Pontificia Universidad
JAVERIANA
Cali

Facultad de Ingeniería
y Ciencias
Ingeniería Electrónica

Monografía de Trabajo de Grado

Desarrollo de un sistema de comunicación multi-robot en ROS (Robot Operating System)

Alejandro Correa Ramos
Émerson Armando Ramírez Gutiérrez

Director
Dr. Alexánder Martínez Álvarez

8 de abril de 2025

Agradecimientos

Agradecemos a Dios por habernos otorgado la fortaleza, determinación y resiliencia necesarias para continuar y culminar este trabajo de grado, a pesar de las dificultades que se presentaron en el camino.

Agradecemos infinitamente a nuestras familias, quienes nos acompañaron y apoyaron incondicionalmente, brindándonos palabras de aliento en los momentos en que más lo necesitábamos. Su esfuerzo, sabiduría y consejos han sido la fuerza primordial que nos ha forjado a lo largo de este camino.

Agradecemos profundamente a nuestro director Alexánder Martínez Álvarez por su infinita paciencia para con nosotros y siempre estar dispuesto a ayudarnos. También extendemos nuestro reconocimiento a todos los profesores que encontramos a lo largo de nuestra carrera, quienes nos brindaron una valiosa guía en el camino hacia el conocimiento y destacaron por su excepcional calidad humana.

Finalmente, expresamos nuestro agradecimiento a todas las personas que nos brindaron consuelo, apoyo y fortaleza, ya sea de manera directa o indirecta. Su ayuda, en cualquiera de sus formas, ha sido fundamental para que hoy podamos concluir este importante capítulo en nuestras vidas.

Agradecemos sinceramente a todos, hoy y siempre.

Glosario

Acrónimos y Abreviaturas

IEEE	The Institute of Electrical and Electronic Engineers
SMR	Sistema Multi-Robot
GUI	Graphical User Interface
ROS	Robot Operating System
IMU	Intertial Measurement Unity
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
RPC	Remote Procedure Call
XML	Extensible Markup Language
VM	Virtual Machine
MAS	Multiagent System
WLAN	Wireless Local Area Network
AVR	Advanced Virtual RISC
OS	Operating System

Términos

Leaflet	Es una biblioteca de código abierto de JavaScript que brinda un conjunto de herramientas para la implementación de mapas interactivos en la Web [1].
LiDAR	Es un sensor que emite pulsos láser para medir distancias y crear modelos 2D o 3D del entorno en tiempo real [2].
Modelo OSI	El modelo OSI (Open Systems Interconnection) es un marco conceptual que estandariza la comunicación en redes de computadoras, el cual se divide en siete capas [3].

Resumen

En la sociedad contemporánea, los Sistemas Multi-Robot (SMR) han adquirido una creciente relevancia debido a sus numerosas aplicaciones y beneficios en diversos campos. Estos sistemas se basan en la coordinación y colaboración de múltiples robots para alcanzar objetivos comunes, lo que les permite llevar a cabo tareas de manera más eficiente, rápida y versátil en comparación con los robots individuales. En este contexto, el sistema de comunicación desempeña un papel fundamental al posibilitar la colaboración entre los robots. Por lo tanto, su implementación resulta de gran importancia para dotar a un SMR con estas características.

En el programa de Ingeniería Electrónica de la Pontificia Universidad Javeriana de Cali, se ha trabajado con un robot móvil denominado DaNI, que cuenta con un sistema de navegación autónoma. Sin embargo, este sistema carece de una infraestructura de comunicación que permita la integración de múltiples robots con capacidades similares (homogéneos) para lograr una navegación conjunta. Esta limitación impide su diversificación hacia nuevas investigaciones o aplicaciones que requieran un SMR.

Para superar esta limitación, se realizó la actualización de un segundo robot DaNI, brindándole las mismas características de navegación autónoma que el primer robot actualizado. Además, se seleccionaron el software y las bibliotecas necesarias para implementar y desarrollar un sistema de comunicación, junto con una interfaz gráfica de usuario (GUI) básica para su control. Como resultado, se ha desarrollado un sistema de comunicación que integra dos robots DaNI y una GUI básica con funcionalidades clave, tales como: la identificación del robot más cercano a un punto de destino específico, la capacidad para guardar coordenadas con un nombre o seudónimo, la activación simultánea de la navegación autónoma en uno o en todos los robots, así como también la detención de la navegación en ambos casos y la visualización de la latitud y longitud actuales de los robots, además de la posición de los robots y de las coordenadas de destino en un mapa proporcionados por la librería de *Leaflet*.

Los resultados obtenidos al poner a prueba el sistema desarrollado fueron satisfactorios, mostrando una correcta integración entre el sistema de navegación que ya estaba presente y el sistema de comunicación empleado para el monitoreo y control de la navegación conjunta.

Este sistema de comunicación, en combinación con el sistema de navegación, establece una base fundamental para desarrollos futuros más sofisticados o para aplicaciones específicas, como en el ámbito agroindustrial, gracias a su modularidad y escalabilidad.

Palabras Clave: ROS, Interfaz gráfica de usuario (GUI), *Leaflet*, Sistemas multi-robot (SMR), modularidad, robots homogéneos, navegación autónoma, sistema de comunicación.

Abstract

In contemporary society, Multi-Robot Systems (MRS) have gained increasing relevance due to their numerous applications and benefits in various fields. These systems are based on the coordination and collaboration of multiple robots to achieve common objectives, allowing them to perform tasks more efficiently, quickly, and flexibly compared to individual robots. In this context, the communication system plays a fundamental role in enabling collaboration between robots. Therefore, its implementation is of great importance to endow an MRS with these characteristics.

In the Electronic Engineering program at the Javeriana University Cali, work has been done with a mobile robot called DaNI, which has an autonomous navigation system. However, this system lacks a communication infrastructure that allows the integration of multiple robots with similar capabilities (homogeneous) to achieve joint navigation. This limitation prevents its diversification towards new research or applications that require an MRS.

To overcome this limitation, a second DaNI robot was upgraded, providing it with the same autonomous navigation characteristics as the first updated robot. Additionally, the necessary software and libraries were selected to implement and develop a navigation system, along with a basic graphical user interface (GUI) for its control. As a result, a communication system has been developed that integrates two DaNI robots and a basic GUI with key functionalities such as: the identification of the robot closest to a specific destination point, the ability to save coordinates with a name or alias, the simultaneous activation of autonomous navigation in one or all robots, as well as stopping navigation in both cases and the visualization of the robots current latitude and longitude, along with the robots position and destination coordinates on a map provided by the *Leaflet* library.

The results obtained when testing the developed system were satisfactory, demonstrating proper integration between the pre-existing navigation system and the communication system used for monitoring and controlling joint navigation.

This communication system, in combination with the navigation system, establishes a fundamental basis for more sophisticated future developments or for specific applications, such as in the agro-industrial field, thanks to its modularity and scalability.

Keywords: *ROS*, Graphical User Interface (GUI), *Leaflet*, Multi-Robot Systems (MRS), modularity, homogeneous robots, autonomous navigation, communication system.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En la última década, los Sistemas Multi-Robot (SMR) han emergido como una solución prometedora para una amplia gama de aplicaciones, que van desde la exploración espacial hasta la agricultura de precisión y la manufactura automatizada. A diferencia de los sistemas robóticos individuales, los SMR ofrecen la capacidad de realizar tareas complejas de manera más eficiente y flexible, distribuyendo el trabajo entre varios robots [12]. Esta cooperación permite enfrentar problemas que serían intratables para un solo robot, como la cobertura de grandes áreas, la coordinación en entornos dinámicos y la ejecución de tareas simultáneas [13].

Las ventajas de los SMR en comparación con los sistemas robóticos individuales se pueden apreciar en contextos particulares. Por ejemplo, en aplicaciones de exploración y cobertura, se ha demostrado que los SMR pueden cubrir hasta un 40 % más de área en comparación con un sistema individual en el mismo periodo de tiempo, gracias a la posibilidad de operar en paralelo [14]. En entornos industriales, el uso de múltiples robots coordinados puede reducir el consumo energético hasta en un 30 %, ya que cada robot tiene una carga de trabajo más específica y optimizada en comparación con un solo robot que tendría que realizar todas las tareas secuencialmente [14]. En escenarios de logística y transporte de objetos, un estudio comparó la eficiencia de un sistema con un robot frente a uno con cinco robots. El SMR mostró una reducción del tiempo total de ejecución de tareas en 60-70 %, dependiendo de la complejidad de la tarea y la capacidad de coordinación entre los robots [15].

La comunicación entre los robots es un pilar para el éxito de cualquier SMR. Sin una comunicación robusta y eficiente, la coordinación y cooperación entre los robots puede verse comprometida, lo que puede llevar a un rendimiento bajo o incluso al fracaso de la misión. La eficiencia en la transmisión de datos, la minimización de la latencia y la capacidad para manejar una red de comunicación escalable son desafíos críticos en el diseño de SMR [16]. La integración de estos sistemas en entornos con alta densidad de robots o con interacciones humanas requiere estrategias avanzadas de comunicación y coordinación, como se ha demostrado en trabajos anteriores [17].

El framework Robot Operating System (ROS) se ha consolidado como una de las plataformas más utilizadas en la investigación y el desarrollo de sistemas robóticos, gracias a su arquitectura modular y su gran colección de herramientas y bibliotecas. ROS facilita la implementación y prueba de sistemas robóticos complejos, permitiendo una rápida integración de hardware y software, así como la reutilización de componentes [18]. Estas características lo convierten en una opción ideal para el desarrollo de SMR, en los que la integración y la comunicación entre varios robots son

esenciales [19].

Sin embargo, a pesar de sus ventajas, el diseño e implementación de un sistema de comunicación para el SMR en ROS presenta varios desafíos. Entre ellos, la necesidad de garantizar la eficiencia en la transmisión de datos, la minimización de la latencia, y la capacidad de escalar el sistema para incluir un gran número de robots [20]. Estos desafíos se agravan en entornos dinámicos o desconocidos, donde los robots deben adaptarse rápidamente a cambios en su entorno y en las condiciones de la red de comunicación [21].

La principal motivación de este trabajo radica en la necesidad de desarrollar e implementar un sistema de comunicación eficiente y escalable para un SMR homogéneo basado en ROS. Este desarrollo se enmarca dentro de las restricciones físicas y de diseño inherentes al robot utilizado, específicamente el robot DaNI, que fue adaptado en la Pontificia Universidad Javeriana, a partir de una plataforma robótica comercial. El sistema propuesto debe garantizar una comunicación fluida entre dos robots DaNI, facilitando el control y la ejecución de la navegación autónoma mediante una interfaz gráfica de usuario (GUI) básica.

Este trabajo está estructurado de manera que inicialmente se presenta el capítulo de Contextualización del proyecto, en el que se encuentra el planteamiento del problema y la justificación que llevaron al desarrollo del sistema de comunicación multi-robot enfocado en los robots DaNI, además de los objetivos planteados para abordar los desafíos asociados y el marco de referencia que sustenta el trabajo. Seguidamente, se detalla el capítulo de Concepción y diseño de la solución, donde se abordan temas como los requisitos generales del sistema en cuanto a hardware y software, la elección de la tecnología utilizada y el proceso de integración de los sistemas de navegación y comunicación. Posteriormente, se presenta el capítulo de Resultados y Discusión, en el que se exponen las pruebas realizadas y los hallazgos obtenidos durante la evaluación del sistema de comunicación junto con el sistema de navegación. Finalmente, se presentan las Conclusiones, las Recomendaciones, los Anexos y las Referencias bibliográficas del proyecto.

Capítulo 2

Contextualización del proyecto

2.1. Planteamiento

La robótica ha adquirido una relevancia significativa en la actualidad, gracias a su aplicación en distintos ámbitos, como la industria, el hogar y la investigación. Debido a esto su mercado es cada vez más demandado según un informe de la empresa de investigación de mercado y consultoría Mordor Intelligence, donde se proyecta que a nivel mundial alcance los USD 226,47 mil millones para 2029 en comparación con los USD 102,75 mil millones que se estima para el 2024, lo que representaría una tasa de crecimiento anual compuesta de 17.10 % [22]. Dentro de este mercado, los SMR están ganando popularidad debido a sus ventajas en comparación con sistemas de un solo robot. Estos beneficios incluyen una mayor eficacia, eficiencia, flexibilidad y tolerancia a fallos, lo que los hace ideales para tareas complejas [23]. Un ejemplo destacado es el uso de robots en los almacenes de Amazon, donde los SMR permiten una coordinación eficiente en la logística de paquetes. Este enfoque ha mejorado significativamente la eficiencia operativa, aumentando la adaptabilidad y robustez del sistema en su conjunto [24]. Ahora otras industrias, como la agricultura, requieren cada vez más el desarrollo de SMR debido a desafíos contemporáneos, como la escasez de mano de obra en países como Estados Unidos e Inglaterra, y la presión por reducir los costos de producción. Estos factores han impulsado la necesidad de optimizar los procesos mediante el monitoreo constante de variables clave [25]. Dentro de este contexto, la comunicación es fundamental para la coordinación de robots en aplicaciones multi-robot, ya que facilita el intercambio de información entre las partes para alcanzar un objetivo común [26].

La Pontificia Universidad Javeriana Cali no es ajena a esta realidad, por lo que existe una línea de investigación en la robótica móvil, donde el primer paso se logró dar con el trabajo de grado "Actualización del sistema de navegación para exteriores de un robot móvil terrestre usando Robot Operating System" de los egresados de ingeniería electrónica Esteban Castaño y Jhoan Solarte. Este proyecto, desarrollado sobre el framework de ROS [27], le brindó la capacidad a un robot móvil llamado DaNI de navegar en exteriores de forma autónoma. Sin embargo, a pesar de los avances logrados, un único robot presenta una menor robustez en comparación con un SMR como se mencionó anteriormente, lo que limita su capacidad para abordar una amplia gama de aplicaciones. Por lo tanto, es fundamental equiparlo con un sistema de comunicación que permita la coordinación y navegación conjunta entre múltiples robots DaNI. Esto no solo superaría las limitaciones dichas, sino que también ofrecería una plataforma más completa para desarrollos futuros o aplicaciones

académicas.

Teniendo en cuenta lo anterior se planteó la siguiente pregunta que orientó la investigación: ¿cómo implementar un sistema de comunicación para varios robots DaNI que permita la realización de navegación conjunta?

2.2. Justificación

El empleo de sistemas multi-robot (SMR) para la ejecución de tareas complejas está ganando una creciente aceptación en la actualidad, debido a sus ventajas distintivas, como la adaptabilidad y la eficiencia. En consecuencia, se está desechando la concepción de utilizar un único robot más robusto para llevar a cabo estas tareas complejas [28]. En este contexto, es importante subrayar que el sistema de comunicación constituye un elemento esencial, ya que facilita la coordinación entre los robots para alcanzar un objetivo común.

Actualmente, la Pontificia Universidad Javeriana Cali dispone de tres robots DaNI para su aplicación en robótica móvil. Estos robots fueron adquiridos a través de National Instruments hace más de una década. Dada su antigüedad y limitaciones en escalabilidad tanto en software como en hardware, se desarrolló un proyecto de grado para actualizar uno de estos robots para navegación autónoma en exteriores. Esta actualización, que incluyó mejoras tanto en hardware como en software, proporcionó al robot una mayor versatilidad, escalabilidad y libertad de desarrollo al basarse en el Sistema Operativo Robótico (ROS). ROS, conocido por su alta modularidad y su extensa variedad de bibliotecas y repositorios en múltiples lenguajes de programación como `C++`, `Python`, `Lisp` y `Java`, ofrece una base sólida creada por la comunidad de usuarios de este framework [29].

En el contexto actual, la implementación de un SMR se vuelve crucial debido a las características y beneficios que ofrece en tareas complejas, en comparación con el uso de un único robot. Para avanzar en el desarrollo de los robots DaNI en la Pontificia Universidad Javeriana de Cali, resulta esencial crear un sistema de comunicación que facilite tareas como la navegación conjunta. Esto no solo representa un avance significativo en las tendencias actuales, sino que también contribuirá al desarrollo futuro de esta línea de investigación. Además, el sistema de comunicación propuesto podría ser una base de la cual se puedan desarrollar aplicaciones en áreas como la agricultura de precisión [30] o en el ámbito académico para el aprendizaje y la enseñanza en robótica móvil dentro de la universidad.

En conclusión, implementar un sistema de comunicación para la realización de navegación conjunta haciendo uso de los robots presentes en la Pontificia Universidad Javeriana de Cali, tendrá un impacto positivo en la misma, al proporcionarle una herramienta innovadora que podría ser utilizada tanto para futuros desarrollos, implementaciones reales o para el aprendizaje académico, acercando así a la comunidad javeriana a la vanguardia de las tecnologías actuales en robótica móvil.

2.3. Objetivos

2.3.1. Objetivo General

Desarrollar un sistema de comunicación multi-robot bajo el framework ROS con el fin de realizar la navegación conjunta.

2.3.2. Objetivos Específicos

- Diseñar un sistema de comunicación multi-robot en ROS que permita realizar la navegación conjunta.
- Implementar el sistema de comunicación en dos robots DaNI.
- Diseñar una interfaz de usuario básica (GUI) para el control de la navegación conjunta en dos robots DaNI.
- Evaluar el sistema de comunicación mediante pruebas de navegación conjunta de dos robots DaNI en el campus Universitario.

2.4. Marco de Referencia

2.4.1. Áreas Temáticas

De acuerdo con la Taxonomía IEEE [31], se considera que el trabajo se encuentra en las siguientes áreas temáticas:

- Robotics and automation - Autonomous systems - Autonomous vehicles.
- Vehicular and wireless technologies - Hardware - Navigation - Satellite navigation systems - Global Positioning System.
- Computers and information processing - Software - Open source software - Computer interfaces.
- Communications technology - Communication systems - Communication networks - IP networks - Machine-to-machine communications.

2.4.2. Marco Teórico

2.4.2.1. Sistema de navegación presente en los robots

El sistema de navegación autónomo presente en el robot DaNI actualizado por parte del anterior trabajo de grado consta de tres procesos, los cuales son:

- **Localización:** Esta etapa se lleva a cabo mediante la fusión sensorial de tres sensores presentes en el robot DaNI, que son el GPS, la IMU (Unidad de Medición Inercial) y los encoders, a través del método Kalman [32].

- **Planeación de trayectoria:** Se realiza mediante un proceso iterativo en el cual se utiliza el escáner láser LiDAR para mapear el entorno donde el robot está ubicado y con uno de los paquetes de ROS llamado `"costmap_2d"`, se generan mapas de costos del entorno que se divide en una rejilla, donde cada celda de esta rejilla contiene un valor que representa el costo o dificultad que el robot tendría al pasar por esa ubicación. Con la información proporcionada por los sensores de localización y estos mapas de costos, se realiza una planificación de ruta eficiente a través de un enfoque combinado de planificación global y local utilizando paquetes como `"global_planner"`, `"navfn"` y `"base_local_planner"`.
- **Control de movimiento:** En esta etapa, se emplea un Arduino para controlar el driver de los motores. Esto es factible gracias a la comunicación establecida entre las tarjetas Jetson Nano y Arduino. Después de la planificación, la Jetson Nano envía un mensaje tipo `"geometry_msgs/Twist"` al Arduino, que contiene información sobre la velocidad y dirección deseadas. El Arduino recibe y procesa estos datos como puntos de referencia (set point) para un controlador PID implementado en él. El objetivo del controlador PID es mantener los valores de las variables en los niveles deseados, permitiendo así un control preciso y estable del sistema.

2.4.2.2. Protocolo UDP

El Protocolo de Datagramas de Usuario (UDP) es un protocolo que opera en la capa de transporte, tanto en el modelo OSI como en el modelo TCP/IP. Se caracteriza por ser un protocolo sin conexión y no confiable, ya que no establece una comunicación previa entre el emisor y el receptor, ni implementa mecanismos para garantizar la entrega exitosa de los datos. A diferencia de su contraparte, el Protocolo de Control de Transmisión (TCP), UDP carece de métodos de verificación que confirmen la recepción de los datos por parte del receptor [33].

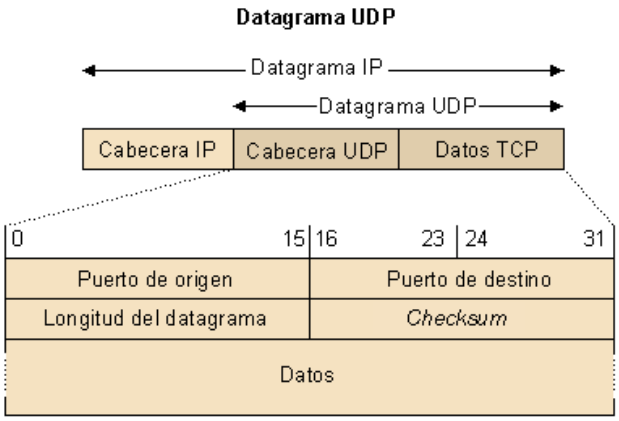

Figura 2.1: Formato cabecera UDP (Fuente: [4]).

A pesar de estas características, en la actualidad el Protocolo UDP es ampliamente utilizado en numerosas aplicaciones que demandan, ante todo, una alta velocidad de transmisión y baja latencia.

Esto se debe en gran parte a que la cabecera de UDP tiene un tamaño fijo de 8 bytes, como se ilustra en la Figura 2.1.

En contraste, el Protocolo de Control de Transmisión (TCP) emplea una cabecera mínima de 20 bytes, como se muestra en la Figura 2.2, y puede ser aún mayor cuando se incluyen opciones adicionales.

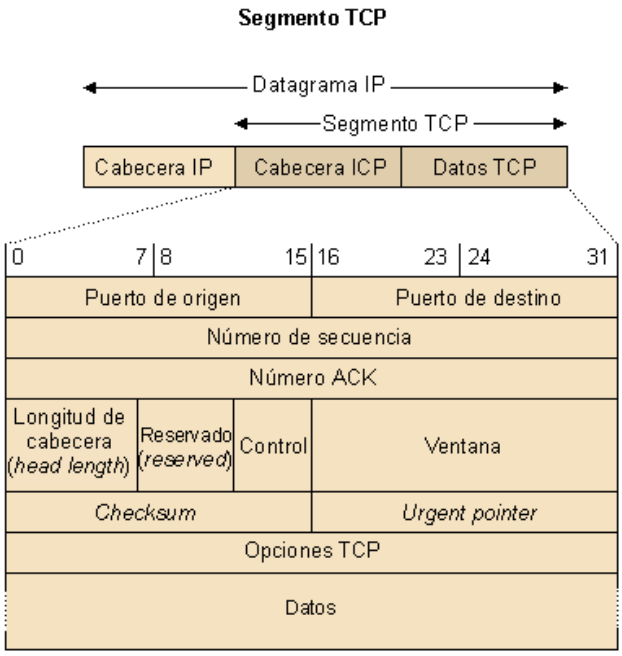

Figura 2.2: Formato cabecera TCP (Fuente: [5]).

Por esto, el protocolo UDP resulta especialmente útil en situaciones donde la rapidez en la transferencia de datos es crucial y la pérdida ocasional de paquetes no compromete la funcionalidad general del servicio. Ejemplos de tales aplicaciones incluyen la transmisión de video y audio en tiempo real, como en videoconferencias, streaming en vivo, llamadas VoIP y juegos en línea, donde se prioriza la fluidez y la continuidad del flujo de datos sobre la corrección de errores. Por ejemplo, en una videollamada, si llega un dato incompleto, el efecto que tendría sería una imagen con píxeles borrosos por un instante [34].

2.4.2.3. ROS Master

Dentro del framework ROS, el Master de ROS (o roscore) desempeña un papel fundamental en la coordinación y gestión de la red de nodos. Actúa como el servidor central que facilita el descubrimiento y la comunicación entre los nodos del sistema. Su principal función es mantener un registro actualizado de todos los nodos que están activos, así como de los tópicos y servicios que estos nodos están publicando o a los cuales se están suscribiendo [35].

Para gestionar esta información, el Master de ROS utiliza la API XML-RPC, un protocolo basado en XML para realizar solicitudes y recibir respuestas a través de HTTP. XML-RPC se emplea para manejar las comunicaciones administrativas con el nodo maestro, lo que incluye el registro de nuevos nodos, la consulta de información sobre tópicos y servicios, y la actualización de esta información cuando los nodos cambian de estado o se desconectan. Este protocolo permite que los nodos se registren con el maestro y obtengan detalles sobre otros nodos y los tópicos que están disponibles para la publicación y suscripción [36].

2.4.2.4. Servicios ROS

Los servicios en ROS representan una forma de comunicación bidireccional entre nodos, permitiendo el intercambio de datos mediante un modelo de solicitud-respuesta, similar al concepto de llamada a procedimiento remoto (RPC). En este modelo, un nodo actúa como cliente y envía una solicitud, mientras que otro nodo actúa como servidor y procesa dicha solicitud para devolver una respuesta [37]. Esto se ilustra claramente en la Figura 2.3, donde se observa el flujo de comunicación entre los nodos involucrados.

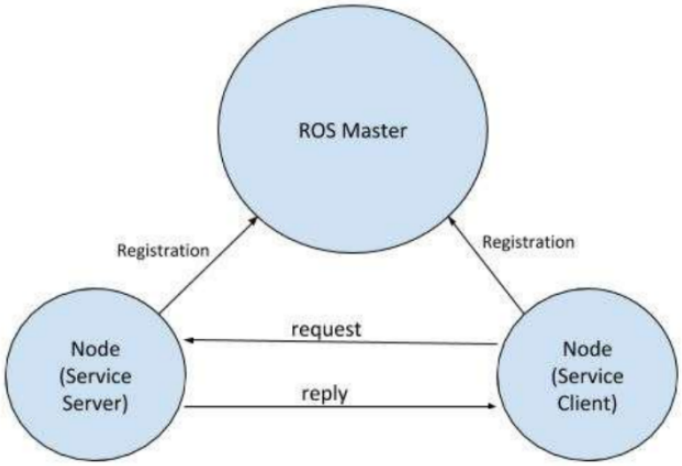

Figura 2.3: Modelo de un servicio en ROS (Fuente: [6]).

La definición de servicios en ROS se realiza mediante archivos con extensión *.srv*. Cada archivo *.srv* describe la estructura de los datos que se intercambian en la solicitud y la respuesta. Este archivo contiene dos partes: la primera define la estructura del mensaje de solicitud, y la segunda, la estructura del mensaje de respuesta. Esto permite que ROS conozca de antemano el formato de datos que se enviarán y recibirán durante la comunicación entre los nodos [38].

Para garantizar la coherencia y la compatibilidad entre el nodo cliente y el nodo servidor, ROS emplea el algoritmo MD5. Este algoritmo genera un hash único a partir de la definición del servicio contenida en el archivo *.srv*. Tanto el cliente como el servidor generan un valor MD5 a partir de la

misma definición de servicio. Cuando se realiza una llamada a un servicio, ROS compara los valores MD5 generados por el cliente y el servidor. Si coinciden, esto garantiza que ambos están utilizando la misma estructura de datos para la comunicación [39]. Esta verificación es esencial para evitar errores y asegurar que el intercambio de datos se realice de manera consistente y confiable.

