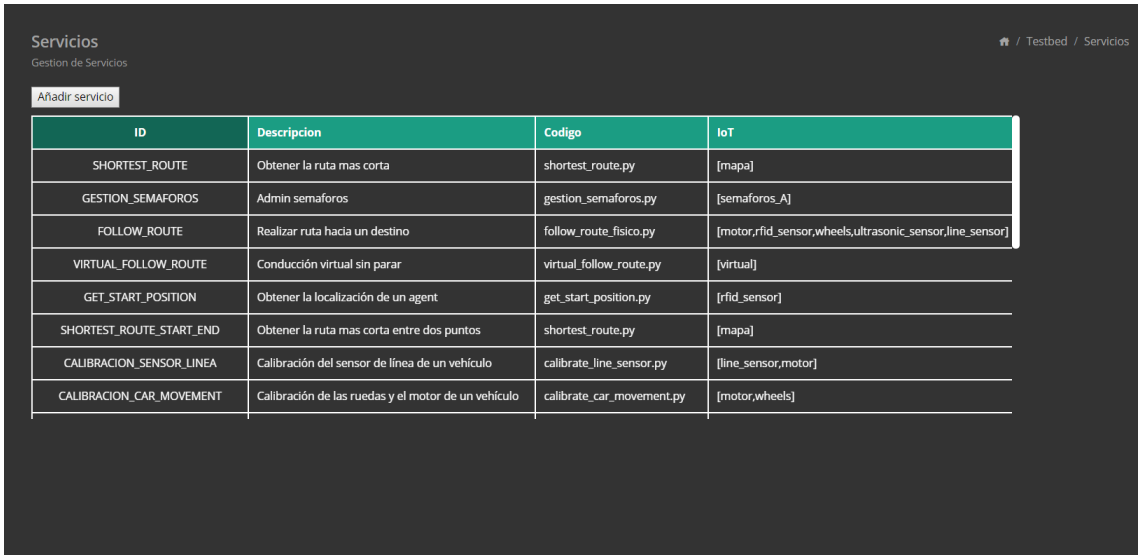

Figura 48: Pantalla de visualización del catálogo de servicios de la ciudad inteligente

Pantalla de visualización de la información de un agent

Como se ha mencionado anteriormente, esta pantalla es cargada una vez se presiona sobre algún agent de la lista de agents de la pantalla de visualización de estos.

La pantalla ha sido creada mediante la programación de dos ficheros, llamados “agent.pug” y “agent.js”.

El fichero “agent.pug” tiene los mismos apartados que los ficheros “.pug” de las dos pantallas anteriores. En cuanto al aparatado de “block content” nos encontramos con que en este fichero se declara lo siguiente:

```
div.page-body
      br
      div.agent-info
        div.agent-info-header
          h3(style="color: white") Información del nodo
        div.agent-info-body(id = "agent_info")
      div.services(id = "services")
```

Figura 49: Apartado "block content" del fichero agent.pug

Podemos ver un apartado llamado “div.agent-info” el cual contiene un subapartado llamado “div.agent-info-header”, que se encarga de mostrar un título, y otro subapartado llamado “div.agent-info-body”, que en el fichero se encuentra vacío. Estos apartados serán los encargados de mostrar la información del agent seleccionado.

A parte, tenemos otro apartado más llamado “div.services”, el cual en el fichero también se encuentra vacío. En la pantalla, este apartado será el encargado de mostrar todos los servicios que puede ejecutar el agent seleccionado.

También disponemos del apartado “append scripts”, en el cual se hace referencia al fichero “agent.js” con la finalidad de ejecutar las funciones que este contenga una vez se cargue la pantalla en un navegador.

Las principales funciones que podemos encontrar en el fichero “agent.js” son las siguientes:

- get_cloud_agent
- request_service
- get_agent_info
- get_services_to_execute
- show_agent_info
- show_service_to_execute

Función get_cloud_agent

Esta función es la encargada de obtener toda la información del nodo cloud_agent de la topología de la red. El nodo cloud_agent es necesario ya que es a quien el administrador del Front-End le solicitará la ejecución de un servicio des de este mismo.

A continuación, se puede ver la programación de esta función:

```
function get_cloud_agent(){
  var url = 'http://'+hostname+':8080/get_topoDB'
  var params = "selec={'nodeID':'0000000000'}"
  $.get(url, params, function(data){
    data.splice(data.length-2,2)
    data = data.join()
    cloud_agent = JSON.parse(data)[0]
  });
}
```

Figura 50: Función get_cloud_agent

Se puede ver como en la función, mediante una petición HTTP, se solicita la información del nodo con identificador “0000000000”, que es el cloud_agent, al host que tiene la base de datos de la topología, es decir, el cloud. Una vez obtenida la información, esta se almacena en una variable llamada “cloud_agent” para poder ser utilizada posteriormente.

Función request_service

Esta función es la encargada de solicitar un servicio al cloud_agent. La función es llamada cada vez que el administrador del Front-End solicita la ejecución de uno de los servicios disponibles para el agent seleccionado, pulsando sobre este y rellenando los parámetros necesarios del servicio si es que este tiene alguno.

El código de la función es el siguiente:

```
function request_service(service_id, params) {
  var service = {
    'service_id': service_id,
    'agent_ip': agent["myIP"],
    'params': params
  }
  $.ajax({
    url: "http://"+cloud_agent["myIP"]+":8000/request_service",
    contentType: "application/json",
    type: 'post',
    data: JSON.stringify(service),
    error: function (request, error) {
          alert("Can't execute because: " + error);
      },
      success: function (response) {
      console.log(response);
          alert(" Done!");
      }
  })
}
```

Figura 51: Función request_service

En la primera parte del código podemos ver como se crea una variable de tipo diccionario llamada “service”, en la cual se almacenan los datos del servicio, que son, el identificador de este (service_id), la IP del agent al que se le va a solicitar el servicio (agent_ip) y los parámetros de entrada del servicio (params).

Posteriormente, mediante una solicitud HTTP al cloud_agent, se solicita la ejecución del servicio y finalmente se comprueba si ha habido algún error en la solicitud de este para poder informar al administrador del Front-End si fuera necesario.

Función get_agent_info

Esta función es la encargada de obtener la información de un agent a partir de su identificador, que es obtenido a partir de la URL de la página web. Un ejemplo de URL sería el siguiente: **http://10.0.2.16:8000/calibration/0000000004**. En este caso el identificador del agent recuperado de la URL seria “0000000004”.

A continuación, podemos ver una imagen con la programación de esta función:

```
function get_agent_info() {
  var url = 'http://'+hostname+':8080/get_topoDB'
  var params = "selec={'nodeID':'"+agent_id+"'}"
  $.get(url, params, function(data){
    data.splice(data.length-2,2)
    data = data.join()
    agent = JSON.parse(data)[0]
    show_agent_info()
  });
}
```

Figura 52: Función get_agent_info

Como podemos ver en la imagen, una vez obtenida la información del agent en cuestión, se llama a la función “show_agent_info”, la cual se encarga de mostrar la información de este en pantalla. Esta función será explicada más adelante.

Función get_services_to_execute

Esta función es la encargada de obtener todos los servicios que hay disponibles en el catálogo de servicios de la ciudad inteligente. La obtención de estos servicios se hace

mediante una petición HTTP a un host, que en nuestro caso es el cloud. Una vez obtenidos todos los servicios, se hace una llamada a la función "show_service_to_execute", la cual, como más adelante veremos, se encarga de mostrar los servicios en pantalla.

A continuación, se puede ver la implementación de esta función:

```
function get_services_to_execute() {
  var params = null
  var url = 'http://'+hostname+':8080/get_serviceDB'
  $.get(url, params, function(data){
    data.splice(data.length-2,2)
    data = data.join()
    services = JSON.parse(data)
    show_service_to_execute()
  });
}
```

Figura 53: Función get_services_to_execute

Función show_agent_info

Esta función, tal y como hemos mencionado anteriormente, es la encargada de mostrar la información del agent seleccionado en pantalla. Esta información está guardada dentro de una variable de tipo diccionario y se muestra haciendo un recorrido por todas las {clave: valor} que se encuentran dentro de este.

Finalmente, para añadir la información, se hace mediante una llamada a la función "agent_info.append(body)", como podemos ver a continuación:

```
function show_agent_info(){
  var body = "<ul>";
  if(agent){
    body += "<li><b>nodeID</b>: " + agent['nodeID'] + "</li>"
    $.each(agent, function( key, value ) {
      if (agent[key] != null && key != "_id" && key != "nodeID"){
        body += "<li><b>" + key + "</b>: " + agent[key] + "</li>"
      }
    });
    body += "</ul>";
    agent_info.append(body)
  }
}
```

Figura 54: Función show_agent_info

Función show_service_to_execute

Esta última función es la encargada de mostrar en pantalla todos los servicios que puede realizar el agent seleccionado. A partir de todos los servicios obtenidos en la función "get_services_to_execute", se hace un filtro para solo obtener los que nosotros queremos.

