16 DE JULIO DE 2020

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

isa

ai2
INSTITUTO DE AUTOMÁTICA E INFORMÁTICA INDUSTRIAL

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CONTROL CENTRALIZADO DE AFLUENCIA DE ROBOTS MÓVILES EN UNA PLANTA CON MÁS DE UNA ESTANCIA

TRABAJO FIN DE MÁSTER

AUTORA: MARTA PAYÁ TORMO
TUTOR: MARTÍN MELLADO ARTECHE

Dedicatoria

A mis dos hermanos postizos, quienes saben que sin ellos esto no habría sido posible.

A mis padres y mis hermanas, por la eterna confianza y el infinito apoyo que me dan.

A mi pareja, por creer en mí y animarme siempre a seguir.

Agradecimientos

A Don Martín Mellado Arteche, por hacer posible este trabajo y todo lo que conlleva.

A todos mis compañeros del proyecto ENDORSE, en concreto a Don Marc Boch Jorge de la empresa Robotnik, Don Francisco Fraile Gil del CIGIP y a mi compañera Sandra Moreno Olivares, por las ayudas incansables, sin quienes este trabajo no habría salido adelante.

A mis compañeros del ai2, por su disponibilidad y sobre todo su dedicación a la investigación y a la ciencia.

Resumen

Este trabajo de fin de máster se va a realizar en el Instituto Universitario de Automática e Informática Industrial (ai2) de la Universitat Politècnica de València (UPV), el cual trabaja con la tecnología más puntera para aproximar los resultados de los trabajos I+D+I que se realizan, a las empresas e instituciones.

En este caso, los avances alcanzados en este trabajo se incluirán en el proyecto europeo ENDORSE (Safe, Efficient and Integrated Indoor Robotic Fleet for Logistic Applications in Healthcare and Commercial Spaces), ya que el ai2 está cooperando junto con otras universidades y empresas nacionales e internacionales para crear un sistema que facilite la labor sanitaria en entornos interiores de trabajo como pueden ser hospitales y residencias de ancianos, con el fin de que el personal de estos lugares trabaje de forma colaborativa con robots que se moverán por una planta con varias estancias para facilitar el trabajo y ayudar en tareas triviales y repetitivas.

Aquí es donde tiene cabida el desarrollo de este trabajo fin de máster, el cual pretende desarrollar una estrategia centralizada de control de afluencia de robots, en este caso del RB-1 (desarrollado por la empresa valenciana Robotnik), de forma que ciertas zonas o estancias de una planta estén restringidas a un número concreto de robots para que estos no obstaculicen el paso al tránsito de personas, o para evitar colisiones o bloqueos entre ellos en zonas de difícil paso o acceso. El desarrollo del proyecto abarcará desde las comunicaciones entre los robots y el nodo máster, hasta la modificación y adición de código que se ejecutará en el middleware de ROS para que cada robot sea capaz de enviar su localización al máster y éste, a modo de semáforo, controle el acceso o restricción de los robots a las distintas zonas del mapa.

A nivel educativo, con el fin de afianzar y ampliar los conocimientos adquiridos en el máster de Automática e Informática Industrial, en este proyecto van a ser necesarias las lecciones aprendidas en asignaturas como Redes y Sistemas distribuidos para control, en lo referente a las comunicaciones de los robots, o Robótica de Servicios para la programación de los robots, para lo cual se hará uso del *middleware* ROS (Robot Operating System).

Palabras clave

ENDORSE, Robótica Móvil, Robots Colaborativos, Sistema Centralizado, ROS, Control de Afluencia.

Resum

Aquest treball de fi de màster es realitzarà en l'Institut Universitari d'Automàtica i Informàtica Industrial (ai2) de la Universitat Politècnica de València (UPV), el qual treballa amb la tecnologia més capdavantera per a aproximar els resultats dels treballs I+D+I que es realitzen, a les empreses i institucions.

En aquest cas, els avanços aconseguits en aquest treball seràn inclosos en el projecte europeu ENDORSE (Safe, Efficient and Integrated Indoor Robotic Fleet for Logistic Applications in Healthcare and Commercial Spaces), ja que el ai2 està cooperant juntament amb altres universitats i empreses nacionals i internacionals per a crear un sistema que facilite la labor sanitària en entorns interiors de treball com poden ser hospitals i residències d'ancians, amb la finalitat de que el personal d'asquests llocs treballe de manera col·laborativa amb robots que es mouran per una planta amb diverses estades per a facilitar el treball i ajudar en tasques trivials i repetitives.

Ací és on té cabuda el desenvolupament d'aquest treball fi de màster, el qual pretén desenvolupar una estratègia de control d'afluència de robots, en aquest cas del RB-1 (desenvolupat per l'empresa Valenciana Robotnik), de manera que certes zones o estades d'una planta estiguen restringides a un nombre concret de robots perquè no obstaculitzen el pas al trànsit de persones, o per a evitar col·lisions o bloquejos entre ells en zones o de difícil pas o accés. El desenvolupament del projecte abastarà des de les comunicacions entre els robots i el node màster, fins la modificació i addició de codi que s'executarà en el middleware de ROS perquè cada robot siga capaç d'enviar la seua localització al màster i aquest, a manera de semàfor, controle l'accés o restricció dels robots a les diferents zones del mapa.

A nivell educatiu, amb la finalitat d'afermar i ampliar els coneiximents adquirits en el màster d'Automàtica i Informàtica Industrial, en aquest projecte seran necessàries les lliçons apreses en assignatures com Xarxes i Sistemes distribuïts per a control, referent a les comunicacions dels robots, o Robótica de Servicis per a la programació dels robots, per a això es farà ús del *middleware* ROS (Robot Operating System).

Paraules clau

ENDORSE, Robòtica mòbil, Robots Col·laboratius, Sistema centralitzat, ROS, Control d'Afluència

Abstract

This final master's project will be carried out at the Instituto Universitario de Automática e Informática Industrial (ai2) of the Universitat Politècnica de València (UPV), which works with the latest technology to approximate the results of I+D+I to companies and institutions.

In this case, the progress achieved in this work will be included at the European project ENDORSE (Safe, Efficient and Integrated Indoor Robotic Fleet for Logistic Applications in Healthcare and Commercial Spaces), because the ai2 is cooperating together with other national and international universities and companies. The aim is to create a system that facilitates health work environments on hospitals and nursing homes. So, the staff in these places could work collaborating with robots, which could be in charge of repetitive tasks, making the work easier for nurses and hospital staff.

On this point, the development of this master´s thesis will take place. The objective is to develop a strategy for controlling the influx of robots, in this case the RB-1 (developed by the Valencian company Robotnik). A specific number of robots will be associate to certain areas or room from a floor. With this, the passage of people will not be hinder and collisions or blockages in areas with difficult access will be avoided. The development of the project will start on the communications between the robots and the master node and will finish on the modification and addition of a code that will be executed in the ROS middleware. Consequently, each robot will be able to send its location to the master and control the access or restriction of the robots to the different areas of the map, as a traffic light.

From an educational point of view, in order to support and expand the knowledge acquired in the Master of Automation and Industrial Computing, this project will require execute the concepts learned in subjects such as Networks and Distributed Systems for control (in relation to communications robots), or Service Robotics for programming robots, for which the ROS (Robot Operating System) middleware will be used.

Key words

ENDORSE, Mobile Robotics, Colaborative Robots, Centralzed System, ROS, Influx Control.

Tabla de Contenido

1. Introducción

Durante años, los robots han sido los grandes pilares sobre los que apoyar las bases de la industria a nivel mundial. Desde el invento de la máquina de vapor en el siglo XVIII y su implicación en el inicio de la Revolución Industrial, hasta la actualidad, donde estos han salido del entorno industrial adentrándose cada día más en nuestras vidas.

Según el estudio anual realizado por la *Asociación española de robótica y automatización tecnologías de la producción,* el parque de robots en España entre los años 2003 y 2016 ha sufrido una notable tendencia creciente, tal y como podemos observar en la siguiente gráfica:

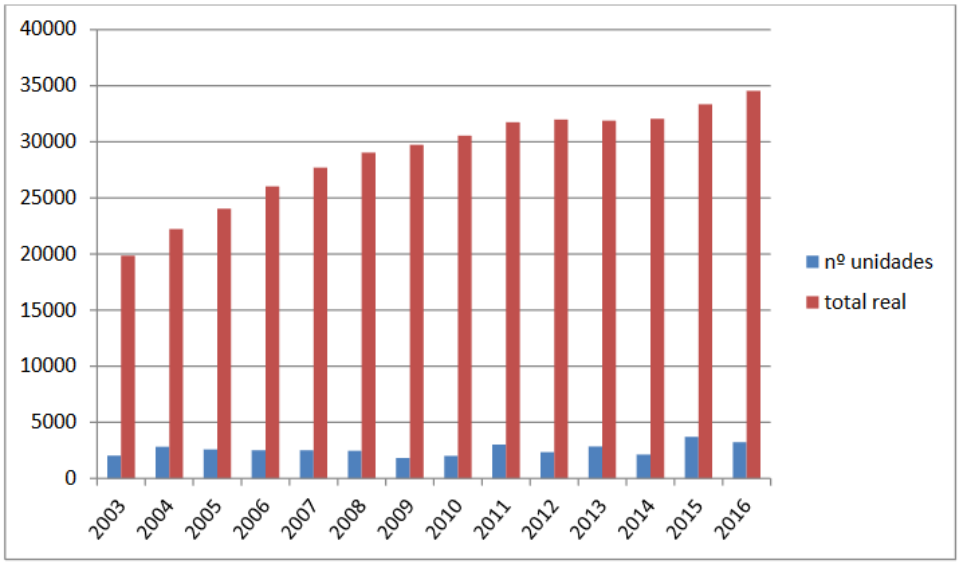

Imagen 1.1. – Evolución del parque de robots en España. (Estudio anual, 2017). Recuperado de: "Estadísticas de robótica industrial en España"

Es importante destacar estos datos tan positivos para el sector de la robótica, ya que reflejan que la repercusión de los robots va a seguir en aumento, aplicándose cada vez en más ámbitos de la industria.

Sin embargo, este proyecto pretende trabajar sobre robots colaborativos, y nos preguntaremos, ¿qué es un robot colaborativo? La Federación Internacional de Robótica admite dos acepciones o tipos de robots colaborativos:

- "Un primer grupo engloba a los robots diseñados para uso colaborativo que cumplen con la norma ISO 10218-1, que especifica los requisitos y las pautas para un diseño seguro, medidas de protección e información de uso."
- "Un segundo grupo engloba a los robots diseñados para uso colaborativo que no cumplen con la norma ISO 10218-1. Esto no implica que estos robots no sean seguros, ya que pueden seguir diferentes estándares de seguridad, por ejemplo, nacionales."

Asociación Española de Robótica y Automatización, IFR y MarketsandMarkets. (2020). *Robótica Colaborativa*.

Según esta misma asociación, nos encontramos ante un "Crecimiento sin precedentes", donde "un mercado de 710 millones de dólares evolucionará hasta superar los 12.303 millones en 2025", es por ello que, cada vez es más interesante adentrarse en el mundo de la robótica colaborativa, sobre todo con la llegada de la Industria 4.0. que pretende revolucionar este sector.

A pesar de los datos de crecimiento aportados, la robótica colaborativa presume de mala fama entre sus compañeros humanos, ya que muchas personas consideran que la robótica colaborativa ha llegado a nuestras vidas para eliminar puestos de trabajo, estando esta postura muy lejos de la realidad. La principal función de la robótica colaborativa es facilitarnos el trabajo, principalmente aquellos que son considerados tediosos y repetitivos, encontrando ejemplos incluso en nuestro hogar en forma de robots de cocina o robots aspiradores.

Este campo de la robótica está teniendo un gran auge en la industria, sin embargo, cada vez es más común encontrarnos ante una robótica colaborativa social, donde los robots interactúan con humanos fuera de las fábricas. Es aquí donde nuestro proyecto ha encontrado un espacio.

El propósito de este trabajo, a través del proyecto europeo ENDORSE, y con la ayuda del Instituto de Automática e Informática Industrial (ai2), los cuales introduciremos en el segundo apartado, es implementar una aplicación capaz de controlar una flota de robots, para que cooperen entre ellos y faciliten el trabajo del personal responsable del lugar donde vayan a ser implantados. Este proyecto ha sido pensado para ser usado principalmente en centros sanitarios como hospitales o residencias, en concreto se va a trabajar en colaboración con la Fundación Ave María de Barcelona.

Los objetivos van a consistir en un estudio minucioso de las distintas partes implicadas en el proyecto, así como la utilización de las partes desarrolladas o en proceso de desarrollo. Primero se realizará un estudio del dispositivo que centraliza la flota de robots, del cual se encarga el Centro de Investigación en Gestión e Ingeniería de Producción de la Universitat Politècnica de València, llegando a acuerdos con ellos con respecto a los mensajes que deberá intercambiar con los robots.

También se cooperará con la empresa valenciana Robotnik para implementar un método de comunicación entre los distintos componentes.

Otro de los objetivos es estudiar el funcionamiento del middleware ROS, alcanzando la desenvoltura suficiente para poder desarrollar una aplicación que funcione en el mismo.

El resultado final deberá ser una aplicación que permita restringir el acceso de los robots a algunas estancias cuando estas presenten cierto número de ocupación, con el único fin de evitar, en la medida de lo posible, que estos robots obstaculicen el trabajo de otros robots o del personal que se encuentre trabajando.

A pesar de que se va a abarcar el problema de forma tangencial, hablando de todas las partes implicadas, nos vamos a centrar principalmente en el trabajo sobre los robots. Haciendo uso, inicialmente, de simulaciones, y, a ser posible, realizar pruebas en robots físicos.

Comenzaremos poniéndonos en contexto tanto a nivel del proyecto ENDORSE, como con respecto a este trabajo fin de máster. Continuaremos con un análisis del problema, donde se tratará de analizar e identificar las posibles soluciones, proponer las que se han escogido, así como el plan de trabajo a seguir.

Seguiremos con un estudio minucioso de las herramientas de trabajo que se van a necesitar, tanto a nivel hardware como a nivel software, para concluir en su uso en el diseño y desarrollo de la solución propuesta. Finalmente, hablaremos de las pruebas realizadas y los resultados obtenidos en las mismas, y de la implantación del proyecto, sobre todo la repercusión del COVID-19 en esta.

2. Contexto y entorno de trabajo

El desarrollo de este proyecto ha sido posible gracias al contrato firmado a través de Garantía Juvenil, una iniciativa europea que pretende facilitar el acceso de los jóvenes al mercado de trabajo. Este servicio permite que jóvenes no ocupados accedan a un puesto de trabajo que les facilite la inserción al mercado laboral. Gracias a él fue posible tener un hueco en el proyecto europeo ENDORSE (*Safe, Efficient and Integrated Indoor Robotic Fleet for Logistic Applications in Healthcare and Commercial Spaces*) trabajando en el Instituto de Automática e Informática Industrial (ai2) de la Universitat Politècnica de València. El ai2 se encuentra Ciudad Politécnica de la Innovación, dentro del campus de la Universidad, y cuenta con más de cien investigadores en sus proyectos. Además, cubre un gran abanico en áreas de investigación, a saber, Control de Procesos, Informática Gráfica y Multimedia, Informática Industrial, Robótica y Visión por Computados, cuya sinergia permite aportar a los proyectos de capacidades complementarias y multidisciplinares.

2.1. Proyecto ENDORSE

El proyecto ENDORSE fue concebido para ser implantado en hospitales, hoteles, o edificios cerrados que puedan requerir de sus servicios. ENDORSE es una colaboración entre las universidades de distintos países de Europa como la Universidad de Chipre, la Universidad de Orleans o la Universidad Politécnica de Valencia, además del trabajo de distintas empresas, como la empresa valenciana Robotnik, entre otras. El trabajo de estas instituciones pretende desarrollar un sistema capaz de orientarse por las distintas estancias de una planta de forma que un determinado número de robots RB-1 (robot desarrollado por la empresa Robotnik, del cual hablaremos en el apartado siguiente), con distintos dispositivos y funcionalidades, puedan moverse a distintos puntos en los que se requiera de su actuación sin entorpecer el trabajo de personas u otros robots. Estos robots serán gestionados por un *Fleet Management System* encargado de que las tareas solicitadas se cumplan en tiempos mínimos y de la forma más eficiente posible.

Este proyecto se centra en cumplir distintos objetivos, organizados en *World Pakcages*. A su vez, estos objetivos están clasificados en dos tipos distintos, objetivos de investigación y objetivos técnicos. Comenzaremos hablando de los objetivos de investigación (OI), para continuar hablando de los técnicos (OT), y finalmente clasificarlos en sus respectivos *World Packages.*

- **OI 1**: Llevar a cabo la investigación y el desarrollo hacia un sistema de robótica logística rentable para espacios comerciales y sanitarios. Buscando la eficiencia en los tiempos de envío y recursos ocupados, para minimizar los costes.
- **OI 2**: Avanzar en la interacción Humano-Robot para aplicaciones logísticas donde los robots navegan en espacios con mucho tránsito de personas.

- **OI 3**: Investigar y desarrollar métodos de localización para la ubicación de flotas de robots móviles en espacios interiores.
- **OI 4**: Introducir servicios en la nube que reúnen soluciones de salud, robótica y logística.

En lo referente a los aspectos técnicos, distinguimos los distintos objetivos:

- **OT 1**: Integrar un sistema de gestión de flotas (*Fleet Management System,* FMS) eficiente y fácil de usar basado en la nube.
- **OT 2**: Desarrollar un paquete de software de navegación con conciencia humana (integrado en ROS) que permita una interacción segura y que minimice el estrés y la incomodidad de los humanos.
- **OT 3**: Desarrollar una estación móvil e inteligentes para diagnósticos médicos.
- **OT 4**: La investigación y los resultados técnicos del proyecto ENDORSE se validarán durante los estudios piloto en las instalaciones del centro de salud Fundación Ave María.

Para cumplir estos objetivos es necesaria una gestión de proyectos sólida, para ello se han definido los *World Packages,* los cuales van a permitir estructurar los distintos objetivos y repartirlos de forma eficiente en los tiempos del proyecto. Para ello, adjuntamos el esquema en la imagen 2.1. propuesto para el proyecto, en el que podemos observar las distintas tareas y objetivos, así como la estructura del mismo, todo ello incluido en sus respectivos paquetes.

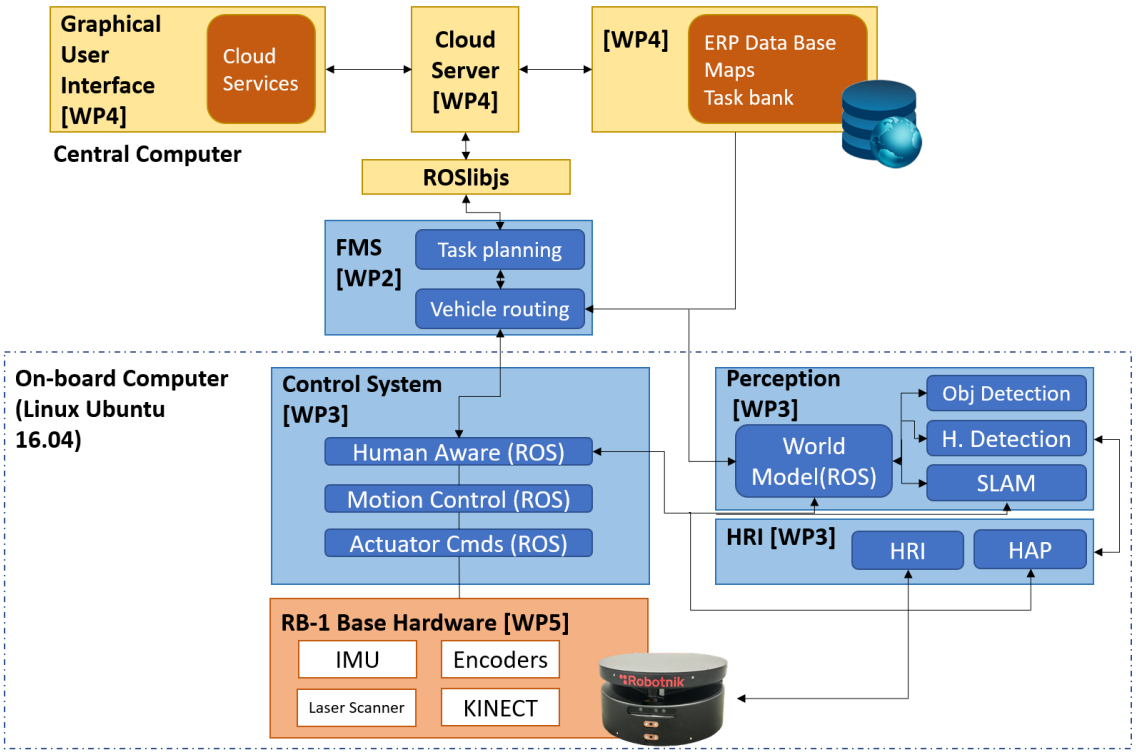

Imagen 2.1. – Estructura del proyecto ENDORSE.