La principal herramienta disponible en ROS para interactuar con los servicios es *rosservice*, la cual ofrece importantes comandos de consola que permiten a los usuarios listar, consultar y gestionar los servicios disponibles en el sistema [40]. Por ejemplo, se pueden listar todos los servicios activos, obtener detalles sobre un servicio específico y llamar a un servicio directamente desde la consola. *Rosservice* es una herramienta fundamental para el diagnóstico y la gestión de servicios en un entorno ROS.

2.4.2.5. Servicios de Acción ROS

Los servicios de acción de ROS se emplean en ciertas situaciones, donde se requiere llevar a cabo tareas específicas, como desplazar un robot de un punto A a un punto B, realizar un escaneo láser que genere una nube de puntos, o capturar imágenes con una cámara, entre otros. Para este tipo de solicitudes, que generalmente implican un tiempo prolongado antes de retornar una respuesta, y en los que el usuario podría necesitar cancelar la ejecución del servicio o realizar un seguimiento del estado de la misma, los servicios de acción son el mecanismo utilizado [41].

Los servicios de acción, proporcionados por la librería *actionlib*, son una forma de comunicación diseñada para manejar tareas complejas y de larga duración. La estructura de los mensajes asociados a estos servicios se define en archivos con extensión *.action*. Estos archivos incluyen tres componentes esenciales:

- Objetivo (Goal): Describe la meta específica que se pretende alcanzar.
- Resultado (Result): Contiene el resultado final una vez que la acción se ha completado.
- Retroalimentación (Feedback): Proporciona actualizaciones periódicas sobre el progreso de la tarea en ejecución.

Esta estructura de mensajes es utilizada tanto por el cliente de acción simple (*SimpleActionClient*) como por el servidor de acción simple (*SimpleActionServer*) para facilitar la comunicación. Además, la arquitectura de los servicios de acción incluye de manera predefinida en su estructura la capacidad para enviar y cancelar metas, permitiendo así un control más eficiente y flexible. Esta comunicación entre el *SimpleActionClient* y *SimpleActionServer* [7], se visualiza en la figura 2.4.

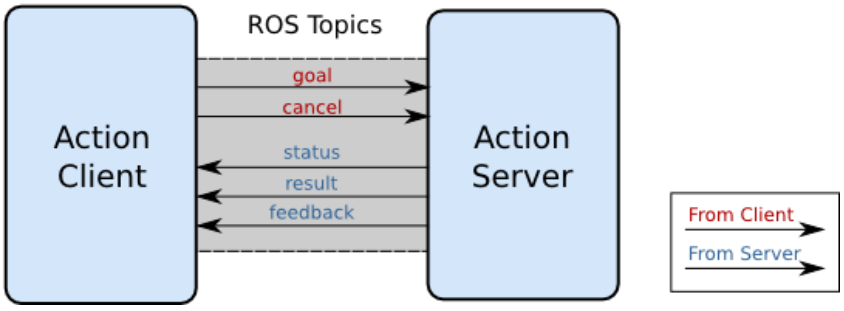

Figura 2.4: Modelo de un servicio de acción en ROS (Fuente: [7]).

2.4.2.6. PyQt5

Qt es un conjunto integral de bibliotecas y herramientas de desarrollo en C++ que ofrece una amplia gama de abstracciones multiplataforma. Estas abstracciones facilitan la creación de interfaces gráficas de usuario, manejo de redes, subprocesos, expresiones regulares y acceso a bases de datos SQL. Además, incluye funcionalidades para trabajar con gráficos SVG, OpenGL y XML, así como para gestionar configuraciones de usuario y de aplicaciones. Qt también proporciona soporte para servicios de posicionamiento y localización, comunicaciones de corto alcance (NFC y Bluetooth), navegación web, animación 3D, gráficos avanzados, visualización de datos en 3D y permite la integración con tiendas de aplicaciones [42].

PyQt5 es la implementación de más de 1000 de estas clases de Qt en forma de módulos de Python. Este paquete incluye no solo el núcleo de PyQt5, sino también una serie de complementos que corresponden a las librerías adicionales de Qt. Estos se distribuyen como paquetes de código fuente (sdist) y ruedas binarias, disponibles para múltiples sistemas operativos como Windows, Linux y macOS.

La compatibilidad de PyQt5 se extiende a varias plataformas, incluyendo Windows, Linux, UNIX, Android, macOS e iOS. Para su funcionamiento, requiere Python en su versión 3.5 o superior. Aunque es posible compilar PyQt5 con Python 2.7 y versiones anteriores de Python 3 utilizando un script de compilación legado (*configure.py*), esta opción no se encuentra actualmente soportada [42]. Esta compatibilidad multiplataforma y la capacidad de trabajar con una amplia variedad de funcionalidades hacen de PyQt5 una herramienta poderosa para el desarrollo de aplicaciones modernas en Python.

2.4.2.7. Docker

Docker es una plataforma de código abierto diseñada para automatizar el desarrollo, la creación, el envío, la ejecución y la gestión de aplicaciones dentro de entornos estandarizados. Esta estandarización se logra mediante el uso de contenedores, que encapsulan todo lo necesario para que el software funcione, incluyendo el código, las bibliotecas y las dependencias. Esta característica es fundamental ya que permite que las aplicaciones se ejecuten de manera coherente en distintos entornos, lo que

reduce significativamente la disparidad entre los entornos de desarrollo, prueba y producción, así como los problemas relacionados con la ejecución del código en diferentes sistemas operativos [43].

Los contenedores de Docker son módulos independientes y ligeros que permiten empaquetar el software y todas sus dependencias. Esta independencia significa que se pueden tener múltiples contenedores en un mismo sistema, cada uno con su propio entorno aislado, incluso si utilizan diferentes versiones de un mismo lenguaje de programación, como Python. Por ejemplo, un contenedor puede ejecutar una aplicación que depende de Python 2.7, mientras que otro utiliza Python 3.8, sin que haya conflictos entre ellos.

Además, los contenedores ofrecen una forma de virtualización más eficiente que las máquinas virtuales (VM). A diferencia de las VM, que requieren un sistema operativo completo y consumen una cantidad significativa de recursos del sistema, los contenedores comparten el mismo núcleo o kernel del sistema operativo anfitrión, lo que reduce su tamaño y mejora su rendimiento. Esto se puede observar en la Figura 2.5. Esta eficiencia se traduce en un menor consumo de recursos y una mayor densidad de aplicaciones en un solo host. Docker, por tanto, proporciona una alternativa más ligera y escalable a las máquinas virtuales, permitiendo la virtualización, el aislamiento y distribución de recursos de manera eficiente [44].

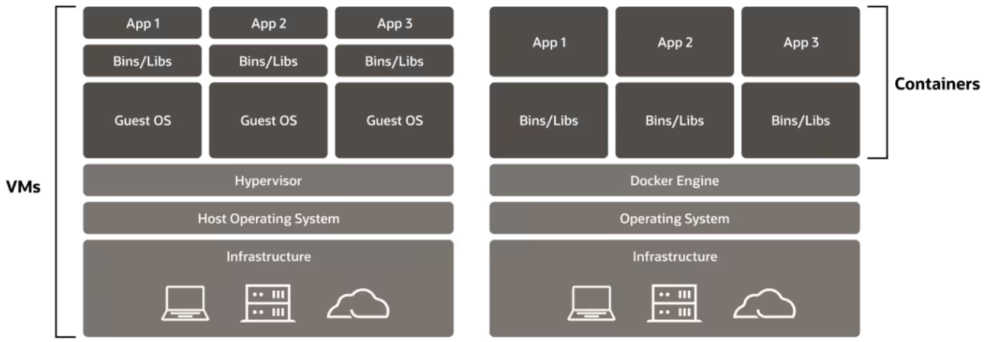

Figura 2.5: Arquitectura VM y Docker (Fuente: [8]).

Debido a estas ventajas, como la portabilidad, escalabilidad y eficiencia en el uso de los recursos, Docker ha ganado una popularidad considerable desde el lanzamiento de su primera versión en 2014. La comunidad de desarrolladores ha adoptado ampliamente esta tecnología, que ha cambiado significativamente la forma en que se desarrollan, despliegan y gestionan las aplicaciones modernas [45]. Docker se ha convertido en una herramienta indispensable en los flujos de trabajo de desarrollo y operaciones (DevOps), facilitando la integración y entrega continua de software, y permitiendo que los equipos de desarrollo aceleren el ciclo de vida de las aplicaciones al simplificar el proceso desde la escritura del código hasta su ejecución en producción.

2.4.3. Antecedentes

Se realizó una revisión de artículos en diversas bases de datos científicas, focalizándose en investigaciones que abordarán desarrollos tecnológicos vinculados a los sistemas de comunicación implementados en ROS. Principalmente, se identificaron dos corrientes de enfoque durante el proceso de búsqueda, dependiendo de si se emplea ROS o ROS 2, pues varios estudios han optado por utilizar ROS 2 ya que ofrece una serie de mejoras y características que lo hacen más adecuado para el desarrollo de sistemas de comunicación multi-robot y aplicaciones robóticas avanzadas.

En el contexto de los sistemas de comunicación multi-robot que emplean distribuciones de ROS, se enfatiza que "para múltiples robots, surgen conflictos en los espacios de nombres con temas de ROS dentro de un único sistema maestro. Esta limitación puede mitigarse mediante el uso de múltiples maestros"[46]. Sin embargo, para abordar esta problemática, se recurre al empleo del paquete "multimaster_fkie"[47]. Este paquete posibilita la comunicación entre dos redes maestras, abordando de manera efectiva el conflicto de nombres de espacio. Este enfoque se implementa ampliamente en numerosos desarrollos diseñados para sistemas multi-robot, tal como el sistema multimaestro creado para los robots móviles MiniLab Enova. Este sistema opera bajo una arquitectura descentralizada y su propósito central es la ejecución eficiente de tareas colaborativas [46].

Por otra parte, se han llevado a cabo investigaciones que posibilitan la modelación de redes de autómatas para un sistema de comunicación, como se observa en [48]. Este estudio tiene como objetivo alcanzar una concordancia entre la simulación y la ejecución práctica. Se explica cómo, mediante el uso de UPPAAL, un software que facilita la modelación de redes de autómatas temporizadas, se pueden generar protocolos distribuidos para Sistemas Multiagente (MAS) que son posteriormente implementados en el entorno de ROS.

Asimismo, dentro de los artículos relacionados con ROS 2 se puede ver en [49] la propuesta de un mecanismo para vincular dinámicamente una implementación adecuada de DDS para ROS2. DDS (Data Distribution Service) es un middleware que facilita la comunicación en tiempo real entre componentes distribuidos, y ROS 2 utiliza DDS como su capa de comunicación subyacente. Así, se permiten vincular nuevos nodos en tiempo de ejecución, lo que significa que es posible incorporar y utilizar diferentes implementaciones DDS para distintos nodos o canales de comunicación en el sistema ROS2 sin tener que reiniciar el sistema completo.

Debido a que DDS es un estándar de middleware orientado a mensajes utilizado para facilitar la comunicación y el intercambio de datos entre sistemas distribuidos en tiempo real popular dentro de las distribuciones de ROS 2, en [50] se comparan distintas implementaciones de este estándar en robots que utilizan la tecnología de ROS 2 con el propósito de analizar diferentes características como el rendimiento, la escalabilidad, la facilidad de integración, entre otras.

Cabe resaltar que existen otros middlewares que proporcionan una herramienta para el desarrollo de sistemas de comunicación multi-robot en base a ROS 2, como lo puede ser Robot Middleware Framework (RMF) [51].

Los avances presentados tanto en ROS como en ROS 2 proporcionan dos alternativas viables y una amplia variedad de herramientas que podríamos emplear en la creación del sistema de comunicación multi-robot destinado a los robots DaNI o cualquier conjunto de robots que pueda desarrollarse en el futuro y que demande un sistema de comunicación en el entorno de ROS.

Capítulo 3

Concepción y Diseño de la Solución

3.1. Requerimientos funcionales

El sistema de comunicación para el sistema multi-robot de los robots DaNI se realizó de tal forma que cumpliera con los siguientes requerimientos:

- El sistema multi-robot debe estar compuesto por dos robots DaNI actualizados, los cuales cuentan con las mismas capacidades de navegación autónoma de un punto A a un punto B y evasión de obstáculos, considerando sus limitaciones físicas.
- El sistema de comunicación debe ser diseñado para una red de pequeña escala, considerando las limitaciones inherentes a la red WLAN y la capacidad de procesamiento del equipo donde se ejecute la interfaz gráfica de usuario (GUI) básica.
- El usuario debe disponer de una GUI básica donde puede encontrar:
  - Una sección para la selección de los robots a utilizar, que incluye una opción para elegir ambos simultáneamente.
  - Un apartado en la sección de selección de los robots donde se muestra la posición actual de cada robot con la latitud y la longitud.
  - Una sección de selección de coordenadas destino, con dos opciones disponibles: ingresar las coordenadas manualmente o seleccionar una ubicación previamente guardada con un seudónimo.
  - Un apartado dentro de la sección de selección de coordenadas destino que permita guardar localmente coordenadas con un seudónimo, así como la posibilidad de eliminar las coordenadas guardadas anteriormente.
  - Una sección donde se pueda identificar cuál de los robots se encuentra más cerca a unas coordenadas de interés ingresadas manualmente.
  - Un mapa en el navegador web, donde se pueda visualizar mediante marcadores tanto la posición actual de los robots como sus destinos seleccionados, junto con el rastro de su trayectoria.
  - Una sección para detener a los robots que se encuentren en navegación.

- Un recuadro de información importante en la sección de inicio de navegación, que muestra avisos relevantes sobre el estado de los robots, como por ejemplo si se encuentran en navegación hacia su destino, si la navegación fue cancelada, si no pudo llegar al destino o si logró alcanzarlo.

3.2. Requerimientos de hardware

3.2.1. Estado pasado

En el trabajo de grado titulado "Actualización del sistema de navegación para exteriores de un robot móvil terrestre usando Robot Operating System", llevado a cabo en el año 2022, se realizó una investigación exhaustiva orientada a la actualización del robot DaNI. Este proceso incluyó una selección rigurosa del hardware más adecuado para cumplir con los objetivos del proyecto. Para la elección de los componentes, se tomaron en cuenta una serie de criterios relevantes. Entre ellos, se destacan la relación calidad-precio, la compatibilidad tecnológica con el sistema existente, la facilidad de implementación, el consumo de energía y la capacidad de suministro en el caso de la batería, así como parámetros técnicos clave como la resolución y la precisión, los cuales se abordaron de acuerdo a cada dispositivo en cuestión [9].

Además, se consideraron otros factores esenciales como la durabilidad, la escalabilidad y el soporte técnico disponible para los componentes seleccionados, con el fin de garantizar que las actualizaciones no solo cubrieran las necesidades actuales del robot, sino que también proporcionaran flexibilidad para futuras mejoras [9]. El conjunto de hardware actualizado, seleccionado para desarrollar el sistema de navegación autónoma, se presenta en la Tabla 3.2.

Por otro lado, aquellos componentes que no requerían actualización debido a su funcionalidad y desempeño satisfactorio en el sistema original se enumeran en la Tabla 3.1. Esta diferenciación permite visualizar de manera clara qué elementos del sistema fueron renovados y cuáles se mantuvieron en el desarrollo del anterior proyecto de actualización del robot DaNI.

Componente	Alimentación	Rango de conexión	Resolución	Torque	Velocidad	Velocidad de transmisión
Batería NiMH	**Entrada (para carga):** -Voltaje de carga: 14,4 V -Corriente de carga: 0,9 A **Salida:** 12 V - 3000 mAh	N/A	N/A	N/A	N/A	N/A
Encoders ópticos de cuadratura	5 V - 15 mA	N/A	100 CPR	N/A	N/A	N/A
Motores Pitsco Education 39530 Tetrix Max DC	12 V - [0,5 A - 1,5 A]	N/A	N/A	320 oz-in	152 RPM	N/A
Router ASUS WL-330N	5 V - 500 mA	Hasta 45 m (campo abierto)	N/A	N/A	N/A	150 Mbps

Tabla 3.1: Hardware preservado en la actualización del Robot DaNI.

Componente	Alimentación	Arquitectura	Velocidad Unidad de Procesamiento	Distancia máxima	Frecuencia de escaneo	Núcleos	Precisión	Rango angular	Sensibilidad antena
Arduino UNO	5V - 500mA	AVR	16 MHz	N/A	N/A	1	N/A	N/A	N/A
Batería Xiaomi Power Bank 3 Pro	**Entrada (Input):** 5 V - 3 A 9 V - 2 A 12 V - 1,5 A **Salida (Output):** USB-A (puerto único): 5 V - 2,4 A, 9 V - 2 A, 12 V - 1,5 A USB-A (Dual USB): 5 V - 3 A USB-C: 5 V - 3 A, 9 V - 3 A, 12 V - 3 A, 15 V - 3 A, 20 V - 2 A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Controlador L298N	12 V - 2 A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
GPS VK-162 G-Mouse	5 V - 80 mA	N/A	N/A	N/A	N/A	N/A	2.5 m - 5 m	N/A	-162 dBm (-192 dBW)
Hub Ugreen - USB 3.0	5 V - 2 A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
IMU BNO055	3,3 V - 50 mA	N/A	N/A	N/A	N/A	N/A	**Giroscopio:** - Rango: ±125°/s a ±2000°/s - Resolución: 16 bits **Acelerómetro**: - Rango: ±2 g a ±16 g - Resolución: 14 bits **Magnetómetro**: - Rango: ±1300 µT (eje x, y), ±2500 µT (eje z) - Resolución: 16 bits	N/A	N/A
Jetson Nano 4 GB RAM	5 V - 2 A	ARM Cortex 57	1,43 GHz	N/A	N/A	**CPU:** 4 núcleos ARM Cortex-A57 **GPU:** 128 núcleos NVIDIA Maxwell	N/A	N/A	N/A
LiDAR A3	5 V - 450 mA	N/A	N/A	25 m (indoor) 20 m (outdoor)	15 Hz	N/A	≤ 3 m: 1 % del rango 3 m - 5 m: 2 % del rango 5 m - 20 m: 2,5 % del rango	360°	N/A
Tarjeta de Red AC8265	3,3 V - 300 mA	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**Banda de 2,4 GHz:** 802.11n (MCS0, 20 MHz): -88 dBm 802.11n (MCS7, 20 MHz): -73 dBm 802.11n (MCS0, 40 MHz): -85 dBm 802.11n (MCS7, 40 MHz): -70 dBm **Banda de 5 GHz:** 802.11ac (MCS0, 20 MHz): -86 dBm 802.11ac (MCS9, 80 MHz): -63 dBm 802.11n (MCS0, 20 MHz): -86 dBm 802.11n (MCS7, 40 MHz): -69 dBm

Tabla 3.2: Hardware nuevo implementado en la actualización del Robot DaNI.

3.2.2. Estado deseado

Conforme a lo expuesto en las secciones 2.3 y 3.1, en las que se detallan los objetivos y los requerimientos funcionales para el sistema de comunicación multi-robot, en las cuales se establece la necesidad de contar con dos robots DaNI actualizados para implementar el sistema de comunicación, se presenta un listado de componentes en la Tabla 3.3 que deberá estar presente en cada Robot DaNI de acuerdo al Trabajo de grado "Actualización del sistema de navegación para exteriores de un robot móvil terrestre usando Robot Operating System" [9].

Componente	**Referencia**
Batería Motores	NiMH de 12V 3000mAh (Modern Robotics)
Batería placa de procesamiento principal	Xiaomi Power Bank 3 Pro
Controlador	Puente H (L298N)
Encoders	Ópticos de cuadratura (Motor Pitsco 39530)
GPS	VK-162 G-Mouse
IMU	BNO055
LiDAR	A3
Microcontrolador	Arduino Uno
Motores	Pitsco Educación 39530 Tetrix Max DC
Placa de procesamiento principal	Jetson Nano (4 GB RAM)
Puertos de expansión USB	Hub Ugreen - USB 3.0
Tarjeta de Red	AC8265

Tabla 3.3: Listado de componentes de cada Robot DaNI.

Por otro lado, en la Figura 3.1, se muestra la arquitectura eléctrica que se deberá implementar en cada Robot DaNI.

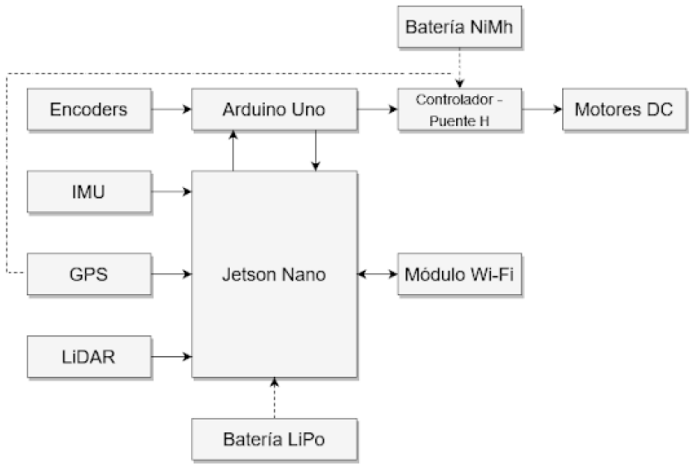

Figura 3.1: Arquitectura eléctrica Robot DaNI (Fuente: [9]).

3.2.3. Selección del hardware

En la etapa final de selección del hardware, se realizaron tres modificaciones significativas debido a las restricciones presupuestarias y, en un caso, a la no disponibilidad, que impidieron la adquisición de todos los componentes previstos para la actualización del segundo Robot DaNI. El consolidado donde se refleja las compras realizadas y el presupuesto usado se visualiza en la Tabla 4.10 de la Sección 4.1. Los componentes que no fueron adquiridos, según lo descrito en la Tabla 3.3, incluyen el LiDAR A3, el GPS VK-162 G-Mouse y Batería Xiaomi Power Bank 3 Pro. En el caso del LiDAR A3, su precio representaba un valor superior al presupuesto total del proyecto. Por lo cual se concluyó que su adquisición no era viable. Como alternativa, se optó por reutilizar un LiDAR A1 mostrado en la Figura 3.2 disponible en el inventario del Centro de Automatización de Procesos (CAP) de la Universidad Javeriana Cali.

Figura 3.2: RPLiDAR A1 (Fuente: [10]).

Aunque el LiDAR A1 presenta algunas limitaciones en cuanto a rango de medición y precisión en comparación con el LiDAR A3, su desempeño se consideró adecuado para cumplir con los requisitos básicos del proyecto. Los análisis previos indicaron que, si bien el LiDAR A3 ofrecía mejoras en cuanto a distancia y resolución [9], el LiDAR A1 puede ser integrado eficazmente en la arquitectura del proyecto, ya que su rango de detección sigue siendo suficiente para las tareas de navegación y mapeo según los parámetros de configuración en los *Launch* relacionados con el LiDAR, estos parámetros se pueden encontrar en la ruta $\backslash catkin\_ws \backslash src \backslash my\_robot\_tools \backslash params$, o se pueden observar en la Tabla 3.4.

Parámetro	**Descripción**	**Valor**
obstacle_range	Establece la distancia máxima desde el robot (en metros) a la que se insertarán obstáculos en el mapa de costos.	2.5 m
raytrace_range	Define la distancia máxima (en metros) en la que se rastrearán los obstáculos.	5.5 m
update_frequency	Indica la frecuencia con la que se actualiza el mapa local.	1 Hz
publish_frequency	Define la frecuencia con la que se publica el mapa local.	2 Hz
width	Establece el ancho del mapa local en metros.	10 m
height	Define la altura del mapa local en metros.	10 m

Tabla 3.4: Valores de parámetros LiDAR.

Las principales caracteristicas del LiDAR A1 se pueden observar en la Tabla 3.5 [10].

Ítem	Unidad	Mín	Típico	Máx
Rango de distancia	Metro(m)	TBD	0.15 - 6	TBD
Rango Angular	Grado	n/a	0-360	n/a
Resolución de Distancia	mm	n/a	<0.5 <1 % de la distancia	n/a
Resolución Angular	Grado	n/a	≤ 1	n/a
Duración de Muestra	Milisegundo(ms)	n/a	0.5	n/a
Frecuencia de Muestreo	Hz	n/a	≥ 2000	2010
Tasa de Escaneo	Hz	1	5.5	10

Tabla 3.5: Características LiDAR A1.

Por otro lado, la batería Xiaomi Power Bank 3 Pro no pudo ser adquirida debido a la falta de stock al momento de realizar la compra. Como alternativa, se optó por un modelo con especificaciones equivalentes en términos de capacidad de almacenamiento de energía (20,000 mAh) y potencia máxima de salida (45W). La opción seleccionada fue la power bank QC 3.0 de EMIGVELA, la cual, además de ofrecer las mismas prestaciones eléctricas, presentó un costo menor. Aunque el tamaño físico de la batería es diferente al de la Xiaomi Power Bank 3 Pro, esta variación no afecta su instalación en el robot, lo que permite cumplir con los requerimientos del proyecto sin comprometer su funcionalidad. En la Figura 3.3 se presenta la batería.

Figura 3.3: Power bank QC 3.0 EMIGVELA (Fuente: [11]).

Finalmente, en lo que respecta al GPS VK-162 G-Mouse, se llevó a cabo un análisis sobre su relevancia en el sistema de comunicación. Si bien es de suma importancia para tener una mayor precisión en la navegación autónoma, dadas las pruebas hechas en el trabajo de grado [9], donde se evidenció que su uso representaba una mayor exactitud hacia la meta de destino, para nuestro

propósito que es lograr el desarrollo del sistema de comunicación no es un factor crítico. Por lo tanto, se optó por mantener el GPS GARMIN 18X, el cual ya se encontraba integrado en el robot DaNI. Aunque este modelo no ofrece el mismo nivel de precisión que el VK-162 G-Mouse, cumple adecuadamente con las funciones de geolocalización requeridas para la navegación autónoma. La decisión de conservar el GPS GARMIN 18X se fundamenta en la necesidad de equilibrar la eficiencia de costos con la funcionalidad del sistema. De este modo, se asegura el cumplimiento de los objetivos del proyecto, sin comprometer su viabilidad. Las principales diferencias entre el GPS VK-162 G-Mouse y el GPS GARMIN 18X se evidencian en la Tabla 3.6.