Este filtro se hace comparando el atributo "IoT" del agent con el atributo "IoT" de cada uno de los servicios obtenidos. Si el agent dispone de todos los "IoT" que tiene el servicio con el que se está haciendo la comparación, se da por válida la comparación y se añade ese servicio al conjunto de servicios que puede ejecutar ese agent. Por el contrario, si existe algún valor dentro de los "IoT" del servicio que el agent no tiene en sus "IoT", la comparación se da como no válida y, por lo tanto, al no poder ejecutar ese servicio, no se le añade a su lista de servicios que puede ejecutar.

Finalmente, después de hacer la comprobación anterior, se añaden estos servicios a la pantalla mediante la función "services_to_execute.append(body)", tal y como podemos ver a continuación:

```
function show_service_to_execute(){
  var body = "";
  $.each(services, function(index, value){
  if(value["IoT"].every(function(val) { return agent["IoT"].indexOf(val) >= 0; })) {
    body += "<button class='accordion'>"+value["description"]+"</button>";
    var form_body = "<form>"
    var service_params = value["params"]
    if(service_params) {
    $.each(service_params, function(name, type) {
      form_body += name+": <input type='"+type+"' name='"+name+"'><br>"
    });
    }
    form_body += "<input type='hidden' name='host_frontend' value='"+hostname+"'>"
    form_body += "<input type='hidden' name='port_frontend' value='"+port+"'>"
    form_body += "<input type='hidden' name='service_id' value='"+value["_id"]+"'>"
    form_body += "<input type='hidden' name='agent_id' value='"+agent["nodeID"]+"'>"
    form_body += "<button type='submit' style='float: right'>Ejecutar servicio</button>"
    form_body += "</form>"
    body += "<div class='panel'>"+form_body+"</div>";
  }
  });
  services_to_execute.append(body)
  hide_accordion()
}
```

Figura 55: Función show_service_to_execute

Una vez explicadas las funciones más importantes de esta pantalla y sabiendo que papel realiza cada una de estas, el resultado de la implementación de la pantalla de visualización de la información de un agent es el siguiente:

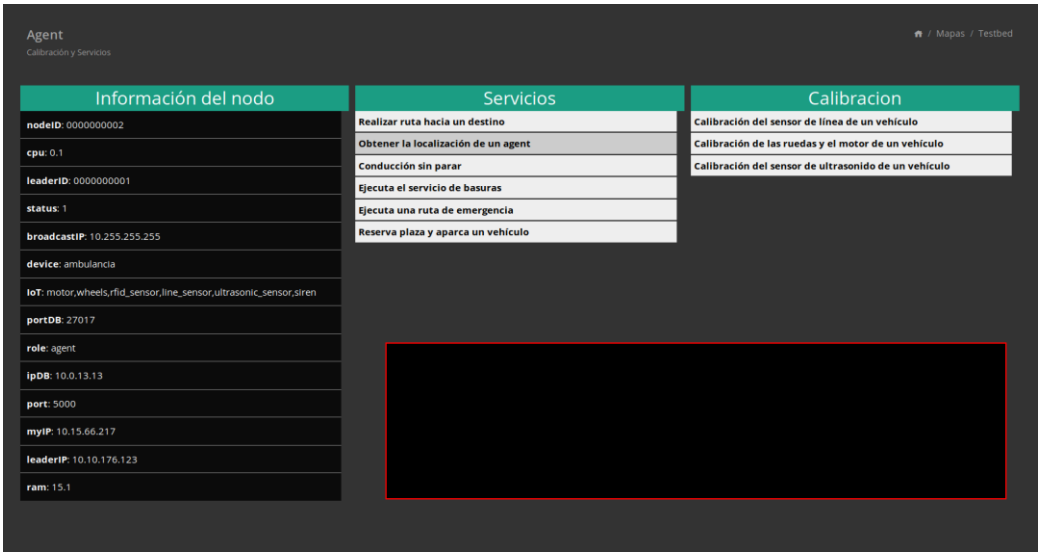

Figura 56: Pantalla de visualización de la información de un agent

Pantalla para añadir servicios al catálogo de servicios de la ciudad inteligente

Esta última pantalla tiene la finalidad de que los usuarios de la ciudad inteligente, que dispongan de un agent, puedan añadir sus servicios al catálogo de servicios de la ciudad inteligente.

Como todas las anteriores pantallas, esta también dispone de dos archivos, uno llamado “add_service.pug” y otro llamado “add_service.js”.

El contenido de “add_service.pug” también consta de los mismos dos apartados principales que el resto de las pantallas. En cuanto al apartado “block_content”, este consta de 11 parámetros de entrada necesarios para poder añadir un nuevo servicio al catálogo de servicios. Estos parámetros son todos y cada uno de los atributos que tiene un servicio, atributos que han sido explicados en el apartado “5.2.4 Servicios”.

Una vez rellenados todos los parámetros necesarios, un usuario del Front-End puede añadir este servicio al catálogo de servicios presionando sobre el botón “Añadir”, situado en la parte inferior derecha de la pantalla. Una vez presionado este botón, se ejecutará una función que se encuentra dentro del fichero “add_service.js”.

Esta función del fichero “add_service.js”, se encarga de recoger todos los parámetros introducidos por el usuario y mediante una petición HTTP, los envía a un host, en

nuestro caso el cloud, para que se añada este nuevo servicio al catálogo de servicios de la base de datos.

El resultado de la programación de esta pantalla es el siguiente:

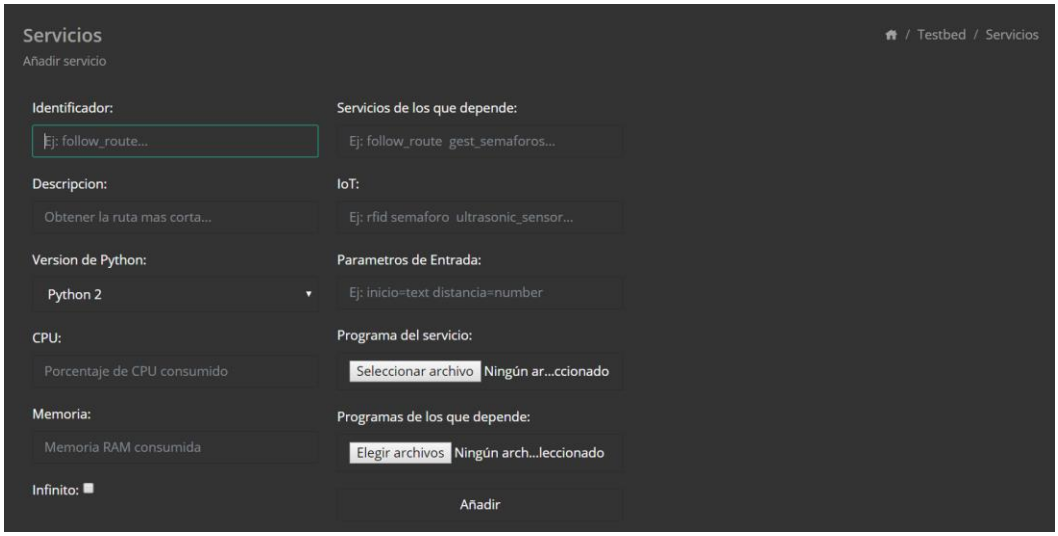

Figura 57: Pantalla para añadir servicios al catálogo de servicios de la ciudad inteligente

7.6 APLICACIÓN CLIENTE DE UN AGENT

Esta aplicación ha sido implementada utilizando el lenguaje de programación “Python”. A continuación, se detallará la implementación de las dos posibles fases de esta aplicación, la configuración inicial de un agent y la solicitud de servicios.

Configuración inicial de un agent

Como ya se ha mencionado anteriormente, la aplicación entra en una fase de configuración de un agent si esta es ejecutada por primera vez. De esta manera, se puede definir que elemento de la ciudad inteligente va a ser el agent en cuestión.

Tal y como se ha explicado en el apartado de diseño, los diferentes parámetros de un agent que pueden ser configurados desde esta aplicación son los siguientes: device y role.