En la imagen 2.1. podemos observar la estructura del proyecto ENDORSE, donde aparecen cuatro World Packages, del WP2 al WP5. Esto es debido a que el WP1 hace

referencia a la organización de todo el proyecto, por parte de cada uno de los miembros. Como podemos comprobar, estos paquetes se dividen desde más bajo a más alto nivel, siendo el WP5 el más bajo y el WP4 el más alto; el WP2 hace referencia al sistema de centralización del proyecto y el WP3 al sistema de control que se ejecutará en los robots RB-1 y que interactuará con el sistema de centralización (FMS).

2.2. Contexto del trabajo

Una vez se ha introducido el entorno de trabajo, es imprescindible poner en contexto este proyecto, lo cual nos lleva directamente a hablar de la robótica móvil. Un robot móvil consiste en una máquina capaz de moverse de un lugar a otro, es además autónomo cuando no requiere de la acción de las personas para alcanzar este propósito. En nuestro caso, aspiramos a trabajar con robots móviles y autónomos; y además de eso con una casi segura interacción con humanos, lo cual es importante destacar ya que las restricciones de seguridad son más duras en este caso.

Dentro de la robótica móvil, encontramos distintos tipos, principalmente clasificados por la forma que tienen de desplazarse de un punto a otro; cabe destacar entre ellos por ser los más recurridos los robots con ruedas, robots oruga o con cadena, robots con patas, robots aéreos, robots marinos o robots humanoides.

Con respecto a la autonomía dentro de los robots móviles, se ha avanzado mucho en los últimos años; sin embargo, es importante destacar que para que esta autonomía sea posible, es importante contar con una sensórica que lo permita. El sensor más importante en este caso es el sensor de distancia, el cual permitirá saber al robot la distancia a un objeto; por supuesto, también es importante la disposición de dicho sensor en el propio chasis del robot. Con respecto a la geolocalización, necesaria para que el robot reconozca en qué posición se encuentra, debe ser con respecto a un sistema de referencia, este sistema de referencia suele ser un mapa, local al robot, como podría ser la estancia en la que se va a mover, o global, como podría ser el mundo, para ello sería necesario un sensor GPS.

En resumen, los sensores de los que dispone un robot autónomo son imprescindibles y es necesario estudiarlos a fondo, es por ello que se les dedicará un apartado. A medida que se añaden nuevos sensores al sistema, resulta más fiable su autonomía, sin embargo, los datos que gestionar aumentan, y con ellos el coste computacional.

Como ya hemos comentado, la robótica móvil es muy recurrida en esta última década, y con ella, la robótica móvil autónoma, pero ¿qué ocurre con la interacción de estos robots autónomos y los humanos? Para responder a esta pregunta hemos indagado en las últimas investigaciones al respecto, tratando de buscar una pequeña clasificación que nos pueda ayudar a entender la interacción de los RB-1 con las personas, y sobre todo la normativa vigente con respecto a tales interacciones, lo cual será muy importante a la hora de implementar una solución.

Tras esta búsqueda hemos encontrado un artículo muy interesante de los autores Saifuddin Mahmud, Xiangxu Lin y Jong-Hoon Kim [5] *Interface for Human machine Interaction for assistant devices: A Review,* publicado este mismo año 2020, donde se habla de la interacción humano-máquina, HCI (*Human Computer Interface),* y que nos va a servir para ponernos en contexto. Para empezar, clasifican esta interacción en cinco tipos: señal cerebral, gestual, ocular, tangible e híbrida. A priori podemos deducir que inicialmente la que más nos va a servir es la tangible, y en ciertos casos la gestual.

En relación con la interacción tangible, lo más importante es que se trate de una interfaz amigable e intuitiva en el caso de que se desarrolle una aplicación. Es interesante el desarrollo de una aplicación ya que está demostrado que los usuarios con una experiencia media-baja, que es el conocimiento que se presupone al mercado que va destinado nuestro producto, en nuevas tecnologías, se encuentran más cómodos con interfaces tangibles.

Con respecto al marco legal vigente en España, en relación a los robots colaborativos, que como bien sabemos son aquellos destinados a interactuar con humanos, debemos tener en cuenta las siguientes leyes. Según la Directiva de Máquinas 2006/42 un robot es considerado legalmente como una cuasi-máquina, ya que se asume que la empresa que lo adquiere debe darle una aplicación en el proceso. Este también es nuestro caso. Cuando adquirimos los RB-1, no tienen una función concreta, nosotros se la vamos a asignar durante el desarrollo del proyecto.

En el caso de los COBOTS, lo que ha ocurrido es que la tecnología ha ido por delante de la normativa, no habiendo una norma que defina los requisitos de seguridad para los COBOTS, y siendo la directiva de máquinas UNE-EN ISO 10218-2 para las células robóticas quien lo rige. Es por ello que la evaluación de riesgos debe apoyarse en la ISO/TS 15066, que especifica los requisitos de seguridad y parámetros de funcionamiento de los COBOTS, de los cuales a continuación vamos a mencionar los más importantes.

Hay cuatro operaciones de obligatorio cumplimiento para una aplicación colaborativa. La primera es que debe existir una parada monitorizada de emergencia, comúnmente conocida como Seta de emergencia; también debe existir una operación de guiado manual, la posibilidad de controlar la velocidad y la separación robot-humano, así como la limitación de potencia y fuerza. Todas estas normas deberán tenerse en cuenta a la hora de desarrollar la aplicación, así como en qué partes del cuerpo puede colisionar el robot con las personas y tratar que esta situación no se dé.

Por otro lado, también resulta necesario ponernos en contexto en lo referente a soluciones y algoritmos centralizados de navegación. Para ello, volveremos a realizar una búsqueda exhaustiva en artículos de investigación, y encontramos varios estudios interesantes; sin embargo, uno en concreto llama nuestra atención. Un estudio sobre navegación autónoma centralizada de múltiples robots, presentado por primera vez en junio del 2019 en la Conferencia Europea de Control (ECC) [3], en el que se habla, además, de la asignación de tareas, muy interesante para nuestro proyecto. En este

artículo se versa sobre dos tipos de planificadores, en función de si hay o no conocimiento previo del entorno, reactivos o deliberativos. Además, estos últimos se pueden descomponer en dos fases: planificación y ejecución. Aunque los planificadores deliberativos son capaces de optimizar trayectorias en función de distintos criterios, no tienen en cuenta posibles cambios en el entorno. Por contraposición, los planificadores reactivos, son capaces de evitar obstáculos, pero no pueden optimizar entre distintas rutas porque no tienen una fase de planificación. Un ejemplo de uso de este tipo de planificación es el SLAM (*Simultaneous Localization and Mapping*), el cual requiere de un elevado nivel computacional. Es por ello que la mejor solución, tal y como se comenta en el artículo, es la híbrida entre deliberativos y reactivos.

Teniendo en cuenta todo lo anterior, podemos comenzar a pensar nuestra solución. Deberemos seguir todas las normas y estandarizaciones relacionadas con los robots colaborativos y así desarrollar una aplicación amigable con el usuario que solucione el problema de control de afluencia en un hospital de la forma más eficaz posible. Se ha decidido desde el inicio que la solución sea centralizada ya que así es posible tener siempre una visión global de cada problema. Para hablar de ello hemos dedicado un apartado, en el que abarcaremos la propuesta de solución realizando un análisis de riesgos y decidiendo cuál de todas las soluciones posibles es la óptima para nuestro proyecto.

3. Análisis del problema

El problema que se nos presenta consiste en desarrollar una aplicación, trabajando mayoritariamente a través de simulaciones, a causa de la imposibilidad de acudir al laboratorio, centralizada, en la que los robots son capaces de moverse en el mapa sin ser un obstáculo para las personas ni para el resto de los robots que están realizando sus propias tareas.

Con el fin de evitar estas obstaculizaciones, se ha decidido desarrollar una solución que permita restringir zonas muy transitadas y de poco espacio a un número limitado de robots, de forma que en el momento en el que se alcance un aforo máximo en cierta zona, el robot que trate de entrar deberá esperar a que salga algún robot para poder hacerlo.

3.1. Análisis de riesgos

Los riesgos con los que debe lidiar este trabajo son bastante claros. Comenzaremos analizando el alcance del proyecto, que, puesto que pretende ser implantado en edificios como hospitales y hoteles, donde existe un importante tránsito de gente, incluso de gente con posibles dificultades motrices (pacientes enfermos o gente mayor), es de obligatorio cumplimiento un sistema capaz de evitar obstáculos, detectar personas y que en ninguno de los casos pueda existir una colisión. Ante este riesgo, este proyecto es en sí mismo una solución, ya que pretende restringir zonas conflictivas para reducir la posibilidad de colisión, esto sumado al algoritmo de navegación del que dispone el robot, más el trabajo de los compañeros del proyecto ENDORSE en identificación de personas, hace que este riesgo esté más que cubierto.

Por otro lado, otro riesgo muy a tener en cuenta es el de la seguridad a nivel de red. Puesto que se está trabajando con un gran número de robots, y no puede ocurrir que una persona con intenciones maliciosas pueda acceder al sistema, pudiendo manejar los robots a su gusto, y provocar accidentes. Estos riesgos se tendrán más en cuenta a la hora de implantar el proyecto, creando una red privada y segura. Sin embargo, a la hora de elegir una solución también es importante tenerlo en cuenta ya que en función de los componentes que se elijan, el resultado será más o menos fiable.

Para finalizar, al tratarse de una solución centralizada, es muy importante barajar la posibilidad de que el dispositivo que centraliza el sistema pueda fallar, lo que provocaría un error fatal, ya que, a pesar de ceder una cierta autonomía a los robots, estos esperan una comunicación con el que, a partir de ahora, llamaremos el nodo máster. En el caso de que esto ocurriera, siempre dependiendo de la programación de la aplicación, es muy probable que al cabo de un tiempo los robots se acabaran perdiendo, y no sabiendo cuál es su tarea. En definitiva, si existe un error en el nodo máster, todo el sistema se viene

abajo, con lo cual hay que tenerlo muy en cuenta a la hora de elegir una solución para que sea lo más fiable y tolerante a fallos posible.

3.2. Identificación y análisis de soluciones posibles

Para identificar las posibles soluciones, primero debemos estructurar el proyecto y diferenciar los distintos niveles y capas. De esta forma podremos distinguir los posibles problemas que se nos puedan presentar y las posibles soluciones a los mismos. De una forma simplificada, podríamos representar la estructura de la siguiente forma:

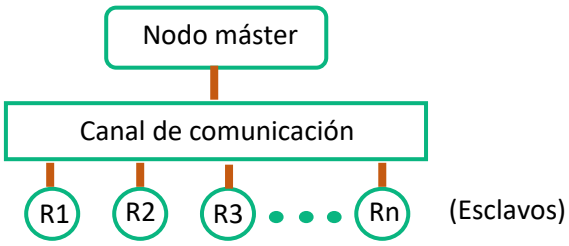

Imagen 3.1. – Estructura del proyecto.

Vamos a hacer una división de arriba abajo, comenzando por el nodo máster y acabando en los esclavos. Para comenzar, con respecto al nodo máster, en referencia al nivel físico, podría dividirse el problema entre dos posibles opciones. La elección de un dispositivo y ejecutar la aplicación a nivel local, o el uso de un servicio en la nube.

En el primer caso, habría que hacer un análisis del nivel de cómputo que requeriría la aplicación, elegir el lenguaje que se va a utilizar para el desarrollo de la misma y el sistema operativo sobre el que se ejecutará, y sabiendo todo esto decidir qué componentes son los mejores calidad-precio o rentabilidad-precio. Una vez realizado este análisis ya podemos elegir un dispositivo donde ejecutarlo. Entre las distintas opciones que nos presenta el mercado actual, encontramos ejemplos tan famosos como la Raspberry Pi o la BeagleBone Black, o incluso existe la posibilidad de diseñar y quemar nuestra propia placa, cuyo coste se dispara pero que puede ser interesante ante un despliegue a gran escala. En el caso de elegir esta opción, es muy importante tener en cuenta que el sistema debe estar replicado de forma persistente, para que en el caso de que ocurra algún fallo, haya consistencia y el sistema continúe funcionando.

Por contraposición, en el caso de decidir ejecutar nuestro nodo máster en un servicio en la nube, nos podríamos olvidar de todo lo anterior, y solo deberíamos preocuparnos en disponer de acceso a Internet. Esto hace que la elección de esta solución sea muy sustanciosa, ya que el propio servicio que se contrata para ejecutar nuestra aplicación se encarga de asignarnos una capacidad de cómputo suficiente y necesaria para que no haya problemas, además de que se encarga de replicar el sistema. Entre los principales proveedores de servicios en la nube, tenemos Microsoft, Amazon o IBM.

Con respecto al canal de comunicación, también diferenciamos entre dos posibilidades. La primera consiste en el uso de un sistema *multi-master* y la segunda en el uso de un

puente MQTT, en el que se asocian *topics* en ROS a *topics* de MQTT para que pueda haber comunicación entre el nodo máster y lo esclavos.

El sistema *multi-master* es una solución proporcionada por el propio ROS, que se sostiene sobre la idea de que cada dispositivo ejecutándose sobre ROS es en sí mismo un nodo máster. Como bien sabemos, según el modelo maestro-esclavo, un dispositivo es el encargado de controlar el proceso, mientras que el resto obedece órdenes. El echo de que todos los robots sean maestros complica el problema, ya que todos ellos en su rol de maestro no pueden ejercer como maestros de otros maestros. Para solucionar este problema, los desarrolladores de ROS crearon el paquete *multimaster_fkie*, el cual, tras unas configuraciones, permite crear una red de maestros capaces de comunicarse entre sí. Sin embargo, esto no es realmente lo que queremos, porque recordemos que nuestro sistema es centralizado, y ya tenemos un nodo máster asignado; esto significaría que este nodo debería soportar ROS para poder comunicarse con el resto de los esclavos. En la imagen 3.2. podemos observar lo que trata de conseguir esta solución. Cada una de las redes locales sería un robot con sus distintos sensores, y cada robot se comunicaría con el resto a través de un canal común con *topics* compartidos.

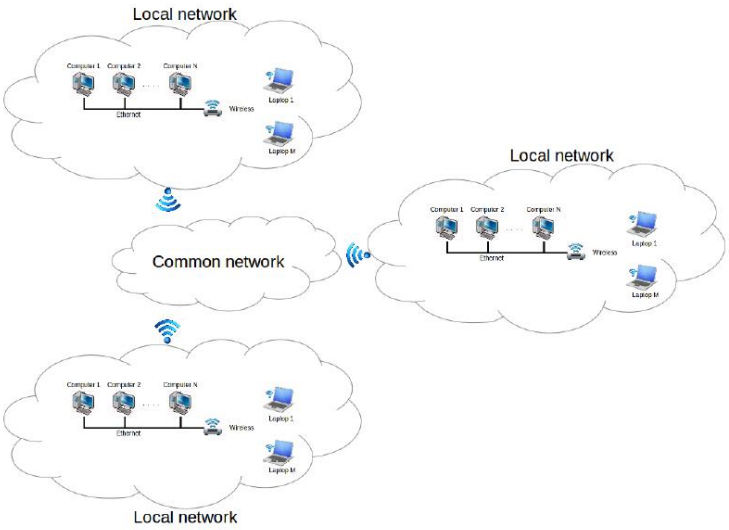

Imagen 3.2. – Estructura multi-mater. Hernandez, S. y Herrero, F. (2015). Imagen de "Multi-master ROS systems"

En el caso de elegir la segunda opción, el uso de un puente MQTT, el problema se simplifica, ya que el modelo que utilizan ambos sistemas, publicación-suscripción, es el mismo. MQTT tiene numerosas ventajas, entre ellas el hecho de que es muy ligero, lo que le permite ser usado en sistemas con pocos recursos. Además, tiene muy buenas respuestas en entornos de poca banda ancha. También permite la posibilidad de encriptar la mensajería, lo cual es muy importante sobre todo en un entorno con potenciales peligros como es un hospital en el que se va a desplegar un elevado número de robots. Por todos estos motivos, la solución que implica el uso de un puente MQTT es muy interesante.

Para finalizar, podría plantearse la disyuntiva de la elección de los robots que se van a utilizar, sin embargo, este problema ya se planteó al inicio del proyecto ENDORSE y se decidió que se iba a usar el robot RB-1 desarrollado por la empresa valenciana Robotnik, el cual ya viene preparado para ser utilizado y con ROS instalado, por lo que a este nivel de desarrollo, solo se nos presentan las posibilidades entre elegir programar nuestra aplicación en Python o en C++, ambos lenguajes compatibles con ROS, y por supuesto compatibles entre ellos, por lo que no es una decisión trascendental.

3.3. Soluciones propuestas

Con respecto al primer problema planteado, se propone utilizar un servicio en la nube. Los motivos son obvios, esta solución facilita mucho el trabajo ya que nos permite preocuparnos solo por el desarrollo de la aplicación sin tener en cuenta posibles fallos en el hardware. A partir de ahora solo habrá que decidir cuál de los mencionados nos ofrece unas mejores prestaciones calidad-precio.

En lo referente al canal de comunicación, se ha decidido utilizar un puente MQTT. Esta decisión ha sido tomada porque tras analizar la opción de *multi-master* llegamos a la conclusión de que la configuración era muy tediosa, además proporciona un funcionamiento dudoso, ya que el propio desarrollo de la aplicación de los robots se complica si se elige esta solución, sobre todo cuando escalamos el problema a un mayor número de robots. Sin embargo, con el puente MQTT el problema se simplifica enormemente. El principal reto será el desarrollo del propio puente, sin embargo, una vez hecho esto, usarlo sería muy sencillo y facilitaría la programación de la aplicación.

Para finalizar, con respecto a la programación de los robots, como ya hemos comentado en el apartado anterior, la decisión es trivial puesto que podemos utilizar el lenguaje que mejor nos convenga en cada momento en función del problema, sin embargo, vamos a tender más a utilizar Python por su simplicidad y legibilidad. Solo debemos tener en cuenta que el sistema operativo que corre dentro de los robots es Linux y sobre él, ROS kinetic.

3.4. Plan de trabajo

El plan de trabajo que se pretende seguir es, colaborando con los participantes del proyecto ENDORSE, compartir las distintas necesidades para el desarrollo de la aplicación centralizada y cuando se disponga de la aplicación que se ejecutará en el nodo máster, y el puente MQTT, comenzar con la programación de los robots.

Con respecto al nodo máster, se va a colaborar con el Centro de Investigación en Gestión e Ingeniería de Producción (CIGIP) de la Universitat Politècnica de València (UPV) con el fin de concretar las funcionalidades y servicios que deberá de prestar este nodo, para que dichas herramientas puedan ser utilizadas por los robots. A esta aplicación la han llamado *Fleet Management System* (FMS), el cual será capaz de, sabiendo las distintas

ubicaciones de los robots, generar rutas eficientes para las distintas tareas solicitadas por el usuario.

Para el desarrollo del puente MQTT, se va a colaborar con la empresa valenciana Robotnik, también colaboradora del proyecto ENDORSE, para poner en marcha un puente que haga posible la comunicación entre el FMS antes mencionado y los robots, para un fin común. Desarrollar una aplicación centralizada, que nos permita restringir distintas zonas del mapa a una máxima capacidad de robots.