Característica	**GPS VK-162 G-Mouse**	**GPS GARMIN 18X**
Sensibilidad	-162 dBm (-192 dBW)	-185 dBW
Frecuencia de actualización	1 Hz por defecto (configurable a 10Hz)	1 Hz (opcionalmente hasta 5 Hz)
Precisión (posicionamiento horizontal)	<2,5 m	<3 metros (con SBAS, como WAAS)
Tiempo de posicionamiento	Cold start: 29 s Warm start: 28 s Hot start: 1 s Re-adquisición: aproximadamente 1-2 s	Cold start: <45 s Warm start: <38 s Hot start: <1 s Re-adquisición: <1 s
Interfaz	USB 2.0	RS-232 y niveles TTL
Consumo de potencia	5 V - 45 mA	8-30 V DC - 80 mA
Temperatura de operación	-40° C a 85° C	-30° C a 80° C
Tasa de Baudios	9600 bps predeterminadamente	4800 bps por defecto (configurable hasta 38400 bps)
Formato de salida	NMEA 0183	NMEA 0183
Dimensiones	46 mm x 38 mm x 14 mm	Diámetro de 61 mm y altura de 19,5 mm

Tabla 3.6: Comparación de características entre GPS VK-162 G-Mouse y GPS GARMIN 18X.

3.3. Requerimientos de software

3.3.1. Estado pasado

Como se ha señalado previamente en las secciones 2.1 y 2.2, la actualización del sistema de navegación autónoma del robot DaNI en el año 2022 [9] se implementó utilizando el framework de ROS (Robot Operating System), el cual se destaca por ofrecer una gran modularidad y escalabilidad para el desarrollo de proyectos robóticos. ROS no solo facilita la integración de componentes, sino que también cuenta con un extenso conjunto de bibliotecas de libre acceso desarrolladas por la comunidad, que permiten la configuración eficiente de sensores, actuadores y otros elementos del sistema robótico. En el marco de la actualización del sistema de navegación autónoma del robot

DaNI, se llevó a cabo una cuidadosa selección de paquetes y bibliotecas, orientada tanto a garantizar el correcto funcionamiento del hardware como a optimizar la planificación de trayectorias y la evasión de obstáculos. Además, se desarrolló un paquete específico para la representación virtual del robot DaNI. Para los componentes de hardware como la IMU BNO055, el GPS VK-162 G-Mouse, el LiDAR A3, el Arduino UNO y el controlador L298N, se seleccionaron los paquetes que se muestran en la Tabla 3.7. Estos paquetes permiten la adecuada integración de cada componente dentro del sistema, asegurando su correcto uso y funcionamiento.

Componente	**Package**
IMU BNO055	ros_imu_bno055
GPS VK-162 G-Mouse	nmea_navsat_driver
LiDAR A3	rplidar_ros
Arduino UNO - Controlador L298N	rosserial

Tabla 3.7: Correspondencia entre componentes y paquetes en la actualización del robot DaNI.

Package	**Descripción**
keyboard_control	Es una herramienta que permite teleoperar el movimiento del robot DaNI utilizando las teclas del teclado.
tf2_ros	Proporciona herramientas para gestionar la transformación de marcos de referencia (frames) en el espacio tridimensional, esto asegura que los datos y transformaciones entre sensores, actuadores y otras partes del robot se alineen tanto en la simulación como en el mundo físico.
robot_localization	Es una herramienta fundamental para la localización y navegación de robots móviles. Su principal objetivo es mejorar la precisión en la estimación de la posición y orientación del robot al fusionar datos de múltiples fuentes de sensores (Filtro Kalman (EKF)).
navigation	Proporciona las herramientas necesarias para planificar y ejecutar trayectorias, localizar el robot en su entorno, evitar obstáculos y adaptarse a situaciones dinámicas con la creación de mapas de costos local y global.
waypoint_nav	Es una herramienta que en conjunto con algunos paquetes de robot_localization permite definir las metas destino, así como las servicio de acción en la navegación, planificación de rutas y evasión de obstáculos.
geographiclib	Proporciona herramientas especializadas para llevar a cabo cálculos geodésicos y geométricos con alta precisión en la superficie terrestre. Resulta particularmente útil para la transformación de coordenadas entre diferentes sistemas de referencia, como el sistema Universal Transverse Mercator (UTM).

Tabla 3.8: Packages complementarios para navegación, planificación y evasión de obstáculos.

Por otro lado, para asegurar un rendimiento óptimo del sistema de navegación autónoma, se implementaron varios paquetes que abarcan áreas clave como la localización, la odometría, la fusión sensorial, la planificación de trayectorias, la detección de obstáculos, la gestión de metas y teleoperación. Estos paquetes, junto con sus respectivas funcionalidades, se detallan en la Tabla 3.8.

Finalmente, se creó un paquete específico para la virtualización del robot DaNI en 3D. Este proceso comenzó con el modelado del robot en SolidWorks, seguido de la conversión a formato URDF, utilizando el paquete *solidworks_urdf_exporter*. El modelo virtual obtenido permite la representación del robot en herramientas como RVIZ y Gazebo, proporcionando una correspondencia precisa entre el mundo físico y el entorno virtual. El package que contiene esta representación se denomina *robotdani*.

La correcta selección y configuración de estos paquetes, adaptados específicamente a las necesidades del proyecto de navegación autónoma, permitió dotar al robot DaNI de una arquitectura modular y flexible. Esta modularidad no solo facilitó la integración eficiente de los diversos componentes, sino que también le brindó al robot la capacidad de realizar navegación autónoma en entornos exteriores, ajustándose de manera efectiva a los desafíos y dinámicas del entorno.

3.3.2. Estado deseado

Para alcanzar los objetivos planteados en la sección 2.3, es necesario integrar al proyecto de navegación autónoma, específicamente dentro del workspace denominado catkin_ws que alberga el proyecto, una serie de bibliotecas y paquetes adicionales. Estos recursos deben facilitar el desarrollo e implementación del sistema de comunicación para el sistema multi-robot que involucra dos robots DaNI actualizados. La integración permite la visualización de nodos, tópicos y servicios esenciales para el monitoreo y control de la navegación autónoma conjunta, desde una interfaz gráfica de usuario.

El control y monitoreo se lleva a cabo mediante una interfaz gráfica de usuario (GUI) básica, diseñada para que un usuario sin conocimientos avanzados en el uso de líneas de comando pueda gestionar la navegación conjunta o individual de los robots. Esta GUI debe proporcionar una experiencia intuitiva, facilitando la puesta en marcha y operación de los robots. Además, la GUI debe estar instalada en la máquina desde la cual el usuario ejecutará las acciones de control, asegurando así una interfaz accesible y funcional que permita la interacción con los sistemas de navegación autónoma de manera eficiente y sin complicaciones técnicas.

3.3.3. Selección del software

Para el desarrollo e implementación del sistema de comuncicación y la GUI básica para el SMR de los robots DaNI, se busca identificar y seleccionar bibliotecas y paquetes que sean compatibles con el framework ROS, enfocándose particularmente en la versión Melodic, que fue previamente elegida en el proyecto de la actualización del robot DaNI [9]. Esta versión de ROS ha sido diseñada y optimizada específicamente para funcionar en la distribución de Linux Ubuntu 18.04 [52], lo que

implica que tanto las herramientas de software adicionales como las bibliotecas complementarias deben cumplir con los requisitos de compatibilidad para este sistema operativo. Además, la correcta integración de estas herramientas y bibliotecas es crucial para garantizar un entorno de desarrollo estable y eficiente, que permita aprovechar al máximo las funcionalidades de ROS en esta versión, evitando problemas de incompatibilidad que puedan surgir durante el desarrollo. Por lo tanto, se prioriza la investigación y el análisis detallado de todas las dependencias necesarias para asegurar un funcionamiento óptimo del sistema en su conjunto.

3.3.3.1. Sistema de comunicación

Se identificaron principalmente dos alternativas para el desarrollo e implementación del sistema de comunicación entre los robots DaNI bajo la versión de ROS Melodic:

- **ROS Network:** La arquitectura de ROS está diseñada como un sistema distribuido, en el cual los nodos, tópicos y servicios pueden residir en diferentes máquinas. En este contexto, el ROS Master se encarga de proporcionar la información necesaria sobre la localización de los tópicos y servicios a aquellos nodos interesados en publicar o suscribirse, tal como se describe en la subsección 2.4.2.3. Cuando los nodos, tópicos y servicios se encuentran distribuidos en distintas máquinas, es necesario configurar las variables de entorno *ROS_MASTER_URI* y *ROS_IP*. La primera, ROS_MASTER_URI, debe contener la dirección IP de la máquina que alojará el ROS Master, y esta configuración debe ser idéntica en todas las máquinas, ya que indica la ubicación centralizada del master. Por otro lado, la variable ROS_IP debe establecerse de manera individual en cada máquina, correspondiendo a su dirección IP dentro de la red WLAN, ya que cada una de ellas posee una dirección única. Este modelo de comunicación se visualiza en la Figura 3.4.

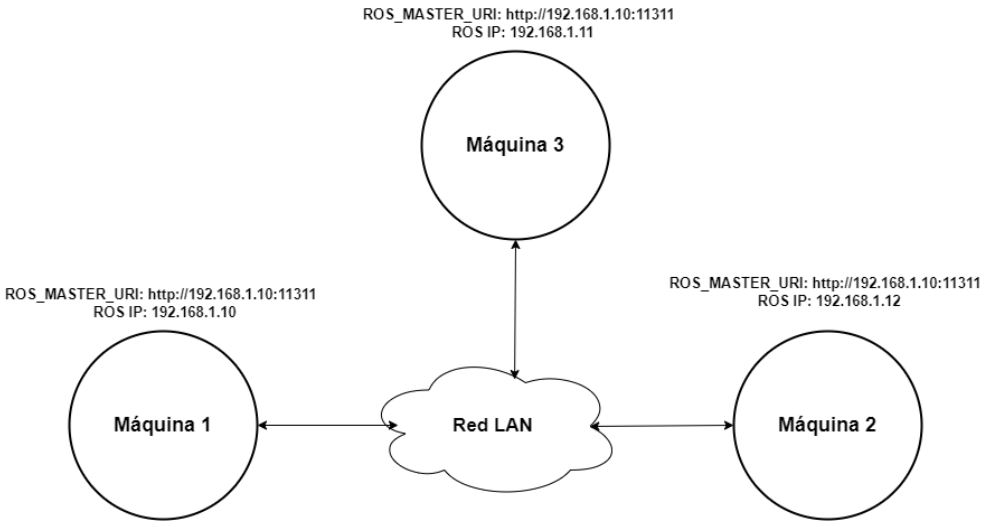

Figura 3.4: Sistema de comunicación con ROS Network (Fuente: Elaboración propia).

- **Multimaster_fkie:** Este paquete permite trabajar con una red multimaster, en la cual ya no se maneja un maestro (ROS Master) de forma centralizada, sino que cada máquina cuenta con su propio maestro (descentralización). Esto se logra a través de los siguientes paquetes:

  - **master_discovery:** Este paquete se encarga de descubrir los ROS Master en la red WLAN mediante el nodo *master_discovery*, que envía mensajes multicast o unicast como heartbeats utilizando el protocolo UDP (el protocolo se describe en la sección 2.4.2.2) en el puerto configurado por el parámetro *mcast_port*. De forma similar, este nodo también recibe mensajes heartbeat de otros nodos *master_discovery* en la red WLAN a través del mismo puerto.

  - **master_sync:** Este paquete incluye un nodo que sincroniza el ROS Master local con los ROS Master remotos detectados por el nodo *master_discovery*. A través de la *ROSMaster_API*, es posible registrar nodos, gestionar tópicos y servicios, resolver nombres y compartir el almacenamiento de parámetros entre máquinas locales y remotas. Además, este nodo proporciona varios parámetros útiles, como *ignore_hosts*, que permite ignorar ciertos ROS Masters remotos dentro de la red WLAN; *sync_hosts*, que especifica con qué ROS Masters se desea sincronizar; y otros parámetros como *ignore_nodes*, *ignore_topics* e *ignore_services*, que permiten excluir nodos, tópicos y servicios específicos de la sincronización.

  Con estos dos paquetes, es posible visualizar los tópicos y servicios de cada ROS Master desde un solo maestro que esté sincronizado con otros master en la misma red WLAN. Este sistema de comunicación se ilustra en la Figura 3.5.

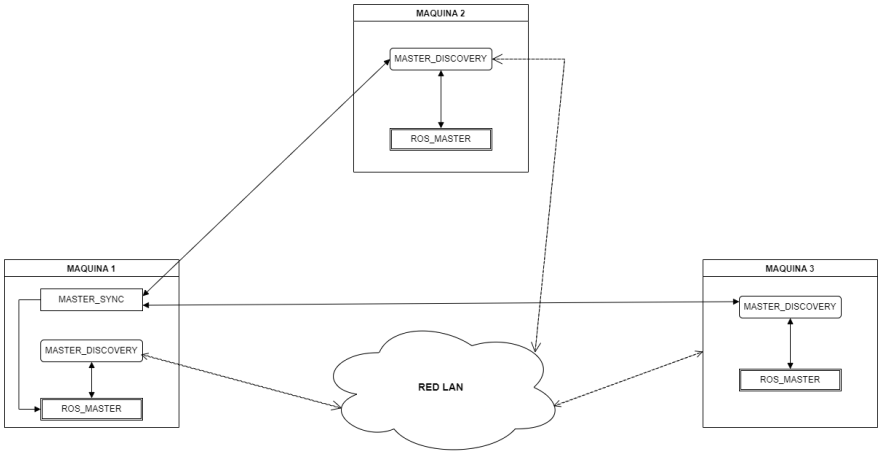

Figura 3.5: Sistema de comunicación multimaster_fkie (Fuente: Elaboración propia).

Finalmente, se opta por la elección del paquete multimaster_fkie para el sistema de comunicación de los robots DaNI, debido a sus claras ventajas frente a la configuración estándar de ROS Network, como la sincronización automática de nodos, tópicos y servicios entre máquinas, características

esenciales en entornos distribuidos. Además, ofrece parámetros avanzados como la capacidad de ignorar nodos o tópicos específicos, y seleccionar los ROS Masters con los que se desea sincronizar. En comparación, ROS Network no proporciona este nivel de flexibilidad, lo que hace a multimaster_fkie la opción óptima para la comunicación dentro de la red WLAN de los robots DaNI.

3.3.3.2. Interfaz gráfica de usuario (GUI)

En el proceso de selección del software adecuado para el desarrollo de la interfaz gráfica de usuario (GUI), se evaluaron diversas herramientas disponibles en el ecosistema de Python, dada nuestra familiaridad con este lenguaje y las ventajas que ofrece, como su sintaxis clara, amplia comunidad y disponibilidad de bibliotecas robustas. El objetivo principal de esta evaluación fue identificar una herramienta capaz de satisfacer nuestros requerimientos para el desarrollo de la interfaz de una manera eficiente. Las tres herramientas más relevantes encontradas fueron:

- **Tkinter:** Tkinter es una biblioteca estándar de Python para el desarrollo de aplicaciones de escritorio multiplataforma, compatible con los sistemas operativos Windows, macOS y Linux/Unix. Al estar incluida por defecto en la mayoría de las distribuciones de Python, es una opción accesible y fácil de implementar para proyectos sencillos. Sin embargo, a pesar de ser la opción estándar para crear GUIs en Python, Tkinter presenta limitaciones en cuanto a la creación de interfaces gráficas complejas o modernas, lo que lo convierte en una herramienta más adecuada para desarrollos básicos o prototipos rápidos.

- **wxPython:** wxPython es un binding que actúa como un enlace entre Python y la biblioteca gráfica wxWidgets, originalmente desarrollada en C++. Esta integración permite utilizar las herramientas avanzadas de wxWidgets para el desarrollo de interfaces gráficas de usuario directamente desde Python. Gracias a esta capacidad, wxPython facilita la creación de aplicaciones de escritorio multiplataforma, compatibles con sistemas operativos como Windows, macOS y Linux. Además, ofrece un conjunto completo de herramientas para el desarrollo de GUIs complejas y robustas.

- **PyQt:** PyQt es un binding que conecta el lenguaje de programación Python con el framework Qt, una de las bibliotecas más completas y poderosas para el desarrollo de interfaces gráficas de usuario multiplataforma. Qt, originalmente escrito en C++, permite crear aplicaciones compatibles con sistemas operativos como Windows, macOS y Linux/Unix. PyQt ofrece acceso a todas las funcionalidades de Qt, lo que lo convierte en una herramienta versátil y flexible para el desarrollo de aplicaciones de escritorio.

  Una de las características destacadas de PyQt es que, desde Qt 4 en adelante, incluye una herramienta visual llamada Qt Designer, la cual facilita la creación de interfaces gráficas ya que permite a los usuarios diseñar GUIs de manera visual lo que optimiza el proceso de desarrollo al ofrecer una representación instantánea de los elementos que se están creando. Esto no solo acelera la implementación, sino que también proporciona una vista previa en tiempo real del diseño final, lo que ayuda a garantizar que los objetivos de diseño se cumplan de manera

efectiva, por esto permite desarrollar interfaces gráficas modernas y altamente personalizables, adaptadas a los requerimientos específicos del proyecto.

Después de un análisis de las capacidades de diseño, la facilidad de uso y la rapidez de implementación de cada herramienta, como se muestra en la Tabla 3.9 (donde se califica de 1 a 5 cada característica), se consideró que la opción más adecuada para nuestro proyecto era PyQt5, en su versión 15.2. Esta versión, lanzada en noviembre de 2020, combina la madurez y estabilidad necesarias para garantizar una base sólida de desarrollo, sin ser una versión obsoleta o demasiado reciente. PyQt5 nos permite aprovechar las últimas funcionalidades de Qt, asegurando la creación de una interfaz gráfica moderna y eficiente que cumple con los objetivos planteados.

Característica	**Importancia (%)**	**Tkinter**	**wxPython**	**PyQt5**
Facilidad de Uso	35 %	3	3	4.5
Rapidez para desarrollo	30 %	2	2	4
Facilidad de Instalación	15 %	4.5	3	3
Multiplataforma	20 %	5	5	5
Total	**100 %**	**3.3**	**3.1**	**4.2**

Tabla 3.9: Matriz de decisión del software para la GUI.

3.4. Procesos e integraciones

3.4.1. Actualización segundo robot DaNI

La actualización del segundo robot DaNI se estructuró en tres etapas principales:

- **Montaje de hardware:** En esta etapa del proyecto, se emplearon prácticamente todos los componentes previamente seleccionados para el primer robot DaNI, como se detalla en la Tabla 3.3 dentro de la sección 3.2.3. Sin embargo, algunos elementos específicos fueron reemplazados o eliminados, de acuerdo con los requerimientos particulares de este segundo prototipo, cambios que también se describen en detalle en dicha sección. La integración eléctrica de los componentes se realizó conforme al diagrama esquemático presentado en el Anexo 1. Este esquema proporcionó una guía en las conexiones de las diferentes partes del sistema.

  El montaje físico del robot se llevó a cabo siguiendo un procedimiento sistemático que ha sido documentado de manera detallada en el Anexo 2, donde se incluyen fotografías y descripciones paso a paso de cada fase del ensamblaje. Esto garantiza replicabilidad y facilita futuras modificaciones o mejoras.

  El resultado final de esta etapa de desarrollo puede observarse en la Figura 3.6, la cual ilustra la configuración final del robot con todos sus componentes ensamblados y operativos, ofreciendo una representación visual del diseño implementado. Este enfoque integral permitió consolidar un segundo prototipo funcional y alineado con los objetivos del proyecto.

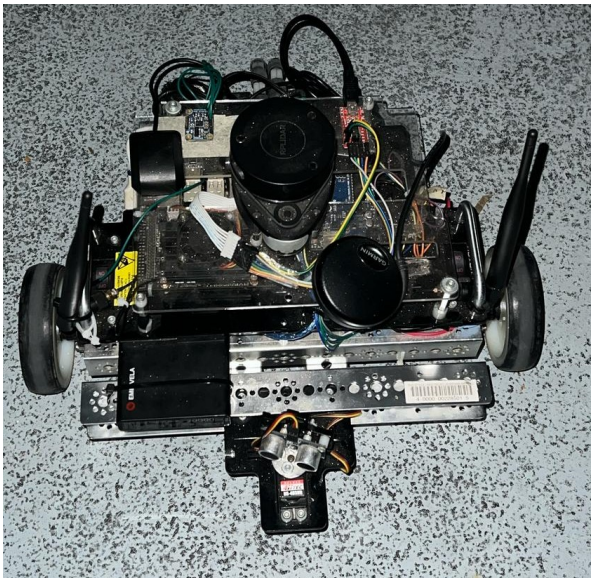

Figura 3.6: Segundo robot DaNI actualizado (Fuente: Elaboración propia).

- **Instalación del Sistema Operativo (OS):** La instalación del sistema operativo en la Jetson Nano se realizó siguiendo las instrucciones oficiales de NVIDIA, utilizando un equipo con Windows para escribir la imagen en la tarjeta microSD. En primer lugar, se empleó el software SD Card Formatter para formatear completamente la tarjeta, eliminando posibles particiones de fábrica y minimizando errores en la escritura de la imagen. Luego, se grabó la imagen JetPack 4.6 que contiene la distribución Linux Ubuntu 18.04 LTS descargada de la página oficial de NVIDIA mediante el programa Etcher. Finalmente, se configuró el sistema operativo en la Jetson Nano.

- **Instalación y verificación del proyecto de navegación autónoma en ROS:** Para este proceso a nivel de software se realizó inicialmente la instalación de ROS Melodic siguiendo los pasos dados en su wiki, por lo cual se instaló, configuró y descargó algunas dependencias de ROS Melodic para su correcto funcionamiento y manejo de proyectos [53].

  Posteriormente, se copió el proyecto de navegación autónoma de ROS, desarrollado originalmente en el primer robot DaNI, y se trasladó al segundo robot DaNI. Esto implicó la configuración del entorno de trabajo del proyecto y entre lo más importante está la instalación de algunas dependencias que se encuentran en el paquete *geographiclib* que es utilizado por los paquetes que hacen uso del GPS para realizar las transformaciones de los marcos de referencia. Para la instalación de estas dependencias, fue necesario acceder a la carpeta de *geographiclib* desde el terminal, ejecutar el comando *./configure*, seguido de *make*, y finalizar el proceso con *sudo make install*, asegurando así la correcta integración de las herramientas de referencia geográfica. Finalmente, el proyecto fue compilado mediante el comando *catkin_ make*, el cual permitió detectar cualquier dependencia faltante e instalarla, logrando de esta manera una

compilación completa y verificada del proyecto de navegación autónoma en el segundo robot DaNI.

Con los paquetes de cada componente de hardware disponibles, se procedió a realizar un chequeo individual de los dispositivos principales: el LiDAR, el GPS, la IMU y los motores del robot. Dado que se empleó una versión A1 de LiDAR diferente a la utilizada en el primer robot DaNI, fue necesario configurar el *baudrate* de 256000 a 115200, además de añadir el parámetro *inverted* y establecerlo en *true*. En el caso del GPS, únicamente fue necesario ajustar el *baudrate* de 9600 a 4800, mientras que para la IMU no se precisaron cambios en los parámetros del archivo *launch*, ya que se utilizó el mismo modelo del primer robot. Finalmente, para validar el funcionamiento correcto de los motores y la movilidad del robot, se empleó el paquete *keyboard_ control*, el cual permitió un control manual del robot y verificó su capacidad de giro y desplazamiento lineal de manera precisa.

Comprobada la correcta instalación y funcionamiento de los componentes del segundo robot DaNI, se procedió a modificar el archivo launch *full_ hw_ test*, configurando los parámetros correspondientes al LiDAR, GPS e IMU para alinearlos con las versiones específicas de cada componente y evitar conflictos en la compilación del proyecto.

Por otra parte, durante las pruebas operativas, se identificó un inconveniente relacionado con los cambios de puerto de los dispositivos GPS, LiDAR e IMU tras cada reinicio del robot. Este cambio constante de puertos implicaba una comprobación manual mediante el comando *ls -l /dev | grep ttyUSB* o *ls -l /dev | grep ttyACM* para asignar correctamente cada puerto en el archivo launch *full_ hw_ test*, lo cual resultaba tedioso y propenso a errores. Para solucionar esta problemática, se decidió implementar reglas *udev* que permite asignar pseudónimos constantes a cada componente basadas en sus características específicas de fabrica. Las reglas *udev* configuran los dispositivos conectados y desconectados mediante el servicio udev, el cual gestiona la creación y eliminación de nodos de dispositivos en respuesta a eventos del kernel del sistema.

Para crear estas reglas, en primer lugar, se identificaron los atributos *idVendor* e *idProduct* de cada dispositivo mediante el comando *lsusb*, que permitió diferenciar cada dispositivo de manera precisa. Con esta información, se generó el archivo *98-devices-robot.rules* que contiene las reglas para LiDAR, IMU y GPS como se muestra en la Figura 3.7.

```
# Asignar nombre persistente a un LiDAR basado en su ID de producto y fabricante
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6001", SYMLINK+="LidarA1"

# Asignar nombre persistente a una IMU basada en su número de serie
SUBSYSTEM=="tty", ATTRS{idVendor}=="10c4", ATTRS{idProduct}=="ea60", ATTRS{serial}=="0001", ATTRS{devpath}=="2.1", \
SYMLINK+="IMUBNO055"

# Asignar nombre persistente a un GPS basado en su ID de producto y fabricante
SUBSYSTEM=="tty", ATTRS{idVendor}=="1a86", ATTRS{idProduct}=="7523", SYMLINK+="GPS"

# Asignar nombre persistente a un GPS basado en su ID de producto y fabricante
SUBSYSTEM=="tty", ATTRS{idVendor}=="1546", ATTRS{idProduct}=="01a7", SYMLINK+="GPSUblox"
```

Figura 3.7: Reglas *udev* (Fuente: Elaboración propia).

En estas reglas, cada dispositivo se renombra de acuerdo a sus atributos, lo que garantiza que

los nombres asignados permanezcan constantes, facilitando la configuración de los parámetros de puerto en el archivo launch *full_hw_test* y simplificando la interacción operativa con el robot, como se observa en la Figura 3.8, donde al hacer uso del comando *ls -l /dev |grep ttyUSB* se observa un parámetro extra en los puertos y que corresponde al pseudónimo configurado en las reglas *udev*.

```
user@jetson:~$ ls -l /dev |grep ttyUSB
lrwxrwxrwx  1 root root           7 oct 29 16:38 GPS -> ttyUSB2
lrwxrwxrwx  1 root root           7 oct 29 16:38 gps1 -> ttyUSB1
lrwxrwxrwx  1 root root           7 oct 29 16:38 IMUBNO055 -> ttyUSB1
lrwxrwxrwx  1 root root           7 oct 29 16:38 LidarA1 -> ttyUSB0
crw-rw----  1 root dialout 188,   0 oct 29 16:38 ttyUSB0
crw-rw----  1 root dialout 188,   1 oct 29 16:38 ttyUSB1
crw-rw----  1 root dialout 188,   2 oct 29 16:38 ttyUSB2
```

Figura 3.8: Atributos de los puertos USB (Fuente: Elaboración propia).