Para seleccionar el tipo de dispositivo que será nuestro agent, se puede hacer mediante la elección de diferentes opciones de un listado. Estas opciones son las siguientes: A continuación, se puede ver una imagen donde se muestra la lista con las diferentes opciones a escoger:

```
Selecciona que tipo de dispositivo sera el agent

=> gestor_farolas:['farola', 'mapa']
   gestor_semaforos:['semaforo', 'mapa']
   gestor_trafico:['mapa']
   camion_basura:['motor', 'wheels', 'rfid_sensor', 'line_sensor', 'ultrasonic_sensor']
   ambulancia:['motor', 'wheels', 'rfid_sensor', 'line_sensor', 'ultrasonic_sensor', 'siren']
```

Figura 58: Lista para seleccionar el device

De la misma manera, para seleccionar el role que tendrá un agent dentro de la topología, se puede hacer mediante la elección de diferentes opciones dentro de un listado. En este caso las opciones que se pueden escoger son agent y leader, como podemos ver a continuación:

```
Selecciona que rol tendra el agent en la topologia

=> agent
   leader
```

Figura 59: Lista para seleccionar el role

Solicitud de servicios

Una vez realizada la configuración anterior, el agent ya queda registrado dentro de la base topológica, y se encuentra disponible para realizar una serie de servicios. Para seleccionar los servicios a ejecutar se hace mediante una lista igual a las dos que hemos visto anteriormente, donde se ven el identificador de cada uno de los servicios juntamente con su descripción.

A continuación, se puede ver el listado de servicios que se muestra en la aplicación:

```
Elige un servicio a solicitar:

=> SHORTEST_ROUTE:Obtener la ruta mas corta
   SHORTEST_ROUTE_START_END:Obtener la ruta mas corta entre dos puntos
   GET_DUMPSTERS:Obtener los containers
   GESTION_TRAFICO:Admin Trafico
   EXIT
```

Figura 60: Lista de los servicios que pueden ser solicitados

8. PRUEBAS DE FUNCIONAMIENTO

Una vez finalizado el desarrollo, tanto de la calibración, como de la conducción autónoma, se han realizado diferentes tests para verificar el correcto funcionamiento de estos. Estos tests se han realizado sobre el conjunto de funcionalidades que nos ofrecen estos.

8.1 TEST DE MOVILIDAD DEL VEHÍCULO

Con este test se pretende comprobar que el movimiento de los vehículos no presenta ningún tipo de problema y es completamente correcto.

Para poder realizar este test se ha creado un programa mediante el cual, de forma manual, se pueden activar tanto el motor del vehículo como modificar el ángulo de giro de las ruedas.

8.2 TESTS DE LOS DIFERENTES SENSORES

Con este conjunto de tests se pretende verificar el correcto funcionamiento de los diferentes sensores instalados en los vehículos. Como hemos visto anteriormente estos sensores son los siguientes: sensor de línea, sensor de ultrasonido y lector de RFID.

Para poder testear cada uno de estos, se han utilizado un conjunto de programas, los cuales nos permiten verificar el correcto funcionamiento de estos. Estos programas son proporcionados por el equipo de SunFounder.

8.3 TEST DE COMUNICACIÓN CON UN AGENT

El objetivo principal de este test es verificar la correcta puesta en marcha de la API de un agent. Esta verificación se ha realizado de la siguiente manera:

Primero de todo, se ha de poner en marcha este servidor API REST que se encuentra dentro del agent. Para hacer esto, basta con encender el agent, ejecutando el comando **$ python3 device.py** desde el directorio donde se encuentra este. Este código se encarga de crear un agent con un conjunto de parámetros predefinidos en el propio programa y posteriormente enciende el servidor API REST.

Para comprobar que este servidor está funcionando correctamente, se realiza una petición HTTP desde un navegador a la URL del agent. Esta URL está compuesta por la IP del agent en cuestión y un puerto por defecto para las API de los diferentes agents, que es el 8000. La URL tiene el siguiente formato: **http://IP_AGENT:8000**.

Suponiendo que disponemos de un agent con IP 10.0.2.16, la petición HTTP que se debe de realizar desde un navegador es la siguiente: **http://10.0.2.16/alive**. Como podemos observar se ha añadido un campo al final de la URL, el cual es **/alive**. Este último campo es necesario para redirigir la petición a una función específica de la API del agent, la cual devuelve el texto “Alive” si esta funciona correctamente, como podemos ver a continuación:

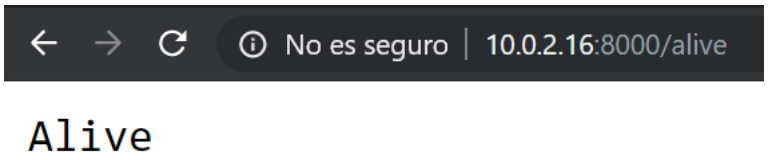

Figura 61: Respuesta de un agent activo

8.4 TEST DE COMUNICACIÓN ENTRE AGENTS

En este test se comprueba que la comunicación entre agents, que es uno de los aspectos más importantes de estos, se puede realizar correctamente. Para probar esto, se han definido dos objetivos para este test, que son los siguientes:

- Verificar el correcto funcionamiento del envío de información.
- Verificar el correcto funcionamiento del recibimiento de información.

Estas verificaciones se han realizado mediante el registro de un agent en la topología. Como ya se ha mencionado anteriormente, en el registro de un agent existe una comunicación bidireccional. Hemos aprovechado esta comunicación bidireccional para verificar el correcto funcionamiento de la comunicación entre agents.

Primero de todo, observamos como un agent que quiere registrarse en la topología, solicita al que será su nuevo leader, que este le añada a la base de datos topológica, con toda la información proporcionada por el propio agent.
El siguiente paso en el registro de un agent, es recibir una respuesta de su nuevo leader, respuesta de la cual el agent recuperará su nuevo identificador.

A continuación, se puede ver una captura con la solicitud de registro de un agent a su leader y la respuesta de este registro que contiene el identificador que se le ha asignado al nuevo agent:

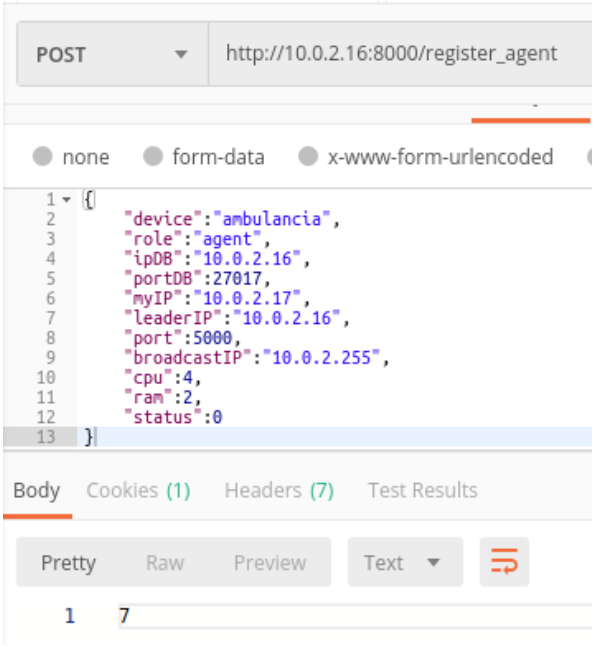

Figura 62: Solicitud de registro y devolución de identificador

Para comprobar que este nuevo agent se ha registrado correctamente en la base de datos topológica, podemos acceder a la pantalla de visualización de agents y ver que este se encuentra ahí. En este caso el identificador asignado ha sido el 7, que corresponde a la última posición de la tabla:

▶ Filtro

nodeID	device	role	zone	myIP	leaderIP	port	broadcast
0000000000	Agent	cloud_agent		10.0.2.16			
0000000003	gestor_semaforos	leader		10.4.159.155	10.0.2.16	5000	
0000000004	gestor_trafico	leader		10.4.159.155	10.0.2.16	5000	
0000000005	gestor_trafico	leader		10.4.159.155	10.0.2.16	5000	
0000000006	gestor_trafico	leader		10.4.159.155	10.0.2.16	5000	
0000000007	ambulancia	leader		10.0.2.17	10.0.2.16	5000	

Figura 63: Tabla de agents de la topología (Front-End)

De esta manera, se puede verificar el segundo objetivo de este test, el cual dice que un agent ha de ser capaz de poder recibir información de otros agents correctamente.

8.5 TEST DE COMUNICACIÓN ENTRE EL FRONT-END Y AGENT

Para que el administrador del Front-End pueda solicitar servicios a los agents, tiene que existir una conexión entre el propio panel Front-End y estos. Esta comunicación ha sido implementada mediante peticiones HTTP directamente a las API de los diferentes agents, tal y como se ha hecho para la comunicación entre agents.

La prueba que se ha realizado consiste en realizar una petición HTTP des del panel Front-End directamente a la API del Cloud agent. Esta petición es la solicitud de ejecución de un servicio llamado “GESTION_TRAFICO”. Como respuesta a esta petición, se obtiene un “success”, indicando que la petición realizada des del Front-End ha podido ser atendida por un agent.

A continuación, se puede ver una imagen que muestra tanto el mensaje de solicitud de servicio recibido en el Cloud agent, como la respuesta a este mensaje recibida en el Front-End:

Figura 64: Solicitud del Front-End y respuesta del Cloud Agent

8.6 TEST DE EJECUCIÓN DE SERVICIOS SIMPLES

A la hora de crear un servicio, tal y como hemos visto anteriormente, este puede ser creado con dependencias a otros servicios, o sin ninguna dependencia. Un servicio sin dependencias a otros es considerado un servicio simple. Por el contrario, si un servicio tiene dependencias a otros servicios, este es considerado un servicio complejo.