Una vez dispongamos de estos dos componentes, podremos proceder a programar la aplicación que se ejecutará en los robots. Para ello, usaremos un entorno de simulación proporcionado por otro compañero del proyecto, Michalis Karamousadakis, de la empresa griega Singular Logic, basado en Gazebo y Rviz. Trabajamos principalmente en simulaciones debido a la situación provocada por el Covid-19, además de que no siempre disponemos de más de un robot RB-1 en el laboratorio del ai2. Sin embargo, una vez tengamos una versión funcional de la aplicación, trataremos de probarla lo más fielmente posible en un entorno real.

4. Herramientas de trabajo

Las herramientas de trabajo las vamos a dividir en dos categorías; elementos hardware y elementos software. Estos apartados abarcarán desde un minucioso estudio de los robots que se van a utilizar en el proyecto, así como los sensores que utilizan, hasta los lenguajes y plataformas utilizadas para el desarrollo del mismo.

4.1. Elementos Hardware

Como bien sabemos, el hardware son los elementos tangibles de cualquier sistema informático. En el caso de este proyecto, el hardware toma un papel muy importante, ya que no deja de ser un trabajo de robótica; sin embargo, esto no implica que sea el tema principal. Aun así, conocer el hardware es imprescindible, ya que para implementar una buena aplicación, es necesario tener claro con qué elementos se va a trabajar.

4.1.1. RB-1

El principal elemento hardware con el que se va a trabajar es el robot RB-1, este robot fue desarrollado por la empresa valenciana Robotnik, destinado a trabajar en espacios interiores. El RB-1 es capaz de transportar cargas o materiales, así como de trabajar con un sistema integrado que suele consistir en un brazo o torso robótico. En el caso del robot RB-1 del que dispone el ai2, se le ha incorporado un brazo robótico UR3, unido a una mano domótica con el fin de poder manipular objetos y cargarlos en el propio robot para poder transportarlos. En estos componentes no incidiremos ya que no entra dentro de nuestra competencia.

Imagen 4.1. – Estructura del robot RB-1 base

Comenzaremos hablando de las especificaciones físicas del robot para posteriormente comentar la sensórica. Las dimensiones robot, como se puede comprobar en la imagen 4.2, son 500 x 500 x 251 mm, un tamaño muy manejable, lo cual es muy importante teniendo en cuenta el entorno en el que se va a trabajar. Este robot pesa 30 kg, pero es capaz de transportar hasta 50 kg, además de alcanzar una velocidad de 1.5 m/s, sin tener

en cuenta el brazo integrado, ya que esto incrementa la altura y en el caso de que el robot se mueva a dicha velocidad, podría volcar al frenar. Este robot consta de una autonomía de 10 horas de movimiento continuo gracias a sus baterías LiFePO4 30Ah@24V. Además, la máxima pendiente que es capaz de asumir este robot es del 8%, volvemos a comentar que siempre se está hablando sin tener en cuenta un posible brazo robot integrado, sin embargo, en este caso no es muy importante ya que en principio el robot va a trabajar en lugares llanos.

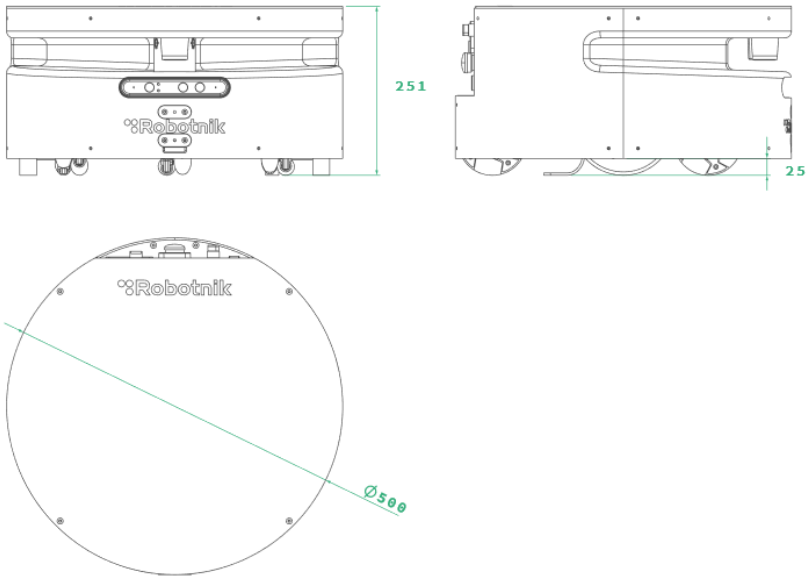

Imagen 4.2. – Diagrama de las dimensiones del robot RB-1

Con respecto a la mecánica, el RB-1 consta de un sistema de tracción diferencial con dos servomotores de 250 W. Además, cuenta con 3 ruedas omnidireccionales de apoyo, que van a permitir al robot girar sobre su propio eje.

El computador del que dispone el RB-1 en su interior es un sistema embebido con sistema operativo Linux, el cual consta de un procesador Intel i7 en su cuarta generación, una RAM de 8 GB y un disco duro donde almacenar los distintos programas de 120 GB. Estas especificaciones son más que suficientes para ejecutar los distintos programas de arquitectura ROS.

Con respecto a la sensórica de la que dispone nuestro RB-1 vamos a destacar los elementos más importantes, comenzando por el sensor de distancia. El RB-1 dispone del sensor 2D Hokuyo UST-20LX, situado en la parte frontal del robot, que nos va a proporcionar una detección media de 20 metros, dentro de un rango de 270º, con una resolución angular de 0’25º y una velocidad de respuesta de 25m/seg. Estas especificaciones nos van a permitir detectar cualquier obstáculo ubicado a una altura de 195mm del suelo.

El robot también dispone de una cámara 3D, modelo Orbbec Astra RGBD, con un rango de detección de 0’4 a 8 metros. El tamaño de imagen de la cámara de profundidad es

de 640*480, y trabaja a 30FPS, mientras que el tamaño de las imágenes RGB alcanzan un tamaño de 1280*960 a una velocidad de 10FPS. Con respecto al campo de visión hablamos de un rango de 60º en horizontal, y 49'5º en vertical (73º diagonal). Además de todo esto, este dispositivo incluye características muy interesantes como la detección de obstáculos, localización del robot (mediante un mapa), o la lectura de códigos QR, además de configurar operaciones de navegación reactiva.

Como es evidente, el robot dispone de otros sensores, tan importantes como los anteriormente mencionados, que ayudan al correcto funcionamiento del RB-1; como una unidad de medición inercial (IMU), en concreto de la marca PixHawk, que permite conocer la velocidad con la que se mueve el robot o la orientación del mismo, entre otros parámetros; sensores para el control de batería; o sensores de velocidad en las ruedas para poder calcular la Odometría, entre otros.

4.2. Elementos Software

En el subapartado de software nos vamos a encontrar con una mayor cantidad de elementos, puesto que se trata de un trabajo muy extenso, y sobre todo muy tangencial, es decir, que abarca desde bajo nivel hasta alto nivel, y por ello requiere de muchas piezas, principalmente software, para poder hacer desde simulaciones hasta comprobaciones a nivel físico.

4.2.1. Robot Operating System (ROS)

El sistema operativo que define la ejecución y los procesos internos de los robots es Ubuntu (Linux) en su versión 16.04. Este sistema operativo, además de ser software libre, permite una perfecta indexación con el *framework* ROS (*Robot Operating System*), necesario para la programación de las funciones de los robots.

ROS es considerado un *framework* o meta sistema operativo de código abierto, que trabaja sobre Linux para proporcionar un entorno de trabajo que facilita la comunicación entre los sensores y el sistema principal en los robots, utilizando un sistema de comunicación entre procesos, IPC (*Inter-Process Communication*), diferente a la tradicional memoria compartida, pudiendo manejarse los distintos datos en forma de publicador-suscriptor. A continuación, vamos a sumergirnos más profundamente en el concepto de *Robot Operating System* y su uso en este proyecto.

Entre otras cosas, ROS nos proporciona una capa de abstracción del hardware, control de los dispositivos a bajo nivel, o funcionalidades propias de la robótica como simulaciones o navegación; IPC, como ya hemos mencionado; o gestión de paquetes. Estos paquetes proporcionan una arquitectura estructurada que implementan gran cantidad de funcionalidades muy útiles a la hora de programar un robot, en concreto el RB-1.

Estos paquetes están formados por: ficheros ejecutables que lanzan las aplicaciones (ficheros *launch*), archivos fuente (ficheros en C++ o Python), y dos ficheros imprescindibles: *CMakeLists.txt* (lista de reglas *cmake* para la compilación) y *package.xml* (información del paquete y sus dependencias).

Los paquetes son una forma de organizar las líneas de código, en ellos puede haber una o más funcionalidades. Estas aplicaciones, para ser ejecutadas de forma independiente, deben lanzar nodos, que no son más que programas hechos en ROS y que tienen una función concreta. Estos nodos se ejecutan en un entorno llamado *roscore*, que es el proceso principal que maneja todo el sistema ROS, y que debe estar en ejecución siempre que queramos trabajar con ROS; a continuación, en la imagen 4.3. podemos observar un ejemplo de un diagrama de la estructura del entorno ROS:

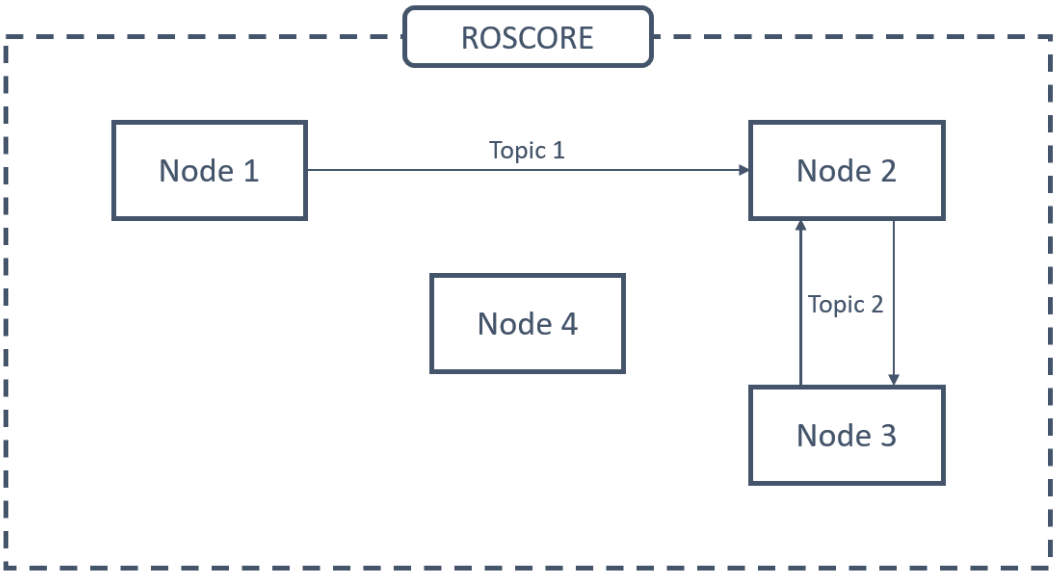

Imagen 4.3. – Ejemplo de una aplicación en ROS con varios nodos. Recuperado de https://www.theconstructsim.com

Como podemos comprobar en el ejemplo de la Imagen 4.3., se dispone de cuatro nodos, encargados de ejecutar cuatro programas diferentes dentro del entorno *roscore.* Este entorno permite que los nodos se comuniquen entre ellos mediante *topics.* Un *topic* es un canal que actúa como un tubo, donde los nodos ROS pueden publicar o leer información. Estos *topics* envían información a través de mensajes, que deben seguir una norma acordada entre la aplicación publicadora y la suscriptora. Para finalizar, hay que comentar que las aplicaciones en ROS suelen consistir en el típico modelo cliente-servidor, donde además existen las acciones, que proporcionan un servicio de forma no bloqueante.

Una vez instalado ROS en el sistema, se puede usar a través de dos lenguajes, C++ y Python. En nuestro caso, por comodidad vamos a utilizar Python. Python es un lenguaje dinámico, interpretado y multiplataforma, de gran abstracción, ya que se centra en una fácil legibilidad. Además, también es considerado multiparadigma, ya que soporta

programación orientada a objetos, programación imperativa y programación funcional. Una de las características más importantes de este lenguaje es que posee una licencia de código abierto, lo que es muy interesante en el mundo de la investigación.

4.2.2. FMS

Para la gestión de los robots, se va a implementar una solución centralizada, de forma que un único sistema se encarga de la gestión de los robots. Para ello se va a utilizar el *Fleet Management System* (FMS), que recibe información de los robots, como su posición, y envía a los mismos, acciones concretas que deben realizar para solventar las tareas solicitadas por los posibles usuarios. Este FMS, se va a ejecutar en un servicio en la nube. Estos servicios en la nube son cada vez más comunes ya que proporcionan muchas ventajas, entre ellas la más importante es que nos despreocupamos totalmente del hardware, y del coste monetario que supone, así como de su mantenimiento, de forma que la empresa a la que se confía la ejecución de nuestro servicio se encarga de proporcionar el nivel computacional suficiente y necesario para que nuestra aplicación funcione correctamente.

Existen varios tipos de servicios en la nube, vamos a mencionar los más importantes e identificar en cuál de ellos encaja nuestra aplicación. Los más populares son tres: SaaS (*Software as a Service*), PaaS (*Platform as a Service*) y IaaS (*Infrastructure as a Service*):

- SaaS: El software está alojado en servidores de los proveedores y el cliente accede a ellos a través del navegador web.
- PaaS: En este tipo de servicio el proveedor ofrece acceso a un entorno en el cual los usuarios pueden crear y distribuir sus propias aplicaciones.
- IaaS: El proveedor de servicios proporciona el software y las aplicaciones a través de internet.

Tras esta explicación, llegamos a la conclusión de que en nuestro caso requeriríamos de un *Platform as a Service*, en el que ejecutar nuestra aplicación.

Con respecto al FMS desarrollado por el CIGIP, deberemos comentar cuál es su funcionalidad y cómo trabaja. Para entender y situar el FMS dentro del proyecto ENDORSE, primero debemos conocer lo que se ha definido como *ENDORSE protocol specification*. Este protocolo es una regla de comunicación que sirve para intercambiar información con los diferentes robots y módulos de forma segura, comprensible y estandarizada. Este protocolo utiliza mensajería publicación-suscripción donde los robots y módulos pueden recibir comandos, proporcionar información acerca de su estado y demás acciones.

Los robots y módulos podrán publicar mensajes en el bróker de mensajes y suscribirse a los temas (*topics*) en los que estén interesados. Esto facilita la gestión y la seguridad de red, porque todos los dispositivos se pueden agrupar en segmentos de red separados detrás de un *firewall* sin comunicaciones entrantes. El siguiente diagrama refleja la estructura que implementa:

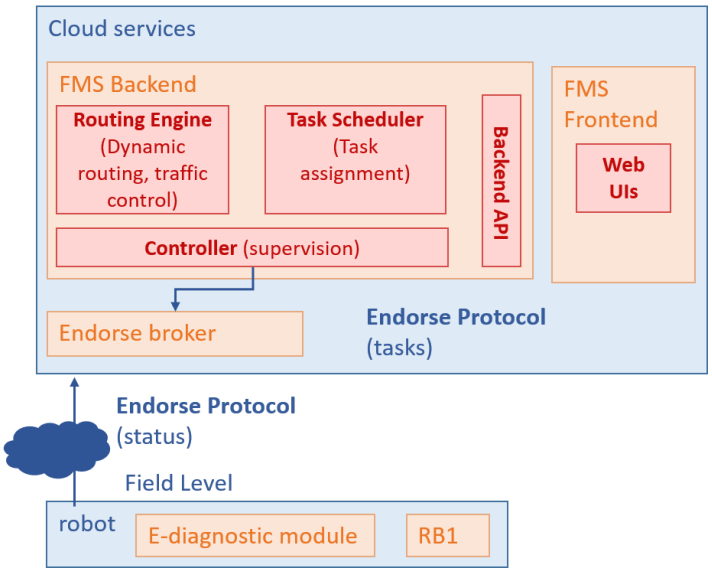

Imagen 4.4. – Diagrama de comunicaciones del proyecto ENDORSE

Como podemos observar en la imagen 4.4., a través de internet (al tratarse de un servicio en la nube) se comunican los robots y los módulos con el bróker, el cual recibe y envía información en forma de publicación-suscripción. Dicha información es procesada en el *backend* del FMS, el cual realiza dos acciones principales, enrutamiento y gestión de tareas.

Explicar el *Fleet Management System* implica de forma directa hablar sobre JSON y MongoDB, en cuyas tecnologías se sustenta principalmente este FMS.

4.2.3. JSON

JSON o *JavaScript Object Notation* es un formato ligero de intercambio de datos [18]. Es un lenguaje muy bueno para el intercambio de información ya que es fácilmente interpretable por los lenguajes de programación más utilizados en el panorama actual. Además, cabe destacar que JSON está constituido por dos estructuras: una colección de pares nombre/valor y una lista ordenada de valores. En la página oficial [18] se explica la estructura de la siguiente forma:

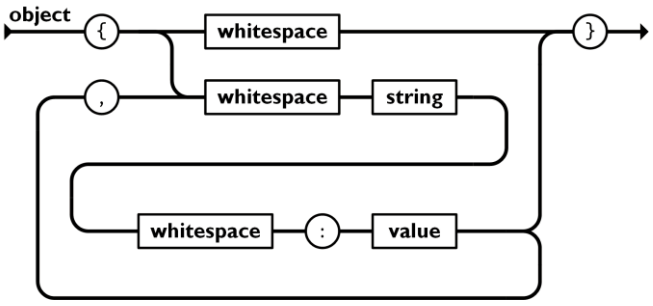

Imagen 4.5. – Esquema de la estructura de los mensajes JSON.

En el caso de nuestro FMS, un ejemplo concreto sería el siguiente:


```
{
"map": [
  { "type": "LineString", "coordinates": [ [ -24.021430274695831, -20.91720237165282, 0.0 ], [ -24.021430274695831, -18.022474770720009, 0.0 ] ] } ,
  { "type": "LineString", "coordinates": [ [ -22.654144532742521, -20.91720237165282, 0.0 ], [ -22.654144532742521, -18.022474770720009, 0.0 ] ] } ,
  { "type": "LineString", "coordinates": [ [ -28.791051874725319, -20.91720237165282, 0.0 ], [ -28.791051874725319, -18.022474770720009, 0.0 ] ] }
  ]
}
```


Imagen 4.6. – Ejemplo de mensaje JSON para el envío de coordenadas.

Como podemos observar, se ha respetado la estructura expuesta en la Imagen 4.5., de forma que, en la imagen 4.6., se ha redactado un mensaje JSON, el cual contiene unas coordenadas concretas con respecto al mapa. Este es solo un pequeño ejemplo de las infinitas posibilidades que presenta JSON y que están expuestas en su página oficial [18].

4.2.4. MongoDB

En lo referente a MongoDB [19], es considerada una base de datos distribuida con gran escalabilidad y flexibilidad, y está basada en documentos de tipo JSON; lo cual nos va a permitir una perfecta indexación con lo expuesto en el apartado anterior, facilitando así el desarrollo de la aplicación. Además, cabe destacar que permite el despliegue en la nube, lo cual, debido a la naturaleza de nuestro proyecto, nos puede resultar muy práctico.

4.2.5. Protocolo ENDORSE

La estructura con la que se trabaja, especificada en el protocolo ENDORSE, es muy sencilla. Para empezar, los *topics* siguen la siguiente estructura: *{namespace}/ {robotId}/{criterion}*, de forma que siempre sabremos a qué hace referencia un *topic* en el caso de que deseemos suscribirnos a este o publicar en él. Donde, *namespace* siempre será *endorse* haciendo referencia al proyecto, *robotId* consiste en el ID del robot y *criterion,* al tipo de mensaje, que puede ser un comando, *commands*, o un *feedback,* para conocer el estado del robot al que se está haciendo referencia después de enviarle un objetivo o tarea. *Criterion* también puede tomar valores como *cancel, errors o map,* sin embargo, estas opciones no las vamos a utilizar con lo que no vamos a incidir en ellas.

En el caso de querer enviar un comando al robot, deberemos publicar en el *topic /endorse/robot_x/commands*, de forma que el mensaje publicado debe seguir la estructura de JSON, donde el primer elemento será *"command"*, y tras los dos puntos una de las siguientes opciones con sus respectivos parámetros:

- **GOTO**: Envía al robot un objetivo al cual debe llegar. Debe especificar tres parámetros que indican la posición (*x,y,theta*).
- **MOVE**: Envía al robot una distancia a moverse en sentido y dirección, al igual que el comando GOTO, debe especificarse tres parámetros (*x,y,theta*).