3.4.2. Interfaz Gráfica (GUI)

El proyecto ROS desarrollado para el computador central desde donde se hará el control de los robots DaNI, por medio de la interfaz gráfica de usuario (GUI), hace uso del software Docker, con el objetivo de facilitar su instalación y uso, independientemente de ciertas limitaciones, como la versión del sistema operativo Linux (en este caso Ubuntu 18.04 LTS, requerida para ROS Melodic) o la instalación de sus dependencias. Esto permite mayor rapidez y facilidad de instalación de la GUI en cualquier máquina con OS Linux, brindándole al sistema mayor flexibilidad y modularidad, debido a que Docker encapsula los desarrollos de software en imágenes, que posteriormente pueden ejecutarse mediante contenedores, tal como se detalla en la sección 2.4.2.7. La imagen del proyecto para el computador central se encuentra disponible en este repositorio de Docker Hub. Para acceder al repositorio puede ser necesario acceder con una cuenta de Docker Hub. Los pasos necesarios para la ejecución del contenedor están descritos en el Anexo 3.

Por otro lado, el script *manager.py* ubicado en el paquete *server_manager*, contiene la lógica y el código de la GUI, desarrollada en Python con la biblioteca PyQt5 (versión 5.15.2), que ofrece un amplio conjunto de herramientas para la creación de interfaces gráficas interactivas y funcionales. Para la implementación de PyQt5 se creó un entorno virtual con Python 3.8, debido a que esta versión de PyQt5 requiere Python 3.5 o superior. Además, PyQt5 permite el uso de Qt Designer, una herramienta visual integrada que simplifica la creación y organización de los elementos de la interfaz de manera intuitiva y estructurada, tal como se especifica en la sección 2.4.2.6 del Capítulo 2. En cuanto al código de la GUI, fue elaborado en el marco de la Programación Orientada a Objetos (POO), a su vez se emplean conceptos como *signals*, *slots* y *events* específicos de los *widgets* de PyQt5. Por otro lado, la navegación se gestionó mediante el uso de hilos y, para la comunicación con el servidor del mapa, el cual se implementó en HTML utilizando la biblioteca Leaflet, se hizo uso de *sockets*.

Finalmente, la arquitectura de la GUI se organizó en dos frames principales: el primero, denominado

Configuración de Robot y Destino, y el segundo, denominado *Navegación*. Cada uno con funcionalidades específicas orientadas al control y monitoreo de los dos robots DaNI. Esta segmentación facilita el acceso y uso intuitivo del usuario en las distintas funciones de operación y supervisión del sistema. A continuación, se presenta una descripción detallada de cada frame y su propósito funcional en la interfaz, así como una representación de la máquina de estados diseñada para guiar el flujo de interacciones de la GUI en su conjunto.

- **Configuración de Robot y Destino:** Este módulo se divide en tres funcionalidades principales: *Elección de Robots*, *Coordenadas Destino* e *Identificar Robot Más Cercano*.

  En la funcionalidad de Elección de Robots, se puede escoger entre tres opciones de un ComboBox, las cuales son: *Robot 1*, *Robot 2* y *Todos los Robots* (en este caso, serían los dos robots DaNI actualmente disponibles). Esta funcionalidad se ilustra en la Figura 3.9.

Figura 3.9: Funcionalidad Elección de Robots (Fuente: Elaboración propia).

Al seleccionar alguna de las opciones en el panel de Elección de Robots se habilitará una opción adicional que permite verificar la latitud y longitud en tiempo real de los robots seleccionados, como se observa en la Figura 3.10, donde se ha elegido *Robot 1*, y en la Figura 3.11, donde se ha seleccionado *Todos los Robots*. Para visualizar esta ubicación, el usuario deberá pulsar el botón llamado *Ubicación Actual*.

Cabe destacar que la funcionalidad de coordenadas en tiempo real se implementó en el archivo de lanzamiento XML *launch_with_map_move_base* mediante el uso de *relay*, una herramienta proporcionada por el paquete *topic_tools*, que actúa como un repetidor para el tópico */fix* que recibe la información de latitud y longitud obtenida del GPS y la retransmite al tópico *$(arg robot_id)/gps* mediante el nodo *relay_node_$(arg robot_id)*. La expresión *$(arg*

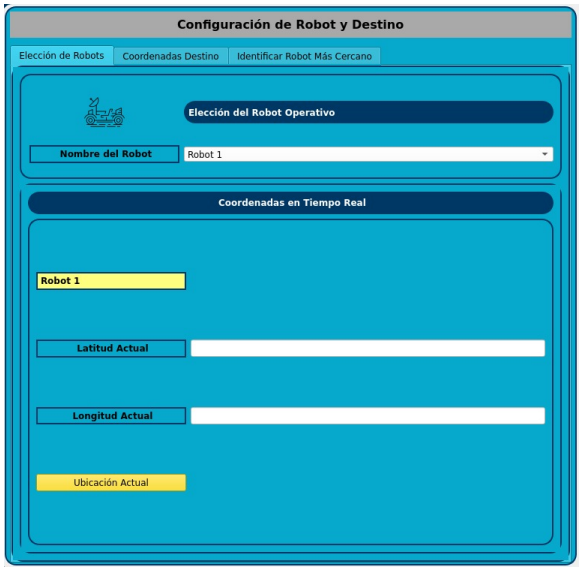

Figura 3.10: Frame Coordenadas en Tiempo Real del Robot 1 (Fuente: Elaboración propia).

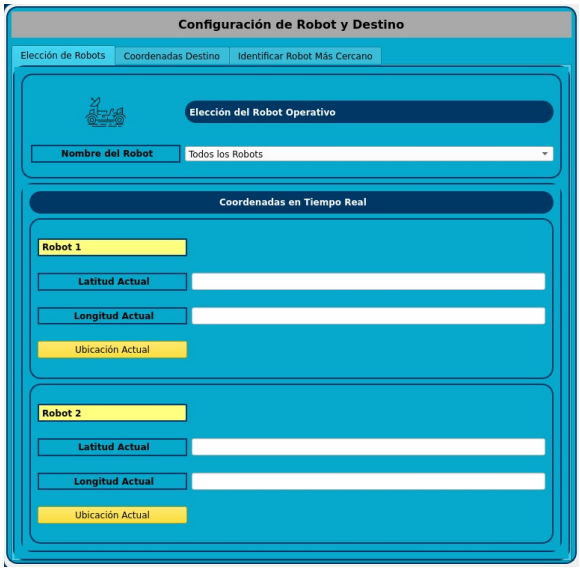

Figura 3.11: Frame Coordenadas en Tiempo Real de Todos los Robots (Fuente: Elaboración propia).

robot_id) hace referencia a un argumento dinámico definido dentro del mismo archivo de lanzamiento, cuyo valor es proporcionado en las variables de entorno de cada robot. Este argumento permite parametrizar el nombre del robot, haciendo que el tópico resultante y el nodo asociado sean únicos para cada robot en la red. Por ejemplo, si el valor de *robot_id* es *robot1*, el tópico será *robot1/gps* y el nodo será *relay_node_robot1*. Esto permite que los

tópicos de GPS de cada robot sean diferenciables en la red y facilite su uso en los métodos (*slots*) de los *widgets* asociados a los botones con nombre *Ubicación Actual* de cada robot correspondientemente.

La segunda funcionalidad se enfoca en la selección de las coordenadas destino, donde se habilitará el frame del modo de coordenadas destino para el robot 1, para el robot 2, para ambos o para ninguno, dependiendo de qué opción se escoja en el ComboBox del panel Elección de Robots. En la Figura 3.12 se muestra cómo se activa solo el frame de modo de coordenadas destino correspondiente al Robot 1 debido a que previamente en la funcionalidad de Elección de Robots se ha elegido "Robot 1", mientras que en la Figura 3.13 se activa el módulo para ambos robots al tiempo debido a que en la funcionalidad de Elección de Robots se ha elegido "Todos los Robots".

Figura 3.12: Funcionalidad Coordenadas Destino para el Robot 1 (Fuente: Elaboración propia).

Ahora para brindar las coordenadas destino se dispone de dos modos: *Coordenadas* o *Lugares*. Si se opta por el modo coordenadas, el usuario deberá ingresar manualmente la latitud y longitud del punto de destino, como se muestra en la Figura 3.14.

Si se selecciona el modo lugares, se mostrará una lista con los pseudónimos de coordenadas previamente guardadas. En caso de que esta lista esté vacía o se necesite incluir un destino adicional, se puede agregar un nuevo destino con latitud y longitud y asignarle un pseudónimo, como "R1", "R2" o "Casa", haciendo uso del botón *Agregar*. Esta función permite decidir entre dos maneras distintas de agregar el destino. La primera es agregando el destino con las coordenadas en tiempo real que recibe el robot del frame en el cual se hizo uso del botón *Agregar*, y la segunda es ingresando las coordenadas manualmente. Este pseudónimo aparecerá en la lista de selección, como se observa en la Figura 3.15.

Adicionalmente, existe la opción de eliminar algún lugar guardado de la lista haciendo uso del

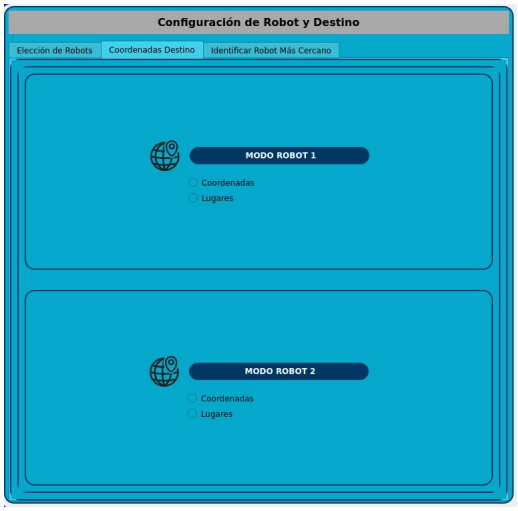

Figura 3.13: Funcionalidad Coordenadas Destino para Todos los Robots (Fuente: Elaboración propia).

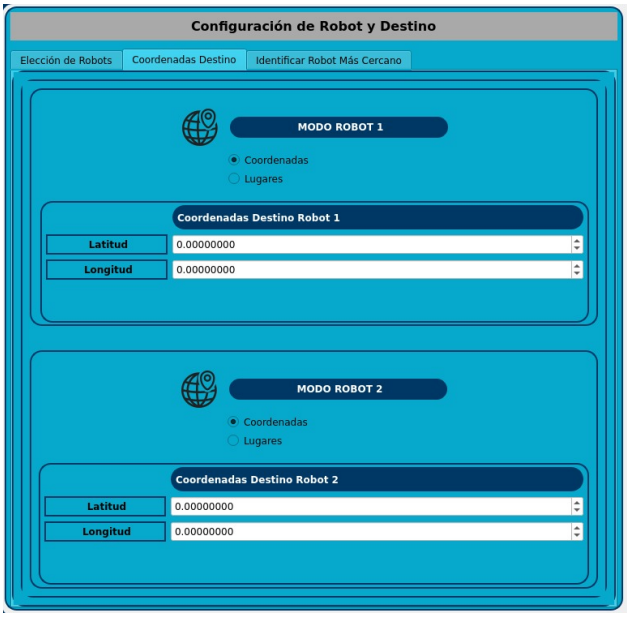

Figura 3.14: Modo Coordenadas para Todos los Robots (Fuente: Elaboración propia).

botón *Eliminar*. La lista de lugares se sincroniza con un archivo JSON llamado '*lugares*', que se guarda localmente en el computador que controla los robots mediante la interfaz gráfica. De esta forma, al cerrar la GUI, la lista se guarda en este archivo, y al abrirla nuevamente, se carga la lista previamente guardada.

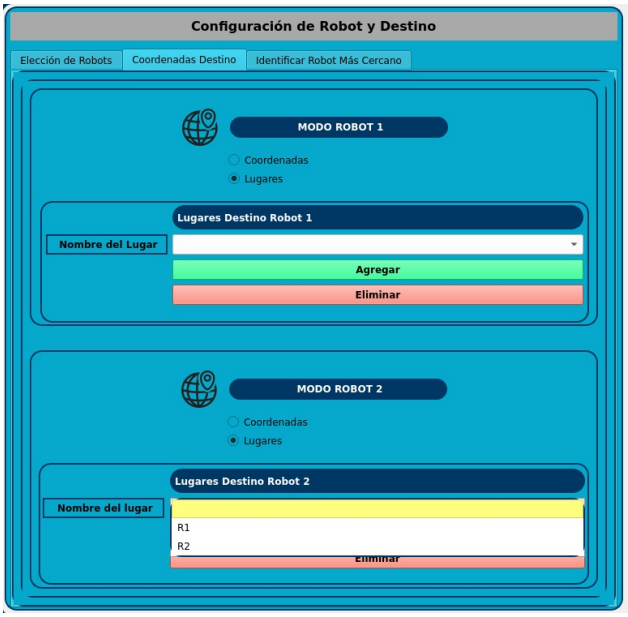

Figura 3.15: Modo Lugares para Todos los Robots (Fuente: Elaboración propia).

La tercera y última funcionalidad, identificación del robot más cercano, permite determinar cuál de los dos robots DaNI está más cerca de un punto de destino ingresado manualmente, como se observa en la Figura 3.16.

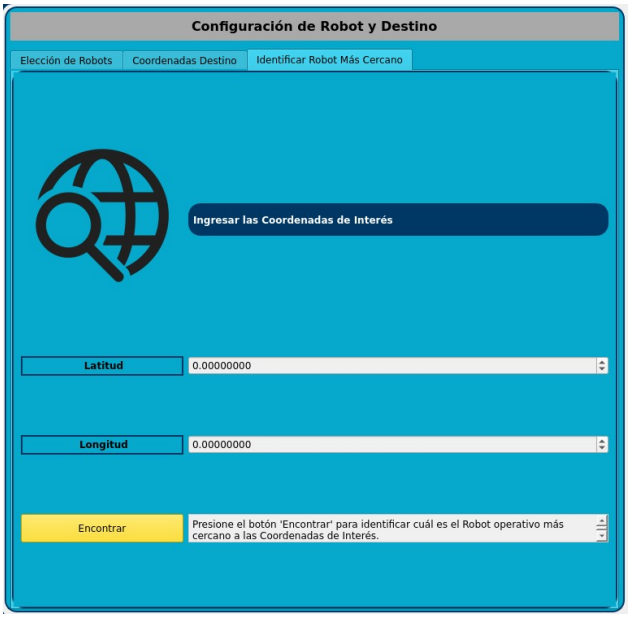

Figura 3.16: Funcionalidad Identificar Robot más Cercano (Fuente: Elaboración propia).

Esta funcionalidad utiliza la fórmula de Haversine, una fórmula trigonométrica que calcula la distancia entre dos puntos en la superficie de una esfera. Aunque la Tierra tiene una forma esferoidal, esta fórmula es ampliamente empleada por su precisión en distancias cortas [54]. La ecuación de Haversine se presenta en la ecuación (3.1).

$$d = 2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\varphi}{2}\right) + \cos(\varphi_1) \cdot \cos(\varphi_2) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right) \tag{3.1}$$

donde:

- d es la distancia entre los dos puntos (en el mismo sistema de unidades que el radio r).
- r es el radio de la Tierra, aproximadamente 6.371 km.
- φ_1 y φ_2 son las latitudes de los puntos 1 y 2 en radianes.
- λ_1 y λ_2 son las longitudes de los puntos 1 y 2 en radianes.
- $\Delta\varphi = \varphi_2 - \varphi_1$ es la diferencia de latitudes.
- $\Delta\lambda = \lambda_2 - \lambda_1$ es la diferencia de longitudes.

- **Navegación:** Este segundo módulo principal está enfocado en iniciar o detener la navegación autónoma de los robots DaNI seleccionados. Para ello, se incluyen dos funcionalidades clave: la opción de iniciar la navegación y la opción de detenerla.

  La primera funcionalidad, denominada *Navegar*, cuenta con dos submódulos. El primero, *Ir a Coordenadas Destino*, dispone de un botón denominado *Ir*, cuyo objetivo es activar la navegación autónoma de los robots seleccionados y dispongan de un punto de destino, como se muestra en la Figura 3.17.

  Este submódulo utiliza el método o slot *send_ data*, un algoritmo creado y permite enviar las coordenadas destino al nodo *gps_ waypoint* correspondiente a cada robot, mediante el servicio *LaunchGoal* de cada robot seleccionado. Dicho nodo recibe las coordenadas de destino y activa el servidor de acciones, el cual emplea esta meta para realizar la navegación. Cabe destacar que el slot *send_ data* hace uso de una *subclase* llamada *GoalThread*, que hereda las características de la clase *QThread*, permitiendo utilizar métodos que gestionan el envío de coordenadas en un hilo separado. Esta estructura asegura que la GUI permanezca responsiva para realizar otras acciones, además de permitir el envío concurrente de coordenadas de navegación a los dos robots DaNI. Por otro lado, se incluye un recuadro blanco designado como el panel de información, que tiene como objetivo mostrar mensajes relevantes sobre el estado del proceso de navegación. Este panel proporciona al usuario información esencial, como la confirmación de la llegada de los robots al destino, advertencias sobre la imposibilidad de alcanzar el objetivo y notificaciones en caso de que no se haya seleccionado un robot o un punto de destino para

Figura 3.17: Funcionalidad Navegar (Fuente: Elaboración propia).

la navegación. Estos y otros casos del panel de control se pueden detallar más precisamente en la máquina de estados dada en la Figura 3.21.

El segundo submódulo, *Visualizar trayectoria en el Mapa*, ilustrado en la Figura 3.17, permite al usuario visualizar la trayectoria de los robots en tiempo real mediante el navegador predeterminado del sistema. Este mapa, desarrollado en HTML, emplea la biblioteca *Leaflet* para obtener una visualización de un mapa mundial, asi como también el uso de sockets para obtener constantemente la información del tópico *$(arg robot_id)/gps*, donde *robot_id* corresponde a cada robot (1 o 2).

En el mapa, el recorrido del robot 1 se representa con una línea roja y el del robot 2 con una línea verde. Además, se incluyen tres botones: uno para limpiar la traza del robot 1, otro para limpiar la traza del robot 2 y un tercero para eliminar ambas trazas al tiempo. Cada robot en el mapa se identifica con un marcador con una leyenda del nombre del mismo, al igual que los puntos de destino. La Figura 3.18 muestra todas las funcionalidades de este mapa. En esta Figura se puede apreciar que los robots y los destinos están situados sobre la Universidad Javeriana de Cali debido a que todas las funcionalidades de la GUI están relacionadas entre sí, lo que quiere decir que en la pestaña de Elección de Robots se seleccionó la opción 'Todos los Robots' y en la pestaña de Coordenadas Destino se ingresaron coordenadas manualmente o se seleccionaron lugares guardados previamente que están localizados en esta zona.

Otro ejemplo donde se puede apreciar el mapa completamente se muestra en la Figura 3.19, donde no se modifican las coordenadas destino, por lo que predeterminadamente quedan en latitud 0,0 y longitud 0,0.

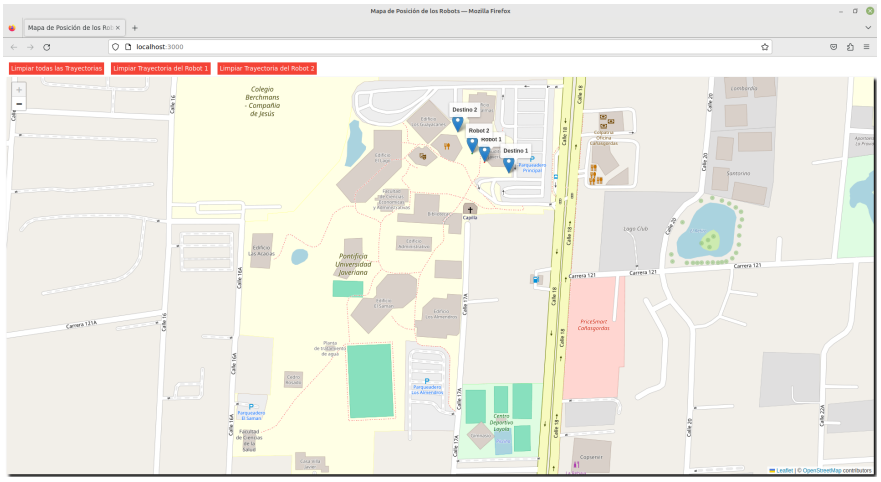

Figura 3.18: Mapa para supervisión de la navegación de los robots (Fuente: Elaboración propia).

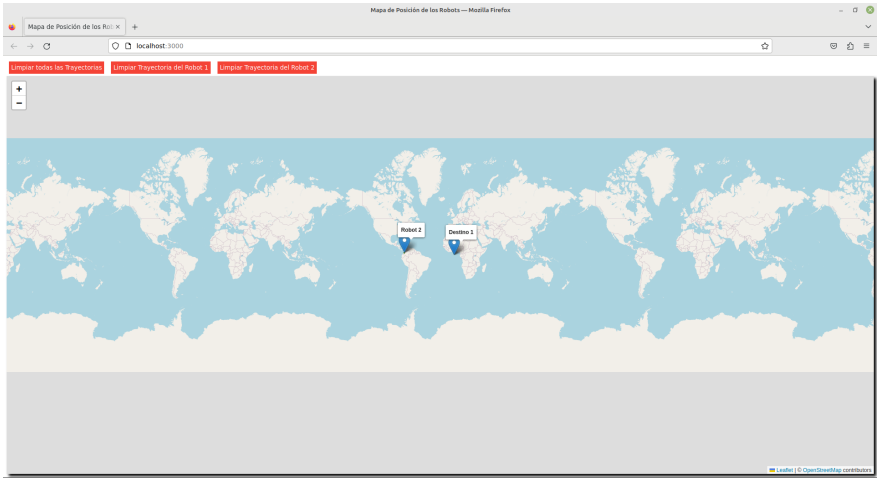

Figura 3.19: Mapa mundial (Fuente: Elaboración propia).

Finalmente, la segunda funcionalidad, *Detener Navegación*, ilustrada en la Figura 3.20, dispone de un ComboBox donde aparecerán automáticamente los robots en navegación hacia un punto de destino. Esta funcionalidad tiene como propósito abortar la acción de alcanzar la meta dada. Para esto se selecciona el robot que se desea intervenir o, en dado caso, se selecciona la opción de 'Todos los Robots' y, posteriormente, se presiona el botón *Detener*.

Figura 3.20: Funcionalidad Detener Navegación (Fuente: Elaboración propia).

La Figura 3.21 muestra la máquina de estados correspondiente a la interfaz gráfica de usuario (GUI), proporcionando una representación clara del flujo de acciones posibles según el estado actual del sistema. Este esquema organiza las transiciones entre estados y las condiciones necesarias para ejecutarlas, facilitando el análisis, la depuración y el mantenimiento del sistema. Además, ofrece una visión estructurada del diseño lógico de la GUI, contribuyendo a su comprensión para posibles cambios futuros.

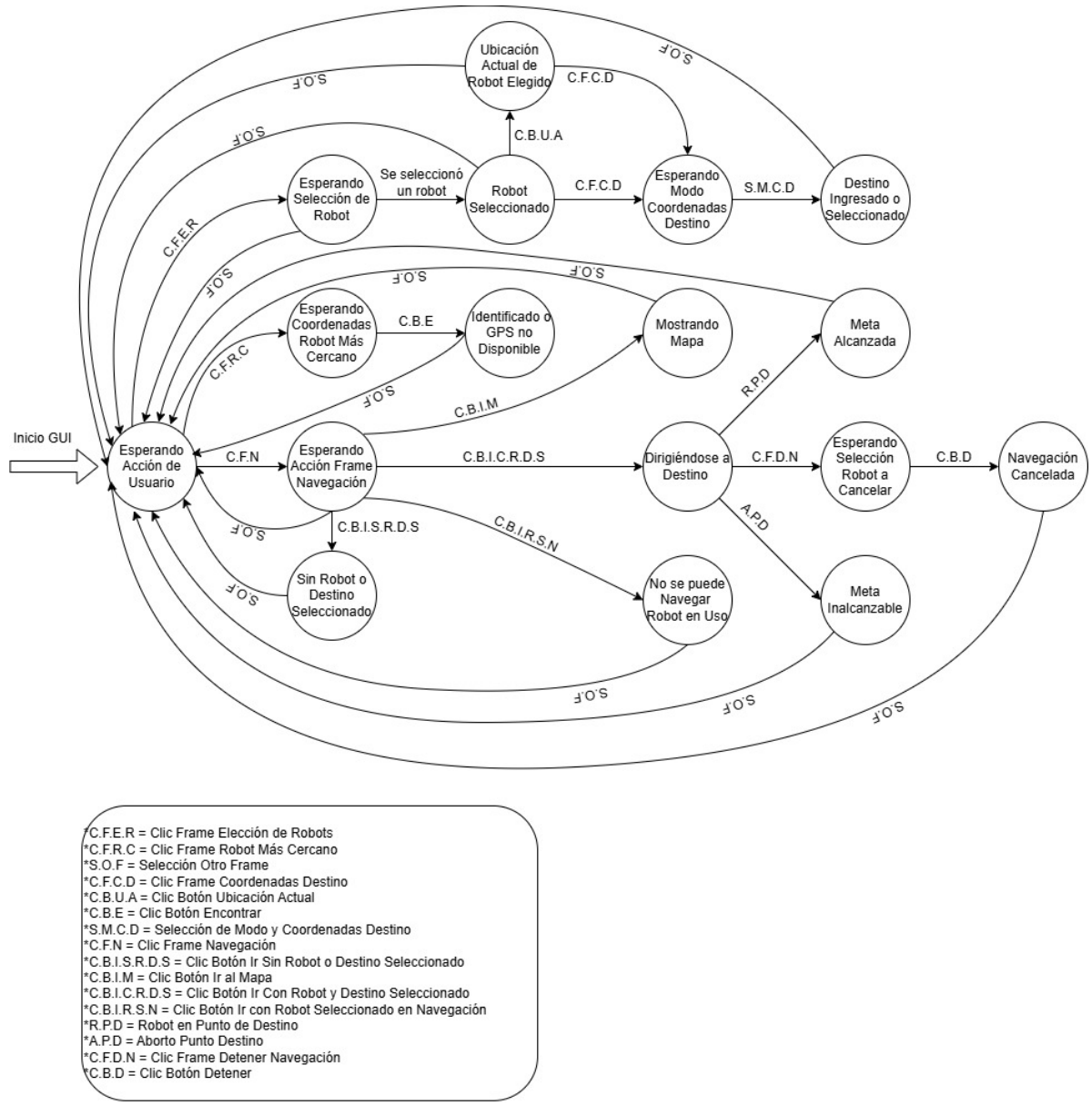

Figura 3.21: Máquina de estados de la GUI (Fuente: Elaboración propia).