En el caso de los servicios simples, las diferentes pruebas que se han realizado son las siguientes:

- Solicitud de un servicio con algún agent activo que pueda ejecutarlo
- Solicitud de un servicio con ningún agent activo que pueda ejecutarlo

En el primer caso, podemos ver como al haber algún agent activo que cumple los requerimientos para poder ejecutar el servicio solicitado, este lo ejecuta sin ningún problema. El resultado de tal ejecución, como hemos visto anteriormente, puede ser "success" o "error".

A continuación, se puede ver el resultado "success" de la ejecución de un servicio simple:

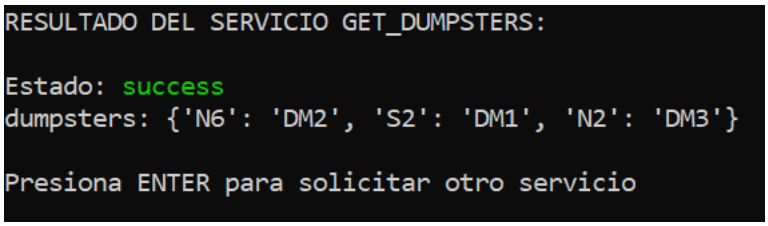

Figura 65: Ejecución con resultado "success" de un servicio simple

En el segundo caso, al no haber ningún agent activo con posibilidad de ejecutar el servicio solicitado, el resultado de tal solicitud es "unattended", indicando que ese servicio no ha podido ser atendido por ningún agent.

8.7 TEST DE EJECUCIÓN DE SERVICIOS COMPLEJOS

Tal y como hemos explicado antes, un servicio complejo es aquel que tiene alguna dependencia a otro, u otros servicios.

En el caso de los servicios complejos, las diferentes pruebas que se han realizado para probar el correcto funcionamiento de la solicitud y ejecución de estos son las siguientes:

- Solicitar un servicio complejo, donde tanto el propio servicio como sus servicios dependientes pueden ser ejecutados por algún agent.
- Solicitar un servicio complejo, donde el propio servicio puede ser ejecutado, pero alguno de sus servicios dependientes no puede ser ejecutado por ningún agent.
- Solicitar un servicio complejo, donde el propio servicio no puede ser ejecutado por ningún agent, pero todos sus servicios dependientes si que pueden ser ejecutados.

- Solicitar un servicio complejo, donde ni el propio servicio ni alguno de sus servicios dependientes pueden ser ejecutados por ningún agent.

Tal y como se ha explicado anteriormente, cuando se solicita la ejecución de un servicio complejo, primero de todo se solicita la ejecución de cada uno de los servicios dependientes. Una vez se han ejecutado todos sus servicios dependientes, entonces se ejecuta el propio servicio. Teniendo en cuenta esto, a continuación, se detalla que es lo que pasa en cada uno de los casos anteriormente mencionados.

En el primer caso, se ejecutarían todas las dependencias del servicio y finalmente el propio servicio. Podemos ver como en este caso, todo ha sido ejecutado, por lo tanto, como hemos visto con los servicios simples, el posible resultado de dicha ejecución podría ser tanto “success”, como “error”.

A continuación, se puede ver una imagen de la ejecución de un servicio complejo, donde el resultado de este es “error”:

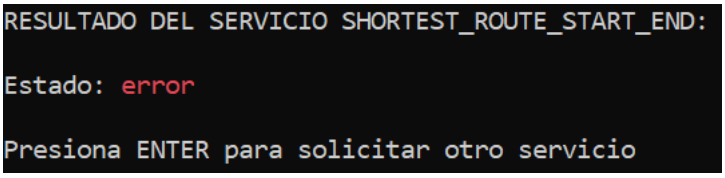

Figura 66: Ejecución con resultado "error" de un servicio complejo

Para los tres casos restantes, nos encontramos con que, si uno de los servicios dependientes no puede ser ejecutado, en ese momento, se devuelve como resultado de ese servicio “unattended”. De esta manera se notifica al servicio solicitado, que este no va a poder ser ejecutado por ningún agent, por lo tanto, de devuelve como resultado de la solicitud del servicio “unattended”.

A continuación, se puede ver una imagen de la ejecución de un servicio complejo, donde una de sus dependencias no puede ser ejecutada y por lo tanto se obtiene como resultado “unattendned”, indicando que el servicio solicitado no ha podido ser ejecutado por ningún agent:

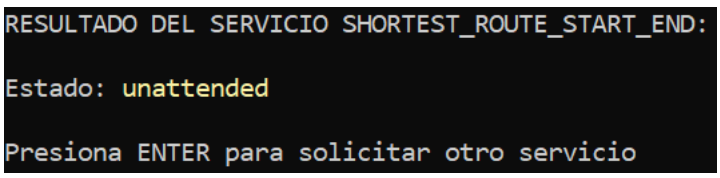

Figura 67: Ejecución con resultado "unattended" de un servicio complejo

8.8 DEMO FINAL

La DEMO final tiene como objetivo principal demostrar el correcto funcionamiento de todas las funcionalidades implementadas, tanto de la conducción autónoma, como de la calibración de dispositivos de una ciudad inteligente.

Esta tiene lugar en el Testbed proporcionado por el CRAAX, lugar donde se probarán servicios de calibración de diferentes elementos de la ciudad inteligente, así como varios servicios de conducción autónoma.

En cuanto a los diferentes actores que tendrán lugar en esta simulación, encontramos los siguientes:

- Administrador (Persona física)
- Persona (Persona física)
- Cloud (Cloud agent)
- Gestor de los semáforos (leader 1)
- Gestor de las farolas (leader 1)
- Gestor del tráfico (leader 2)
- Ambulancia (agent 1)
- Camión de la basura (agent 2)

De esta manera, disponemos de un Cloud agent, dos leaders y dos agents, obteniendo la siguiente estructura de la topología de la red:

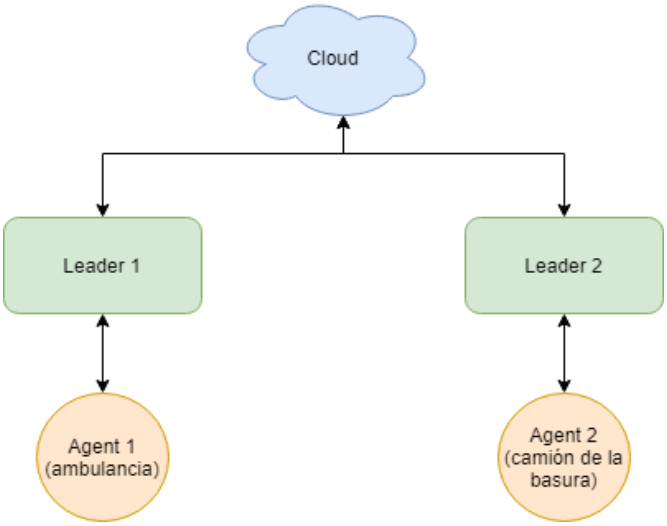

Figura 68: Estructura de la topología de red

Una vez sabemos quiénes serán los diferentes actuadores en esta DEMO, pasamos a explicar cada uno de los pasos a realizar:

1. El administrador del Front-End solicitará el servicio de recogida de basura.
2. El camión de la basura empezará a realizar las diferentes rutas correspondientes de forma autónoma y continua.
3. Se simulará un accidente en algún punto del Testbed.
4. Una persona a través de un agent se solicitará el servicio de emergencia hacia el lugar donde se ha simulado el accidente.
5. La ambulancia saldrá del hospital y realizará la ruta óptima para llegar al sitio del accidente, interactuando con los diferentes elementos de la ciudad inteligente, es decir, semáforos, farolas y otros vehículos.
6. Una vez la ambulancia llegue al punto del accidente, se parará unos segundos simulando una atención a las posibles víctimas y posteriormente volverá al hospital, solicitando una nueva ruta óptima. De la misma manera, la ambulancia seguirá interactuando con los diferentes elementos de la ciudad inteligente.
7. Una vez la ambulancia llegue al hospital de nuevo, esta quedará estacionada y el servicio de emergencia se dará por finalizado.

9. INTEGRACIÓN CON OTROS PROYECTOS

Una vez realizadas las pruebas de funcionamiento de todo el proyecto, se ha realizado una integración con un TFG con nombre "*Sistema d'aparcament intel·ligent basat en F2C*" [34] realizado por un compañero, el cual también ha desarrollado un agent y un servicio de aparcamiento, entre otras cosas.

Esta integración consiste en poder añadir agents diseñados por diferentes TFG dentro de una misma topología y probar que estos pueden comunicarse entre ellos, con la finalidad de poder ejecutar un servicio conjunto.