- **TURN**: Especifica en ángulo en el que debe girar el robot. En este caso tan solo se requiere de un parámetro (*theta*).

4.2.6. Puente MQTT

Para que sea posible la comunicación entre el FMS y los robots, es necesario un puente MQTT. Esto es debido a que, a pesar de que el sistema ROS también trabaja con mensajería publicación-suscripción, cualquier sistema ejecutando ROS, es en sí mismo un nodo máster, de forma que para poder comunicarse con otros sistemas ROS, debería configurarse un sistema *multi-master*, lo que es complejo, difícil de implementar y poco seguro, además de, como hemos comentado anteriormente, complicarse a medida que incrementamos el número de robots. Por todo esto es que, como se comentó en el apartado 3, se ha considerado una mejor solución el desarrollo de un puente MQTT como interfaz entre el FMS y los robots.

Ya se han expuesto en el punto 3.2. algunas de las ventajas de trabajar con MQTT, y en este apartado se va a hablar de las especificaciones técnicas que hacen posibles estas ventajas. MQTT fue inventado por el Dr. Andy Stanford-Clark de IBM, y Arlen Nipper de Arcom en el año 1999, y desde entonces su uso se ha extendido a gran escala, haciendo sombra a todos sus competidores en el campo de las comunicaciones. Esto ha sido debido principalmente por la popularidad que ha adquirido en los últimos años el Internet de las cosas (IoT, *Internet of Things*), cuyos dispositivos usan casi exclusivamente comunicaciones MQTT, sobre todo por su poco uso de recursos, imprescindible en dispositivos de poco tamaño.

Las conexiones de este protocolo están orientadas a TCP, lo que reduce notablemente las latencias, al ser un protocolo sin conexión. También es importante destacar que soporta persistencia en los mensajes, ya que el bróker es capaz de almacenar información a pesar de que el comunicador deje de funcionar, permitiendo así que el receptor reciba el último mensaje del comunicante. El bróker es un programa servidor que recibe, almacena y envía mensajes basados en *topics.* Cada *topic* puede contener como máximo un mensaje a la vez, el cual debe conservarse. El cliente se conectará al bróker MQTT, de forma que cuando se conecta este le responde con un identificador único.

En el caso del puente MQTT que vamos a utilizar en este proyecto, ha sido desarrollado por la empresa Robotnik, y es soportado por el bróker de Eclipse, Mosquitto. El envío de mensajes entre los robots y el puente se ha desarrollado basándose en el paquete ROS *mqtt_bridge,* de forma que, a través de un fichero YAML de configuración, es posible asignar a un *topic* en ROS, un *topic* MQTT, permitiendo así que exista comunicación entre un sistema con ROS (como es el caso de los robots), y un sistema sin ROS (como es el FMS).

Para comprobar que esta comunicación funciona, se va a hacer uso de la herramienta MQTT Explorer, que permite conectarse a un bróker, dado de alta en una IP y un puerto,

de forma que podremos visualizar todos los *topics*, y sus respectivos mensajes, que están siendo publicados en el bróker.

Para finalizar, parece necesario hablar sobre las simulaciones, ya que ha sido la principal forma de trabajo durante el transcurso del proyecto. En estas simulaciones se va a trabajar con tres robots, a los cuales nos referiremos como *robot_1, robot_2 y robot_3*.

4.2.7. Docker

Para estas simulaciones, dentro del proyecto ENDORSE se ha desarrollado un contenedor en Docker. Un contenedor no es más que un paquete de Software ejecutable que incluye todo lo necesario para poder lanzar una aplicación, desde el código, hasta las herramientas del sistema y las configuraciones, entre otras cosas, lo cual facilita mucho el trabajo. En este caso, dentro de la imagen contenedora se ha incluido tanto los paquetes de ROS necesarios para el movimiento de los robots, como el propio mapa en el que se va a trabajar y el entorno de simulación con tres robots RB-1.

Es importante destacar que, para la visualización de la simulación, así como para facilitar la programación de aplicaciones en ROS, se hace uso de los programas Gazebo y Rviz respectivamente, los cuales también vienen incluidos en el propio Docker.

4.2.8. Gazebo

Gazebo es una herramienta de simulación que permite recrear escenarios realistas tanto de entornos interiores como exteriores, permitiendo construir un posible entorno con el que se podría encontrar el robot, con el fin de hacer pruebas en el caso de no disponer de robots para hacerlas o simplemente para que no haya accidentes que puedan deteriorar el propio entorno de trabajo, el robot o incluso afectar a personas.

Cuando lancemos las simulaciones y cargue Gazebo podremos observar una interfaz como la de la imagen 4.7.

En Gazebo es posible tanto visualizar el comportamiento de los robots, como diseñar uno desde cero, así como el entorno en el que se va a mover. Esta última funcionalidad no nos corresponde como parte del proyecto, puesto que ya existía un modelo tanto para el robot RB-1 como para el mapa; sin embargo, nos va a ser muy útil a la hora de programar la aplicación ya que vamos a poder ver en tiempo real cómo interactúan los robots con el entorno.

Además, se dispone de un modelo de las instalaciones del centro Fundación Ave María, lo cual es muy práctico ya que es el lugar en el que se implantará el proyecto.

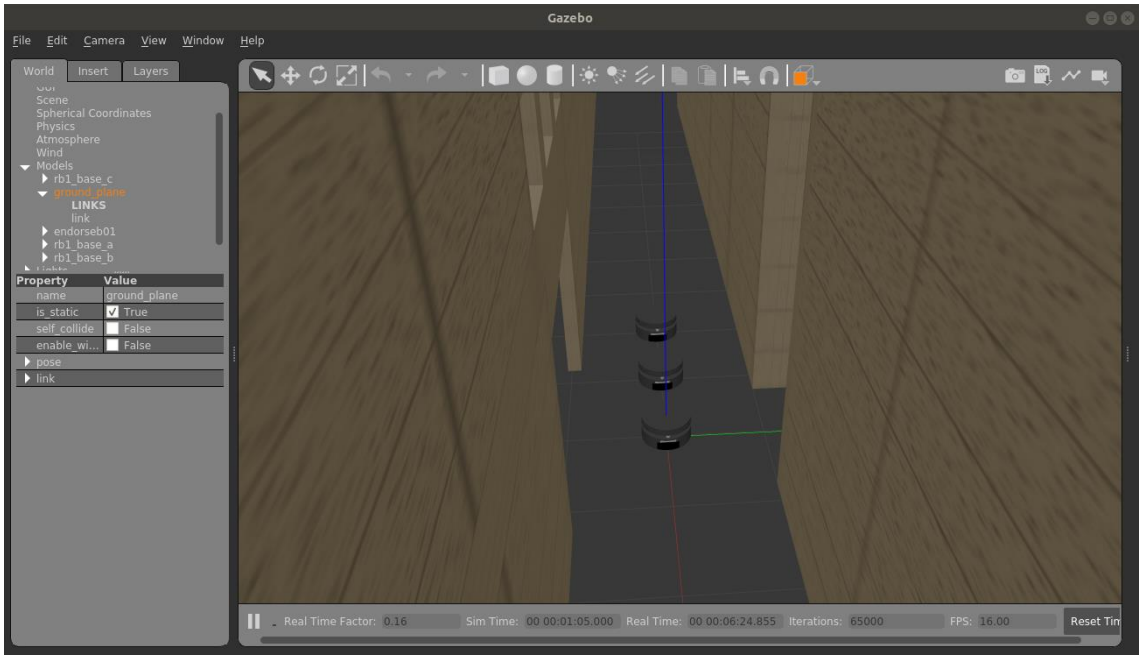

Imagen 4.7 – Interfaz de Gazebo.

4.2.9. Rviz

Con respecto a Rviz, es también una herramienta de simulación, pero adaptada al entorno de ROS, esto quiere decir que nos va a permitir funcionalidades como enviar mensajes a los robots sin necesidad de redactar código, o visualizar lo que está detectando el robot en tiempo real por cualquiera de sus sensores, lo cual es muy práctico a la hora de desarrollar una aplicación de esta envergadura.

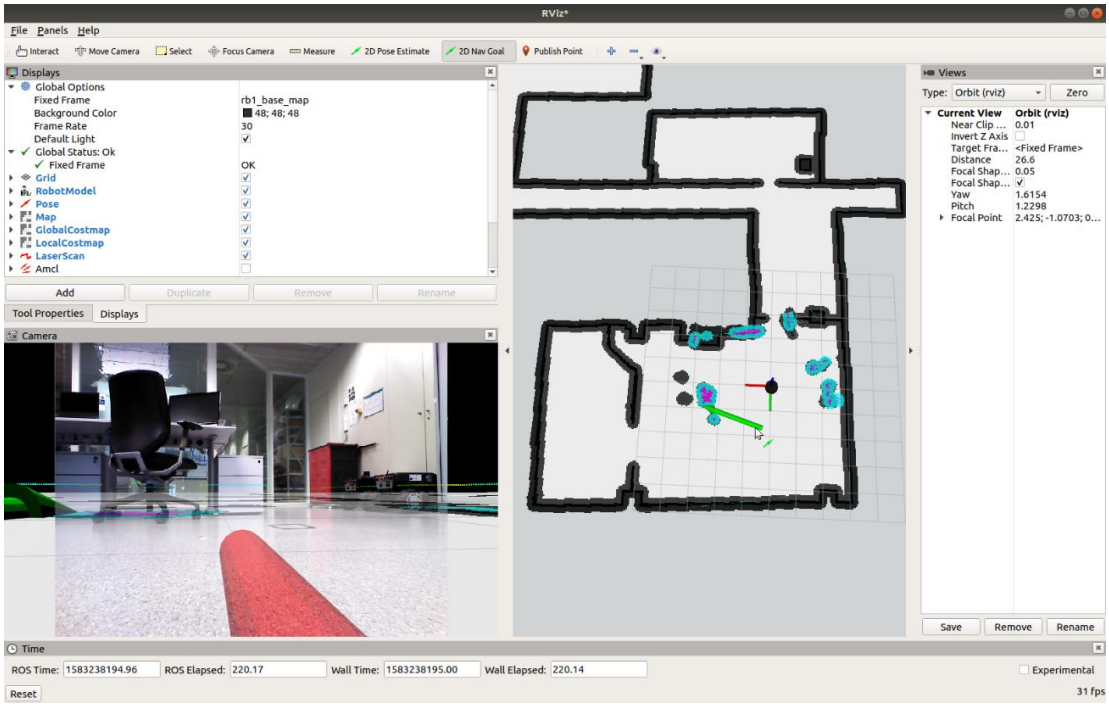

Imagen 4.8 – Interfaz de Rviz mostrando el laboratorio del ai2.

Esta aplicación es útil incluso cuando se está trabajando con robots físicos. En la imagen 4.8 podemos observar un ejemplo de ello, donde mediante la aplicación Rviz podemos observar lo que está detectando el robot RB-1 del que disponemos en el ai2.

Rviz nos proporciona distintos componentes, capaces de leer los *topics* de ROS e interpretarlos, de forma que, entre otras cosas, podemos observar qué se está viendo a través de la cámara integrada en el robot, tal y como vemos en la imagen 4.8, o los obstáculos que está detectando a través del sensor de proximidad, que vienen expresados en el mapa de color azul y rosa. También se está mostrando el mapa que tiene cargado el propio robot, y el cual va a utilizar para localizarse en el espacio, de forma que las líneas negras son las paredes de las distintas estancias.

4.2.10. Repositorios

Para finalizar, resulta imprescindible mencionar los repositorios en la nube. Estos se definen como depósitos de documentos digitales, cuya finalidad es poder compartir de forma fácil y segura dicha información entre los usuarios que lo requieran. Encontramos un gran abanico de repositorios en el mercado actual, sin embargo, en este proyecto se va a hacer uso de dos: GitLab y Docker Hub.

En el caso de GitLab, es una herramienta para compartir todo tipo de información digital, aunque se use más comúnmente para la distribución programas y aplicaciones informáticas. Sin embargo, Docker Hub está más enfocada al desarrollo y distribución de imágenes contenedoras, y por ello, se requieren ambas plataformas en el proyecto.

5. Diseño y desarrollo de la solución propuesta

Durante este capítulo hablaremos del diseño de la solución, y del desarrollo de la misma, con lo que se pretenderá alcanzar los objetivos propuestos durante la introducción.

5.1. Diseño de la solución

Como se ha comentado en repetidas ocasiones, se pretende diseñar una solución centralizada que permita evitar, en la medida de lo posible, colisiones, basándose en la restricción de estancias que puedan considerarse conflictivas.

Esta restricción se pretende que sea dinámica, y que cuando la aplicación se encuentre implantada, mediante la interfaz de usuario, sea posible decidir el aforo máximo de robots en cada una de las estancias definidas, incluso poder definir estancias nuevas.

5.1.1. Planteamiento

FMS

Para que nos hagamos una idea de cómo podría ser, mediante un mapa del laboratorio ai2, obtenido por uno de los robots RB-1 de los que se dispone, se ha hecho una subdivisión en posibles áreas, tal y como se muestra en la imagen 5.1.

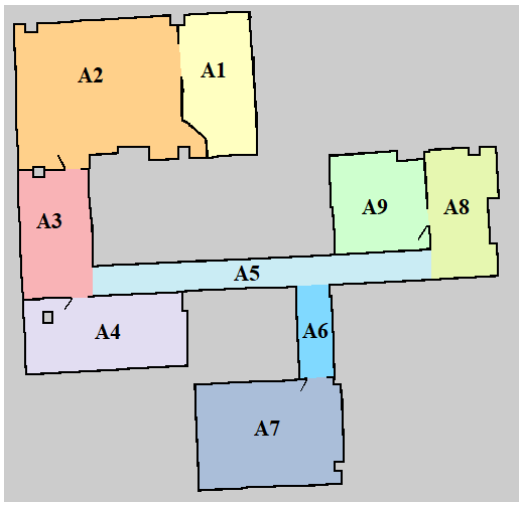

Imagen 5.1. – Plano del edificio ai2 subdividido en zonas.

Para tratar de que el diseño de la solución sea lo más realista posible, podríamos imaginar que, por ejemplo, la sala A9 es la farmacia de la planta del hospital. Esta farmacia, como es obvio, va a ser muy transitada por personal del hospital, como personal de enfermería o médico, además de por nuestros robots; es por ello que,

seguramente, deberemos restringir a un robot su aforo, con el fin de que no entorpezca a las personas, ya que se trata de una estancia no muy grande. Además, la zona A5, que es un pasillo, podría restringirse a dos robots, con el fin de que puedan esquivarse sin demasiadas dificultadas y así evitar bloqueos. Este podría ser un buen punto de partida para comenzar con el diseño de nuestra solución.

Sabemos que el FMS es capaz de, mediante el uso del grafo de Voronoi y con la disponibilidad de un mapa de la planta, generar las rutas de menor coste de un punto a otro. El grafo de Voronoi se define de la siguiente manera: *"Dado un conjunto de vértices V en el plano, el diagrama de Voronoi de V, es una descomposición en el plano en regiones relacionadas a cada uno de los V"* (Moreno & Ordóñez; 2009); en resumidas cuentas, este grafo es capaz de descomponer los píxeles del plano en distintos puntos o nodos, por los cuales podrán circular los robos, y mediante la unión de los mismos se generan rutas de distintos costes de un punto a otro del plano. Un ejemplo de la generación de caminos es la siguiente imagen. Debemos tener en cuenta que, para que los robots no se pierdan, la distancia máxima entre dos puntos deberá ser de tres metros.

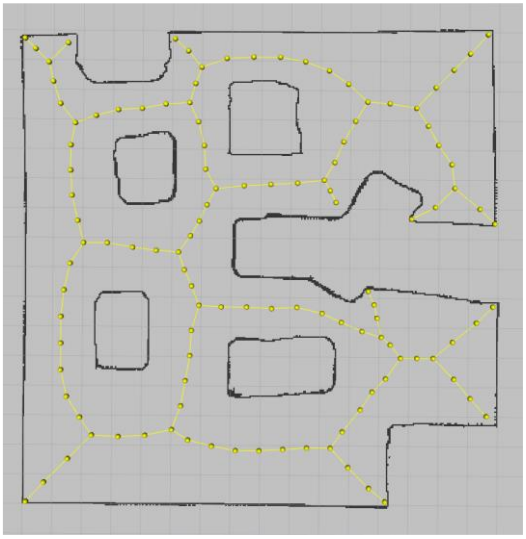

Imagen 5.2. – Ejemplo de posibles rutas generadas por el FMS. Recuperado de: http://wiki.ros.org/tuw_voronoi_graph

Mediante estas rutas y diferentes algoritmos, el FMS es capaz de calcular los costes de cada uno de los posibles caminos que existen de un punto a otro cuando se asigna una tarea a un robot.

Resumiendo, un usuario, mediante la aplicación, solicitará una tarea, y el FMS, comprobará los robots que están libres en ese momento, calculará las posibles rutas desde el punto en el que se encuentran hasta el punto objetivo, y elegirá la de menor coste.

Todo esto es posible gracias al constante *feedback* o retroalimentación entre el propio FMS y cada uno de los robots. Estos, enviarán al FMS, a través del puente MQTT, su

ubicación en el mapa. Esta información será enviada en periodos de tiempo predefinidos, para que el FMS sepa en todo momento dónde se encuentra cada robot.

Una vez el FMS ha elegido una ruta eficiente hacia un objetivo, el procedimiento se puede dividir en los siguientes pasos:

1. El FMS envía al robot elegido los tres siguientes nodos que debe seguir hasta el objetivo.
2. El robot recibe estos tres puntos, y comienza el movimiento.
3. A medida que avanza el robot, este sigue enviando en qué punto se encuentra.
4. Cuando el FMS considere que el robot está lo suficientemente cerca del tercer nodo, volverá a enviar los tres nodos siguientes.

Este procedimiento se repetirá hasta que el robot alcance su objetivo.

Cuando el FMS genera la ruta, almacenará las zonas que debe cruzar el robot en su paso por el camino. Además, también podrá guardar listas de ocupación de cada una de las zonas. De forma que, a medida que avanza el robot por su camino y envía su ubicación, estas listas se van actualizando. Lo aclaramos con un ejemplo.

Supongamos que un robot trata de ir de un punto a otro. Sabemos los puntos exactos de cada uno de los nodos de la ruta, además de las zonas que implican, cuya información se almacena en una base de datos MongoDB, como hemos comentado en el apartado 4.1. Vamos a suponer que el punto de origen se encuentra en el área A1 con respecto a la imagen 5.1., y el punto destino se encuentra en el área A9. El FMS almacenará, entre otras cosas, la siguiente información en el instante inicial (suponemos que solo existe un robot):

Ruta de áreas, robot 1:

Lista de ocupación de cada área:

A1:

…

An:

Como podemos observar, el robot 1, para llegar hasta su objetivo debe pasar las áreas A1, A2, A3, A5, A8 y A9. Además, observamos que, inicialmente, las listas de ocupación de las distintas áreas están vacías, excepto el A1, que contiene un robot.

A medida que este robot vaya avanzando, enviará, como ya hemos comentado anteriormente, su ubicación al FMS, el cual sabrá a qué zona se corresponde cada ubicación recibida. Llegará un punto en el que el robot envíe una ubicación, la cual llamaremos de transición, que supondrá el paso de un área a otra diferente. Cuando esto ocurra, el FMS reaccionará comprobando en la lista de ruta del robot, a qué área

trata de dirigirse, y en la lista de ocupación de la misma si, con respecto a una variable de ocupación, el robot puede entrar en la zona o no. En caso negativo deberá enviarle una señal de espera.

Esta situación no se va a dar en este ejemplo ya que se está suponiendo que solo existe un robot, sin embargo, habrá que tenerlo en cuenta para problemas de múltiples robots.

Una vez el robot pasa a la siguiente área, la situación en el FMS será la siguiente:

Ruta de áreas, robot 1:

Lista de ocupación de cada área:

A2:

…

An:

Volviendo a repetirse la situación en la que todas las áreas están vacías excepto, en este caso, la dos. Como podemos observar, la ruta de áreas también ha variado, eliminando el área 1 de la lista. Este proceso se repetirá hasta alcanzar el objetivo.