3.4.3. Sistema de comunicación de los robots DaNI

El sistema de comunicación multi-robot enfocado en la navegación autónoma conjunta, se desarrolló fundamentado en la biblioteca *fkie_ multimaster*, en su versión *melodic-v1.2.4*. Este paquete proporciona una solución práctica para establecer una red multimaster en un entorno ROS, lo que simplifica la comunicación entre varios dispositivos que comparten la misma red. Este desarrollo se

integra con la arquitectura general de la GUI en el computador central, permitiendo una gestión centralizada y eficiente del sistema. La facilidad de configuración de *fkie_multimaster* es una de sus principales ventajas, ya que, según se indica en la sección 3.3.3.1, su implementación requiere de modificaciones mínimas y su lanzamiento se ejecuta con tan solo dos archivos *.launch*: uno para el descubrimiento de los master en la red y otro para su sincronización. El desarrollo del sistema de comunicación, junto con sus distintos componentes, se organizó en dos etapas fundamentales:

Etapa 1: Adaptación en cada robot DaNI

En esta primera etapa, se enfocó en integrar el paquete *fkie_multimaster* en el proyecto de navegación autónoma de cada robot DaNI. Esto se logró mediante la inclusión del paquete como parte del proyecto individual de cada robot, realizándose una copia del mismo y su posterior compilación en el entorno ROS de cada robot. Esta inclusión garantiza que cada robot pueda participar en la red multimaster, esencial para la sincronización de sus nodos, tópicos y servicios.

Además, se llevaron a cabo modificaciones específicas en el *script gps_waypoint.cpp*, cuya función es recibir coordenadas de destino y enviar el robot hacia dichas ubicaciones. Previamente, el destino se especificaba en un archivo de texto local, pero ahora se implementó un nuevo servicio llamado *LaunchGoal*. Este servicio es fundamental para la comunicación con la GUI del sistema, ya que permite que *gps_waypoint.cpp* reciba las coordenadas de destino enviadas desde un cliente de la GUI. En esta configuración, *gps_waypoint.cpp* actúa como el servidor del servicio *LaunchGoal*, mientras que el cliente se encuentra en *manager.py*, el script principal de la GUI. De esta manera, cuando la GUI envía una solicitud con las coordenadas deseadas, el servidor activa la función que asigna dichas coordenadas como meta para el cliente del servicio de acción en ROS, quien se encarga de iniciar el desplazamiento hacia el destino.

En cuanto a la cancelación de la navegación, se implementó un servicio denominado *CancelGoal*, cuyo servidor se inicializa junto con el nodo *cancel* de cada robot. Este servicio permite interrumpir la ejecución de la tarea de navegación en curso. El mecanismo se basa en un callback que, al ser invocado por una solicitud, ejecuta un proceso en segundo plano. Específicamente, este callback abre una línea de comandos y ejecuta el comando *rostopic pub /move_base/cancel actionlib_msgs/GoalID -- {}*. Este comando se encarga de enviar un mensaje de cancelación a la acción de navegación en curso, lo que provoca que el robot detenga la navegación que se encuentra ejecutando en ese momento. La implementación de este servicio proporciona una forma efectiva de interrumpir la navegación sin necesidad de que el usuario interactúe directamente con el sistema de navegación.

Otra adaptación fundamental en esta etapa fue la creación de una variable de entorno llamada *ROBOT_ID*, que se configura en el archivo *~/.bashrc* y se ejecuta automáticamente cada vez que se abre una nueva terminal. Esta variable permite identificar de forma única a cada robot en la red multimaster (con valores 1 o 2 según corresponda a cada robot).

Por otro lado, para asegurar que el sistema de comunicación y navegación funcione de manera íntegra, el archivo de lanzamiento *with_map_move_base* fue modificado en cinco aspectos clave:

- **Definición del argumento *robot_id*:** En el encabezado del archivo *with_map_move_base* se define el argumento *<arg name="robot_id" default="$(env ROBOT_ID)" />*, donde *robot_id* representa el identificador del robot, y *ROBOT_ID* corresponde al valor asignado a la variable de entorno configurada previamente en el archivo *~/.bashrc*. Esta variable toma los valores 1 o 2, según el robot en cuestión. Cabe resaltar que posteriormente, el argumento *robot_id* es empleado en los item dos, tres y cuatro que se observa a continuación.
- **Incorporación de *send_goals.launch*:** Se incorporó el código XML correspondiente al archivo de lanzamiento *send_goals.launch*, el cual permite desplegar el robot hacia una meta predefinida inicializando el nodo *gps_waypoint*, como se ilustra en la Figura 3.22. Esta integración tiene como objetivo simplificar el proceso de inicialización al consolidar varias etapas en un solo comando. Específicamente, se logra iniciar simultáneamente la odometría, el entorno de visualización RViz y el nodo que crea y gestiona el cliente de acción de ROS responsable del desplazamiento del robot hacia el destino establecido.

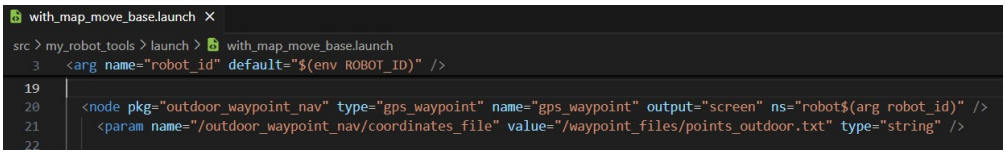

Figura 3.22: Código XML de lanzamiento del nodo *gps_waypoint* (Fuente: Elaboración propia).

En la configuración previa, este proceso requería dos comandos separados. En primer lugar, se utilizaba *roslaunch my_robot_tools full_setup.launch* para inicializar la odometría y el entorno RViz. Posteriormente, se ejecutaba *roslaunch outdoor_waypoint_nav send_goals.launch*, que activaba el nodo *gps_waypoint* encargado de establecer la meta y movilizar el robot mediante el cliente de acción de ROS.

- **Incorporación del nodo *cancel*:** Se añadió código encargado de inicializar el nodo que permite la cancelación de una meta en curso, esto con el fin de que sea inicializado también al momento de ejecutar el archivo de lanzamiento y evitar así el tener que realizar procedimientos extras. El código se incorporó como se muestra en la Figura 3.23.

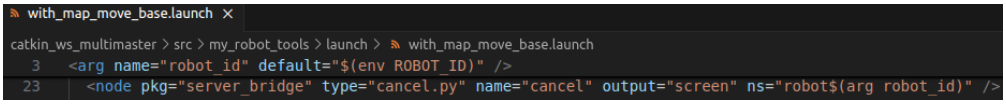

Figura 3.23: Código XML de lanzamiento del nodo *cancel* (Fuente: Elaboración propia).

- **Implementación del nodo repetidor:** Se implementó el código XML para la creación de un nodo repetidor mediante la herramienta *relay*, proporcionada por el paquete *topic_tools*, el cual está incluido por defecto en ROS. Este nodo se configuró con el propósito de duplicar el tópico */fix*, que recibe la información de latitud y longitud obtenida desde el GPS. Tal configuración permite que los datos originales del GPS sean reenviados al tópico específico

/robot$(arg robot_id)/gps. Por otro lado, su uso se detalla en la sección 3.4.2, en el ítem relacionado con la Configuración de Robot y Destino, y su código se observa en la Figura 3.24.

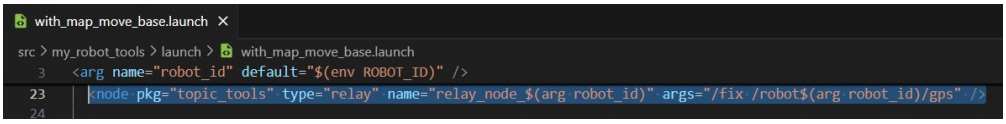

Figura 3.24: Código XML para la duplicación del tópico */fix* mediante el nodo *relay* (Fuente: Elaboración propia).

- **Incorporación de los archivos de lanzamiento *fkie_multimaster*:** El último ajuste realizado en el archivo de lanzamiento consistió en la incorporación de un bloque de código XML como se muestra en la Figura 3.25 que permite la inicialización de los nodos *master_discovery* y *master_sync*, ambos fundamentales para establecer y mantener la comunicación multimaster entre los robots DaNI y el computador que gestiona la interfaz gráfica de usuario (GUI). El nodo *master_discovery* se encarga de detectar automáticamente los demás maestros ROS presentes en la red, mientras que el nodo *master_sync* facilita la sincronización de sus nodos, tópicos y servicios, asegurando una comunicación coherente y eficiente entre los distintos dispositivos.

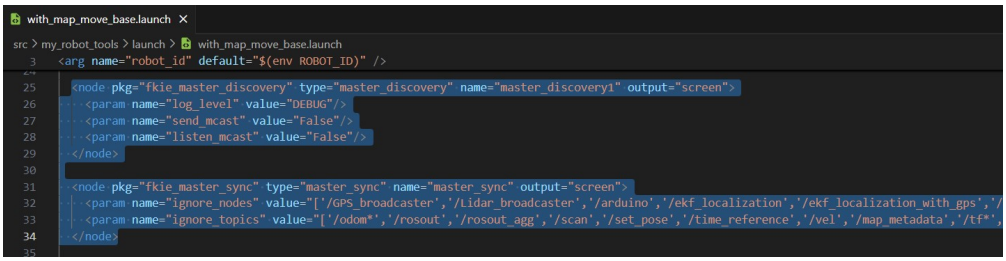

Figura 3.25: Código XML para lanzamiento de los nodos de descubrimiento y sincronización (Fuente: Elaboración propia).

Las modificaciones introducidas en el archivo *with_map_move_base* permiten unificar las funcionalidades relacionadas con la comunicación y la navegación en un único archivo de lanzamiento denominado *full_setup*. Esta integración no solo simplifica la estructura de los archivos de configuración, sino que también optimiza significativamente el proceso de inicio del sistema. Al consolidar las configuraciones necesarias en un único archivo, se elimina la necesidad de ejecutar múltiples comandos para inicializar las distintas partes del sistema.

Finalmente, la integración del paquete *fkie_multimaster*, en conjunto con las modificaciones implementadas en los archivos de configuración correspondientes, fue fundamental para dotar a cada robot de las capacidades necesarias para habilitar la comunicación en un entorno multimaster. Esto incluyó no solo la configuración de parámetros clave para garantizar una sincronización eficiente

entre los nodos de los robots, sino también el establecimiento de un canal de comunicación directo con la interfaz gráfica. Esta interfaz, diseñada para centralizar la gestión del sistema, se convirtió en un elemento clave para recibir información en tiempo real de cada robot y emitir comandos de manera eficaz, facilitando así la coordinación general del sistema.

La herramienta *rqt_graph* fue empleada para generar los grafos del sistema que permiten analizar y comprender mejor las conexiones y relaciones establecidas entre nodos y tópicos. Estas representaciones gráficas, mostradas en las Figuras 3.26 y 3.27, ilustran la arquitectura del sistema para el robot 1 y el robot 2, respectivamente.

Etapa 2: Configuración en el computador para la GUI

La última etapa del desarrollo se centró en la adaptación del computador que aloja la GUI. Aquí, *fkie_multimaster* también fue integrado en el proyecto ROS del computador, siguiendo el mismo procedimiento de copia y compilación del paquete. En este entorno, se configuraron los servicios ROS *LaunchGoal* y *CancelGoal*, junto con sus respectivos clientes en *manager.py*, permitiendo así que la GUI gestione el envío de coordenadas y el control del movimiento del robot de manera remota.

Para lograr una comunicación fluida entre el servidor local del mapa y la GUI, se estableció una conexión mediante sockets. Esto permite que el cliente en *manager.py* se comunique con el servidor del mapa (escrito en HTML) para enviar y recibir información en tiempo real. Esta configuración mejora significativamente la interactividad de la GUI, permitiendo un control y seguimiento preciso de la navegación autónoma de los robots DaNI.

El grafo correspondiente al computador central, encargado de gestionar el control de los robots DaNI a través de la interfaz gráfica, se visualiza en la Figura 3.28. Este grafo presenta una estructura considerablemente más pequeña en comparación con los grafos de los robots 1 y 2. Esta diferencia se debe a que, mediante la configuración del paquete *fkie_multimaster*, se especificó que no se compartieran la mayoría de nodos y tópicos, ya que no eran necesarios para el envío de las metas a los robots. En consecuencia, solo se visualizan los nodos GPS duplicados de cada robot por medio de *relay*, así como los nodos involucrados en la comunicación multimaster.

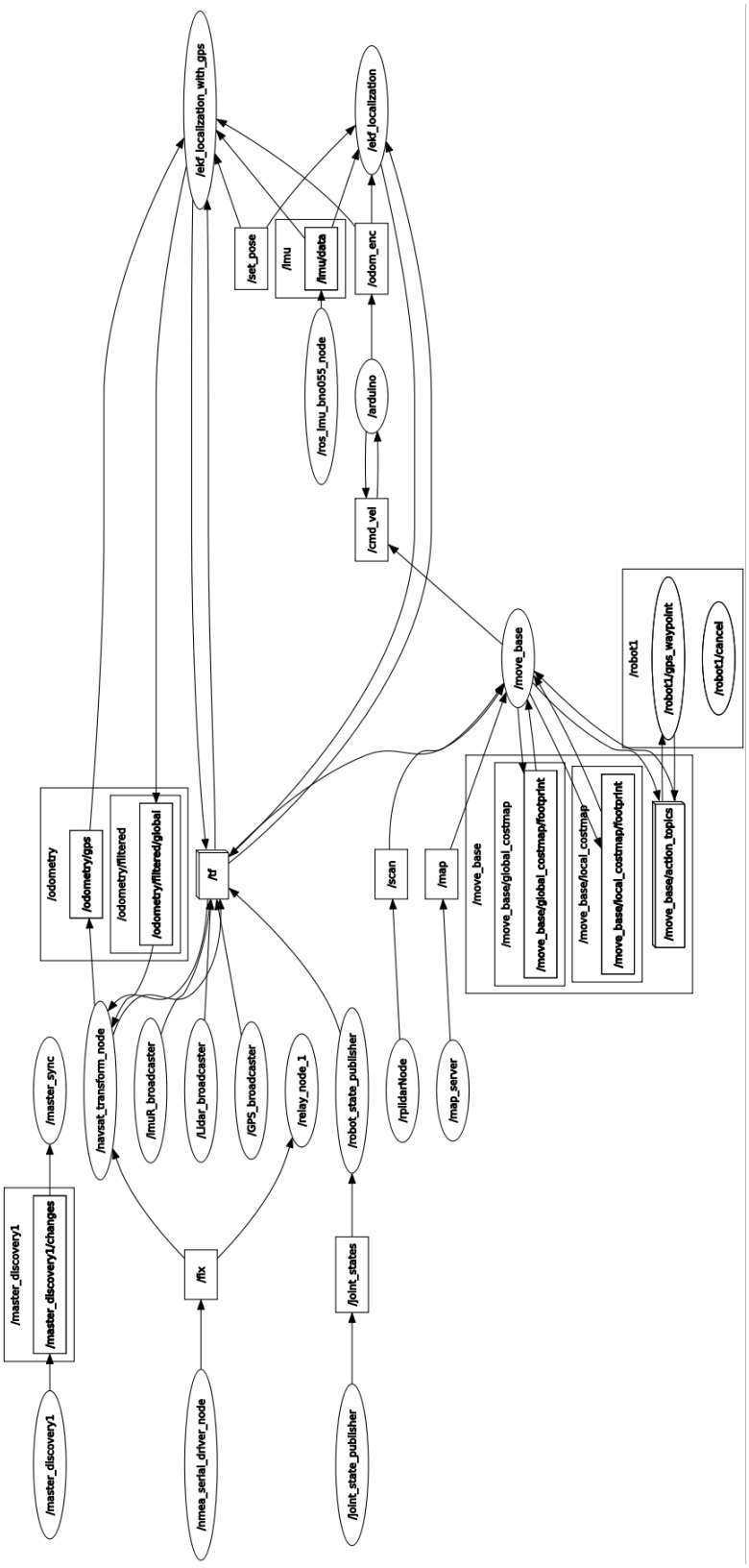

Figura 3.26: Grafo de nodos y tópicos del robot 1 (Fuente: Elaboración propia).

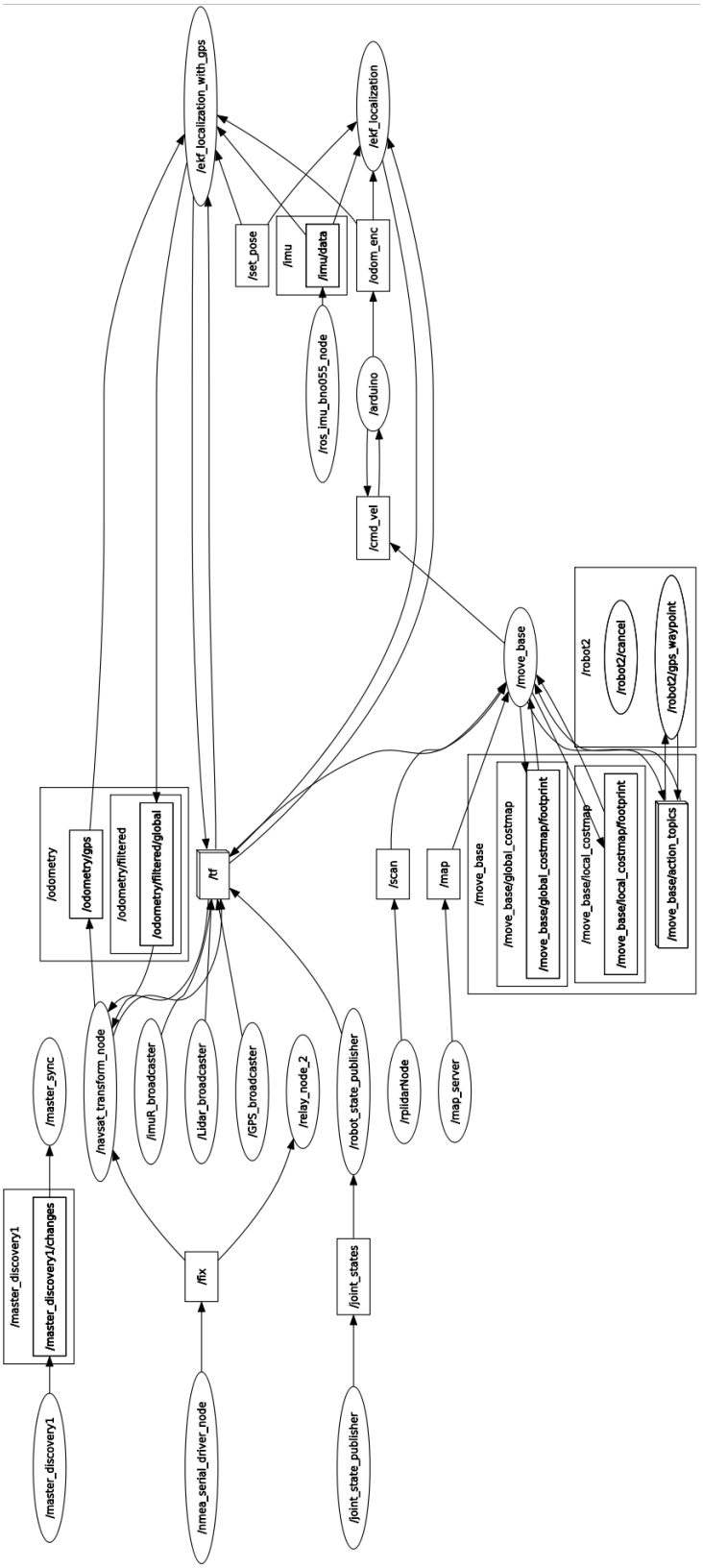

Figura 3.27: Grafo de nodos y tópicos del robot 2 (Fuente: Elaboración propia).

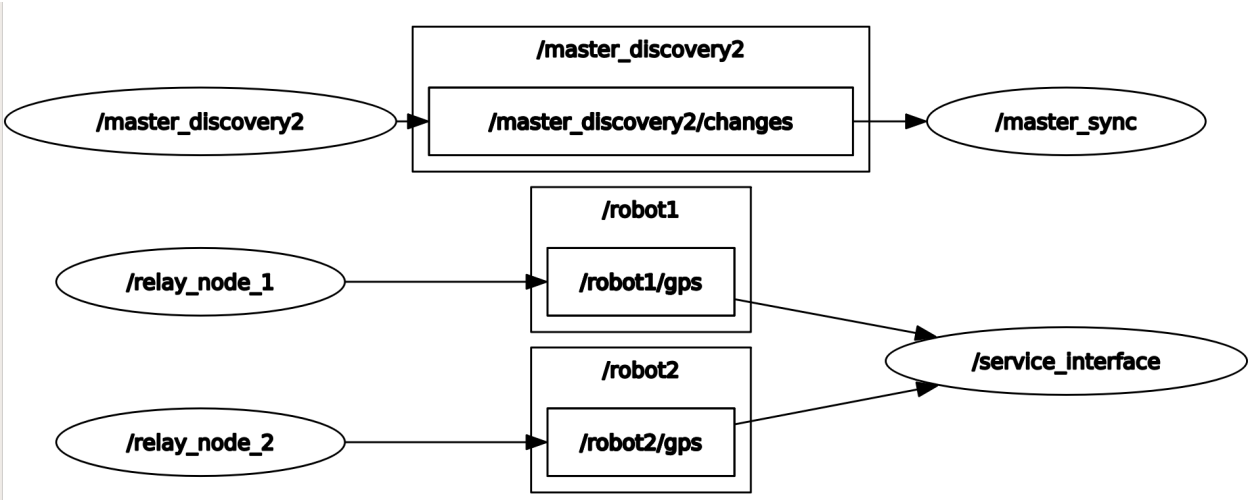

Figura 3.28: Grafo de nodos y tópicos del computador central (Fuente: Elaboración propia).

Capítulo 4

Resultados y Discusión

4.1. Resumen de hardware y software construido

En el ámbito del hardware, el proyecto culminó exitosamente con la construcción y entrega de un segundo robot DaNI, diseñado y actualizado para tareas de navegación autónoma. Este desarrollo implicó el uso de componentes adquiridos y elementos disponibles en el laboratorio de la universidad. La Tabla 4.10 proporciona un detalle de los componentes utilizados, clasificándolos entre aquellos comprados y los suministrados por la institución. Además, en la Figura 4.1, se comparan los dos robots: a la izquierda, el modelo original, y a la derecha, la nueva versión construida utilizando los componentes disponibles.

Componente	**Comprado**	**Precio (COP)**
NiMH de 12V 3000mAh Modern Robotics	Si	315.000
Power bank QC 3.0 EMIGVELA	Si	240.000
Puente H L298N	Si	25.000
Encoders Motor Pitsco 39530	No	N/A
GPS VK-162 G-Mouse	No	N/A
IMU BNO055	Si	274.000
LiDAR A1	No	N/A
Arduino Uno	No	N/A
Motores Pitsco Educación 39530 Tetrix Max DC	No	N/A
Jetson Nano (4 GB RAM)	Si	717.000
Modulo Wi-Fi Jetson Nano (AC8265)	Si	118.000
Ventilador Jetson Nano	Si	12.000
Hub Ugreen USB 3.0	Si	71.000
Total		1.772.000

Tabla 4.10: Resumen de componentes de hardware y precios.

En lo que respecta al desarrollo de software, se implementó un sistema de comunicación entre los robots DaNI, basado en el paquete *fkie_multimaster*. Este sistema habilita la interconexión y sincronización en una misma red de nodos, tópicos y servicios, optimizando así la navegación autónoma conjunta de los robots. Este sistema permite una administración centralizada a través de una interfaz gráfica de usuario (GUI) instalada en un equipo desde donde se realiza el control.

Figura 4.1: Robots DaNI actualizados (Fuente: Elaboración propia).

Cabe destacar que el proyecto ROS, actualizado con el sistema de comunicación descrito, ha sido documentado y puesto a disposición pública en un repositorio de GitHub. El enlace para acceder al repositorio se encuentra aquí y en [55], para su consulta y posible extensión por parte de futuros desarrollos.

Adicionalmente, como parte de las entregas de software, se encapsuló el proyecto ROS en una imagen de Docker, disponible en Docker Hub. Este método garantiza una mayor flexibilidad, portabilidad y facilidad para futuras implementaciones, permitiendo que el proyecto pueda ejecutarse en cualquier máquina con sistema operativo Linux compatible con Docker. El enlace a la imagen en Docker Hub también está documentado aquí y en [56].

La GUI desarrollada incluye una serie de funcionalidades diseñadas para maximizar la capacidad de control y monitoreo de los robots DaNI. Entre estas, se destacan:

- Control en tiempo real de la navegación, tanto conjunta como individual, de los robots.
- Opciones para detener los robots en cualquier momento durante la operación.
- Visualización en tiempo real de la ubicación de los robots, incluyendo un mapa dinámico que muestra las coordenadas actuales y los puntos de destino.
- Determinación del robot más cercano a un punto específico, lo cual optimiza la asignación de tareas.

- Almacenamiento local de coordenadas de destino bajo pseudónimos, lo que facilita su reutilización en futuras operaciones.

Finalmente, se presenta en la Tabla 4.11 un resumen consolidado del software como paquetes, programas y herramientas utilizadas para el desarrollo del sistema de comunicación, incluyendo sus roles específicos en el proyecto.

Paquete o herramienta	**Funcionalidad**
fkie_multimaster	Permite la comunicación, sincronización y descubrimiento entre múltiples maestros ROS de los robots DaNI y el sistema centralizado que alberga la interfaz gráfica de usuario (GUI).
fkie_master_discovery	Es un paquete encargado de realizar el descubrimiento de los maestros (masters) en el entorno ROS.
fkie_master_sync	Es un paquete diseñado para sincronizar los nodos, tópicos y servicios entre los maestros (masters) detectados.
Docker	Facilita el empaquetado de la distribución Linux, la versión de ROS, el proyecto y las dependencias de la interfaz gráfica de usuario (GUI), permitiendo su distribución y ejecución eficiente mediante contenedores.
server_bridge	Incluye las dependencias desarrolladas específicamente para los servicios LaunchGoal y CancelGoal.
PyQt5	Facilitó el desarrollo de la interfaz gráfica de usuario (GUI) mediante un enfoque visual y eficiente.
venv	Permite crear un ambiente virtual específico para la versión de Python 3.8 requerida para correr la interfaz desarrollada con PyQt5.
threading	Es una biblioteca de Python que permite la creación y gestión de hilos, facilitando la ejecución concurrente de las tareas de navegación de los robots.
webbrowser	Es una biblioteca de Python que permite abrir el navegador predeterminado, donde se mostrará el mapa correspondiente.
socketio	Es una biblioteca de Python que facilita la comunicación entre el script *manager.py* de la interfaz gráfica de usuario (GUI), que recibe en tiempo real las coordenadas de los robots, y el archivo HTML que desarrolla el mapa. Esta comunicación permite transmitir las coordenadas de los robots al segundo script, asegurando la actualización y visualización precisa en el mapa.
Leaflet	Es una biblioteca de HTML que facilitó la creación de un mapa interactivo, incluyendo marcadores que representan en tiempo real la posición de los robots y los destinos establecidos en la interfaz gráfica de usuario.