Los diferentes nodos de la topología que forman parte de la ejecución de este servicio son los siguientes:

- Cloud Agent
- Leader A
- Leader Parking
- Agent

El servicio que se ha ejecutado es un servicio de aparcamiento el cual consiste en lo siguiente:

1. Un ciudadano de la ciudad inteligente, solicita el servicio a su agent mediante una aplicación.
2. El agent solicita el servicio a su leader, en este caso es el leader A.
3. El leader A se descarga de la base de datos la información del servicio y comprueba sus dependencias.
4. La primera dependencia que encuentra es el servicio "Reserva plaza", el cual reserva una plaza de aparcamiento en el parking más cercano.
5. El leader A solicita el servicio de reservar plaza a su leader, que en este caso es el cloud agent.
6. El cloud agent solicita la ejecución del servicio de reservar plaza al leader correspondiente que lo puede ejecutar, en este caso el leader del parking.
7. El leader del parking ejecuta el servicio y devuelve el resultado al cloud agent. El resultado de este servicio es la plaza que ha sido reservada en el parking.
8. El cloud agent devuelve la ejecución del servicio al leader que se lo había solicitado anteriormente, el leader A.
9. El leader A comprueba la siguiente dependencia, que en este caso es el servicio de posición inicial, el cual manda ejecutar el agent.

10. El agent ejecuta el servicio y le devuelve al leader como resultado su posición actual.
11. Con esa información el leader A ejecuta el servicio de generar la ruta óptima hasta el parking.
12. Finalmente, el leader A solicita la ejecución del servicio de conducción autónoma hacia un destino al agent, para que este pueda ir correctamente al parking.

A continuación, se puede ver el diagrama de comunicación del servicio de aparcamiento:

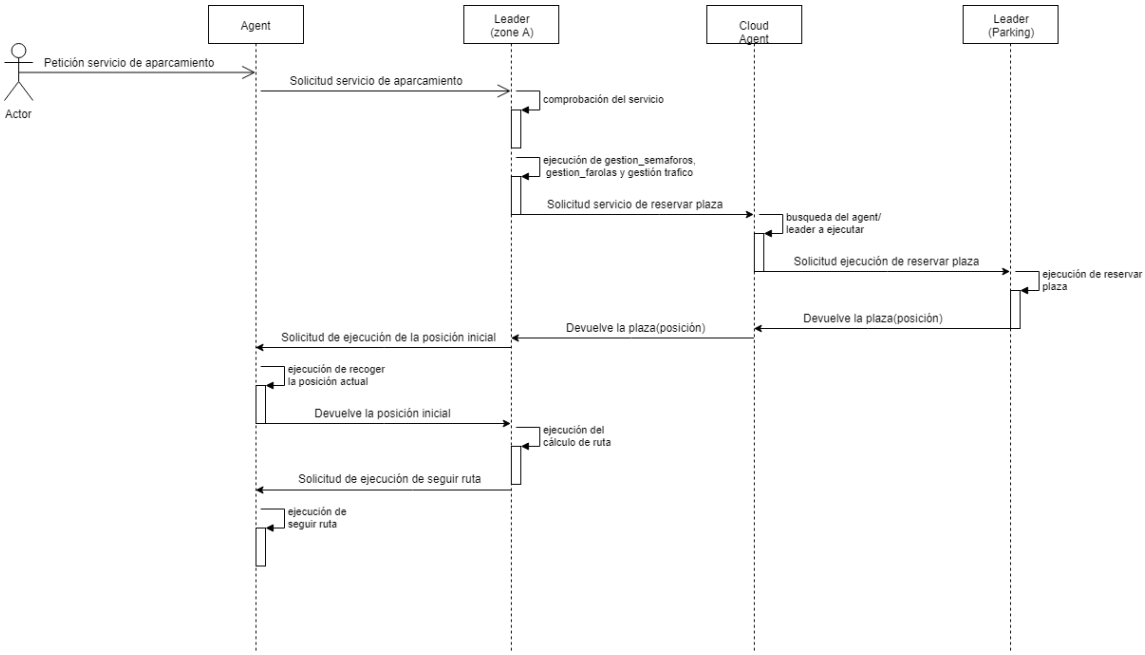

Figura 69: Diagrama de comunicación del servicio de aparcamiento

La estructura topológica que se ha creado para la ejecución de este servicio es la siguiente:

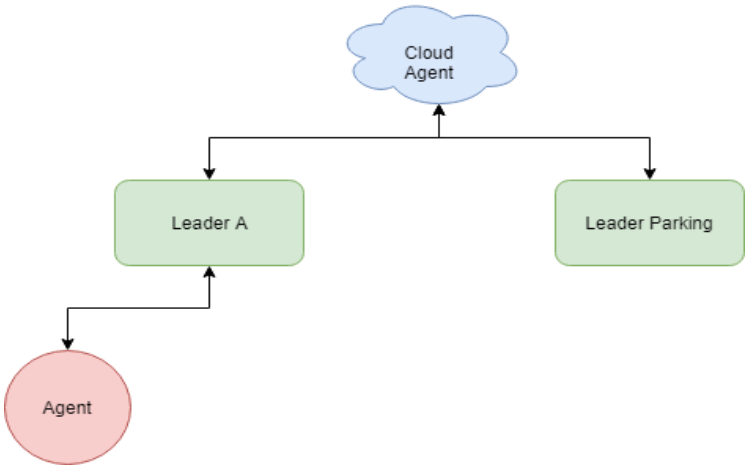

Figura 70: Estructura de la topología del servicio conjunto

Durante la ejecución del servicio, se ha creado un cluster entre el leader A y el agent. Este cluster, como hemos visto anteriormente, es utilizado para el intercambio de información inmediato, como puede ser para solicitar el estado de un semáforo, solicitar la activación de una farola, etc.

Durante la ejecución del servicio, la estructura de la topología sería la siguiente:

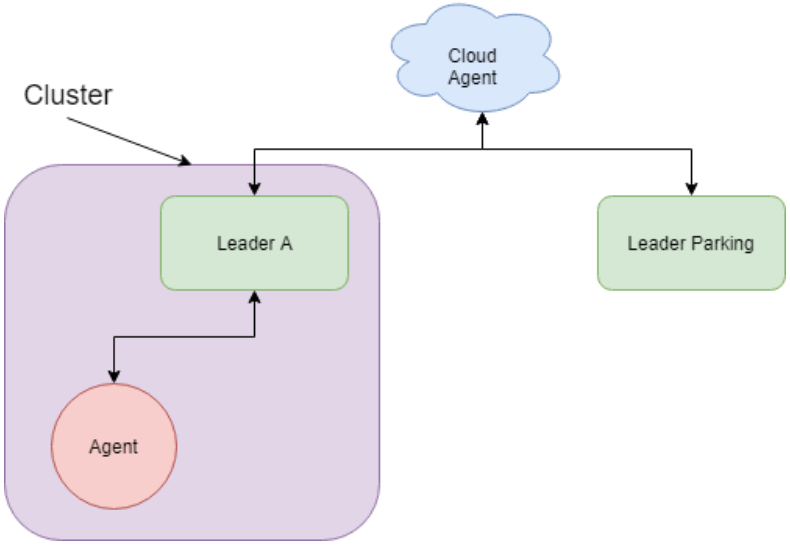

Figura 71: Estructura de la topología durante la ejecución del servicio

10. CONTINUIDAD DEL PROYECTO

En este proyecto se muestra el desarrollo e implementación de un sistema de conducción autónoma que utiliza las ventajas que nos proporciona F2C dentro de un entorno de Smart City, así como el desarrollo e implementación de la calibración de los diferentes dispositivos que podemos encontrar en una Smart City.

Una propuesta de mejora es la integración de nuevos dispositivos dentro de la ciudad inteligente, los cuales sean capaces de proporcionar información útil a la hora de realizar la conducción autónoma. De esta manera se podrían tener en cuenta más fuentes de información para tomar decisiones más precisas o simplemente para incrementar el número de posibles acciones a realizar.

De la misma manera, en cuanto a la calibración una idea que podría dar continuidad al proyecto sería añadir más sensores a los diferentes dispositivos de la ciudad para de esta manera poder simular fallos y ver que efectivamente estos son detectados por los sistemas de calibración.

Otra mejora (quizá fuera del contexto de este proyecto), sería incrementar el número de tags RFID con el fin de hacer más preciso el sistema de posicionamiento de los vehículos, lo cual permitiría realizar ciertas acciones de una manera más fluida y fiable, como la detección del estado de los semáforos, la proximidad del resto de vehículos, etc. También se podría añadir un segundo carril al Testbed, lo cual permitiría comprobar el funcionamiento de la conducción autónoma cuando los vehículos pueden circular en ambos sentidos al mismo tiempo, pudiendo llevar a cabo la realización de ciertas mejoras en esta.