Esta solución nos facilitará el proceso de restricción de zonas, al conocer en todo momento el volumen de ocupación de cada una de las áreas y manteniendo constantemente una retroalimentación entre el FMS y los robots.

Al fin y al cabo, esta solución corre a cargo del CIGIP y no es más que una propuesta, puesto que todavía no ha sido implementado. Al tratarse de información requerida por los robots, y no disponer de la misma, durante el apartado de desarrollo se asumirá que efectivamente se dispone de dicha información.

Puente MQTT

Con respecto al puente MQTT, desarrollado por parte de Robotnik, haciendo uso del paquete de ROS *mqtt_bridge*, tal y como comentamos en el apartado 4.2, solo cabría realizar las modificaciones que pudiéramos necesitar. Este puente ha sido desarrollado y configurado en un nuevo paquete de ROS llamado *command_interface*, el cual ya está preparado para funcionar intercambiando información entre los robots y el FMS.

En el caso de nuestra aplicación, deberemos modificar el fichero de configuración, *mqtt_bridge.yaml*, especificando las correspondencias entre *topics* ROS y *topics* MQTT, y viceversa. Esta configuración será replicada y adaptada para cada uno de los tres robots de los que se dispone en la simulación.

Primero necesitamos que se publique en el puente MQTT la Odometría, para que el FMS sea capaz de leerla y así utilizarla en sus algoritmos de búsqueda de rutas óptimas.

Además, necesitaremos un *topic* en el que enviar al robot una señal de parada. Estas modificaciones se realizarán en el fichero de configuración del paquete *command_interface.*

Debemos tener muy en cuenta que, al trabajar sobre simulación, el puente debe traducir los *topics* de tres robots a la vez, sin embargo, cuando pasemos a un entorno real, cada robot ejecutará su propio paquete, trabajando única y exclusivamente sobre sus *topics*.

Aplicación ROS

Para finalizar, debemos hablar de la aplicación ROS, la cual se ejecutará en cada uno de los robots disponibles. Tal y como hemos comentado unas líneas más arriba, es importante tener en cuenta que, al trabajar en una simulación, estamos trabajando en lo que podríamos llamar modo supervisor, es decir, toda la información que necesitamos se encuentra a nuestra disposición en nuestro equipo, lo cual es una situación irreal, puesto que en un entorno real esto no va a ocurrir.

Esto significa que, un único código podría realizar la tarea de gestión de parada, suscribiéndose a los *topics* de parada de todos los robots mediante el símbolo reservado en MQTT "+". Sin embargo, esto no es real y sería un error tomar ese camino.

La manera correcta de trabajar será crear una aplicación distinta para cada uno de los robots, de forma que cada una se suscriba a sus propios *topics*. Estos cambios se prevén muy poco significativos al estar estos estandarizados por el protocolo ENDORSE; incluso cabe la posibilidad de que, en el entorno real, los nombres de los robots sean los mismos, sin dar lugar a errores puesto que son privados para cada uno. En cualquiera de los casos, sería interesante el uso de una variable global que defina el nombre del robot para que, en el caso de que varíe de un robot respecto a otro, no sea tedioso buscar los puntos en los que se hace referencia al nombre, siendo en el caso de la simulación, *robot_base_a (robot_1), robot_base_b (robot_2) y robot_base_c (robot_3).*

La aplicación deberá suscribirse tanto al *topic* de parada como al *topic* que solicita los comandos predefinidos y ya programados por Robotnik. Cada vez que se reciba una solicitud de parada, la aplicación almacenará el comando que estaba ejecutando el robot y lo cancelará. Cuando se solicite a este robot que continúe, se enviará a sí mismo el objetivo que estaba tratando de cumplir antes de su interrupción, y así poder retomar el movimiento.

A continuación, se incidirá en ideas más técnicas, necesarias para el desarrollo de la solución.

5.1.2. Especificaciones técnicas

Con respecto a las especificaciones técnicas, principalmente se va a hacer uso del protocolo ENDORSE, el cual ya comentamos en el apartado 4.2.5. Sin embargo, va a ser necesaria la adición de *topics* y mensajes, de los cuales hablaremos en este subapartado.

Para empezar, conocemos que el FMS es capaz de generar rutas eficientes de un punto a otro del mapa. La información sobre los puntos que debe seguir el robot se envía, como ya hemos comentado, a través del *topic endorse/robot_x/commands.* Este FMS enviará consecutivamente puntos relativamente cercanos al robot para que este vaya alcanzando sus metas a corto plazo. Para saber si el robot está cumpliendo sus objetivos, este FMS recibe una retroalimentación a través del *topic endorse/robot_x/feedback.*

Con estos mensajes *feedback,* el FMS deberá almacenar un contador de ocupación de cada una de las estancias, además conocer en todo momento qué robots se encuentran en cada zona, para que en el caso de que obstaculice una tarea de emergencia por parte de otro robot, poder solicitar su salida.

Para poder implementar la solución expuesta en el apartado 5.1.1., el FMS deberá almacenar en buffers las listas de ocupación de cada una de las zonas para que, en el caso de que un robot se disponga a entrar a una zona restringida, pueda comprobar si puede o no entrar, y en caso negativo hacerle esperar. Para esta función, se va a configurar el *topic endorse/robot_x/stop,* por el cual el FMS podrá enviar un número entero con el que mandar parar al robot, o permitir que continúe el con la tarea que estaba realizando. Las opciones son las siguientes:

- **1**: Solicita al robot que se pare, debido que la zona a la que trata de acceder tiene el aforo completo.
- **0**: Solicita al robot, una vez está parado, que continúe con el objetivo que trataba de cumplir.

Para poder hacer uso de estos *topics*, es necesario configurar el puente MQTT. Primero decidiendo qué tipo de mensajes en ROS cumplen nuestras necesidades. Esto es el mensaje *Int32* del paquete *std_msgs.*

Adicionalmente, se ha añadido un *topic* que se suscribe a la Odometría y la publica en el puente MQTT, ya que esta información será necesaria para que el FMS gestione y reparta las distintas tareas entre los robots. El mensaje requerido para este propósito es *Odometry* del paquete *nav_msgs*.

Con respecto a la aplicación ROS, deberá estar suscrita a los *topics* ROS en los que se han traducido los *topics* MQTT. Estos siguen la estructura */robot_base_x /commands_manager/stop* y */robot_base_x/commands_manager/commands,* tal y como se ha definido en el fichero de configuración del puente. Mediante estas suscripciones, la aplicación podrá comprobar si le envían una solicitud de parada, y en el caso de hacerlo, almacenar el comando que estaba ejecutando el robot y cancelarlo, para que cuando reciba una señal que le permita continuar, poder reenviársela a sí mismo.

Python simplifica mucho este proceso, ya que no va a ser necesario conocer el tipo de mensaje que utiliza el *topic commands,* ya que Python es un lenguaje no tipado, y por lo tanto podemos asignar a una variable cualquier tipo valor, incluso que este varíe durante el transcurso de la ejecución. Crearemos una variable global a la que llamaremos *command* en la que almacenar dicho valor.

Algo que sí deberemos tener en cuenta es que el hecho de suscribirse a dos *topics* distintos no es algo trivial, va a ser necesario el uso de clases de forma que se generen distintos hilos de ejecución para no perder información. Estas clases proporcionaran funciones *get* con las que poder obtener los valores recibidos a través de los *topics*.

5.2. Desarrollo de la solución

En este subapartado se va a tratar de plasmar cada uno de los pasos a seguir para hacer posible el desarrollo expuesto en el apartado anterior, diseño de la solución.

5.2.1. Preparación del entorno

Al inicio del proyecto se nos proporcionó una imagen Docker completamente funcional, la cual nos permitía lanzar el entorno de simulación sin necesidad de ninguna instalación previa, tan solo debemos tener en nuestro equipo *git* y Docker. Con estos dos programas podremos descargarnos las imágenes y lanzarlas.

La estructura del desarrollo del proyecto RB-LOG la hemos plasmado en el esquema de la imagen 5.3. En ella podemos observar que el proyecto RB-LOG se divide al menos en dos subproyectos, *rb_log_sim* y *endorse_command_interface.* Estos subproyectos son dos imágenes contenedoras, donde *rb_log_sim* lanza el entorno y las simulaciones, y *endorse_command_interface* contiene paquetes para ejecutar el puente MQTT.

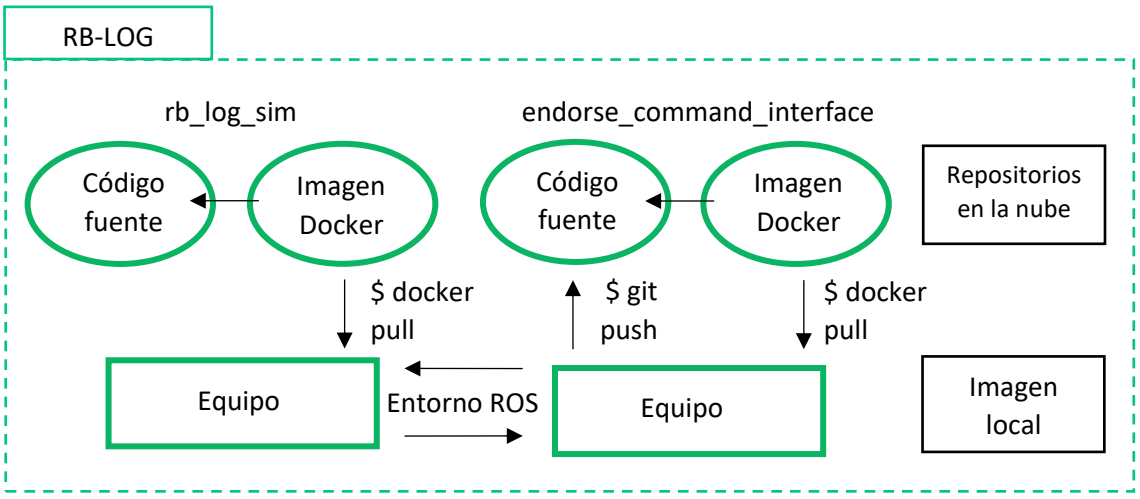

Imagen 5.3. – Estructura del desarrollo del proyecto.

El código de estas imágenes contenedoras se encuentra en GitLab, de forma que Docker Hub toma dicho código para montar las imágenes que ejecutaremos en nuestro equipo.

En el caso de querer modificar alguna imagen, en *rb_log_sim* no podremos porque no tendremos permisos, pero en el repositorio *endorse_command_interface* sí que se nos está permitido realizar cambios; para ello, deberemos hacer un *push* sobre el repositorio de GitLab, para que Docker compruebe el código y nos permita descargar una imagen sin errores a través de un *pull.*

Para descargar ambas imágenes vamos a requerir de permisos por parte de los desarrolladores, los cuales deberemos solicitar, además de, generarnos una clave SSH en GitLab.

La primera imagen, que nos permitirá lanzar las simulaciones, se encuentra en el repositorio Docker Hub con nombre *rb_log_sim*, dentro del proyecto RB-LOG. Podremos descargarlo ejecutando:

```
$ docker login
$ docker login registry.gitlab.com/rb-log/rb_log_sim
```

Donde se nos pedirá que iniciemos sesión con nuestro usuario de Docker Hub para el primer comando, y con el usuario de GitLab para el segundo. Por ello, debemos habernos creado usuarios en ambos servicios previamente.

Continuaremos descargando la imagen del repositorio mediante:

```
$ git clone https://gitlab.com/RB-LOG/rb_log_sim/
```

Y haciendo el *pull* para actualizar la imagen a su última versión del repositorio Docker.

```
$ docker pull registry.gitlab.com/rb-log/rb_log_sim:default
```

En este punto ya podemos lanzar la simulación ejecutando el siguiente script, el cual se encuentra en el directorio *rb_log_sim* que acabamos de descargar:

```
$ ./.docker/docker_run_rb_log_sim.sh
```

Este script, mediante el comando d*ocker run* y varios parámetros de configuración, prepara el entorno ROS y lanza las simulaciones de los robots con Rviz y Gazebo en nuestro equipo. Posteriormente, si deseamos cargar todas las características en Rviz deberemos ejecutar, además, los siguientes scripts:

```
$ ./.docker/docker_run_mlm_server.sh
$ ./.docker/docker_run_mlt_server.sh
```

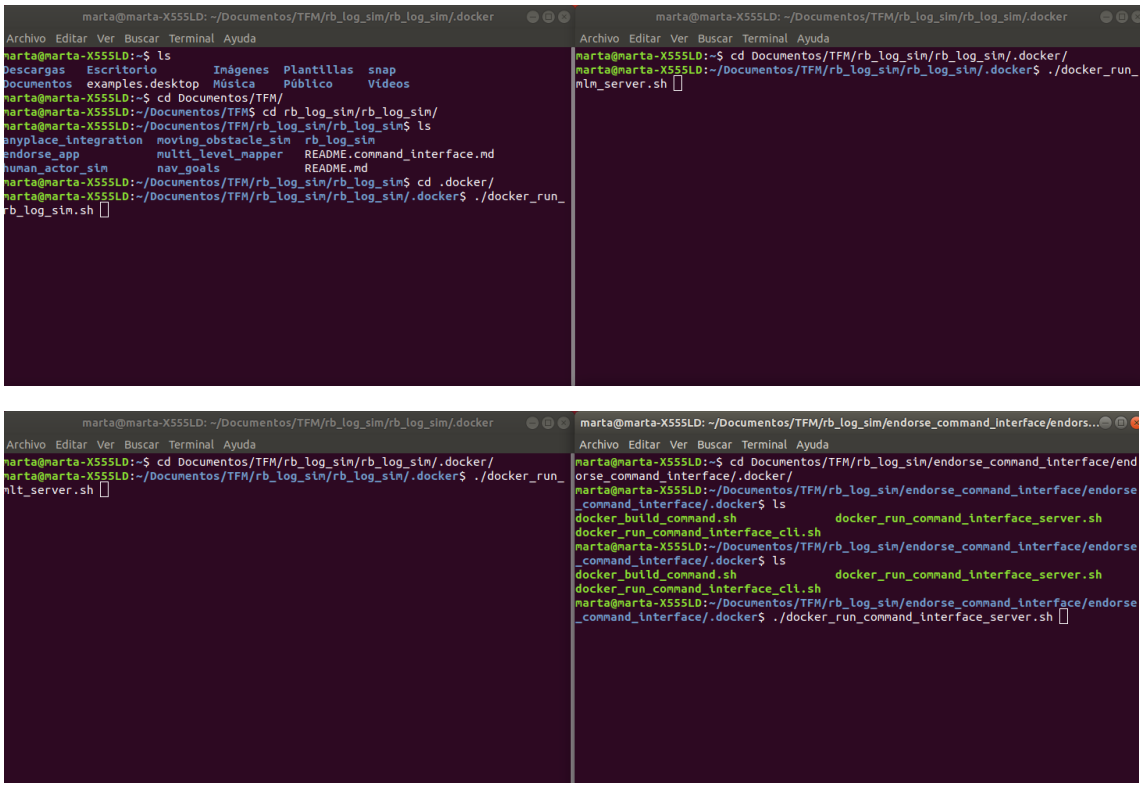

Imagen 5.4. – Terminales con los scripts listos para ser ejecutados.

En la imagen 5.4. podemos observar los terminales listos para lanzar los scripts que se han comentado anteriormente. Se ha decidido plasmarlo para reflejar que deben ejecutarse en terminales distintos, o, en su defecto haciendo uso del símbolo “&” al final de cada comando. Este símbolo lanza comandos en segundo plano, de forma que se puede reutilizar la instanciación del terminal. Sin embargo, esto complica un poco la acción de matar los procesos, puesto que tenemos que especificar el ID del proceso que el sistema ha asignado para poder finalizarlo. Por contraposición, si hacemos uso de distintos terminales, para matar un proceso basta con teclear Ctrl+c sobre el terminal que lo ha lanzado.

Una vez hecho esto, podremos observar lo siguiente:

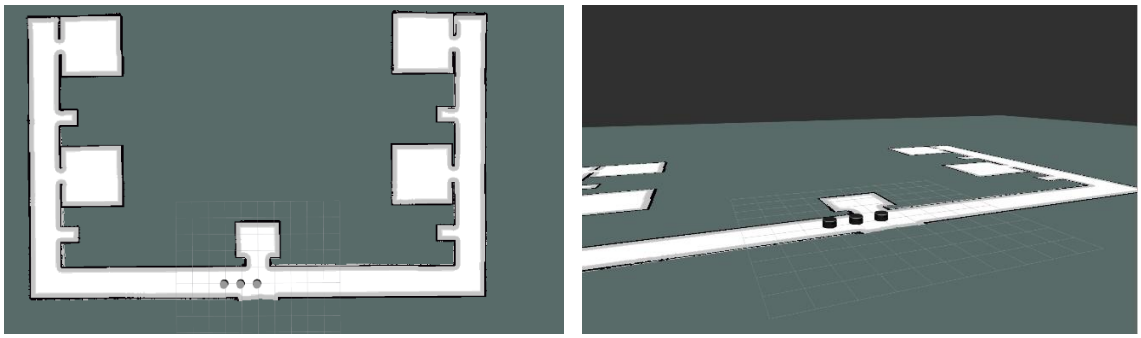

Imagen 5.5. – Captura de Rviz al lanzar las simulaciones.

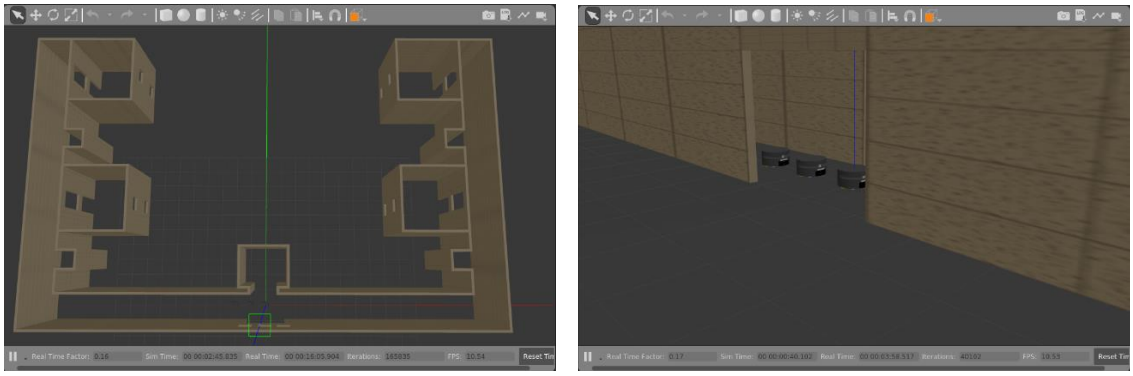

Imagen 5.6. – Captura Gazebo al lanzar las simulaciones.

Realmente las imágenes 5.5. y 5.6. han sido modificadas, eliminando las herramientas de cada una de las aplicaciones, para dar más importancia y poder observar mejor el entorno de cada una de las aplicaciones, es decir, el mapa y los tres robots. El mapa se corresponde con las instalaciones de la fundación Ave María.

A continuación, procedemos a descargar el segundo repositorio, el cual nos permitirá iniciar el servicio *mosquitto* y lanzar el puente MQTT. Para ello, ejecutamos las siguientes líneas de comando:

```
$ docker login
registry.gitlab.com/rb-log/endorse_command_interface

$ git clone
https://gitlab.com/RB-
LOG/endorse_command_interface:command-interface

$ docker pull
registry.gitlab.com/rb-log/
endorse_command_interface:command-interface
```

Llegados a este punto ya podemos ejecutar el script que nos permitirá lanzar el servicio *mosquitto* y el puente MQTT:

```
$ ./.docker/docker_run_command_interface_server.sh
```

En este caso dentro del directorio que acabamos de descargar de GitLab, *endorse_command_interface*.