Tabla 4.11: Resumen software y funcionalidad.

Este esfuerzo integrado en hardware y software refleja el logro de los objetivos planteados, destacando la capacidad de los robots para operar en conjunto con alta eficiencia y flexibilidad.

4.2. Análisis y validación del sistema de comunicación

En esta sección se presenta el análisis y validación del sistema de comunicación implementado, considerando las variables clave que determinan su desempeño y robustez. Cada una de estas variables se describe detalladamente junto con los procedimientos de prueba realizados para evaluar su impacto en el sistema. A continuación, se enumeran las métricas analizadas:

4.2.1. Latencia de envío de meta y cancelación

Descripción general de las pruebas

La prueba de latencia se centró en medir, en primer lugar, la diferencia temporal entre la emisión de la orden de navegación desde el computador central, que ejecuta la interfaz gráfica de usuario (GUI), y la recepción de dicha orden por el robot correspondiente a través del servicio *LaunchGoal* y, en segundo lugar, la diferencia temporal entre la emisión de la cancelación de la navegación desde el computador central y su respectiva recepción en el robot a través del servicio *CancelGoal*. Cada una de estas pruebas se realiza utilizando la infraestructura de comunicación proporcionada por la red multimaster. Este análisis se llevó a cabo en dos escenarios distintos para cada caso:

- Puesta en marcha de navegación y emisión de solicitud de cancelación únicamente para un robot DaNI.
- Puesta en marcha de navegación y emisión de solicitud de cancelación para ambos robots DaNI.

Para cada uno de los escenarios se realizaron 20 pruebas. En el caso de la puesta en marcha de navegación, se varió la distancia entre el PC central y cada robot desde 1 metro hasta 20 metros, incrementando en pasos de un metro. Esta variación en la distancia no se hizo para el caso de la emisión de solicitud de cancelación debido a que, como se verá más adelante, se identificó que esta distancia no genera ningún tipo de efecto en el funcionamiento del sistema.

Resultados y análisis de las pruebas

Para el cálculo de la diferencia de tiempo en la puesta en marcha de navegación, se emplearon los valores de *timestamp* de *UNIX* correspondientes al momento en que se generaba la solicitud desde la GUI y el momento en que el robot correspondiente la recibía. De esta manera, se obtuvieron las mediciones realizadas para el robot 1 de manera individual, presentadas en la Figura 4.2, así como las mediciones para ambos robots al ejecutar la orden de navegación de manera concurrente, las cuales se muestran en la Figura 4.3.

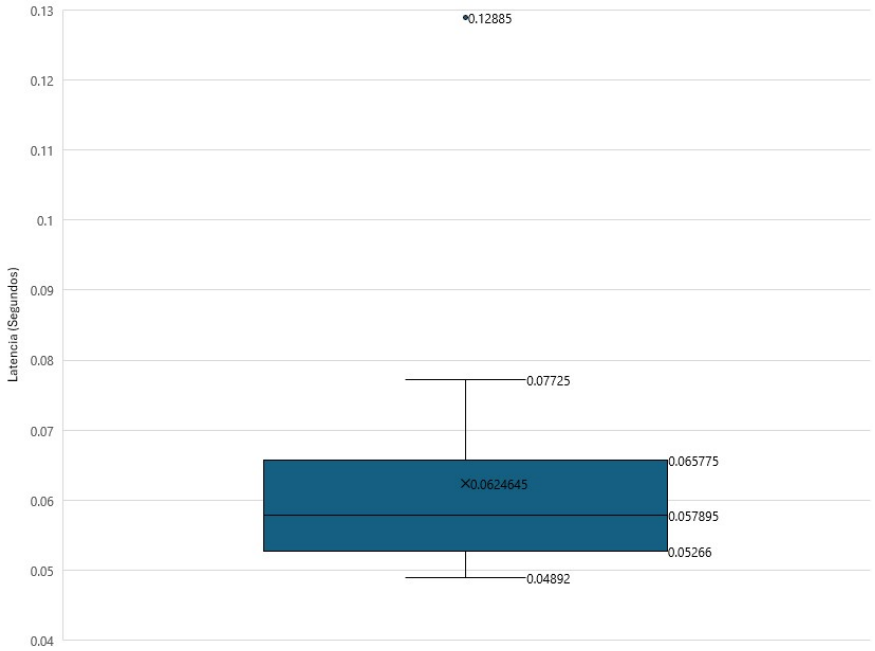

Figura 4.2: Latencia del servicio *LaunchGoal* para el robot 1 (Fuente: Elaboración propia).

Figura 4.3: Latencia del servicio *LaunchGoal* para ambos robots concurrentemente (Fuente: Elaboración propia).

Las pruebas realizadas en los dos escenarios mencionados han demostrado que el sistema responde de manera rápida y eficiente ante las solicitudes generadas a través del servicio *LaunchGoal*. Se observa que la latencia del sistema no se ve significativamente afectada por el aumento de la distancia entre el

robot y la meta, tal como se observa en la Figura 4.4. Esto indica que el sistema es capaz de gestionar las peticiones de navegación de forma constante, con poca afectación de la distancia. Sin embargo, es relevante señalar que, debido a las limitaciones impuestas por los mapas de costo utilizados en el sistema de navegación previamente realizado [9], la distancia máxima permitida entre el robot y la meta es de 15 metros. Si la distancia supera este umbral, el sistema no considerará la meta como válida, lo que impide que el robot la utilice para la navegación autónoma. Esta restricción asegura que las pruebas estén dentro de los límites óptimos para su navegación.

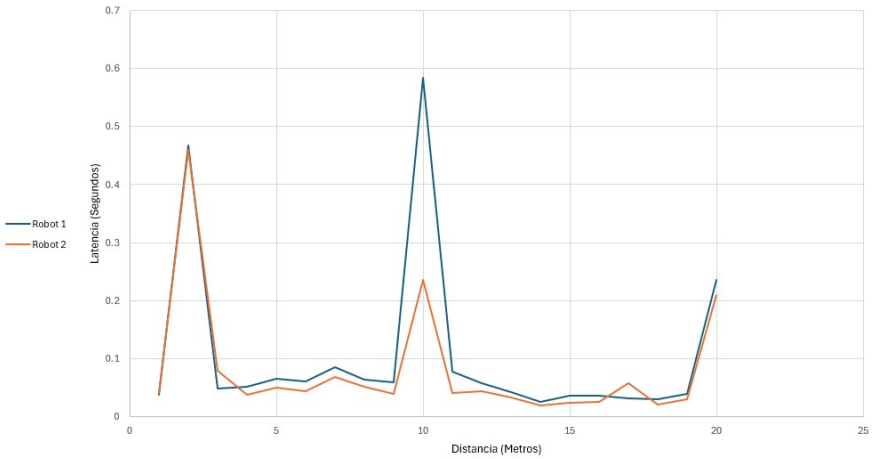

Figura 4.4: Latencia en función de la distancia entre computador y robots (Fuente: Elaboración propia).

Por otro lado, de manera similar se realizó la prueba de latencia para la emisión de la solicitud de cancelación, de manera que se obtuvieron las mediciones para el caso únicamente del robot 1 que se presentan en la Figura 4.5, mientras que para el caso de ambos robots en conjunto arrojó los resultados que se exponen en la Figura 4.6.

Como se puede observar en las dos figuras, en este caso se presentó una diferencia considerable en el tiempo de recepción de la solicitud de cancelación entre el robot 1 y el robot 2. En primer lugar, la mediana del tiempo requerido por el robot 1 se encontró en 2,01 segundos para el caso individual, mientras que para el caso en conjunto la mediana fue de 2,08 segundos. Esto sugiere que la presencia de otro robot no afectó significativamente el tiempo de recepción de la solicitud por parte del robot 1. En contraste, la mediana del tiempo de recepción de la solicitud por parte del robot 2 se situó en 7,3 segundos, lo que representó una diferencia de más de 5 segundos. Teniendo en cuenta que la velocidad máxima alcanzada por un robot es de 0,35 m/s, según lo establecido en el trabajo de grado [9] (página 73), los más de 5 segundos de diferencia corresponderían a una distancia de frenado adicional de más de 1,75 metros debido al tiempo de espera para la detención del robot 1.

La diferencia entre la solicitud de navegación y la de cancelación radica en que la primera hace uso de hilos para ejecutar procesos de manera concurrente, mientras que la segunda se lleva a cabo de forma secuencial. Esto significa que la solicitud debe enviarse de acuerdo con el orden establecido

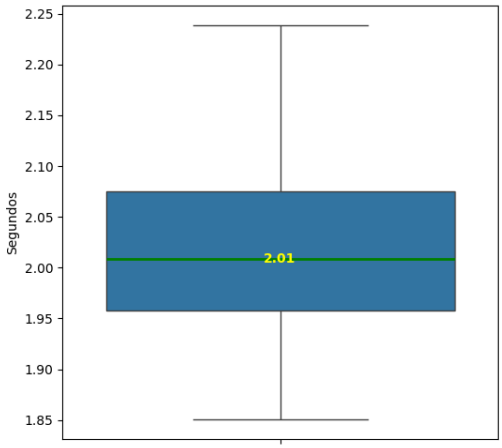

Figura 4.5: Latencia del servicio *CancelGoal* para el robot 1 (Fuente: Elaboración propia).

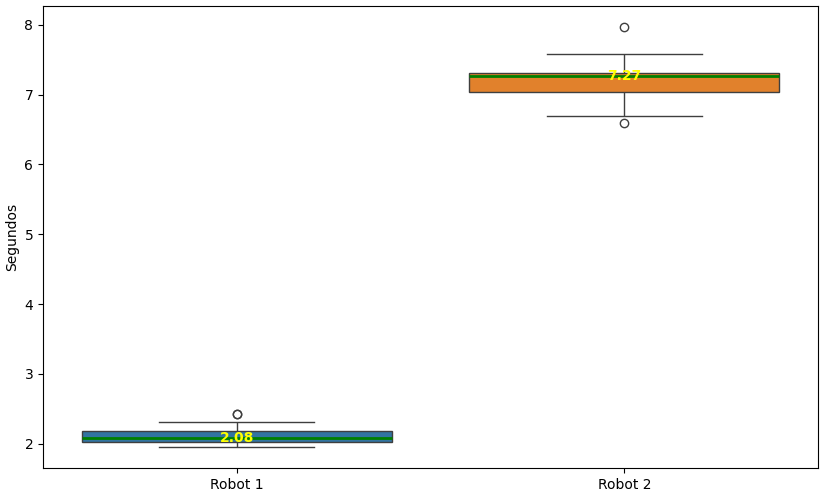

Figura 4.6: Latencia del servicio *CancelGoal* para ambos robots en conjunto (Fuente: Elaboración propia).

en el script de la interfaz gráfica de usuario (GUI), y el sistema debe esperar a que se complete el algoritmo de cancelación para el primer robot antes de enviar la solicitud para el segundo robot.

Esta situación no resulta viable para aplicaciones que requieren acciones rápidas. Mientras el segundo

robot espera a recibir la solicitud, puede ejecutar acciones no deseadas. Además, puede inferirse que, a medida que se integren más robots al sistema, los tiempos de espera aumentarán considerablemente al seguir este método.

Debido a esto, se implementó un algoritmo concurrente mediante hilos para el servicio Cancel-Goal, de manera similar a como se hizo para el servicio LaunchGoal. Con esta implementación se obtuvieron mejores resultados, como se puede observar en la Figura 4.7.

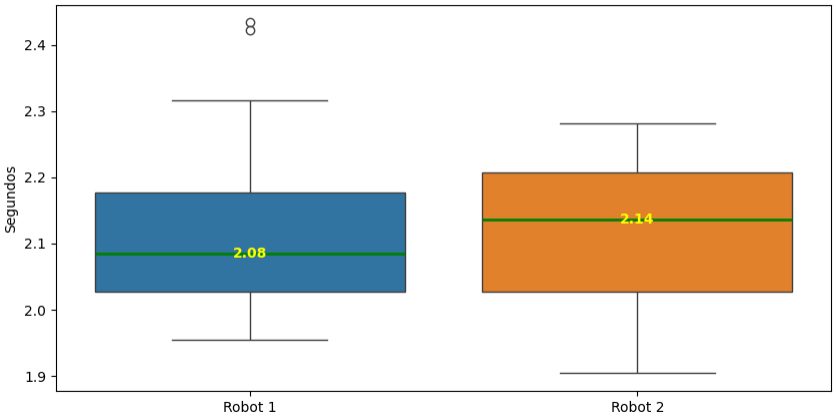

Figura 4.7: Latencia del servicio *CancelGoal* para ambos robots en conjunto, implementando hilos (Fuente: Elaboración propia).

En esta nueva implementación, se obtuvo que la mediana del tiempo de latencia en la cancelación para el robot 1 fue de 2,08 segundos, mientras que para el robot 2 fue de 2,14 segundos, en 20 pruebas realizadas en cada caso de forma similar a lo hecho en el servicio de LaunchGoal. Esto representa una distancia de frenado de 72 cm y 74 cm, respectivamente, lo cual muestra una notable diferencia en comparación con la distancia de frenado del robot 2 en la forma secuencial que se implementó inicialmente.

4.2.2. Latencia simulada

Descripción general de la prueba

Después de realizar la prueba de latencia en condiciones normales, se procedió a implementar una simulación de incremento de latencia en el sistema para analizar la respuesta que se tenía al momento de emitir una orden de navegación y validar qué tan robusto puede ser el sistema. Esta simulación se hizo únicamente para el caso de la navegación con ambos robots a la vez, ya que, como se vio anteriormente, no se presenta una diferencia relevante en la latencia entre emitir una orden de navegación para un solo robot y una orden para ambos. Para poder hacer esto, se necesitó usar una herramienta de control de red en Linux llamada *Traffic Control* (tc), la cual permite emular

retrasos, pérdida de paquetes, jitter y limitación de ancho de banda. En la mayoría de distribuciones de Linux, tc viene preinstalado como parte del paquete *iproute2*. En caso de no venir preinstalado, se puede instalar con:

- *sudo apt install iproute2*

Para introducir la latencia simulada con esta herramienta se usó el siguiente comando:

- *sudo tc qdisc add dev wlo1 root netem delay 25ms*

Aquí se especifica un retraso de 25 ms para los paquetes que viajan por la interfaz de red *wlo1*, que es la interfaz de red que se está utilizando. Además del retraso de 25 ms, también se aplicaron retrasos de 50 ms, 100 ms, 200 ms, 500 ms, 1000 ms, 2500 ms, 5000 ms, 7500 ms, 10000 ms y 100000 ms.

Es importante tener presente que después de introducir el retraso simulado para hacer las pruebas, se debe limpiar las reglas para regresar la red a la normalidad. Esto se hace ejecutando el siguiente comando:

- *sudo tc qdisc del dev wlo1 root*

Resultados y análisis de la prueba

De manera similar a lo hecho en la anterior prueba, el cálculo de la diferencia de tiempo entre la emisión y recepción de la orden de navegación se realizó empleando los valores de *timestamp* de *UNIX*. Los resultados obtenidos se pueden observar en la Figura 4.8

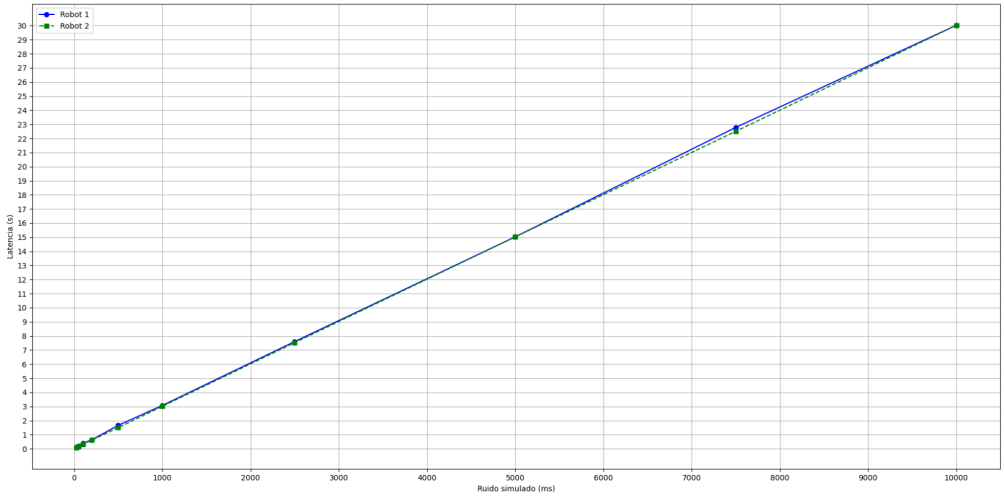

Figura 4.8: Latencia simulada para ambos robots (Fuente: Elaboración propia).

De estos resultados se pueden resaltar los siguientes puntos clave:

- **Tendencia lineal:** A medida que se agregó latencia simulada a la red WLAN con la herramienta Traffic Control, la latencia del servicio LaunchGoal en ambos robots aumentó de manera casi proporcional, indicando una correlación directa entre el aumento en la latencia agregada en la red WLAN y el aumento en la latencia entre el envío del servicio LaunchGoal desde el computador a los robots.
- **Efecto notable en latencias bajas y altas:** Para niveles de ruido bajos (25-200 ms), la latencia es relativamente pequeña y los robots tienen una recepción casi inmediata de la orden. Por otro lado, para niveles altos de ruido (1000 ms o más), las latencias aumentan significativamente, lo cual puede ser crítico en el caso del servicio de CancelGoal, donde cada segundo de latencia extra significa una distancia de frenado adicional de 35 cm. Adicionalmente, al inducir una latencia simulada extremadamente alta (en este caso de 100000 ms) el sistema falla en su operación, como se puede apreciar en el mensaje de error de la Figura 4.9. Este mensaje de error indica que el sistema intentó realizar una llamada a un servicio ROS pero falló debido a un tiempo de espera excedido (*timeout*). Esto ocurre cuando el cliente que realiza la solicitud no puede establecer conexión con el servicio en el tiempo definido como máximo. En caso de que esto sucediera, los robots permanecerían en el estado en el que se encuentran, ya sea en reposo o en medio de una orden de navegación, ya que ninguna petición de servicio sería recibida.

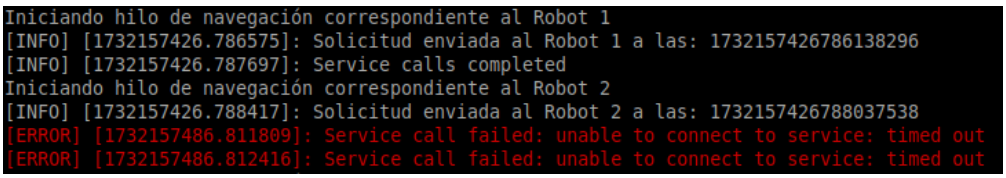

Figura 4.9: Incapacidad de conectar al servicio *LaunchGoal* por *timeout* (Fuente: Elaboración propia).

En aplicaciones reales, estos escenarios donde se agregan latencias de magnitudes altas a la red son muy poco probables, pero posibles en situaciones en las que se presentan interferencias de otros dispositivos, congestión en la red, distancias y obstáculos físicos o saturación en los canales.

4.2.3. Ancho de banda

Descripción general de la prueba

El consumo de ancho de banda de la red de parte de cada robot en la red multimaster constituye un parámetro crucial para evaluar la capacidad de la red, ya que determina uno de los límites prácticos acerca de la cantidad de robots que pueden operar simultáneamente sin sobrecargar la red WLAN. Para este análisis, se empleó la herramienta *iftop*, la cual permite el monitoreo en tiempo real del tráfico de red, tanto entrante (RX) como saliente (TX), de los dispositivos conectados, identificándolos mediante sus respectivas direcciones IP. El consumo promedio de ancho de banda para cada dispositivo se calculó utilizando la Ecuación (4.1):

$$\text{Ancho de banda consumido promedio} = TX \text{ (saliente)} + RX \text{ (entrante)} \tag{4.1}$$

Con el objetivo de realizar un análisis completo, se llevaron a cabo evaluaciones en tres escenarios específicos, tanto antes como después de ejecutar el servicio *LaunchGoal* en cada caso:

- Evaluación únicamente para el robot 1.
- Evaluación únicamente para el robot 2.
- Evaluación para ambos robots operando simultáneamente.

Resultados y análisis de la prueba

En el primer caso, correspondiente al robot 1 en operación individual en estado de reposo, se observó un consumo de ancho de banda promedio de 25,93 kbps antes de la ejecución del servicio *LaunchGoal*, como se muestra en la Figura 4.10, donde la columna resaltada representa el consumo promedio de los últimos 2 segundos para cada dispositivo conectado a la red.

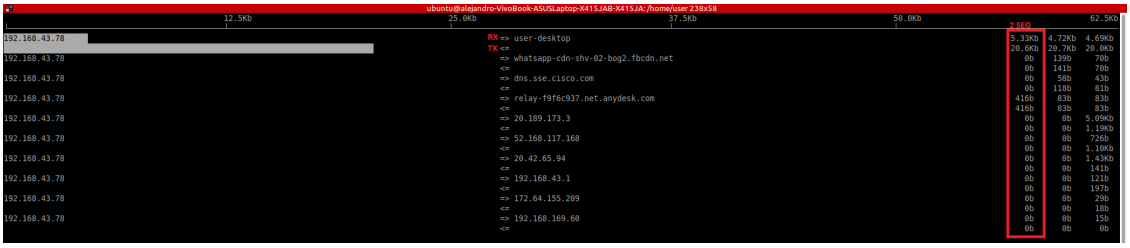

Figura 4.10: Tráfico de red del robot 1 en reposo (Fuente: Elaboración propia).

Posteriormente, se realizaron varias pruebas consecutivas de medición del consumo de ancho de banda con la misma herramienta, pero con el envío de la meta por medio del servicio *LaunchGoal*. El consumo del ancho de banda aumentó, como se observa en la Figura 4.11 que representa la información de las pruebas en un diagrama de caja y bigotes, donde el pico máximo es de 59,10 kbps y la mediana es de 55,70 kbps.

En el segundo caso, correspondiente al robot 2, se obtuvo un consumo promedio de ancho de banda de 27,89 kbps antes de la ejecución del servicio *LaunchGoal*, como se ilustra en la Figura 4.12.

Tras realizar varias pruebas consecutivas ejecutando la navegación tal como se hizo con el robot 1, se registró un incremento en el consumo de ancho de banda promedio, como se muestra en la Figura 4.13, donde el pico máximo es de 89,30 kbps y la mediana es de 64,50 kbps.

Finalmente, en el tercer caso, se analizaron los consumos de ancho de banda promedio de ambos robots operando simultáneamente. En la Figura 4.14 se representan los datos en diagramas de cajas y bigotes para ambos robots de cinco pruebas realizadas consecutivamente en el caso de navegación conjunta. Durante estas pruebas se observó un pico máximo de 72 kbps para el robot 1

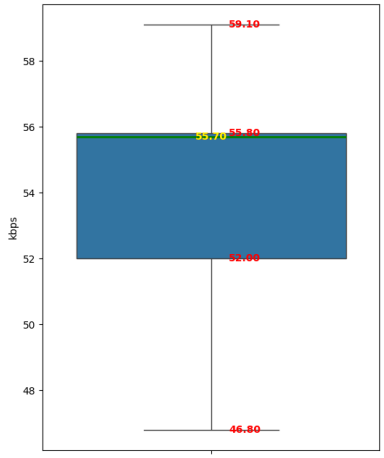

Figura 4.11: Ancho de banda consumido promedio del robot 1 al recibir una orden de navegación (Fuente: Elaboración propia).

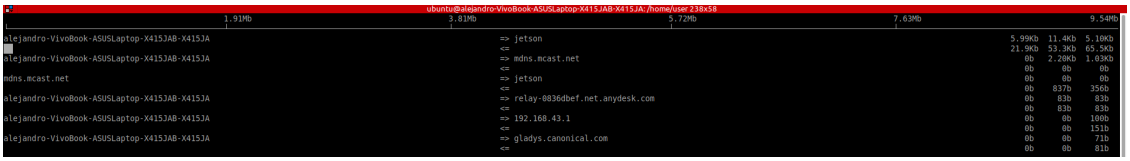

Figura 4.12: Tráfico de red del robot 2 en reposo (Fuente: Elaboración propia).

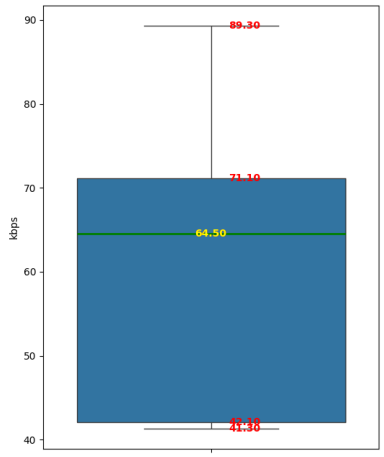

Figura 4.13: Ancho de banda consumido promedio del robot 2 al recibir una orden de navegación (Fuente: Elaboración propia).

y de 67 kbps para el robot 2, mientras que las medianas obtenidas fueron 62,30 kbps y 62,60 kbps, respectivamente.

Esto indica que el consumo de ancho de banda presenta un comportamiento similar tanto para cuando los robots navegan de forma individual como para cuando operan de manera conjunta, lo que quiere decir que el consumo de ancho de banda promedio para cada robot es independiente de la cantidad de robots que estén presentes en el sistema de comunicación.

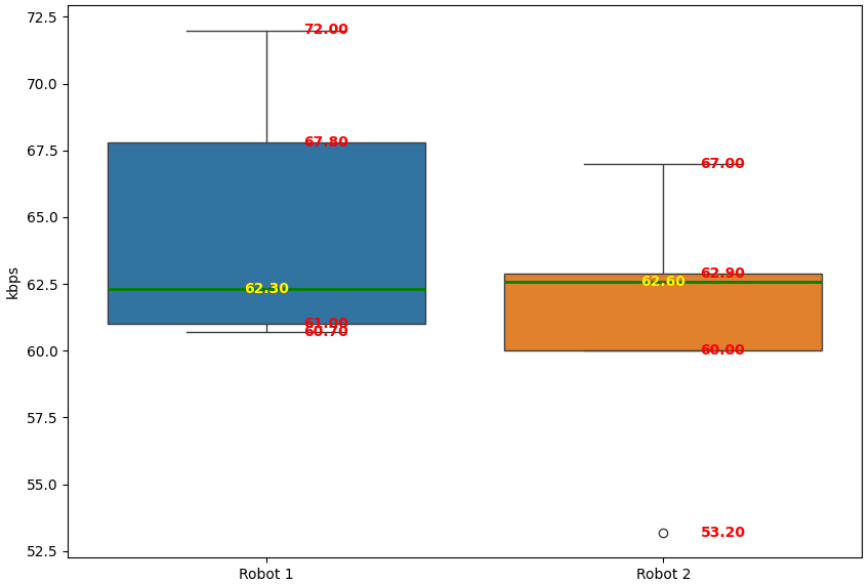

Figura 4.14: Ancho de banda consumido promedio de ambos robots al recibir una orden de navegación conjunta (Fuente: Elaboración propia).