11. PRESUPUESTO

En este apartado se detalla el conjunto de gastos del proyecto, tanto a nivel material, como a conjunto de horas trabajadas en el desarrollo de este.

En cuanto a nivel material, los gastos que podemos encontrar en nuestro proyecto son los siguientes:

- **Servidor donde alojar el Front-End:** Como servidor donde alojar el Front-End hemos elegido uno llamado “SiteGround” [35], el cual dispone de precios des de 3,95€/mes hasta 11,95€/mes, dependiendo del plan a contratar.

En cuanto a los gastos relacionados con el conjunto de horas trabajadas encontramos los siguientes:

- **Horas trabajadas:** 800horas/persona
- **Precio por hora [36]:** 7,63€

Teniendo en cuenta los precios anteriores, el gasto total para desarrollar el proyecto es el siguiente:

Precio del servidor: 11,95€/mes x 11 meses = 131,45€
Precio de las horas trabajadas: 7,63€/hora x 800horas/persona x 2personas = 12.208€

Precio final = Precio del servidor + Precio de las horas trabajadas = 131,45€ + 12.208€ = **12.339,45€**

12. CONCLUSIONES

Una vez finalizado el proyecto, podemos ver que, en cuanto a los objetivos propuestos al inicio de este, todos han sido cumplidos satisfactoriamente.

Se ha realizado con éxito tanto el diseño como la implementación de la conducción autónoma, dentro de una ciudad inteligente, ciudad en la cual también se puede realizar la calibración de diferentes dispositivos que se encuentren dentro de esta. Todo esto, teniendo en cuenta el gran papel que tiene el F2C, el cual hace que la conducción autónoma desarrollada en este proyecto disponga de grandes ventajas respecto a la conducción autónoma existente hoy en día.

Como hemos visto a lo largo del documento, en el diseño y la implementación de la conducción autónoma, también intervienen varios elementos de dicha ciudad. Con esta interconexión entre los vehículos y los elementos de la ciudad, logramos tener una conducción eficiente y segura.

También hemos observado la importancia de que los elementos de dicha ciudad inteligente estén funcionando correctamente y para ello es que se pueda realizar una calibración sobre estos, ya que, si los elementos de la ciudad no están correctamente calibrados, esto puede llegar a repercutir en la conducción autónoma haciendo que esta sea peligrosa e inestable.

En cuanto al positivo resultado del proyecto, este ha sido conseguido en gran parte, gracias a la metodología utilizada en el desarrollo de este. Al ser un tipo de metodología en la cual, periódicamente se han realizado reuniones con el cliente, esto nos ha permitido que, en caso de haber tenido que realizar algunos cambios, estos no hayan tenido una gran repercusión a la hora realizarlos y de esta manera poder seguir con el desarrollo del proyecto sin ningún problema.

Para concluir, no solo se ha conseguido desarrollar un gran proyecto, cumpliendo de manera exitosa cada uno de los objetivos propuestos, sino que también se han ido mejorando y ampliando para llegar a un gran producto final.

12.1 VALORACIÓN PERSONAL

Durante el desarrollo del proyecto nos hemos encontrado con dos obstáculos principales.

El primero de ellos ha sido que, al ser un proyecto “sin fin”, es decir, que puede seguir desarrollándose y mejorándose de forma indefinida, no se han definido correctamente las fases finales de este. Al no haber definido estas de forma precisa, nos hemos encontrado en que en todo momento había nuevas cosas a desarrollar, por lo que, en

las fases finales de este, hemos tenido alguna dificultad para indicar hasta que punto se iba a seguir implementado y cuando debíamos parar.

El segundo obstáculo con el que nos hemos encontrado ha sido que, en ciertas ocasiones, los objetivos propuestos por el cliente han sido entendidos de forma errónea, lo cual nos ha implicado realizar cambios más grandes de lo necesario, o directamente rehacer lo que ya había sido desarrollado.

Pese a estos obstáculos que han aparecido a lo largo del desarrollo del proyecto, se ha logrado sobrepasar estos lo cual nos ha permitido cumplir los objetivos finales del proyecto dentro de la planificación general.

13. REFERENCIAS

[1] CRAAX (Centre d'Arquitectures Avançades de Xarxes): https://www.craax.upc.edu/index.php/about-us

[2] MF2C: https://www.mf2c-project.eu/objectives/

[3] Marlon Díaz, Roger Figueras, Marc Medrano, Carlos Padial (2018). *Diseño e implementación de un Testbed para una Smart City.*

[4] Trello: https://trello.com/es/about

[5] Github: https://github.com/about

[6] Python: https://docs.python.org/

[7] TCP: https://tools.ietf.org/html/rfc793

[8]: Adrián Roldán (2018). *Panel Front-End para el control de una Smart City.*

[9] Norman Bel Geddes: https://en.wikipedia.org/wiki/Norman_Bel_Geddes

[10] Ernst Dickmanns: https://en.wikipedia.org/wiki/Ernst_Dickmanns

[11] Niveles de autonomía: https://www.carmagazine.co.uk/car-news/tech/autonomous-car-levels-different-driverless-technology-levels-explained/

[12] Pavel Schilling: https://es.wikipedia.org/wiki/Pavel_Schilling

[13] Incremento de la seguridad con la conducción autónoma: https://www.lavanguardia.com/motor/actualidad/20181130/453239114130/coche-autonomo-mayor-seguridad-vial.html

[14] CityMobil2: https://cordis.europa.eu/project/rcn/105617/reporting/en

[15] Robosoft: https://robosoft.be/en/about

[16] European Truck Platooning Challenge: https://www.eutruckplatooning.com/About/default.aspx

[17] *Longgang, Shenzhen: Giving Birth to a Smart City*: https://e.huawei.com/en/publications/global/ict_insights/201806041630/success/201808170845

[18] Daimler: https://www.daimler.com/en/

[19] Waymo: https://waymo.com/

[20] Tesla: https://www.tesla.com/es_ES

[21] Embark: https://embarktrucks.com/

[22] Uber: https://www.uber.com/es/es-es/about/

[23] Lyft: https://www.lyft.com/

[24] Departamento de Arquitectura de Computadores (DAC): https://www.ac.upc.edu/es

[25] Raspberry Pi: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[26] Arduino: https://www.arduino.cc/

[27] Processing: https://processing.org/

[28] HTTP: https://tools.ietf.org/html/rfc2616

[29] MongoDB: https://www.mongodb.com

[30] CherryPy: https://docs.cherrypy.org/en/latest/

[31] Sqlite: https://www.sqlite.org/about.html

[32] Jade: http://jade-lang.com/api

[33] Javascript: https://www.javascript.com/about

[34] Nil Salvat (2019). *Sistema d'aparcament intel·ligent basat en F2C.*

[35] SiteGround: https://www.siteground.es/hosting-web.htm

[36] Precio por hora: https://www.boe.es/eli/es/res/2019/07/22/(2)

14. BIBLIOGRAFÍA

- Adrián Pedraza (2019). *Multi Inteligent Agent Unit per a una Smart City.*
- Python. *Documentation.* Disponible en: https://www.python.org/doc/
- UNE 178201-2 Sistemas de gestión de la Ciudad Inteligente e Indicadores de cuadros de mando. Disponible en: https://www.aenor.com/certificacion/administracion-publica/ciudad-inteligente-indicadores
- Asociación Española para la Calidad: *Smart Cities Ciudades Inteligentes.* Disponible en: https://www.aec.es/c/document_library/get_file?uuid=c8b4e9a6-1796-4e90-9ddd-4ceb3ac66a81&groupId=10128
- Artículo en academia.edu: *Smart City Roadmap.* Disponible en: https://www.academia.edu/21181336/Smart_City_Roadmap
- Recorte de noticia: *Connected Vehicles in Smart Cities: The Future of Transportation.* Disponible en: https://interestingengineering.com/connected-vehicles-in-smart-cities-the-future-of-transportation

  Artículo en gradiant.org: *Edge/Fog Computing: del Cloud hacia la computación en los dispositivos*. Disponible en: https://www.gradiant.org/blog/edge-fog-computing-cloud/
- Artículo en saiconference.com: *Towards Service Protection in Fog-to-Cloud (F2C) Computing Systems.* Disponible en: https://saiconference.com/Downloads/FTC2017/Proceedings/73_Paper_4-Towards_Service_Protection_in_Fog-to-Cloud.pdf
- Blog de la empresa Tesla. Disponible en: https://www.tesla.com/es_ES/blog
- Press de la empresa embarktrucks. Disponible en: https://embarktrucks.com/press.html
- Blog de la empresa Daimler. Disponible en: https://blog.daimler.com/en/
- Blog de la empresa waymo. Disponible en: https://waymo.com/tech/
- Comisariado europeo del automovil: *Los niveles de la conducción autónoma.* Disponible en:

https://www.cea-online.es/blog/213-los-niveles-de-la-conduccion-autonoma

15. ANEXO

En este apartado se explica de forma detallada el conjunto de aspectos a tener en cuenta para poder poner en marcha el proyecto desarrollado y de esta manera probar el funcionamiento de este.