En este punto ya podremos comprobar que se ha dado de alta el puente en la IP local 127.0.0.1, localhost, en el puerto típicamente usado para MQTT, 1883. En la siguiente imagen podemos observar cómo está el servicio dado de alta, de forma que podremos publicar en él, descargando previamente la aplicación *mosquitto-clients*:

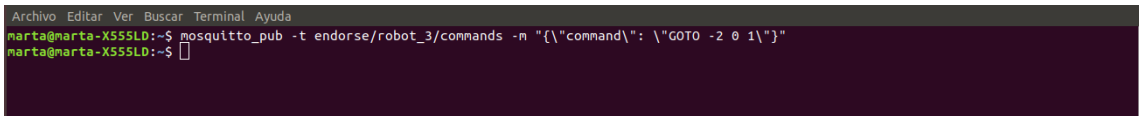

Imagen 5.7. – Publicación en el puente MQTT a través de la terminal.

Imagen 5.8. – Mensaje en MQTT Explorer y respuesta del robot (feedback).

Hasta este punto, podemos incluso enviar un punto en el mapa a través del puente, tal y como hemos hecho en la imagen 5.7. y el robot al que se lo indiquemos se moverá hasta dicho punto.

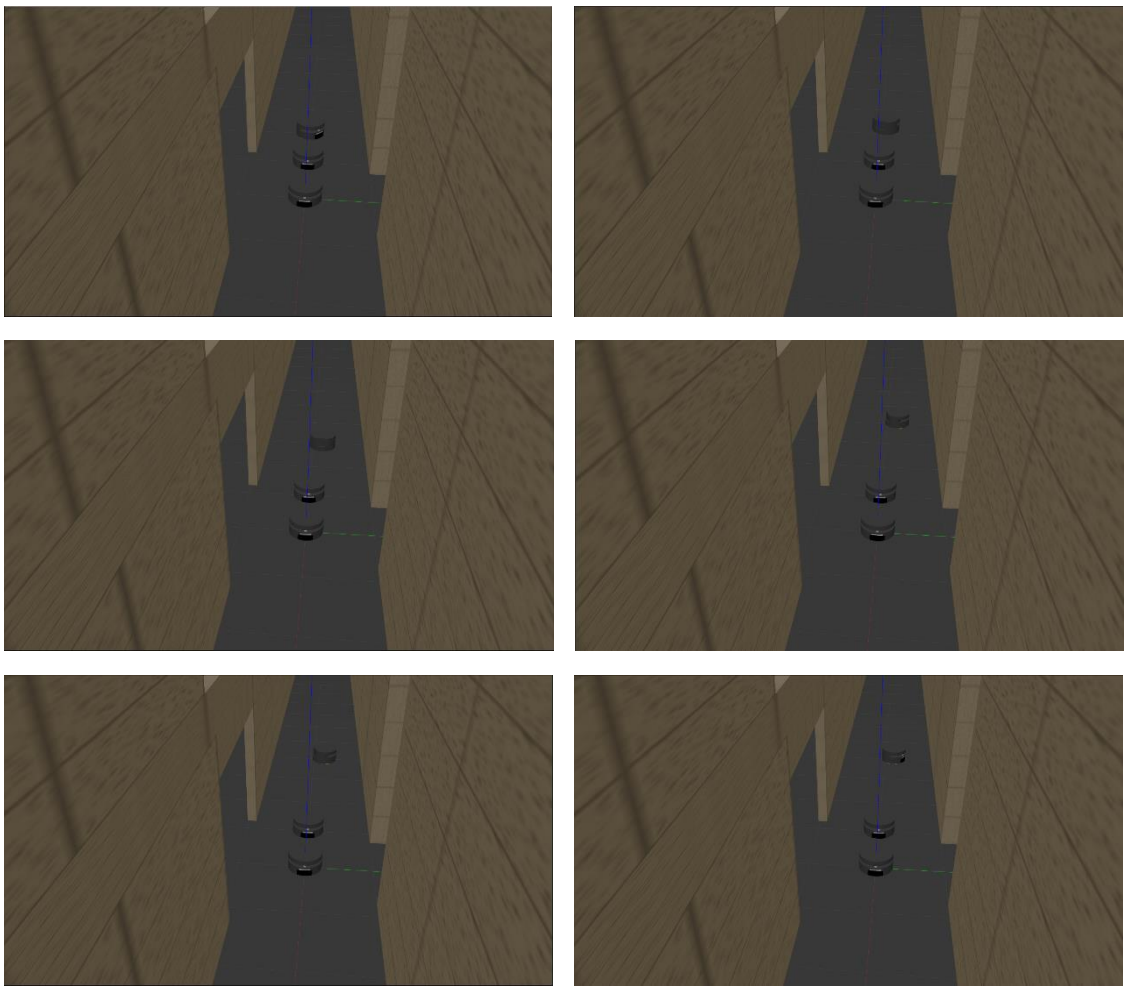

Imagen 5.9. – Secuencia de Imágenes donde el robot 3 se desplaza hasta su objetivo.

5.2.2. Configuración del puente MQTT

Una vez hemos preparado el entorno para trabajar, podemos comenzar a realizar los cambios pertinentes para que el FMS pueda comunicarse con los robots haciéndolos parar o continuar en función de la ocupación de las distintas salas.

Para comenzar, debemos dirigirnos al paquete *command_interface* dentro del proyecto *endorse_command_interface,* y en el directorio *config* modificar el fichero

mqtt_bridge.yaml, que, tal y como hemos comentado en apartados anteriores, es el fichero de configuración que define a qué *topics* ROS se corresponden los *topics* MQTT, en el caso de que deseemos una comunicación desde el FMS hasta los robots; o qué *topics* MQTT se corresponden con los *topics* ROS en el caso de que deseemos una comunicación desde los robots hasta el FMS.

La estructura que sigue este fichero es la siguiente:

```
mqtt:
  client:
    protocol:
  connection:
    host:
    port:
    keepalive:
bridge:
  - factory:
    msg_type:
    topic_from:
    topic_to:
```

Donde podremos añadir todos los *topics* que deseemos traducir. Donde *host* y *port* serán la ip y el puerto donde publicaremos el bróker MQTT. Con respecto al apartado *bridge,* en *factory* deberemos especificar si la comunicación va a ser de MQTT a ROS o de ROS a MQTT; en *msg_type* el tipo de mensaje que usan los *topics* que vamos a configurar; y *topic_from* y *topic_to* indican qué *topic* en un sistema, se corresponde con otro topic en el sistema paralelo.

En nuestro caso, deseamos dar de alta dos *topics*, uno que publique en el puente la odometría de los robots para que el FMS pueda conocer sus ubicaciones, al que hemos llamado *endorse/robot_x/ubic,* cuyo tipo de mensaje es *Odometry* del paquete de mensajes de ROS *nav_msgs.* Con respecto al segundo *topic* que vamos a dar de alta es uno que nos permita enviar al puente un mensaje de parada, donde publicando un 1 o un 0 el robot deberá parar o continuar, respectivamente, de forma que al publicar un mensaje en el puente, sobre el *topic endorse/robot_x/stop* el robot podrá leerlo en el *topic /rb1_base_x/command_manager/stop*. Este *topic* usará un mensaje de tipo entero (*Int32*), perteneciente al paquete *std_msgs.*

Cada vez que realicemos un cambio en la imagen local, deberemos subir los ficheros modificados al repositorio y posteriormente volver a descargar la imagen Docker en nuestro equipo. Hay que tener en cuenta que la actualización de la imagen no es inmediata, ya que la propia aplicación Docker Hub compila la imagen que hemos subido para comprobar que no existe ningún error, reportándolos automáticamente en el caso de que los localice y de ser así, no actualiza el repositorio para que la versión en la nube sea siempre una imagen sin fallos. Esto hace de Docker Hub una herramienta muy útil.

Los pasos a seguir para actualizar el repositorio son los siguientes:

```
$ git add .
```

```
$ git commit -m "Comentario sobre la actualización"

$ git push
```

Una vez hecho esto, y tras unos minutos, podremos volver a actualizar la imagen local ejecutando el siguiente comando:

```
$ docker pull
registry.gitlab.com/rb-
log/endorse_command_interface:command-interface
```

Llegados a este punto, el sistema está listo para que se intercambie la información necesaria.

5.2.3. Aplicación ROS

La aplicación ROS, que se ejecutará en los robots, es muy sencilla. Consiste en tres ficheros Python, dos clases y un script. La clase *Stop_sub()*, creada en el fichero *stop_sub_class.py,* se va a encargar de suscribirse al *topic /rb1_base_x /command_manager/stop* y la clase *Command_sub(),* al *topic /rb1_base_x /command_manager/command*, actualizando constantemente una variable global cada vez que uno de estos *topics* reciba un mensaje, y pudiendo obtener dichos valores con una función *get*. A continuación, el script principal (*app.py*) importará ambas clases para utilizarlas leyendo los valores que estas le proporcionan, tal y como se trata de reflejar en la imagen 5.10.

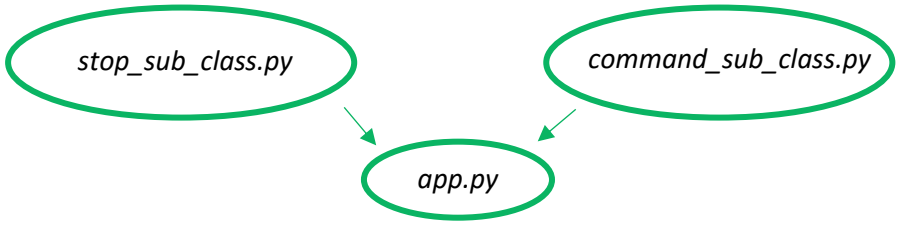

Imagen 5.10. – Estructura de la aplicación endorse_app.

En este script *app.py*, nos encontramos un bucle infinito que solo finaliza cuando lo paramos manualmente a través de Ctrl+c, y en él que se comprueba constantemente si el robot ha recibido un señal de parada, a través de la función *get* de la clase *Stop_sub(),* de forma que, en caso afirmativo, almacena el último comando recibido, obteniéndolo de la clase *Command_sub().* En el momento en el que la función *get* de la clase *Stop_sub()* devuelva un 0, significará que se ha solicitado al robot que retome su camino, y por lo tanto la aplicación volverá a enviar el comando que había almacenado.

Para crear un paquete en ROS, un entorno con *Robot Operating System*, permite una funcionalidad muy práctica en la que, mediante el comando *catkin_create_pkg,* podemos crear un paquete de forma rápida y añadiendo las dependencias necesarias sin necesidad de modificar a mano los ficheros de configuración. Sin embargo, al trabajar sobre una imagen Docker, y al tratarse de una aplicación sencilla, no hemos visto

necesario instalar ROS en nuestro equipo, y hemos escrito los ficheros de configuración desde cero.

La dependencia principal de este paquete es a *rospy,* que no es más que una librería de ROS que proporciona una API para usar Python en el entorno ROS. Además, también son necesarias las dependencias a los paquetes de mensajes que se van a utilizar, estos son *std_msgs* y el paquete de mensajes creados por la empresa Robotnik, *robot_simple_command_manager_msgs.* Estas dependencias habrá que añadirlas al fichero de configuración XML bajo la etiqueta *<exex_depend>,* y en el caso de *rospy,* también con la etiqueta *<build_depend>.* Mientras que en el fichero TXT, tan solo deberemos añadir *rospy* en la función *catkin_package()* con la etiqueta *CATKIN_DEPENDS.*

En este punto, ya podremos probar nuestra aplicación en el enrono de simulación. Para ello, debemos generar un fichero LAUNCH que ejecute el script y lance el nodo que hemos definido en el mismo. Los ficheros LAUNCH son muy útiles y comunes en entornos ROS, ya que proporcionan una forma sencilla de ejecutar múltiples nodos o inicializar distintos parámetros del entorno. El contenido de este fichero, *endorse_app_launch.launch,* es el siguiente:

```
<launch>
    <node pkg="endorse_app"
        type="app.py"
        name="endorse_app_node"
        output="screen">
    </node>
</launch>
```

Donde podemos distinguir dos etiquetas: *launch* y *node*. Dentro de *node*, nos encontramos con cuatro argumentos, los cuales deberemos definir con respecto a nuestra aplicación. Donde *pkg* hace referencia al nombre del paquete, *type* contiene el nombre del script a ejecutar, en *name* debemos indicar el nombre del nodo que hemos definido en el fichero Python, y *output* hace referencia a la salida estándar por donde el robot mostrará cierta información, como los mensajes log.

Finalmente, ya podemos ejecutar nuestra aplicación lanzando el fichero LAUNCH que acabamos de mencionar. Para ello, deberíamos usar el siguiente comando:

```
$ roslaunch endorse_app endorse_app_launch.launch
```

Sin embargo, nosotros estamos trabajando sobre una imagen Docker, y es por ello que debemos crear nuestro propio script que se encargue de lanzar la aplicación y al cual llamaremos *docker_rub_endorse_app.sh*. Este script contendrá el comando *docker run* con una serie de parámetros que permiten ejecutar una imagen contenedora, en nuestro caso, esta imagen contenedora se encuentra en el repositorio de GitLab, *endorse_command_interface.* Cabe destacar en este script la función *bash,* donde a

través de la etiqueta *-c,* podemos lanzar un comando en la terminal de la imagen Docker, que en nuestro caso será el comando especificado unas líneas más arriba.

Una vez generados todos estos ficheros, debemos actualizar el repositorio tanto en la nube como en nuestro equipo, y es por ello que ejecutamos de nuevo los comandos del apartado 5.2.2., que usábamos para actualizar el fichero *mqtt_bridge.yaml*. Pasados unos instantes se habrá actualizado la imagen Docker y podremos lanzar nuestra aplicación. Para ello, preparamos el entorno tal y como comentamos en el apartado 5.2.1. y ejecutamos el script:

```
$ ./docker_run_endorse_app.sh
```

Cabe destacar que, la aplicación es distinta para cada uno de los robots, puesto que cada aplicación se ejecuta en el propio robot y hace referencia a sí mismo. La modificación es pequeña, ya que se ha especificado una variable global llamada *robot* en la que deberemos indicar el nombre que recibe el robot en el sistema.

En el apartado siguiente hablaremos de las pruebas tanto en el entorno simulado como en un entorno real con un robot RB-1.

6. Pruebas y resultados

En este apartado se va a tratar de explicar de forma detallada el proceso de realización de pruebas y resultados tanto en el entorno simulado como en el entorno real, que en este caso va a ser el ai2. Las pruebas se van a realizar con un único robot, con el cual trataremos de comunicarnos a través de MQTT y mediante la configuración que hemos comentado en el apartado anterior. Además, usaremos nuestra aplicación para gestionar la solicitud de parada al robot cuando sea necesario, así como que continúe su trayectoria hasta su objetivo en el caso de solicitarlo.

6.1. Entorno simulado

Las pruebas en simulación han consistido en preparar el entorno, lanzar el puente MQTT y finalmente ejecutar la aplicación, tal y como comentamos en el apartado 5.2. A continuación, deberemos probar a publicar un mensaje para que uno de los robots se mueva, y posteriormente pararlo a través del *topic* que hemos configurado para ello. Comprobar que efectivamente el robot se para, y que tras mandarle una señal para que continúe, este sigue su camino hasta su objetivo inicial.

Una vez hemos preparado el entorno, mediante la herramienta MQTT Explorer nos conectamos al bróker, en el host y puerto en la que lo hayamos lanzado, en este caso, al haberlo configurando en nuestro host local podemos especificar tanto la palabra clave *localhost* como la ip 127.0.0.1, tal y como podemos observar en la imagen 6.1.

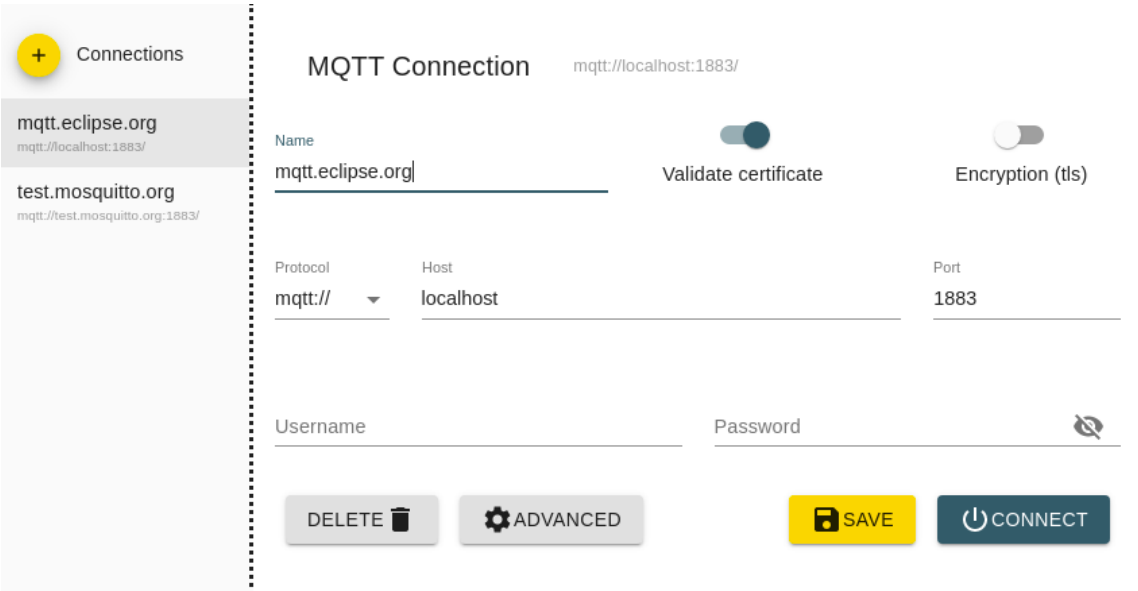

Imagen 6.1. – Configurando MQTT Explorer para conectarse a localhost.

Al conectarnos, comprobaremos que se publican una serie de mensajes $SYS, los cuales ignoraremos puesto que hacen referencia al estado del bróker. A continuación,

enviamos a través de la terminal un comando para poner uno de los robots en movimiento, en este caso el robot número dos:

```
$ mosquitto_pub -t endorse/robot_2/commands -m
"{\"command\": \"MOVE 1 0\"}"
```

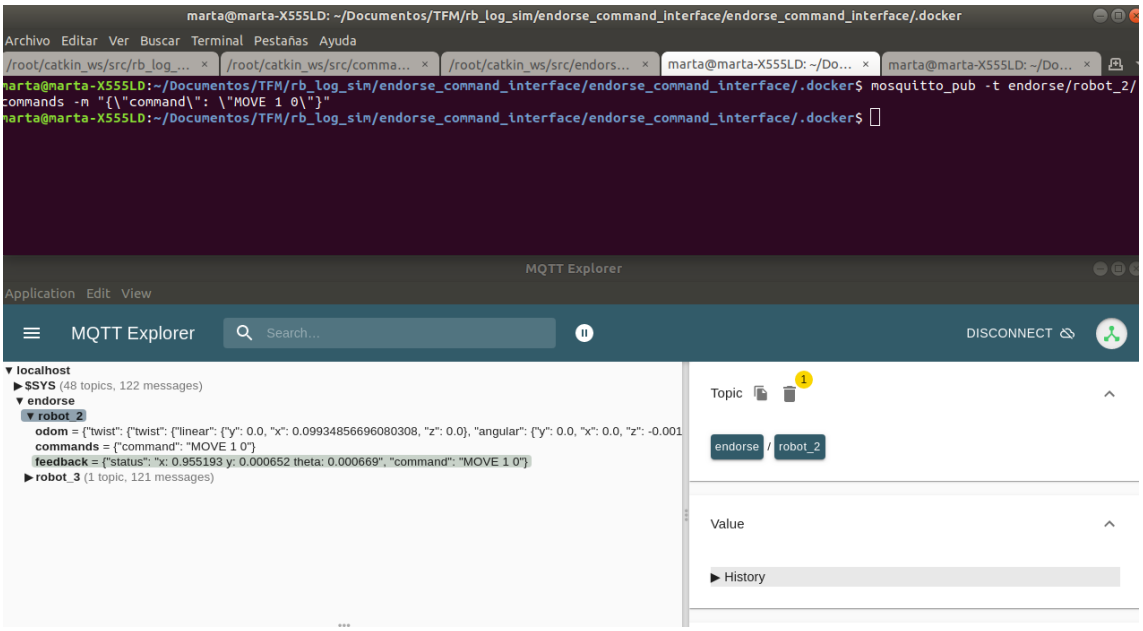

Imagen 6.2. – Captura enviando comando MOVE.