4.3. Análisis y validación del sistema de navegación autónomo en conjunto

En esta sección se lleva a cabo el análisis y la validación del comportamiento del sistema de navegación autónomo implementado en los dos robots DaNI, mediante su integración y funcionamiento en conjunto con el sistema de comunicación y la interfaz gráfica de usuario (GUI). Para ello, se realizarán tres pruebas: una para el robot DaNI 1, otra para el robot DaNI 2, y finalmente, una tercera prueba que implicará la operación simultánea de ambos robots. Cabe resaltar que los resultados obtenidos de las pruebas no se pueden observar de una manera más clara debido a las limitaciones en el aumento del zoom con la herramienta Leaflet, en la que se puede lograr un zoom máximo correspondiente a x18, que es como se muestra en las figuras. Por otro lado, se debe tener en cuenta que, a partir de la planeación de la trayectoria, tal como se describe en la sección 2.4.2.1, se generan las velocidades deseadas para las ruedas. Dichas velocidades son alcanzadas por el control de movimiento, descrito en [9] (página 49), mediante un controlador PID que utiliza como setpoint las velocidades generadas y como realimentación los datos dados por los encoders.

4.3.1. Navegación Robot DaNI 1

Descripción general de la prueba

Para esta prueba, se selecciona únicamente el robot 1 desde la interfaz gráfica de usuario (GUI), asignándole un punto de destino al cual debe dirigirse. A través del mapa de Leaflet, se verificará la posición de los marcadores correspondientes al punto de origen del robot y al destino seleccionado. Posteriormente, se procederá a verificar el trayecto seguido por el robot y la ubicación final alcanzada.

Resultados y análisis de la prueba

En la Figura 4.15 se muestra la posición inicial del robot, indicada por el marcador denominado Robot 1, así como la ubicación a la que deberá llegar, representada por el marcador identificado como Destino 1.

Figura 4.15: Robot 1 antes del lanzamiento al punto Destino 1 (Fuente: Elaboración propia).

Ahora, en la Figura 4.16 se muestra la posición final del robot 1, la cual tuvo un error de 1,43 metros de diferencia con la ubicación del marcador de destino 1.

Una segunda prueba realizada, cuyos resultados se muestran en la Figura 4.17 y en la Figura 4.18, ilustra tanto el punto de origen como el de destino, antes y después del lanzamiento del robot 1 correspondientemente. En esta prueba, el punto de llegada tuvo un error de 2,29 metros respecto al punto de destino.

Con las pruebas de navegación realizadas en el robot 1, se valida el correcto funcionamiento del

Figura 4.16: Robot 1 en posición final de navegación al punto Destino 1 (Fuente: Elaboración propia).

Figura 4.17: Robot 1 antes del lanzamiento al punto Destino 1 (Navegación 2) (Fuente: Elaboración propia).

Figura 4.18: Robot 1 en posición final de navegación al punto Destino 1 (Navegación 2) (Fuente: Elaboración propia).

robot 1 en conjunto con el sistema de comunicación y la interfaz gráfica de usuario (GUI).

4.3.2. Navegación Robot DaNI 2

Descripción general de la prueba

Para esta segunda prueba, se selecciona únicamente el robot 2 desde la interfaz gráfica de usuario (GUI), asignándole un punto de destino al cual debe dirigirse. A través del mapa de Leaflet, se

verificará la posición de los marcadores correspondientes al punto de origen del robot y al destino seleccionado. Posteriormente, se procederá a verificar el trayecto seguido por el robot y la ubicación final alcanzada.

Resultados y análisis de la prueba

En la Figura 4.19 se muestra la posición inicial del robot, indicada por el marcador denominado Robot 2, así como la ubicación a la que deberá llegar, representada por el marcador identificado como Destino 2.

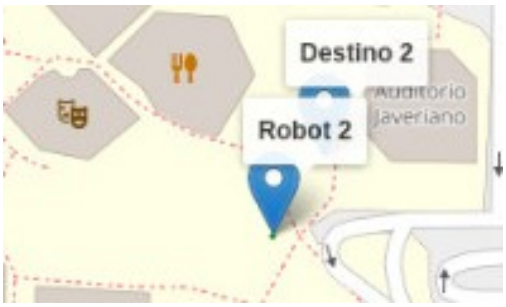

Figura 4.19: Robot 2 antes del lanzamiento al punto Destino 2 (Fuente: Elaboración propia).

Ahora, en la Figura 4.20 se muestra la posición final del robot 2, la cual tuvo un error de 2,05 metros respecto al punto de destino.

Figura 4.20: Robot 2 en posición final de navegación al punto Destino 2 (Fuente: Elaboración propia).

Una segunda prueba realizada, cuyos resultados se muestran en la Figura 4.21 y en la Figura 4.22, ilustra tanto el punto de origen como el de destino, antes y después del lanzamiento del robot 2, correspondientemente. En esta prueba, el error de posicionamiento fue de 2,87 metros.

Con las pruebas de navegación realizadas en el robot 2, se valida su correcto funcionamiento en conjunto con el sistema de comunicación y la interfaz gráfica de usuario (GUI). Sin embargo, cabe resaltar que, en este caso, el robot se dispersó un poco alrededor de los puntos de llegada tratando de determinar el correcto alcance de las metas, lo que explica la apariencia de la trayectoria.

Figura 4.21: Robot 2 antes del lanzamiento al punto Destino 2 (Navegación 2) (Fuente: Elaboración propia).

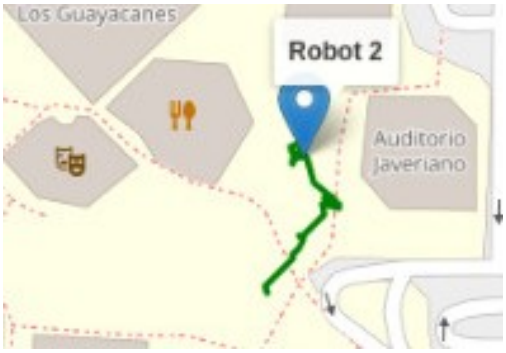

Figura 4.22: Robot 2 en posición final de navegación al punto Destino 2 (Navegación 2) (Fuente: Elaboración propia).

Este comportamiento se debe en su mayoría a la precisión del GPS y a la robustez del mismo en condiciones ambientales desfavorables.

4.3.3. Navegación simultanea de robots DaNI

Descripción general de la prueba

Para esta última prueba, se selecciona "Todos los Robots"desde la interfaz gráfica de usuario (GUI), asignándoles un punto de destino a cada robot al cual deben dirigirse. A través del mapa de Leaflet, se verificará la posición de los marcadores correspondientes al punto de origen de los robots 1 y 2 y a los destinos seleccionados. Posteriormente, se procederá a verificar el trayecto seguido por ambos robots en su navegación en conjunto y la ubicación final alcanzada por cada uno.

Resultados y análisis de la prueba

En la Figura 4.23 se muestra la posición inicial de los robots, indicada por los marcadores denominados Robot 1 y Robot 2, así como la ubicación a la que deberán llegar, representada por los

marcadores identificados como Destino 1 y Destino 2. Lamentablemente la herramienta no permite obtener una mejor visualización.

Figura 4.23: Robot 1 y 2 antes del lanzamiento a los puntos Destino 1 y 2 (Fuente: Elaboración propia).

Ahora, en la Figura 4.24, se muestran las posiciones finales de los robots 1 y 2, las cuales se asemejan a la ubicación de los marcadores de destino 1 y 2 en el mapa, pero con errores de posicionamiento en la navegación de 1,76 metros y de 2,58 metros, respectivamente.

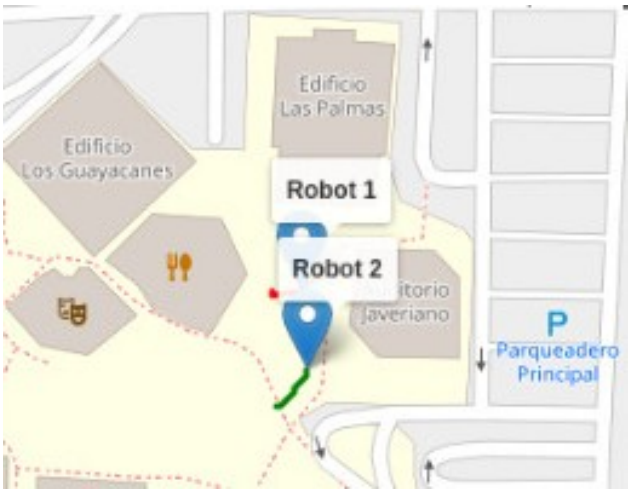

Figura 4.24: Robot 1 y 2 en su posición final de navegación al punto Destino 1 y 2 (Fuente: Elaboración propia).

Una segunda prueba realizada, cuyos resultados se muestran en las Figuras 4.25 y 4.26, ilustra tanto el punto de origen de los robots 1 y 2 como sus destinos 1 y 2, antes y después de su lanzamiento.

En este caso, los errores en la navegación fueron de 2,34 metros para el robot 1 y de 2,95 metros para el robot 2.

Figura 4.25: Robot 1 y 2 antes del lanzamiento a los puntos Destino 1 y 2 (Navegación 2) (Fuente: Elaboración propia).

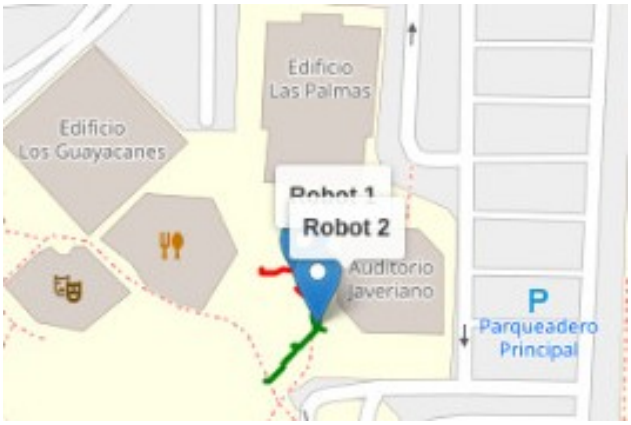

Figura 4.26: Robot 1 y 2 en posición final de su navegación a los puntos Destino 1 y 2 (Navegación 2) (Fuente: Elaboración propia).

Con las pruebas de navegación realizadas en los robots 1 y 2 simultáneamente, se valida el correcto funcionamiento de los mismos en conjunto con el sistema de comunicación y la interfaz gráfica de usuario (GUI), pero con los errores en la precisión de la navegación correspondientes.

La validación del sistema de navegación se centró en el correcto funcionamiento de los robots, tanto de manera individual como en conjunto, en su navegación autónoma, en integración con el sistema de comunicación y la interfaz gráfica de usuario (GUI). Durante las pruebas, se comprobó

el correcto funcionamiento de los robots en la navegación autónoma con errores de 2,28 metros de media. Adicionalmente, en el proceso de realizar las pruebas, se quiso comprobar el funcionamiento de la capacidad de evasión de obstáculos, por lo que en medio de algunas de las trayectorias de los robots se interpuso una persona y los robots la rodeaban satisfactoriamente. Sin embargo, por la forma de visualización del mapa, esto no se puede apreciar. Por otra parte, se evidenció que los GPS presentes en los robots DaNI son poco robustos bajo condiciones de clima nublado. Además, se verificó que, a lo largo de las pruebas, el GPS Garmin ofrece una mayor estabilidad en comparación con el GPS Ublox.

Capítulo 5

Conclusiones

En este trabajo se desarrolló un sistema de comunicación multi-robot basado en el framework ROS con el propósito de habilitar la navegación conjunta. Previamente, se había implementado un sistema de navegación autónoma en ROS para un robot terrestre denominado DaNI, lo que representó un avance significativo. Sin embargo, la ausencia de un sistema de comunicación diseñado para la navegación conjunta entre varios robots, así como la falta de una interfaz gráfica de usuario para su control, limitaban su aplicabilidad en diversos escenarios, tanto en el ámbito académico como en desarrollos futuros que requieran la operación de más de un robot. Además, el uso del sistema dependía de la ejecución de comandos en la línea de comandos de Linux a través de ROS, lo que demandaba conocimientos previos y dificultaba su accesibilidad para usuarios sin experiencia en la plataforma.

Se logró construir un segundo robot DaNI cumpliendo con las conexiones eléctricas correspondientes. También se logró la instalación de las dependencias necesarias, compilación y adecuación de los paquetes del proyecto de navegación autónoma, con lo cual se obtuvo finalmente un segundo robot que navega autónomamente y que, pese a no contar con el LiDAR A3 de mejores características de precisión y rango en comparación con el A1 que fue implementado, cumple de manera óptima la navegación autónoma. A su vez, se mejoraron algunos aspectos como la detección de los periféricos GPS, IMU y LiDAR por parte de la Jetson Nano de forma automática con el uso de reglas udev para mayor comodidad de uso. También se modificó el archivo de lanzamiento (launch) con el fin de que desde éste se lancen todos los nodos necesarios en un solo paso.

Se logró desarrollar el sistema de comunicación para los robots DaNI por medio de un sistema multimaster centralizado, implementado por medio del paquete fkie_multimaster, el cual posibilita visualizar nodos, tópicos y servicios en una red WLAN. El sistema es fundamental para enviar y cancelar las metas de cada robot correspondientemente. A su vez, permite especificar nodos, tópicos y servicios no esenciales en el envío y cancelación de metas con el fin de no generar conflictos y optimizar el uso de ancho de banda de la red.

Se crearon con éxito los servicios ROS LaunchGoal y CancelGoal, necesarios para enviar y cancelar metas a cada robot correspondientemente desde la interfaz gráfica de usuario. Las metas enviadas o canceladas serán transmitidas por el cliente de servicio de acción y gestionadas por el servidor de servicio de acción.

Se logró el desarrollo de una interfaz gráfica de usuario (GUI) intuitiva y portable en sistemas Linux, utilizando Docker para facilitar su implementación. La GUI, basada en PyQt5, permite futuras

expansiones y modificaciones. La versión final de la interfaz incluye módulos para la selección de robots, visualización de latitud y longitud, gestión de coordenadas destino, identificación del robot más cercano a un punto dado, navegación con visualización de metas en un mapa en tiempo real y un panel para la detención de robots en operación.

En la evaluación del sistema de comunicación, se determinó que la distancia entre el computador y los robots no constituye un factor relevante en la variación de la latencia. Por otro lado, se determinó la importancia del uso de hilos y los procesos concurrentes en aplicaciones de este tipo, pues el realizar los procesos de manera secuencial retrasa mucho los tiempos de ejecución. Esto quedó evidenciado durante la ejecución del servicio CancelGoal, al compararse la implementación secuencial con la posterior implementación concurrente, ya que se obtuvo una diferencia media de más de 5 segundos en el proceso secuencial a comparación del proceso concurrente. Aquí, cada segundo de retraso podría resultar en un incremento de 35 centímetros en la distancia de frenado.

Las pruebas de simulación de latencia, realizadas mediante la introducción de delay a la red WLAN, permitieron identificar los efectos que puede tener la latencia en el sistema. Para niveles de latencia bajos, el sistema aún tiene una recepción rápida de las solicitudes de servicios, mientras que, para niveles altos (1000 ms o más), el sistema empieza a tener retrasos considerables que pueden ocasionar situaciones críticas.

Se observó que el consumo de ancho de banda por parte de los robots presenta un comportamiento similar, ya sea que la navegación se realice de manera individual o conjunta durante la ejecución de LaunchGoal, alcanzando un pico máximo de 89,30 kbps de consumo por un robot.

En las pruebas de navegación conjunta se confirmó un comportamiento comparable al observado en el trabajo de grado previo relacionado con la actualización del sistema de navegación autónoma, presentando una correcta navegación hacia un punto de destino, pero presentando errores en promedio de 2,28 metros, esto debido a la precisión de los GPS usados en los robots, así como también la baja robustez de los mismos.

En resumen, se logró desarrollar un sistema de comunicación eficiente para la navegación conjunta de dos robots DaNI, acompañado de una interfaz gráfica de usuario que facilita su operación. La escalabilidad del sistema depende principalmente de la capacidad del procesador donde se ejecuta la GUI, debido a la concurrencia de tareas asociadas al envío de metas y la cancelación de las mismas, así como del ancho de banda disponible en la red para la comunicación con cada robot. Este trabajo representa un avance significativo en la integración de sistemas multi-robot en ROS para la Universidad Javeriana de Cali, sentando las bases para futuras mejoras y aplicaciones en entornos colaborativos.

Capítulo 6

Recomendaciones

A partir de los resultados obtenidos en el desarrollo e implementación del sistema de comunicación multi-robot y su integración con el sistema de navegación autónoma de los robots DaNI, se proponen las siguientes recomendaciones con el fin de mejorar el rendimiento y la eficiencia de la solución, así como para futuras expansiones del sistema:

- **Automatizar la escalabilidad de nuevos robots DaNI:** Se propone la implementación de algoritmos que permitan la integración automática de nuevos robots DaNI a la interfaz gráfica de usuario (GUI). Actualmente, añadir un nuevo robot al sistema implica modificaciones manuales en la interfaz para crear módulos adicionales y configurar las comunicaciones necesarias. Para superar esta limitación se podría hacer uso del nodo master_discovery el cual se encarga de monitorear los masters disponibles en la red, y publica información sobre los cambios en los masters (conexiones, desconexiones, etc.) a través del tópico master_discovery/changes. Este tópico publica mensajes de tipo MasterState que contienen información relevante sobre los masters, como su nombre, URI y estado. A partir de esta información se podría generar una integración con algún algoritmo que se desarrolle en el script de la GUI y que permita crear los módulos correspondientes a cada robot dependiendo de la cantidad disponible, así como también de su actualización en el ComboBox que muestra la lista de los robots disponibles.

- **Implementación de algoritmos para la gestión de tareas multi-robot, aprovechando el sistema de comunicación:** A partir del sistema de comunicación desarrollado para los robots DaNI y su interfaz gráfica de usuario (GUI), se presenta la oportunidad de implementar algoritmos avanzados y nuevas funcionalidades que podrían ser aplicables a una amplia gama de campos, tales como la cartografía, en la recopilación de datos en tiempo real para la creación de mapas topográficos; la agricultura de precisión, en el análisis detallado de cultivos; y la logística, en el manejo y transporte de mercancías. La GUI podría desempeñar un papel fundamental en estas aplicaciones, actuando como el eje central desde donde se programen y coordinen los algoritmos, así como para que se desarrollen nuevas funcionalidades que permitan gestionar y supervisar de manera eficiente todas las actividades de los robots. Además, dependiendo de la aplicación específica a la que se destine el sistema, será necesario adecuar tanto el hardware como el software, adaptándolos al entorno en el que se ejecutarán las tareas. Esto incluiría tanto la adaptación de los parámetros de entrada requeridos por el software, como la optimización del hardware para garantizar el buen funcionamiento del sistema en condiciones específicas de operación.

- **Mejoramiento de la precisión del sistema de navegación:** A lo largo de las pruebas de navegación, se observó el desempeño en cuanto a la precisión de los sistemas GPS de los robots, los cuales presentaron ciertos problemas debido a diversos factores, como su precisión limitada, la falta de robustez en condiciones de clima nublado y la estabilidad del GPS, especialmente en el caso del modelo Ublox. Estos factores ocasionaron que, en algunos momentos, el sistema de navegación no operara de la manera más óptima posible. En vista de ello, se recomienda considerar la sustitución de estos dispositivos por otros de mayor robustez, que puedan ofrecer una mayor fiabilidad en diversos entornos. Esto permitiría garantizar un mejor desempeño del sistema en condiciones variables, así como la capacidad de abordar con éxito nuevas aplicaciones que se puedan implementar en el sistema multi-robot en su conjunto.

Capítulo 7

Anexos

Anexo 1 – Diagrama eléctrico

El diagrama eléctrico utilizado en el montaje del segundo robot DaNI de acuerdo al trabajo de grado de navegación autónoma [9] se presenta en la Figura 7.1. Este diagrama proporciona una representación detallada de las conexiones eléctricas entre los distintos componentes del sistema, asegurando una comprensión integral del ensamblaje y de las interacciones de cada dispositivo involucrado en el robot.

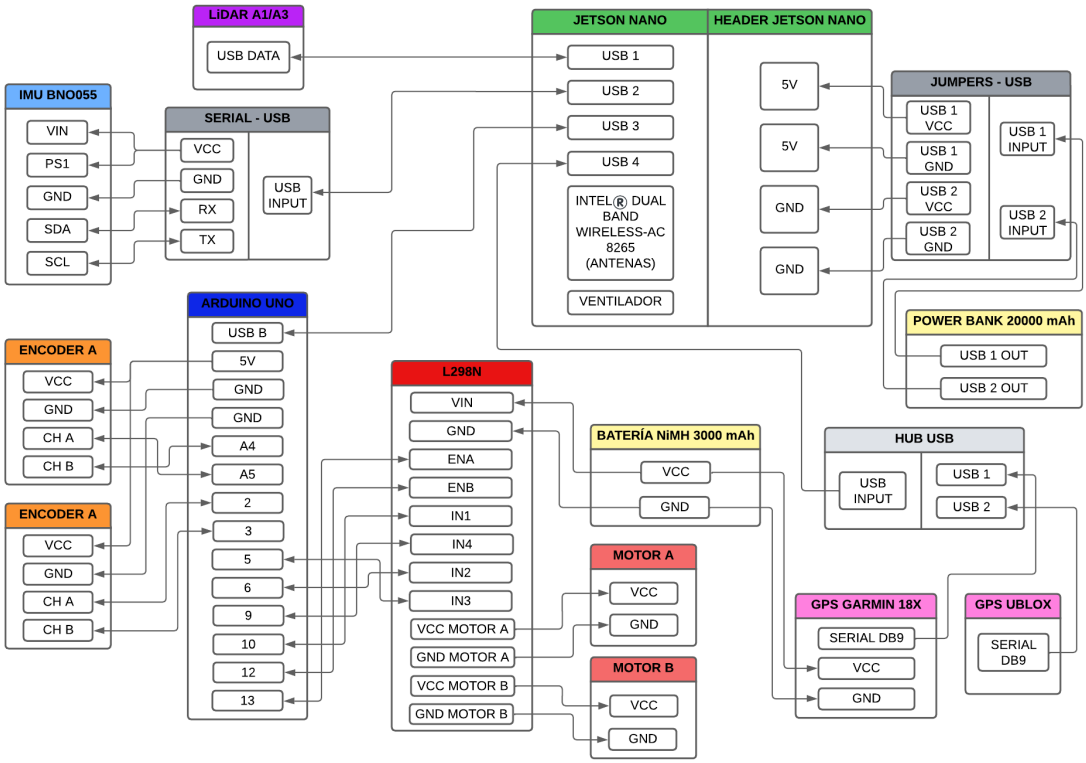

Figura 7.1: Diagrama de conexiones eléctricas Robot DaNI (Fuente: Elaboración propia).

Anexo 2 – Montaje segundo robot DaNI

Las conexiones eléctricas correspondientes a los pasos 1 y 2 se presentan en el Anexo 1.

Paso 1

En este paso, es fundamental asegurar una adecuada disposición de los componentes dentro del chasis para optimizar tanto la funcionalidad como el equilibrio del robot. En primer lugar, se deben posicionar los motores en los costados del chasis, garantizando una instalación firme que permita una transmisión eficiente del movimiento. La batería NiMH y el power bank deben ubicarse en los espacios libres de la parte superior del chasis, de modo que queden fácilmente accesibles y con una distribución de peso equilibrada.

Adicionalmente, es necesario organizar correctamente el puente H, la placa Arduino Uno, y la Jetson Nano en la plataforma de montaje negra situada sobre el chasis. Es importante destacar que, en la parte inferior de esta plataforma, solo debe colocarse el puente H, mientras que en la parte superior se instalarán la placa Arduino Uno y la Jetson Nano.

La organización completa del montaje en este primer paso, incluyendo la ubicación precisa de cada uno de los componentes en el chasis y en la plataforma de montaje, se presenta en las Figuras 7.2, 7.3 y 7.4.

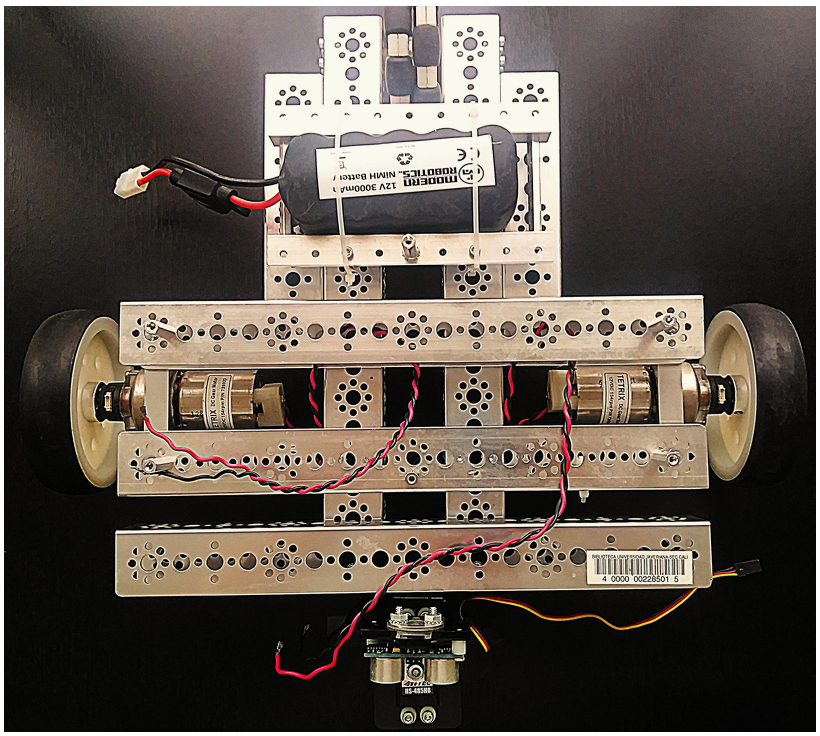

Figura 7.2: Distribución de motores y bateria NiMH (Fuente: Elaboración propia).