Hay que tener en cuenta, que el desarrollo de este se ha hecho utilizando máquinas con el sistema operativo Raspbian (instalado en las Raspberry Pi) y el sistema operativo Ubuntu 18.04.3 (instalado en el resto de las máquinas que no son Raspberry Pi). De esta forma, todo manual de instalación que se encuentra a continuación ha de ser realizado en uno de los dos sistemas operativos indicados.

También es necesario que toda configuración sea hecha estando conectado a la red wifi "Zephyrus" proporcionada por el CRAAX.

15.1 MANUAL DE INSTALACIÓN DEL FRONT-END

El Front-End debe de ser instalado en una máquina con el sistema operativo Ubuntu, ya que, en una Raspberry Pi, este no se instala correctamente debido a las limitaciones que estos pequeños ordenadores presentan.

A continuación, se muestra de forma detallada el conjunto de pasos necesarios para poder instalar correctamente el nuevo Front-End con los cambios realizados a lo largo de este proyecto:

Primer paso:
Primero de todo, debemos de descargarnos el repositorio del Front-End situado en la página web Github.

Para hacer esto podemos descargar directamente un archivo ZIP, o des de un terminal podemos hacer un clon del repositorio, lo cual nos creará un directorio con el contenido de este.
Para descargar el archivo como ZIP, primero de todo debemos de acceder al siguiente enlace a través de un navegador: https://github.com/lowi1996/front-end

Una vez abierta la página web, en la parte derecha nos aparecerá un botón con el texto

"Clone or download". Debemos de presionar sobre este y a continuación hacer clic sobre "Download ZIP". De esta manera la descarga del archivo ZIP comenzará automáticamente.

Una vez finalice la descarga, debemos de descomprimir el archivo para que de esta manera se cree una carpeta con el contenido de este, a la cual nosotros podamos acceder.

Otra forma de obtener la carpeta con el contenido necesario del repositorio es haciendo un clon de este. Para hacer esto debemos de seguir los siguientes pasos:

- Primeramente, abriremos un terminal y ejecutaremos el siguiente comando:
  $ sudo apt-get install git
- Una vez se nos haya instalado el paquete "git" debemos de situarnos sobre el directorio en el cual queremos crear la carpeta con el contenido del Front-End.
- Posteriormente debemos de ejecutar el siguiente comando:
  $ git clone https://github.com/lowi1996/front-end

De esta manera, se habrá creado una carpeta con el contenido del Front-End en el directorio que hayamos seleccionado.

Segundo paso:

Una vez disponemos de la carpeta con el contenido del Front-End, debemos de instalar un entorno llamado "Node.js" en nuestro sistema, el cual se encarga de proporcionar la capa de servidor del Front-End.

Para instalar este entorno, es necesario descargar dos paquetes llamados "nodejs" y "npm". La descarga e instalación de ambos debe de hacerse ejecutando los siguientes comandos:

$ sudo apt-get install nodejs

$ sudo apt-get install npm

Primero de todo, comprobaremos que "node" se haya instalado correctamente ejecutando el siguiente comando en un terminal: **$ node -v**. Si el paquete ha sido instalado correctamente, deberemos de obtener como resultado, la versión de node que se haya instalado.

De la misma manera, debemos de comprobar que el paquete "npm" haya sido instalado correctamente, ejecutando el siguiente comando: **$ npm -v**. Si el paquete ha sido instalado correctamente, deberemos de ver la versión de npm que haya sido instalada.

Tercer paso:

Una vez disponemos del código del Front-End y de los paquetes necesarios para hacer funcionar este, ya podemos comenzar con la puesta en marcha de este. Primero de todo, debemos de abrir un terminal y situarnos dentro de la carpeta que se nos haya creado con el nombre "front-end".

A continuación, debemos de ejecutar el comando **$ sudo npm install**, para que de esta manera se descarguen e instalen los paquetes necesarios en un nuevo directorio llamado "node_modules".

Cuarto paso:

Si la instalación ha finalizado correctamente, ya podremos inicializar el Front-End para poder acceder a este a través de un navegador. Para inicializar este, debemos de ejecutar el siguiente comando **$ npm start**.

Una vez veamos que el comando se ha ejecutado correctamente, para poder visualizar el Front-End debemos de abrir un navegador y acceder a la siguiente URL: **http://localhost:3001**.

15.2 MANUAL DE INSTALACIÓN DE UN AGENT

La finalidad de este manual es conseguir instalar el software del agent diseñado e implementado en este proyecto para posteriormente poder poner este en marcha y de esta manera poder crear una topología capaz de ejecutar cualquier servicio del catálogo de servicios de la ciudad inteligente.

Un agent puede ser instalado tanto en un ordenador con sistema operativo Raspbian (una Raspberry) como en un ordenador con el sistema operativo Ubuntu.

A continuación, se detallan los diferentes pasos a seguir para poder instalar el agent correctamente y ponerlo en marcha:

Primer paso:

Para poder instalar un agent, primeramente, es necesario descargar e instalar un conjunto de paquetes necesarios para que este pueda ser ejecutado correctamente.

La descarga e instalación de estos paquetes se ha de hacer mediante la ejecución de un *script* programado en el lenguaje *Bash*.

La obtención de este fichero puede hacerse mediante un navegador, o directamente desde un terminal.

Para obtener el fichero des del terminal, debemos de hacer lo siguiente:

- Abrir el terminal y acceder al directorio donde queremos descargar el fichero.
- Una vez nos encontremos en el directorio, debemos ejecutar el siguiente comando para descargar el fichero:

$ wget
https://raw.githubusercontent.com/adrianARL/Agent2.0/master/install_agent.sh

Para obtener el fichero des de un navegador, hay que hacer lo siguiente:

- Acceder al siguiente enlace: https://raw.githubusercontent.com/adrianARL/Agent2.0/master/install_agent.sh
- Una vez abierta la página web, debemos pulsar el botón derecho del ratón y clicar sobre “Guardar como”.
- Finalmente debemos seleccionar un directorio donde guardar el fichero, el cual ha de tener como nombre “install.sh” y hacer clic sobre “Guardar”.

Segundo paso:

A continuación, abre un terminal y accede al directorio donde hayas situado el fichero de instalación. Una vez en el directorio, debemos de dar permisos de ejecución al fichero descargado. Para hacer esto, ejecuta el siguiente comando:
$ chmod +x install_agent.sh.

De esta manera, ya se puede ejecutar el fichero para descargar e instalar las dependencias correspondientes.

Tercer paso:

Para comenzar con la instalación de los diferentes paquetes, debemos ejecutar el fichero. Para ejecutarlo, y siguiendo dentro del mismo directorio donde se encuentra este, debes ejecutar el siguiente comando: **$ ./install_agent.sh**.

Una vez ejecutado el programa, se solicitará la contraseña de usuario para comenzar con el proceso de descarga e instalación de paquetes.

Una vez finalice este proceso, verás el mensaje “Proceso de instalación del agent finalizado” en el terminal donde hayas ejecutado el comando anterior, indicando que este ha finalizado correctamente.

Esta instalación, además de descargar e instalar los paquetes necesarios, también realiza otras acciones necesarias para que el agent pueda funcionar correctamente. Estas acciones son las siguientes:

- Creación de una carpeta llamada “agent”, dentro del directorio “/etc/” de nuestro sistema operativo. Dentro de esta carpeta se encuentra la configuración del agent.
- Creación de una carpeta llamada “codes”, dentro del directorio “/etc/agent/” donde se guardarán los diferentes códigos de los servicios que ejecute el agent.
- Creación de un archivo llamado “map.db”, el cual es una base de datos utilizada para la creación de rutas óptimas de la ciudad. Este archivo se encuentra dentro del directorio “/etc/agent/”.
- Creación de un archivo llamado “start_agent.py”, el cual es un programa que al ser ejecutado pone en marcha un agent. Este último fichero también se encuentra en el directorio “/etc/agent”.
- Creación de un archivo llamado “device.py”, el cual es un programa que sirve para configurar un agent por primera vez y poder solicitar servicios a este. Este fichero es la aplicación cliente de un agent diseñada e implementada a lo largo del proyecto. Este fichero se encuentra dentro del directorio “/etc/agent/”, pero puede moverse a cualquier otro directorio según nos convenga.

Cuarto paso:

Una vez se ha creado el directorio de configuración del agent dentro de nuestro sistema operativo, ya estamos listos para poder ponerlo en marcha. Hay que tener en cuenta que para que el agent pueda ser configurado y ejecutado correctamente, en la base de datos topológica tiene que existir como mínimo el Cloud Agent.