Como podemos observar en la imagen 6.2., el robot al que se le ha enviado el comando está respondiendo al puente con su ubicación en todo momento. Esto podemos observarlo porque, cuando el puente recibe un mensaje, la aplicación MQTT Explorer lo refleja iluminando el *topic* en color gris.

Imagen 6.3. – Captura de robot moviéndose.

Una vez hecho esto, y tras haber observado que el robot inicia su movimiento, procedemos a pararlo lanzando el siguiente comando, el cual veremos reflejado en la aplicación MQTT Explorer, y en Gazebo deberemos observar como el robot se para:

```
$ mosquitto_pub -t endorse/robot_2/stop -m "{\"data\": 1}"
```

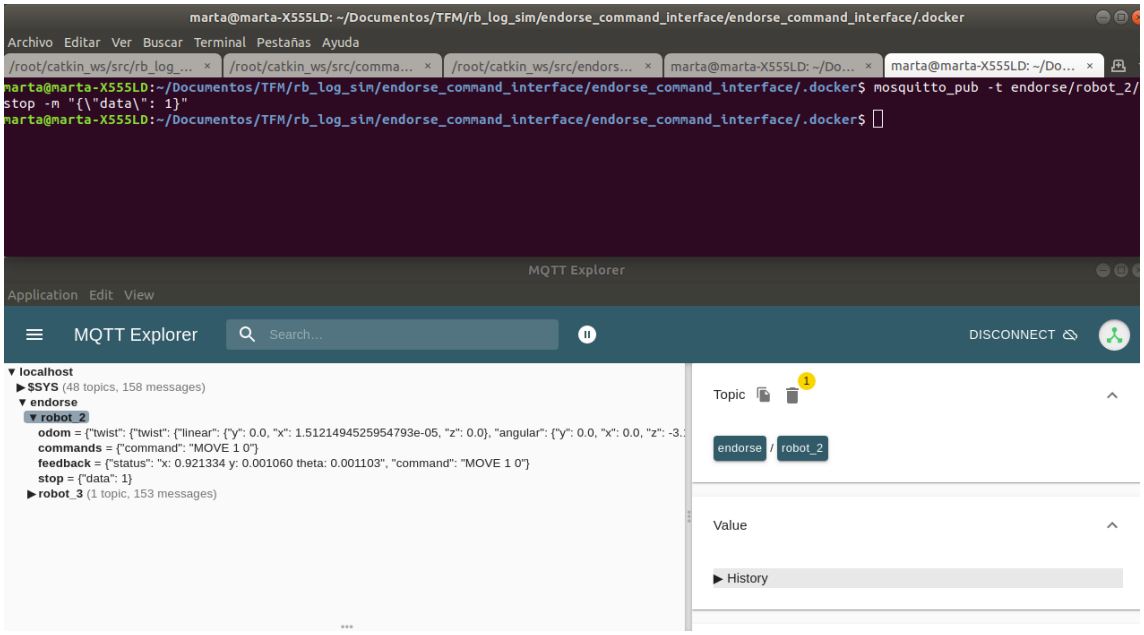

Imagen 6.4. – Imagen parando al robot.

Como podemos comprobar, en este caso no existe retroalimentación por parte del robot (el *topic* no se ilumina como ocurría en la imagen 6.2.), al haber recibido este una solicitud de parada.

Imagen 6.5. – Robot parando tras recibir comando de parada.

Podemos observar que, efectivamente, el robot se para cuando se lo solicitamos. Para finalizar, ordenaremos al robot que continúe su camino, y deberemos observar como en Gazebo se mueve hasta alcanzar su objetivo:

```
$ mosquitto_pub -t endorse/robot_2/stop -m "{\"data\": 0}"
```

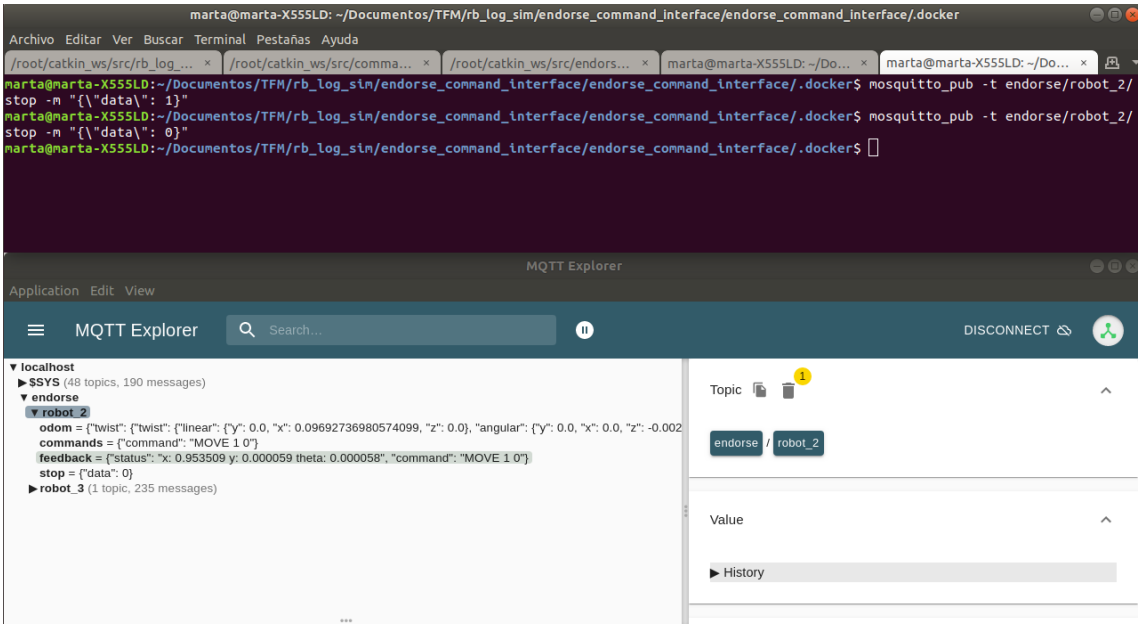

Imagen 6.6. – Captura enviando comando para continuar.

En este caso, como se puede comprobar en la imagen 6.6., se vuelve a iluminar el *topic feedback*, tal y como ocurría en la imagen 6.2., al recibir el puente de nuevo mensajes por parte del robot.

Imagen 6.7. – Captura de robot continuando su trayectoria.

Como Podemos observar, al enviar un 0 a través del *topic endorse/robot_2/stop* el robot lo entiende como que puede continuar su camino. Comprobamos que el robot se mueve de nuevo, y podemos dar por hecho que la aplicación funciona correctamente.

El cambio entre las figuras 6.3, 6.5 y 6.7 es prácticamente imperceptible. Sin embargo, el *robot_1* da problemas en la simulación, y decidimos que para reflejar el funcionamiento en imágenes lo más representativo era utilizar el comando MOVE, en este caso sobre el *robot_2,* y así poder observar una diferencia en la distancia con respecto a los otros dos robots.

También se ha programado la aplicación para que se muestre en todo momento las acciones que se le ha solicitado al robot y cómo las ha interpretado este, tal y como podemos observar en la imagen 6.8. Esta información se almacenará en los *logs* del sistema, lo cual es muy práctico a la hora de localizar un posible *bug.*

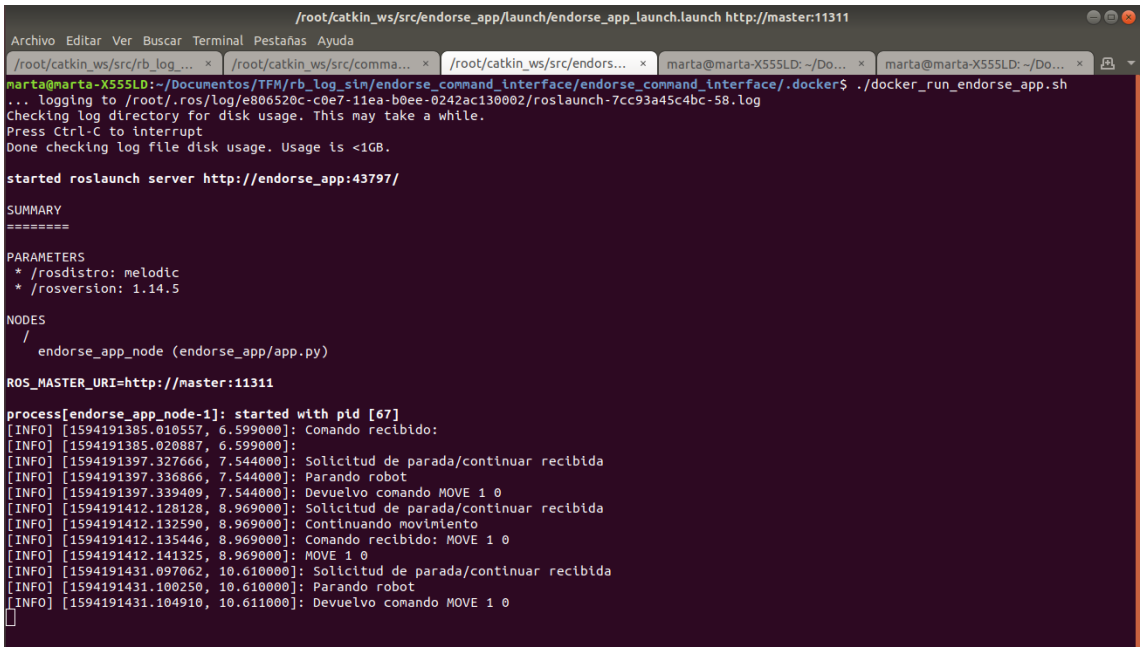

Imagen 6.8. – Captura de los mensajes que imprime la aplicación durante su ejecución.

6.2. Entorno real

Una vez hemos realizado las pruebas pertinentes en el entorno simulado y hemos comprobado que nuestra aplicación funciona sin problemas, podemos comenzar a realizar pruebas en un entorno real en el caso de disponerlo. Al estar trabajando en el ai2, contamos con un robot RB-1 que nos permitirá realizar pruebas iniciales sobre todo en lo referente a la configuración y las comunicaciones.

Como ya hemos comentado en alguna ocasión, es importante tener en cuenta que no es lo mismo trabajar en una simulación que en la realidad. Es por ello que habrá que realizar ciertas modificaciones en los paquetes que hemos estado utilizando hasta ahora, sobre todo por el hecho de que, en este caso, tan solo disponemos de un robot,

y no de tres como era el caso de las simulaciones. Los paquetes que deberemos modificar son los siguientes:

- **command_interface**: paquete que utiliza el paquete *mqtt_explorer* para traducir *topics* ROS a *topics* MQTT. En este fichero deberemos modificar todos los ficheros en los que se hace referencia a los distintos robots, tanto ficheros YAML de configuración como ficheros LAUNCH, lanzadores. En ellos eliminaremos las referencias a *robot_base_b* y *robot_base_c*, y mantendremos *robot_base_a*, sustituyéndolo por *robot_base*. Los ficheros a los que hacemos referencia son los que se muestran en la imagen 6.9., donde podemos observar dónde se encuentran cada uno de los ficheros a los que hacemos referencia, así como la forma en la que se estructuran dentro del paquete *command_interface*:

```
marta@marta-X555LD:~/Documentos/TFM/robot_pkgs/command_interface$ tree
.
├── CMakeLists.txt
├── config
│   ├── command
│   │   └── rb1_base_handlers.yaml
│   ├── diff_docker.yaml
│   ├── move.yaml
│   └── mqtt_bridge.yaml
├── launch
│   ├── command_interface_complete.launch
│   ├── command_manager.launch
│   ├── dockers.launch
│   ├── move.launch
│   └── mqtt_bridge.launch
└── package.xml
```

Imagen 6.9. – Captura de la estructura del paquete command_interface.

- **endorse_app**: Con respecto a la aplicación que hemos desarrollado, debemos modificar la variable global *robot*, la cual mencionamos en el apartado 5.2.3., sustituyéndola por *robot_base*, que es el nombre que recibe el robot del que disponemos en nuestro laboratorio.

Una vez hecho esto, podemos proceder a copiar los paquetes necesarios en nuestro robot a través del comando Linux, *scp*. Los paquetes necesarios son los siguientes:

- command_interface
- endorse_app
- mqtt_bridge
- rcomponent
- robot_simple_command_manager
- robot_simple_command_manager_msgs
- robotnik_msgs

Gracias a ellos, podemos hacer uso de los comandos de los que hablamos en el apartado 4.2.5., y que el robot los comprenda y ejecute. Pero antes de esto, debemos asignar al robot una IP fija dentro de la red por la que nos vamos a comunicar, es este caso la red de la UPV. En nuestro caso le hemos asignado la IP 158.42.165.114. Este trabajo corresponde al servicio técnico de la Universidad, sin embargo, en una red local no sería

necesario este paso intermedio, podríamos hacerlo desde nuestro propio router o a través del fichero de configuración del robot.

Una vez tenemos al robot conectado a la red de la UPV, podemos acceder a él de forma remota a través de *ssh*, y una vez dentro modificar el host de trabajo del robot a la IP que nos han asignado. Los pasos serán los siguientes:

1. Nos conectamos a la red de la universidad.
2. Establecemos un canal seguro por el que acceder al robot (para ello deberemos saber el usuario del sistema del RB-1, en este caso *rb1,* así como su contraseña):

```
$ ssh rb1@158.42.165.114
```

3. Una vez dentro, podemos modificar la variable de entorno ROS_MASTER_URI, la cual se encarga de definir el host donde los nodos localizarán el nodo máster. Al conectar el robot a una red común, y asignarle una IP, podremos comunicarnos con él a través de la misma, así como añadir tantos robots como direcciones de IP nos queden libres:

```
$ export ROS_MASTER_URI=158.42.165.114
```

Una vez hecho esto, copiamos los paquetes en el directorio de trabajo del robot, llamado *catkin_ws.* En el directorio donde tenemos los paquetes necesarios ejecutamos lo siguiente*:*

```
$ scp -r * rb1@158.42.165.114:/home/rb1/catkin_ws/src
```

A continuación, dentro de una terminal en *ssh,* compilamos el espacio de trabajo situándonos en el directorio *catkin_ws* y ejecutando lo siguiente:

```
$ catkin_make
```

Debemos tener en cuenta que para poder ejecutar el puente, debemos instalar unas librerías, sin embargo, se nos ha facilitado el trabajo a través de un fichero llamado *requirements.txt* con lo que solo deberemos situarnos en la raíz del paquete *mqtt_bridge* y ejecutar lo siguiente:

```
$ pip install -r requirements.txt
```

El siguiente paso será instalar el servidor mosquitto en el robot y lanzarlo, para ello ejecutamos los siguientes comandos:

```
$ sudo apt-get install mosquitto
```

```
$ service mosquitto start
```

Una vez hecho esto, podemos lanzar nuestro puente mediante el comando:

```
$ roslaunch command_interface
command_interface_complete.launch
```

En este punto, podremos conectarnos al bróker mosquitto siempre y cuando estemos conectados en la misma red. Para comprobarlo, en la aplicación MQTT Explorer

indicamos la IP del robot, y observaremos que, efectivamente, podemos conectarnos al bróker y escuchar los mensajes de estado que este envía a través de MQTT.

También podremos lanzar comandos desde cualquier punto de la red, y deberemos poder verlos en la aplicación MQTT Explorer, tal y como podemos ver en la imagen 6.10. Es importante destacar que en este caso debemos especificar la IP del bróker, ya que por defecto toma *localhost*:

```
$ mosquitto_pub -h 158.42.165.114 -t
endorse/robot_1/commands -m "{\"command\": \"MOVE" 1 0\"}"
```

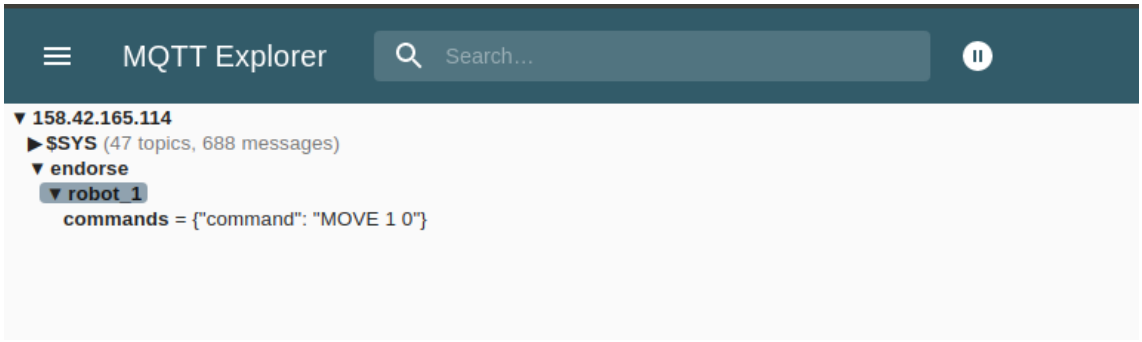

Imagen 6.10. – Captura del puente mostrando el mensaje que acaba de recibir.

Una vez hecho esto, y habiendo comprobado que el robot se mueve, lo pararemos enviándole un 1 a través del *topic endorse/robot_1/stop*:

```
$ mosquitto_pub -h 158.42.165.114 -p port -t
endorse/robot_1/stop -m "{\"data\": 1}"
```

Observaremos que el robot se para, y a continuación retomaremos su marcha enviando:

```
$ mosquitto_pub -h 158.42.165.114 -p port -t
endorse/robot_1/stop -m "{\"data\": 0}"
```

En este punto podemos dar por concluido el apartado de pruebas, al habernos podido comunicar sin problemas con un robot a través de MQTT. El siguiente paso será hacer uso de más de un robot para poder implantar nuestra flota.

7. Implantación y Trabajos futuros

Con respecto a la implantación, la situación actual ha complicado mucho el proceso. Durante el mes de febrero de este año 2020, pudimos hacer uso de los tres robots que se pueden ver en la imagen 7.1., los cuales entrarán en el proceso de implantación del proyecto ENDORSE. Durante unos días pudimos probarlos, trabajar con ellos y observar cómo interactúan con el entorno.

Imagen 7.1. – Equipo ENDORSE durante el Term Review en Valencia, 25 de febrero de 2020.

Durante esta Term Review, acordamos poder disponer de los tres robots para las fechas de entrega de este TFM, para así poder comprobar el correcto funcionamiento de las comunicaciones a través de los puentes MQTT y la interacción entre los robots y un dispositivo centralizado como el FMS. Sin embargo, debido al COVID-19, que nos obligó a aislarnos en nuestras casas durante tres meses, no ha sido posible contar en el ai2 con estos tres robots, de forma que no hemos podido hacer uso de ellos. Debido a esto, el proceso de implantación se ha visto paralizado durante un tiempo, no alcanzando la fecha actual de entrega. A pesar de ello, prevemos poder empezar a hacer pruebas a partir de septiembre.

En su defecto hemos llegado a tiempo de utilizar el RB-1 del que sí disponemos en el ai2, tal y como hemos mencionado en el apartado 6.2., lo cual nos ayuda a comprender cómo habrá que configurar el sistema una vez dispongamos de la flota de robots.

Con respecto a los trabajos futuros, una vez nuestra aplicación sea capaz de gestionar una planta entera de un edificio, sería muy interesante la adaptación de esta a edificios con más de una planta. Los robots podrían comunicarse con los ascensores a través de MQTT tal y como hacen ahora, para solicitar que bajen o suban en función de sus intereses. Pudiendo así controlar que todas las plantas estén cubiertas en la medida de

lo posible, sin saturaciones en los ascensores ni dejando desatendida ninguna de las plantas.

Esta solución puede ser muy interesante, ya que, al fin y al cabo, los edificios para los que está destinado nuestro proyecto, suelen ser lugares de gran tamaño y con más de una planta.

8. Conclusiones

Teniendo en cuenta los objetivos que se plantearon en la introducción de esta memoria, y una vez hemos finalizado esta parte del proyecto, podemos concluir que, al menos en gran parte, los objetivos han sido satisfechos.

Para empezar, hemos podido estudiar las herramientas y los sistemas utilizados de forma profunda, hasta comprenderlos para así poder usarlos y modificarlos. Comenzamos estudiando el *Fleet Management System* (FMS), contactando con los miembros del CIGIP para que nos explicaran ciertas partes de su trabajo y para ponernos de acuerdo sobre la estandarización. Principalmente, hubo que mantener el contacto con el fin de acordar el tipo de información que deberían intercambiar los robots y el FMS, de qué forma y por qué canal, para que pudieran entenderse mutuamente.

A continuación, comenzamos a estudiar las distintas formas de comunicación entre sistemas con ROS; de hecho, llegamos a realizar pruebas con sistemas *multi-master*, y en seguida llegamos a la conclusión de que no era una manera óptima de trabajar. Nos pusimos en contacto con Robotnik, empresa que también forma parte del proyecto, y nos explicaron que el mejor modo de trabajar era a través de puentes MQTT. En esta ocasión, ya habíamos trabajado con MQTT durante la asignatura de Redes y Sistemas distribuidos para control, del máster de Automática e Informática Industrial. Este trabajo previo fue de gran ayuda durante el transcurso del proyecto.

A pesar de que se nos facilitaron entornos de simulación ya desarrollados, lo cual ha sido muy útil para el desarrollo del proyecto, hemos tenido la oportunidad de trabajar con herramientas nuevas como Docker, o recordar otras que teníamos un poco olvidadas como git. Nos hemos topado de pleno con problemas totalmente reales y gracias a los conocimientos adquiridos durante el grado de Ingeniería Informática y el máster AII, hemos tenido las herramientas suficientes para solventarlos.

Finalmente, para el desarrollo de la aplicación en ROS, fue necesario tomar unas nociones previas antes de abordar el problema. *Robot Operating System* es un sistema cada vez más común en la programación de robots y ha sido muy enriquecedor tener la oportunidad de probarlo en un robot real. Ya que, a pesar de que las simulaciones siempre son útiles, trabajar sobre un entorno real nos va a presentar problemas que solventar muy prácticos de cara a un futuro laboral.

En conclusión, el desarrollo de este proyecto ha sido muy enriquecedor. Desde problemas técnicos que nos han proporcionado conocimientos sobre distintas herramientas, tecnologías y lenguajes, hasta la cooperación grupal. Este proyecto nos ha permitido valorar la importancia de una buena comunicación entre los distintos miembros del proyecto, para poder trabajar en equipo, y la responsabilidad de formar parte de un proyecto común, en el cual se otorgan ciertas responsabilidades que cumplir.

A nivel técnico considero que este TFM ha sido el broche final a un máster multidisciplinar como es el máster de Automática e Informática Industrial. Ha sido muy satisfactorio para mí sacar adelante una parte del proyecto, en la que he tenido la oportunidad de trabajar con personas muy cualificadas que me han ofrecido siempre su ayuda y me han empujado a aprender.

Además, he tenido la oportunidad de desarrollar este proyecto trabajando simultáneamente en el Instituto de Automática e Informática Industrial, lo que me ha permitido trabajar de primera mano con todo tipo de tecnología puntera en el campo I+D+I dentro del panorama actual, como es el caso del RB-1 utilizado en este proyecto, así como cooperar al lado de compañeros con grandes aptitudes y siempre dispuestos a colaborar y ayudar.

9. Referencias

[1] Robot Móvil RB-1 Base [En línea] [fecha de consulta: 2 de junio de 2020]. Disponible en: https://robotnik.eu/es/productos/robots-moviles/rb-1-base/

[2] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng. "ROS: an open-source Robot Operating System" [fecha de consulta: 2 de junio de 2020]

[3] European Conference on Mobile Robots [fecha de consulta: 2 de junio de 2020]. Disponible en: https://www.ecmr2019.eu/

[4] MecatrónicaLATAM. "Tipos de Robots" [En línea] [fecha de consulta: 2 de junio de 2020]. Disponible en: https://www.mecatronicalatam.com/es/tutoriales/tipos-de-robots/

[5] S. Mahmud, X. Lin and J. Kim, "Interface for Human Machine Interaction for assistant devices: A Review," 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 2020, pp. 0768-0773, doi: 10.1109/CCWC47524.2020.9031244. [fecha de consulta: 3 de junio de 2020]

[6] All about Python. [En línea] [fecha de consulta: 4 de junio de 2020]. Disponible en: https://www.python.org/about/

[7] A Platform to Learn/Teach Robotics form Zero [En línea] [fecha de consulta: 4 de junio de 2020]. Disponible en: https://www.theconstructsim.com/

[8] Clinic Cloud [En línea] [fecha de consulta: 5 de junio de 2020]. Disponible en: https://clinic-cloud.com/blog/servicios-en-la-nube-tipos-ejemplos/

[9] J. A. Dulce-Galindo, M. A. Santos, G. V. Raffo and P. N. Pena, "Autonomous Navigation of Multiple Robots using Supervisory Control Theory," 2019 18th European Control Conference (ECC), Naples, Italy, 2019, pp. 3198-3203, doi: 10.23919/ECC.2019.8796261. [fecha de consulta: 9 de junio de 2020]

[10] David Miquel. "Seguridad en robots colaborativos (COBOTS)" [En línea] [fecha de consulta: 10 de junio de 2020]. Disponible en: https://www.integra-sti.com/marcado-ce/seguridad-en-robots-colaborativos-cobots/

[11] ROS Documentation. "multimaster_fkie" [En línea] [fecha de consulta: 15 de junio de 2020]. Disponible en: http://wiki.ros.org/multimaster_fkie

[12] Sergi Hernandez Juan, Fernando Herrero Cotarelo, "Multi-master ROS systems", January, 2015, IRI. [fecha de consulta: 15 de junio de 2020]

[13] ¿Qué es garantía juvenil? [En línea] [fecha de consulta: 16 de junio de 2020]. Disponible en: https://www.sepe.es/HomeSepe/Personas/encontrar-trabajo/Garantia-Juvenil/que-es-garantia-juvenil

[14] Y. Orlando, C. Castro. "Aproximación da los grafos de Voronoi y diagramas de Delaunay". Encuentro Distral de Educación Matemárica EDEM. Volumen 3, año 2016. ISSN 2422-037X [En línea] [fecha de consulta: 22 de junio de 2020]. Disponible en: http://funes.uniandes.edu.co/10253/1/Orlando2016Aproximaci%C3%B3n.pdf

[15] Frequently Asked Questions. [En línea] [fecha de consulta: 25 de junio de 2020]. Disponible en: http://mqtt.org/faq

[16] What is a Container? [En línea] [fecha de consulta: 25 de junio de 2020]. Disponible en: https://www.docker.com/resources/what-container

[17] Gazebo. [En línea] [fecha de consulta: 25 de junio de 2020]. Disponible en: http://gazebosim.org/

[18] Introducción a JSON [En línea] [fecha de consulta: 25 de junio de 2020]. Disponible en: https://www.json.org/json-es.html

[19] mongoDB [En línea] [fecha de consulta: 25 de junio de 2020]. Disponible en: https://www.mongodb.com/es

[20] Repositorio Digital [En línea] [fecha de consulta: 30 de junio de 2020]. Disponible en: https://www.bib.upct.es/repositorio-digital

[21] Instituto de Automática e Informática industrial de la UPV [En línea] [fecha de consulta: 1 de julio de 2020]. Disponible en: https://www.ai2.upv.es/

[22] Clearpath. "Launch files" [En línea] [fecha de consulta: 6 de julio de 2020]. Disponible en: http://www.clearpathrobotics.com/assets/guides/kinetic/ros/Launch%20Files.html

[23] N. Martínez Medina. "James Watt, la máquina de vapor y el origen de la revolución industrial" [En línea] [fecha de consulta: 7 de julio de 2020]. Disponible en: https://www.rtve.es/noticias/20110211/james-watt-maquina-vapor-origen-revolucion-industrial/404679.shtml

[24] AERATP. "Estadísticas de robótica industrial en España" (Año 2015) [fecha de consulta: 7 de julio de 2020]

[25] AER Automation. "Robótica Colaborativa" [En línea] [fecha de consulta: 7 de julio de 2020]. Disponible en: https://www.aer-automation.com/mercados-emergentes/robotica-colaborativa/

[26] Wiki ROS. "EnvironmentVariables" [En línea] [fecha de consulta: 7 de julio de 2020]. Disponible en: http://wiki.ros.org/ROS/EnvironmentVariables

[27] S. Hernandez Juan, F. Herrero Cotarelo. "Multi-master ROS systems" (Junio de 2015) [fecha de consulta: 7 de julio de 2020]

[28] N. Al-Sabbagh, A. Al-Omary. "A centralized Multi-Floor Indoor Navigation System for a Large Mall" (2019 8ª ICMSAO) [fecha de consulta: 8 de julio de 2020]

[29] HostDimeBlog. “FRAMEWORK” [En línea] [fecha de consulta: 9 de julio de 2020]. Disponible en: https://blog.hostdime.com.co/que-es-un-framework-informatica-programacion/

[30] Orix. “Qué es un Framework” [En línea] [fecha de consulta: 9 de julio de 2020]. Disponible en: https://www.orix.es/que-es-un-framework-y-para-que-se-utiliza

[31] Desarrolloweb. “Qué es la programación orientada a objetos” [En línea] [fecha de consulta: 9 de julio de 2020]. Disponible en: https://desarrolloweb.com/articulos/499.php

[32] Digital Guide IONOS. “Programación imperativa: resumen del paradigma de programación más antiguo” [En línea] [fecha de consulta: 9 de julio de 2020]. Disponible en: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/programacion-imperativa/

[33] Digital Guide IONOS. “Programación funcional: ideal para algoritmos” [En línea] [fecha de consulta: 9 de julio de 2020]. Disponible en: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/programacion-funcional/

[34] Bigeek. “Bróker de mensajería” [En línea] [fecha de consulta: 9 de julio de 2020]. Disponible en: https://blog.bi-geek.com/broker-mensajeria/

[35] Platzi. “Qué es Frontend y Backend” [En línea] [fecha de consulta: 9 de julio de 2020]. Disponible en: https://platzi.com/blog/que-es-frontend-y-backend/

[36] Guillaume Bresson, Zayed Alsayed, Li Yu, Sébastien Glaser. “Simultaneous Localization And Mapping: A Survey of Current Trends in Autonomous Driving”. IEEE Transactions on Intelligent Vehicles, Institute of Electrical and Electronics Engineers, 2017, XX, pp.1. 10.1109/TIV.2017.2749181. hal-01615897. [fecha de consulta: 9 de julio de 2020]

[37] Deloitte. “IoT – Internet of Things”. [En línea] [fecha de consulta: 9 de julio de 2020]. Disponible en: https://www2.deloitte.com/es/es/pages/technology/articles/IoT-internet-of-things.html

10. Glosario

API:

Abreviatura de las siglas en inglés de Application Porgramming Interface. Una API se define como un conjunto de definiciones y protocolos que facilitan el desarrollo e integración del software de las aplicaciones informáticas.

Backend:

El backend es la parte de un programa informático que no es directamente accesible por los usuarios. Suele desarrollar la función de acceso a datos y uso de los mismos.

Bróker:

Un bróker es considerado un middleware orientado a mensajes, facilitando la transferencia de los mismos, entre dos o más aplicaciones. Estas aplicaciones pueden ser emisoras o receptoras.

Framework:

La palabra framework es una composición de dos palabras, frame (marco) y work (trabajo). Esto significa que un framework es un marco ya diseñado del cual puede hacer uso el desarrollador para conseguir una mejor organización y desarrollo del software.

Frontend:

Es la parte de un programa informático al que puede acceder el usuario, pudiendo interactuar con él de forma directa.

Interfaz amigable:

Podemos definir una interfaz amigable como una estructura intuitiva para el usuario, que permita un uso fluido de la aplicación, y con utilización de elementos reconocibles.

Internet of Things:

IoT se define como “Agrupación e interconexión de dispositivos y objetos a través de una red, dónde todos ellos pueden interaccionar” [37].

Middleware:

El middleware, como su propio nombre indica, es un software intermedio que permite comunicarse o interactuar dos aplicaciones distintas.

Programación funcional:

La programación funcional, como su propio nombre indica se centra en las funciones. En un programa funcional, todos los elementos pueden ser concebidos como funciones y el código puede ser ejecutado mediante llamadas a funciones secuenciales. Un ejemplo de ello son lenguajes como Haskell, Scala o LISP.

Programación imperativa:

La programación imperativa se define como "Secuencia claramente definida de instrucciones para un ordenador" [32]. Ejemplos de ello son los lenguajes Java, C, C++ o Python entre otros.

Programación OO:

Abreviatura de Programación Orientada a Objetos. Esta es definida como un paradigma de la programación donde el código se organiza en elementos que denominamos clases, mediante las cuales se pueden crear objetos que se relacionan entre sí para conseguir los objetivos de las aplicaciones que los utilizan.

SLAM:

Abreviatura de las siglas Simultaneous Localization and Mapping. Hace referencia a un algoritmo que permite obtener la posición y, simultáneamente, un mapa del entorno.

Tecnología maestro-esclavo:

La técnica maestro-esclavo es un modelo de comunicación asimétrico donde uno de los dispositivos (maestro) controla al resto (esclavos).

11. Anexos

En el apartado de anexos, vamos a facilitar los códigos que se han modificado para el uso de este proyecto en un entorno real. Con un único robot tal y como comentamos en el apartado 6.2.

ANEXO 1 – Fichero mqtt_bridge.yaml

```yaml
mqtt:
  client:
    protocol: 4      # MQTTv311
  connection:
    host: localhost
    port: 1883
    keepalive: 60
bridge:
  # Robot A bridges
  - factory: mqtt_bridge.bridge:MqttToRosBridge
    msg_type: robot_simple_command_manager_msgs.msg:CommandString
    topic_from: endorse/robot_1/commands
    topic_to: /rb1_base/command_manager/command
  - factory: mqtt_bridge.bridge:MqttToRosBridge
    msg_type: std_msgs.msg:Empty
    topic_from: endorse/robot_1/cancel
    topic_to: /rb1_base/command_manager/cancel
  - factory: mqtt_bridge.bridge:MqttToRosBridge
    msg_type: robot_simple_command_manager_msgs.msg:CommandString
    topic_from: endorse/robot_1/commands
    topic_to: /rb1_base/command_manager/command
  - factory: mqtt_bridge.bridge:RosToMqttBridge
    msg_type: robot_simple_command_manager_msgs.msg:CommandStringFeedback
    topic_from: /rb1_base/command_manager/feedback
    topic_to: endorse/robot_1/feedback
  - factory: mqtt_bridge.bridge:RosToMqttBridge
    msg_type: nav_msgs.msg:Odometry
    topic_from: /rb1_base/robotnik_base_control/odom
    topic_to: endorse/robot_1/odom
  - factory: mqtt_bridge.bridge:MqttToRosBridge
    msg_type: std_msgs.msg:Int32
    topic_from: endorse/robot_1/stop
    topic_to: /rb1_base/command_manager/stop
```

ANEXO 2 – Fichero command_interface_complete.launch

```
<?xml version="1.0"?>
<launch>

  <arg name="command_input_name" default="command"/>
  <arg name="id_robot" default="rb1_base"/>
  <arg name="launch_robot" default="true"/>

  <include file="$(find command_interface)/launch/command_manager.launch"
>
    <arg name="command_input_name" value="$(arg command_input_name)"/>
    <arg name="id_robot" value="$(arg id_robot)"/>
  </include>

  <include file="$(find command_interface)/launch/dockers.launch" />

  <include file="$(find command_interface)/launch/move.launch" />

  <include file="$(find command_interface)/launch/mqtt_bridge.launch" />

</launch>
```

ANEXO 3 – Fichero command_manager.launch

```
<?xml version="1.0"?>
<launch>
  <arg name="command_input_name" default="command"/>
  <arg name="id_robot" default="rb1_base"/>
  <arg name="launch_robot" default="true"/>

  <!-- ROBOT A COMMAND INTERFACE-->
  <group if="$(arg launch_robot)" ns="rb1_base">
    <node name="command_manager" pkg="robot_simple_command_manager" type=
"simple_command_manager_node.py" output="screen">
      <param name="command_input_name" value="$(arg command_input_name)"/
>
      <rosparam command="load" file="$(find command_interface)/config/com
mand/$(arg id_robot)_handlers.yaml"/>
    </node>
  </group>

</launch>
```

ANEXO 4 – Fichero dockers.launch

```
<?xml version="1.0"?>
<launch>

  <node ns="rb1_base" name="diff_docker" pkg="robotnik_docker" type="diff
_docker_node" respawn="false" output="screen">
    <param name="robot_base_frame" type="string" value="rb1_base_base_foo
tprint" />
    <param name="fixed_frame" type="string" value="rb1_base_odom" />

    <rosparam command="load" file="$(find command_interface)/config/diff_
docker.yaml"/>
  </node>

</launch>
```

ANEXO 5 – Fichero move.launch

```
<?xml version="1.0"?>
<launch>

  <node ns="rb1_base" name="move" pkg="robotnik_move" type="move_node" re
spawn="false" output="screen">
    <param name="robot_base_frame" type="string" value="rb1_base_base_foo
tprint" />
    <param name="fixed_frame" type="string" value="rb1_base_odom" />

    <rosparam command="load" file="$(find command_interface)/config/move.
yaml"/>
    <param name="differential_robot" value="1" />
  </node>

</launch>
```

ANEXO 6 – Fichero mqtt_bridge.launch

```
<?xml version="1.0"?>
<launch>

  <node name="mqtt_bridge" pkg="mqtt_bridge" type="mqtt_bridge_node.py" o
utput="screen">
    <rosparam file="$(find command_interface)/config/mqtt_bridge.yaml" co
mmand="load" />
  </node>

</launch>
```

ANEXO 7 – Fichero app.py

```
#! /usr/bin/env python
import rospy

from stop_sub_class import Stop_sub
from command_sub_class import Command_sub
from std_msgs.msg import Empty
from robot_simple_command_manager_msgs.msg import CommandString

ctrl_c = False
moving = True

robot = "/rb1_base_b"
topic_pub_cancel = robot + "/command_manager/cancel"
topic_pub_command = robot + "/command_manager/command"


def shutdownhook():
    # works better than the rospy.is_shutdown()
    ctrl_c = True


rospy.init_node('endorse_app_node')
rospy.on_shutdown(shutdownhook)

cancel_pub = rospy.Publisher(topic_pub_cancel, Empty, queue_size=1)
command_pub = rospy.Publisher(topic_pub_command, CommandString, queue_siz
e=1)
command = ""

msg_cancel = Empty()

stop_sub_obj = Stop_sub()
command_sub_obj = Command_sub()

while (not ctrl_c):
    if stop_sub_obj.get_data() == 1 and moving: #significa que se tiene q
ue parar
        rospy.loginfo("Parando robot")
        command = command_sub_obj.get_data()
        cancel_pub.publish(msg_cancel)
        moving = False

    if stop_sub_obj.get_data() == 0 and (not moving):
        rospy.loginfo("Continuando movimiento")
        command_pub.publish(command)
        moving = True
```

ANEXO 8 – Fichero stop_sub_class.py

```
#! /usr/bin/env python
import rospy
from std_msgs.msg import Int32


class Stop_sub():

    def __init__(self):
        robot = "/rb1_base_b"
        topic_sub = robot+"/command_manager/stop"
        self.data = 0

        #rospy.init_node('stop_sub_node') # inicializa el nodo llamado to
pic_subscriber
        sub = rospy.Subscriber(topic_sub, Int32, self.on_message)

    def get_data(self):
        return self.data

    def on_message(self, msg):
        rospy.loginfo("Solicitud de parada/continuar recibida")
        self.data = msg.data
```

ANEXO 9 – Fichero command_sub_class.py

```
#! /usr/bin/env python
import rospy
from robot_simple_command_manager_msgs.msg import CommandString

class Command_sub():

    def __init__(self):
        robot = "/rb1_base_b"
        topic_sub = robot+"/command_manager/command"
        self.command = ""

        #rospy.init_node('command_sub_node') # inicializa el nodo llamado
 topic_subscriber
        sub = rospy.Subscriber(topic_sub, CommandString, self.on_message)

    def get_data(self):
        return self.command

    def on_message(self, msg):
        rospy.loginfo("Comando recibido: "+self.command)
        self.command = msg.command
```

ANEXO 10 – Fichero endorse_app_launch.launch

```
<launch>
    <node pkg="endorse_app"
        type="app.py"
        name="endorse_app_node"
        output="screen">
    </node>
</launch>
```