El fichero que debemos ejecutar es el fichero llamado “device.py”. Primero de todo, debemos abrir un terminal y acceder al directorio donde hayamos colocado este fichero. Si este fichero no ha sido movido, por defecto se encuentra en el directorio “/etc/agent/” tal y como se ha mencionado anteriormente.

Una vez situados en el directorio correspondiente, podemos correr el programa ejecutando el siguiente comando: **$ python3 device.py**.

De esta manera, podemos configurar por primera vez nuestro agent para añadirlo a la base de datos topológica y que posteriormente este pueda ejecutar los servicios del catálogo de servicios de la ciudad inteligente.

15.3 MANUAL DE CONFIGURACIÓN DEL CLOUD

El objetivo de este manual es configurar la base de datos topológica. Esta configuración tiene como finalidad añadir un nodo cloud agent a una colección de la base de datos, y también añadir el conjunto de servicios creados a otra colección de la base de datos que será la que hará la función de catálogo de servicios de la ciudad inteligente.

La configuración debe de hacerse en la misma máquina en la que se haya hecho la instalación del Front-End, teniendo en cuenta de que previamente, se ha de haber realizado la instalación de un agent sobre esta, ya que es esta última instalación la que nos crea ciertos directorios necesarios para configurar el cloud.

A continuación, se detallan los diferentes pasos a seguir para poder realizar dicha configuración:

Primer paso:
Primero de todo debemos de descargar el paquete llamado “MongoDB”, el cual nos permitirá crear una base de datos no relacional sobre la cual se crearán la colección que almacena los nodos de la topología como la colección que guarda los servicios. Para hacer esto, dentro de un terminal ejecutamos el siguiente comando:
$ sudo apt-get install mongodb

Podemos comprobar que este se ha instalado correctamente ejecutando el comando **$ mongo –-version**, el cual nos devolverá la versión que ha sido instalada.

Segundo paso:

A continuación, debemos de modificar la configuración del fichero de configuración que nos ha creado MongoDB. Para hacer esto hemos de ejecutar el siguiente comando:

$ sudo nano /etc/mongodb.conf

Una vez se abra el fichero, se ha de modificar la línea con el contenido "bind_ip = 127.0.0.1" por lo siguiente: **bind_ip = 127.0.0.1,ip_frontend**.

El parámetro "ip_frontend" es la dirección IP de la máquina sobre la cual se esta realizando la configuración. Esta IP se puede obtener ejecutando el siguiente comando:

$ hostname -I | awk '{print $1}'

Para guardar los cambios se ha de presionar las teclas Ctrl+X -> Y -> Enter. Para que estos cambios se tengan en cuenta, hemos de reiniciar el servicio de mongo ejecutando el siguiente comando:

$ sudo service mongodb restart

Ejecutando el comando **$ sudo service mongodb status** se puede comprobar que la configuración de mongo ha sido exitosa si nos sale el siguiente mensaje en una parte del output: "Active: active (running)".

Tercer paso:

A continuación, hemos de descargar un script llamado "configure_database.sh", el cual una vez ejecutado nos creará las dos colecciones nombradas anteriormente y añadirá a cada una de ellas el contenido correspondiente.

Para obtener este script, abriremos un terminal y nos situaremos sobre un directorio cualquiera en el que será guardado el fichero ejecutable. A continuación, ejecutaremos el siguiente comando el cual nos descargará el fichero:

$ wget https://raw.githubusercontent.com/adrianARL/database_configuration/master/configure_database.sh

Cuarto paso:

Una vez descargado el fichero, y sobre el mismo directorio que se encuentra este, ejecutaremos el comando **$ chmod +x configure_database.sh** el cual dará permisos de ejecución sobre el fichero para que este pueda ser ejecutado.

Quinto paso:
Finalmente, corremos el script ejecutando el comando siguiente:
./configure_database.sh

15.4 PUESTA EN MARCHA DEL PROYECTO

En este apartado se pretende explicar cómo poner en marcha el proyecto. Para hacer esto, a continuación, se muestran el conjunto de pasos a seguir para poder instalar todo correctamente y crear una topología simple sobre la cual se pueda ejecutar algún servicio de prueba.

A continuación, se muestran los diferentes pasos a seguir:

- Primero de todo, debemos de instalar en una máquina (que no sea Raspberry) con sistema operativo Ubuntu 18.04.3 el Front-End, siguiendo el manual de instalación del apartado “15.1 MANUAL DE INSTALACIÓN DEL FRONT-END”.

- Una vez hecho esto, se ha de instalar un agent en la misma máquina, para que de esta manera se instalen el conjunto de paquetes necesarios y se creen los directorios que un agent necesita para poder funcionar.

- A continuación, y de nuevo sobre la misma máquina, se debe de configurar el Cloud. Con esta configuración se añadirá a la base topológica el nodo Cloud Agent y también se añadirán todos los servicios al catálogo de servicios de la ciudad inteligente.

Una vez realizados los tres pasos anteriores, ya se pueden configurar más agents para que estos sean añadidos a la topología y se puedan ejecutar los diferentes servicios del catálogo de servicios que han sido implementados.

15.5 EJECUCIÓN DE UN SERVICIO DE CONDUCCIÓN

En este apartado se detallan los diferentes pasos a seguir para ejecutar un servicio de conducción autónoma, partiendo de la base que ya se ha instalado correctamente tanto el Front-End, el Cloud y los diferentes agents que tienen que formar parte en la ejecución del servicio.

Los agents que hay que tener configurados para poder ejecutar el servicio de conducción son los siguientes:

- El cloud agent
- Un leader que tenga como tipo de *device* “gestor_total”
- Un agent que tenga como tipo de *device* algún tipo de vehículo (ambulancia, camión basura, etc.)

Teniendo en cuenta los datos anteriores, la topología de red que se debe de haber creado para ejecutar el servicio de conducción hacia un destino es la siguiente:

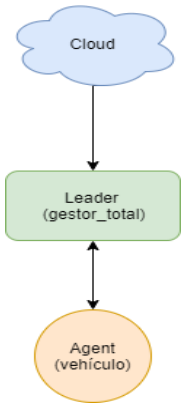

Figura 72: Topología mínima necesaria para ejecutar un servicio de conducción autónoma

A continuación, se detallan los diferentes pasos a seguir para solicitar el servicio de conducción autónoma hacia un destino:

- Primero de todo, debemos de encender cada uno de los agents nombrados anteriormente para que estos puedan comunicarse entre si.

- Para poner en marcha el cloud agent, debemos de abrir un terminal y dirigirnos al directorio donde se encuentre el fichero “device.py” para posteriormente ejecutar el comando **$ python3 device.py** desde un terminal, comando que pone en marcha el cloud agent.

- Para poner en marcha el leader, debemos de conectarnos a la Raspberry Pi que utilizará este de forma remota, mediante *ssh* utilizando el siguiente comando: **$ ssh pi@10.10.176.123**. A continuación, debemos de introducir como contraseña “raspberry” y finalmente dirigirnos al directorio donde se encuentre el fichero “device.py” para ejecutar el comando **$ python3 device.py** y de esta manera poner en marcha el leader.

- Para poner en marcha el agent hemos de conectarnos remótamente a la Rasperry Pi que se encuentra instalada en el vehículo. Para hacer esto debemos de ejecutar el comando **$ ssh pi@10.15.66.217** o **$ ssh pi@10.0.0.200** dependiendo el vehículo al que se quiera conectar. La contraseña que hay que introducir posteriormente es la misma que en el caso anterior, es decir "raspberry". Una vez hecho esto, nos dirigimos al directorio donde se encuentre el fichero "device.py" y lo ejecutamos mediante el comando **$ python3 device.py**.

- Finalmente, y con los nodos necesarios activos, des de la conexión remota establecida con la raspberry del vehículo, y habiendo ejecutado el fichero device.py, ya podemos seleccionar cualquiera de los servicios que nos aparezca en la lista de servicios. Para hacer esto, nos posicionamos sobre el servicio que queremos ejecutar utilizando las flechas del teclado y posteriormente presionamos la tecla Enter para solicitar la ejecución de este. En este caso, deberiamos de seleccionar el servicio de conducción hacia un destino e introducir como parámetro cualquier posición de la ciudad. Las diferentes posiciones que se pueden introducir como final son las siguientes:
    - N1, N2, N3, N4, N5, N6, N7 (calle horizontal superior)
    - W1, W2, W3, W4, W5 (calle vertical izquierda)
    - S1, S2, S3, S4, S5, S6, S7 (calle horizontal inferior)
    - E1, E2, E3, E4, E5 (calle vertical derecha)
    - C1, C2, C3, C4 (calle vertical central)
    - B1, B2, B3, B4, UB, B5, B6, B7 (puente horizontal)
    - NW, NE, SW, SE (esquinas de la ciudad)

- Una vez seleccionada alguna de las posibles posibles posiciones finales de la ruta, se debe de presionar sobre la tecla Enter de nuevo para que el servicio de conducción autónoma hacia un destino comience a ser ejecutado.