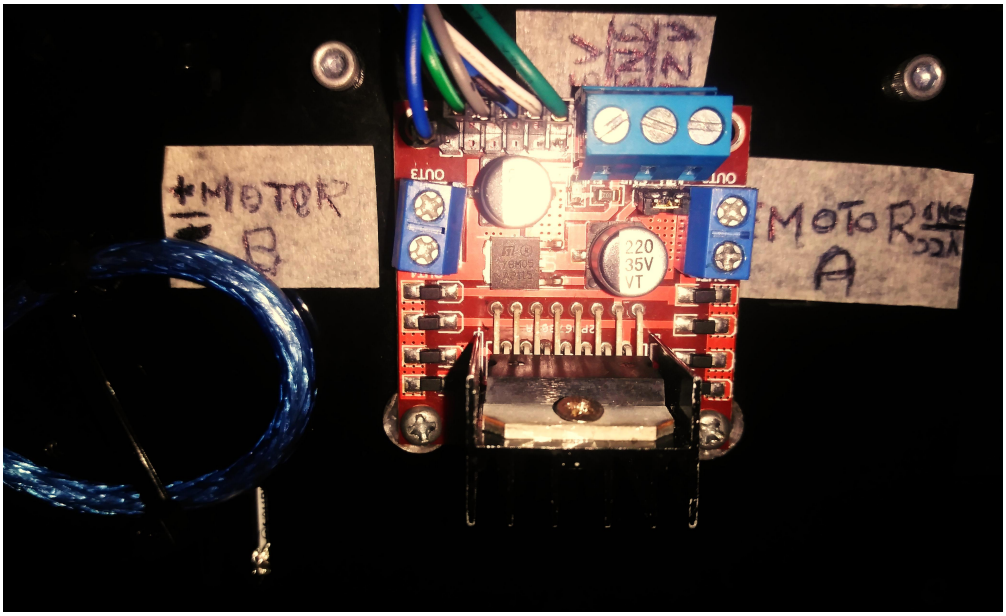

Figura 7.3: Plataforma de montaje inferior con puente H (Fuente: Elaboración propia).

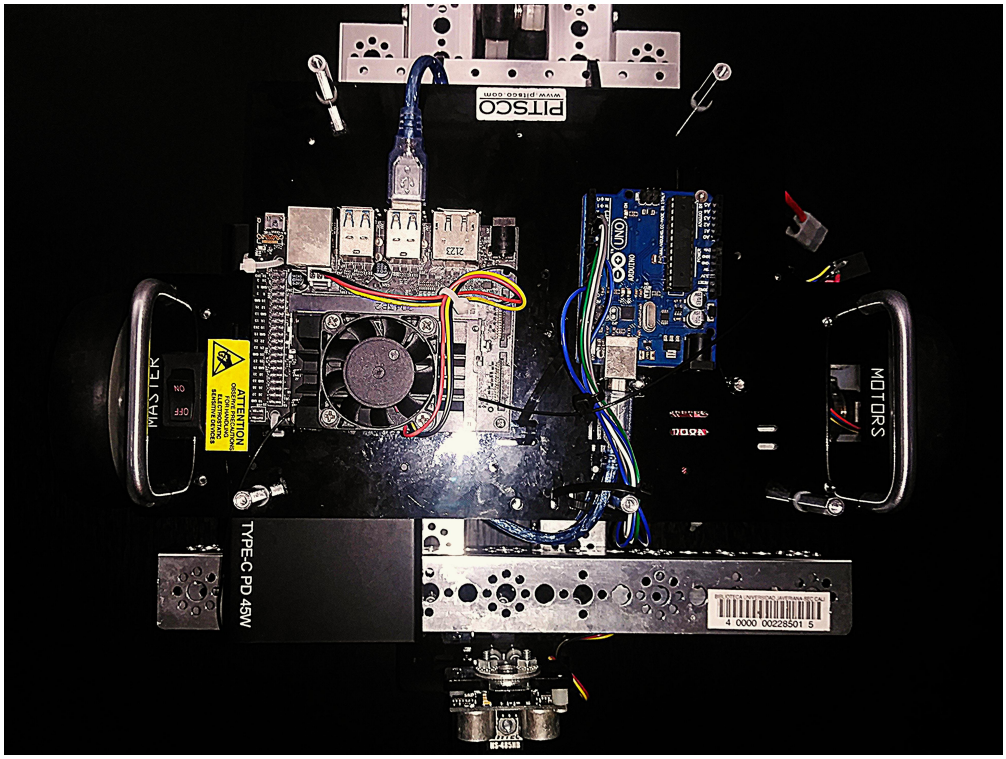

Figura 7.4: Plataforma de montaje superior y Power Bank (Fuente: Elaboración propia).

Paso 2

En esta segunda y última fase de montaje, se procede a la distribución de los sensores, tales como el GPS, la Unidad de Medición Inercial (IMU) y el LiDAR, en la segunda plataforma del robot, la cual es de material transparente. En esta plataforma se ubican cuidadosamente cada uno de estos sensores en posiciones estratégicas para asegurar una funcionalidad óptima y minimizar posibles interferencias.

Adicionalmente, se llevan a cabo las conexiones eléctricas correspondientes, integrando estos sensores con los otros componentes electrónicos del sistema, como el Arduino Uno, la Jetson Nano y Hub. La distribución de los sensores y las conexiones eléctricas resultantes se pueden observar detalladamente en la Figura 7.5.

Figura 7.5: Plataforma de montaje de los sensores (Fuente: Elaboración propia).

Anexo 3 – Manual de usuario

El sistema de comunicación multi-robot en ROS cuenta inicialmente con dos robots DaNI completamente actualizados y preparados para su aplicación de navegación autónoma por medio de un computador central que controlará todo el sistema. Para ello se puede hacer uso del router ASUS WL-330N que viene incorporado también junto con los robots, debido a que el sistema ya viene configurado para conectarse automáticamente a la red que provee dicho router, pero se pueden configurar los robots para conectarse a otra red si así se desea.

En [9], en el *Anexo C – Manual de Usuario*, se puede obtener una primera guía acerca de diferentes aspectos relacionados con el robot 1 para su funcionamiento, como lo pueden ser los elementos básicos del robot, indicadores de funcionamiento de los distintos elementos del robot (específicamente indicadores de alimentación), puertos de conexión, una sección de configuración inicial en la que se menciona cómo alimentar el sistema para habilitar el uso de todos los componentes, cómo conectarse remotamente al robot, cómo verificar los puertos que están en uso, cómo se debe configurar previamente el sistema para ser usado en la navegación autónoma y, por último, una sección de solución de problemas. Esta guía puede servir en su mayoría para explicar el funcionamiento del robot 2, pero con algunas diferencias en cuanto al montaje de los elementos básicos, la forma de alimentación y la IP estática para conexión remota al router ASUS WL-330N. Sólo estas partes mencionadas podrían servir como complemento a este *Manual de usuario*, ya que lo relacionado a la operación de la aplicación de navegación autónoma cambió con el trabajo realizado en este proyecto.

1. Diferencias entre el robot 1 y el robot 2

En términos generales ambos robots son muy similares, pero las principales diferencias a encontrar entre ambos robots son las siguientes:

- La versión del LiDAR, ya que en el robot 1 es un LiDAR A3 y en el robot 2 es un LiDAR A1.
- Los switches de potencia general y de motores, ya que en el robot 1 son dos switches diferentes en los que el switch general habilita la potencia de la batería NiMH para alimentar el GPS y el switch de motores habilita la potencia de la batería NiMH para alimentar los motores y el GPS en paralelo, mientras que en el robot 2 está puenteada la alimentación del GPS y los motores para habilitar la potencia de todo este conjunto únicamente con el switch de motores.
- La alimentación de la Jetson Nano, pues en el robot 1 se implementaron unos switches para habilitar la potencia de la Power Bank y darle inicio al sistema del robot solo por medio de estos switches, mientras que a la Jetson Nano del robot 2 se le habilita la potencia de la Power Bank conectando el cable USB al correspondiente puerto de la batería y posteriormente presionando un botón ubicado en un lateral de la batería.
- La visualización del nivel de carga de las Power Bank, pues en la Power Bank del robot 1 el nivel de carga se representa con 4 leds blancos los cuales indican un 25 % de la carga total cada uno, mientras que en la Power Bank del robot 2 el nivel de carga se muestra numéricamente,

de 0 % a 100 %.

- La conexión remota con el router ASUS WL-330N, debido a que cuentan con direcciones IP diferentes. Como se menciona en [9], la dirección IP fija para el robot 1 en la red de este router es 192.168.1.99 para una conexión WiFi y 192.168.1.24 para una conexión por Ethernet, mientras que la dirección IP fija para el robot 2 es 192.168.1.142 para una conexión WiFI y 192.168.1.197 para una conexión por Ethernet. Esta conexión remota se puede hacer desde SSH en el PC central (o cualquier otro dispositivo que se quiera conectar) o desde NoMachine. La conexión por SSH permite una mayor fluidez en el uso de la terminal, por lo que es recomendable conectarse por este método. Sin embargo, si se requiere de una visualización gráfica, como lo puede ser el uso de RViz, será necesaria la conexión por NoMachine. Es importante tener presente que ambos robots cuentan con un usuario llamado **user** y contraseña **user**, ya que con estas credenciales se permitirá acceder a la conexión remota y a cualquier otra operación que requiera autenticación de usuario. El robot 1 tiene un hostname "user-desktop" y el robot 2 tiene un hostname "Jetson2", lo que significa que así serán reconocidos en la red en la que estén presentes.

2. Configuración de la imagen del PC central

Como se ha mencionado a lo largo del documento, el PC central que controla el sistema de comunicación usa una imagen de Linux guardada en un contenedor de Docker. Esta imagen se podría usar en un computador con sistema operativo Windows pero se requeriría realizar unos procesos adicionales como lo son la instalación de Docker Desktop, el uso de WSL2 (Windows Subsystem for Linux) que viene por defecto en la mayoría de dispositivos Windows actuales, entre otros procesos de configuración que podrían resultar tediosos. Por todo esto, se recomienda hacer uso de un computador con sistema operativo Linux nativo, pues facilita mucho la configuración del PC para incluirlo en el sistema de comunicación.

Si no se tiene Docker instalado, aquí se explica cómo instalarlo en distribuciones basadas en Debian/ Ubuntu. Antes de instalar Docker, hay que asegurarse de cumplir con los siguientes requisitos:

- Un sistema operativo Linux actualizado.
- Acceso a un usuario con privilegios de superusuario (*root*) o la capacidad de usar *sudo*.
- Conexión a internet para descargar los paquetes necesarios.

Una vez se cumpla con esto, se procede a realizar los siguientes pasos:

- **Actualizar el sistema:** Abra una terminal y ejecute el siguiente comando para actualizar los paquetes disponibles:
  sudo apt update && sudo apt upgrade -y
- **Instalar dependencias requeridas:** Ejecute el siguiente comando:
  sudo apt install -y ca-certificates curl gnupg lsb-release

- **Agregar el repositorio oficial de Docker:** Ejecute los siguientes comandos:
  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
  *echo "deb [arch=$(dpkg --print-architecture)] https://download.docker.com/linux/ubuntu *
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
  sudo apt update

- **Instalar Docker:** Ejecute el siguiente comando:
  sudo apt install -y docker-ce docker-ce-cli containerd.io

- **Verificar la instalación:** Ejecute los siguientes comandos para verificar que Docker está instalado correctamente:
  docker --version
  docker run hello-world
  Si la instalación fue exitosa, verá un mensaje indicando que el contenedor de prueba se ejecutó correctamente, como se muestra en la Figura 7.6.

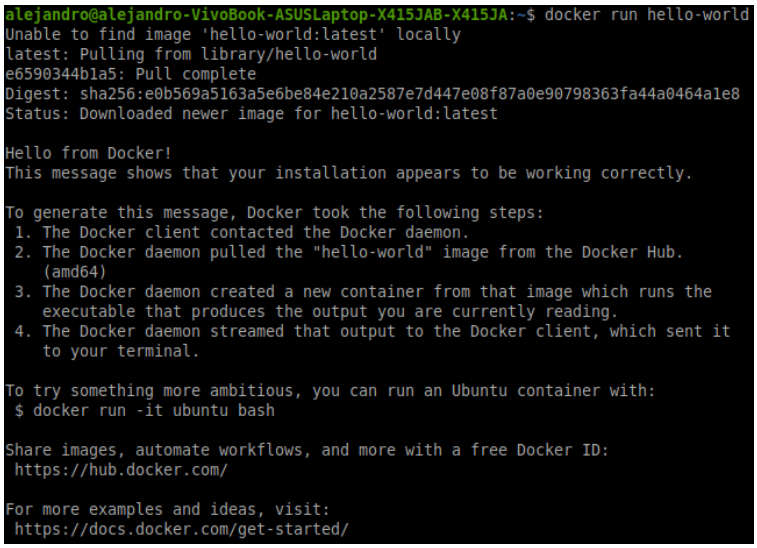

Figura 7.6: Instalación exitosa de Docker (Fuente: Elaboración propia).

Con todo esto, Docker debería estar instalado y listo para usar en el sistema Linux. Si se requiere más información o algo falló, se puede consultar la documentación oficial de Docker aquí.

Ahora, lo que sigue es descargar la imagen que va a ejecutarse en el PC central para el control de la comunicación multi-robot y la navegación. Esta imagen está alojada en un repositorio de *Docker Hub*. Docker Hub es el repositorio oficial de imágenes de Docker, donde se almacenan y distribuyen imágenes de contenedores listas para su uso.

Se deben seguir los siguientes pasos:

- **Descargar la imagen:** Para descargar la imagen desde Docker Hub, utilice:
  docker pull alejocr18/manager_multimaster_ros
- **Ver las imágenes descargadas:** Puede listar todas las imagenes que haya descargado en su sistema con el siguiente comando:
  docker images
  Debería ver algo como lo que se muestra en la Figura 7.7

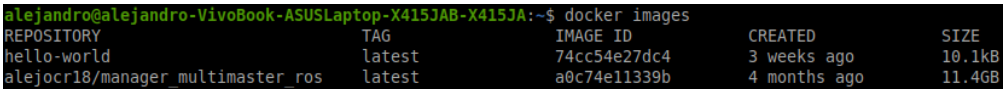

Figura 7.7: Imágenes de Docker guardadas (Fuente: Elaboración propia).

- **Eliminar imágenes:** Si desea eliminar imágenes que ya no utilice puede usar el siguiente comando:
  docker rmi alejocr18/manager_multimaster_ros:latest
  Puede cambiar el nombre de la imagen a la que desea eliminar del sistema. La sentencia ":latest" se usa para referirse a la última versión creada o guardada de la imágen a la cual hace referencia, ya que puede tener al mismo tiempo varias versiones diferentes de la misma imágen.

En este punto ya se cuenta con todo lo necesario para usar el sistema de comunicación para la navegación conjunta de los robots. Si se tiene alguna duda o se quiere entrar más en detalle sobre Docker Hub, puede visitar su página oficial aquí.

3. Lanzamiento del sistema de comunicación

Con todo el sistema ya configurado, se procede a lanzar un contenedor que contenga la imagen que se ha descargado. Este contenedor se puede lanzar de varias formas, pero inicialmente se recomienda lanzarlo usando el siguiente comando:

sudo docker run --rm -it --network=host --privileged -v /tmp/.X11-unix:/tmp/.X11-unix -e DISPLAY=$DISPLAY --name master_pc alejocr18/manager_multimaster_ros

Este comando se explica de la siguiente manera:

- **sudo docker run:** Ejecuta un contenedor Docker.
- **--rm:** Elimina automáticamente el contenedor una vez que finaliza su ejecución.
- **-it:** La bandera -i mantiene la entrada estándar abierta para permitir la interacción, y la bandera -t asigna una terminal virtual (pseudo-TTY).
- **--network=host:** Hace que el contenedor use la red del host directamente, sin crear una red virtual. Esto es útil cuando el contenedor necesita acceso a la red sin restricciones, por

ejemplo, para comunicar procesos entre el contenedor y el sistema anfitrión. En este caso, le permite al contenedor usar la red del host para comunicarse con los robots presentes en la red local.

- **--privileged:** Otorga privilegios extendidos al contenedor, permitiéndole acceder a dispositivos y configuraciones del sistema anfitrión.
- **-v /tmp/.X11-unix:/tmp/.X11-unix:** La bandera -v permite montar un volumen del sistema anfitrión al contenedor. En este caso comparte el socket de X11 entre el host y el contenedor para permitir aplicaciones gráficas dentro del contenedor.
- **-e DISPLAY=$DISPLAY:** La bandera -e define una variable de entorno dentro del contenedor la cual es DISPLAY=$DISPLAY, que permite que las aplicaciones gráficas ejecutadas dentro del contenedor se muestren en la pantalla del host.
- **--name master_pc:** Asigna el nombre master_pc al contenedor.
- **alejocr18/manager_multimaster_ros:** Especifica la imagen que se va a ejecutar.

Al ejecutar este comando se debería ver algo como lo que se ve en la Figura 7.8.

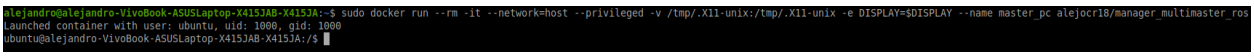

Figura 7.8: Lanzamiento del contenedor (Fuente: Elaboración propia).

Una vez aquí, se encontrará en la terminal del contenedor, en la cual podrá moverse a su gusto para inspeccionarla y observar qué tanto contenido tiene, de igual manera a como se hace en el dispositivo host. En un inicio se estará ubicado en la raíz del sistema (*root*), así que si se ingresa al directorio */home* se encontrarán dos usuarios: *ubuntu* y *user*. El usuario que contiene el proyecto es *user*, así que accediendo a él se encontrará el espacio de trabajo con el cual se va a lanzar el sistema de comunicación y la GUI para controlar la navegación. Antes de ejecutar el archivo de lanzamiento, se debe asegurar de configurar las variables de entorno. Se deben ejecutar los comandos que se muestran a continuación para realizar todo lo mencionado:

cd /home/user/catkin_ws/ source devel/setup.bash

Al hacer esto, ya se debería poder ejecutar el archivo de lanzamiento, pero aún se debe realizar un paso más para que la interfaz pueda generarse correctamente. Debido a que la herramienta que se usa para desarrollar la GUI es PyQt5, y muchos de los comandos necesarios en el script de desarrollo de la interfaz requieren de una versión de PyQt5 que no está disponible en los repositorios de Ubuntu 18.04 (versión de Ubuntu de la imagen), es necesario activar un entorno virtual de Python que permita utilizar una versión específica de Python y sus paquetes sin interferir con la instalación del sistema. El comando es el siguiente:

source /home/user/catkin_ws/src/server_manager/entorno/myenv/bin/activate

El comando *source* ejecuta un script en el mismo entorno de la terminal en lugar de hacerlo en un proceso separado. En este caso se usa para ejecutar el script de activación del entorno virtual *activate* que se encuentra ubicado en la ruta que está descrita en el comando. Como se ve, *myenv* es el nombre del entorno virtual creado previamente con la herramienta *venv*. Al ejecutarlo, se cambia el entorno de Python en la terminal a uno aislado. En la Figura 7.9 se puede apreciar cómo cambian las versiones de Python y PyQt5 después de activar el entorno virtual. También se puede ver que se reconoce el entorno virtual como activo al tener el nombre del entorno (*myenv*) al inicio de la línea de la terminal.

```
ubuntu@alejandro-VivoBook-ASUSLaptop-X415JAB-X415JA:/home/user$ python --version
Python 2.7.17
ubuntu@alejandro-VivoBook-ASUSLaptop-X415JAB-X415JA:/home/user$ python3 --version
Python 3.6.9
ubuntu@alejandro-VivoBook-ASUSLaptop-X415JAB-X415JA:/home/user$ python -c "import PyQt5.QtCore; print(PyQt5.QtCore.QT_VERSION_STR)"
5.9.5
ubuntu@alejandro-VivoBook-ASUSLaptop-X415JAB-X415JA:/home/user$ source /home/user/catkin_ws/src/server_manager/entorno/myenv/bin/activate
(myenv) ubuntu@alejandro-VivoBook-ASUSLaptop-X415JAB-X415JA:/home/user$ python --version
Python 3.8.0
(myenv) ubuntu@alejandro-VivoBook-ASUSLaptop-X415JAB-X415JA:/home/user$ python -c "import PyQt5.QtCore; print(PyQt5.QtCore.QT_VERSION_STR)"
5.15.2
```

Figura 7.9: Activación del entorno virtual (Fuente: Elaboración propia).

Si se quiere desactivar el entorno virtual, solo es necesario ejecutar el comando *deactivate* en la terminal.

Para hacer todo este proceso más eficiente y no tener que ejecutar todos los comandos mencionados, al final del archivo *~/.bashrc* se añaden las líneas que se muestran en la Figura 7.10. Este archivo es un script de configuración que se ejecuta automáticamente cada vez que se abre una nueva sesión interactiva de *Bash*, es decir, al abrir una terminal nueva, en este caso del contenedor.

```
#Atajo cd
cd(){
    if [ "$1" = "u" ]; then
        builtin cd /home/user/
        source /home/user/catkin_ws/devel/setup.bash
    elif [ "$1" = "vir" ]; then
        source /home/user/catkin_ws/src/server_manager/entorno/myenv/bin/activate
    else
        builtin cd "$@"
    fi
}

source /opt/ros/melodic/setup.bash
source `catkin locate --shell-verbs`
source /home/user/catkin_ws/devel/setup.bash
```

Figura 7.10: Configuración de variables de entorno automática y atajos (Fuente: Elaboración propia).

Como se puede apreciar de la Figura 7.10, al abrir una terminal del contenedor, automáticamente se configurarán las variables de entorno del espacio de trabajo. Además, se han creado unos atajos, de manera que al ejecutar "*cd u*" se dirigirá al directorio */home/user* y se volverán a configurar las variables de entorno del espacio de trabajo de una manera más rápida, lo que servirá en caso de

haber configurado las variables de entorno de otro proyecto diferente. Por otro lado, al ejecutar "*cd vir*" se activará el entorno virtual de una manera mucho más fácil.

Ya con el entorno virtual activo, ahora sí se puede ejecutar el archivo de lanzamiento de la interfaz y los nodos del paquete *multimaster_fkie* que permiten la comunicación con los robots. El comando es el siguiente:

roslaunch server_manager server.launch

Una vez ejecutado, se deberá ver en pantalla algo como lo siguiente:

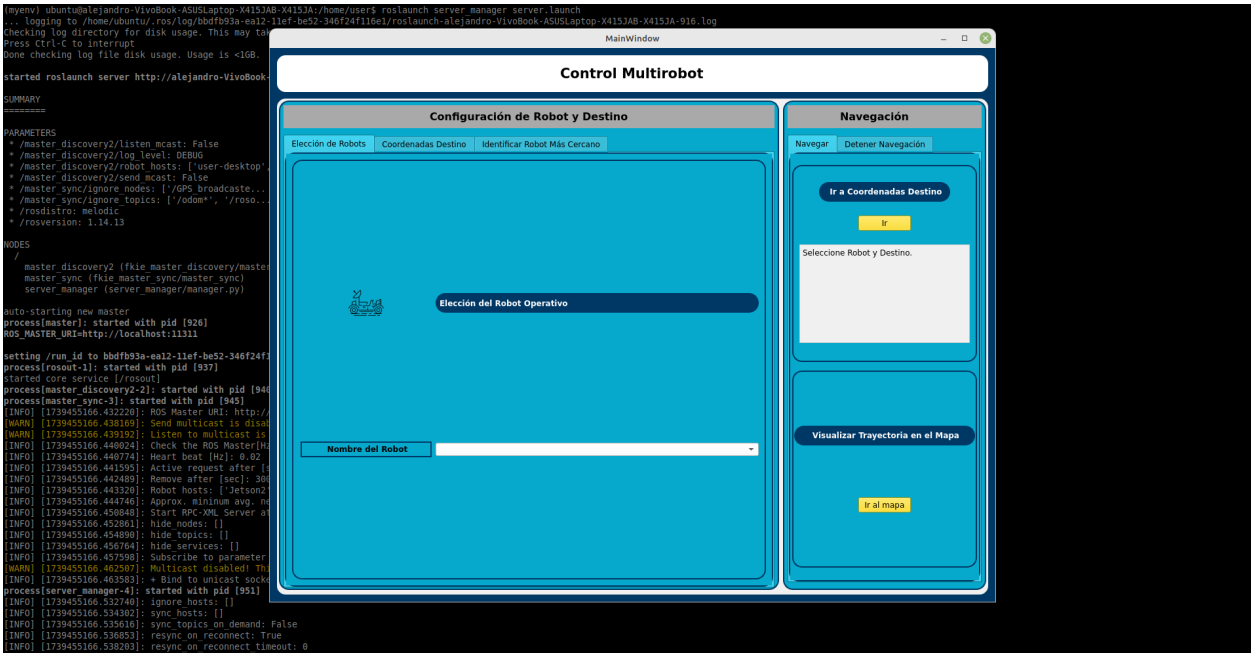

Figura 7.11: Lanzamiento de la GUI (Fuente: Elaboración propia).

En este punto ya se está listo para utilizar la GUI como se desee. En la sección 3.4.2 se explicó en detalle cada apartado de la GUI para saber cómo darle manejo. Si se quiere familiarizar con ella, dirigirse a dicha sección.

Es importante tener en cuenta que para el funcionamiento del sistema de comunicación, en los robots también se debe ejecutar el archivo de lanzamiento correspondiente:

roslaunch my_robot_tools full_setup.launch

Con todos los dispositivos lanzados y verificando que se encuentran en la misma red, se deberán identificar los robots en el computador y se comunicarán los datos necesarios de cada uno, como lo es la información de posicionamiento, por ejemplo.

Es recomendable abrir más de una terminal del contenedor para tener más libertad en el manejo del PC, pues al ejecutar el archivo de lanzamiento se ejecutan los nodos del paquete *multimaster_fkie* y el nodo correspondiete a la GUI, que se llama *server_manager* y es del tipo *manager.py*, por lo que si se cierra la interfaz y se quiere volver a abrirla, se deberá parar la ejecución del archivo de lanzamiento completo, lo que implica que se interrumpa la comunicación entre el PC y los robots. Para evitar esta interrupción, simplemente se abre otra terminal con el siguiente comando:

sudo docker exec -it -u ubuntu master_pc bash

Este comando abre una terminal interactiva dentro del contenedor llamado *master_pc* usando el usuario *ubuntu* en lugar de *root*, lo que permite ejecutar comandos dentro del contenedor.

Así, dentro de esta nueva terminal y ejecutando el comando "*cd vir*", se podrá lanzar únicamente el nodo de la GUI para abrirla y cerrarla cuantas veces se desee sin necesidad de afectar los demás nodos. Para lanzar el nodo de la GUI se emplea el siguiente comando:

rosrun server_manager manager.py

Hay que tener presente que para poder visualizar el mapa que viene incluido en la GUI se debe estar conectado a una red con conexión a internet ya que el mosaico y demás características del mapa se descargan desde un almacenamiento en la nube, por lo que si se está usando el router ASUS WL-330N en el modo predeterminado, no se podrá visualizar.

En caso de que se quiera lanzar el contenedor de manera que se ejecute el archivo de lanzamiento automáticamente y no se necesite hacer uso de la terminal, se puede usar el siguiente comando:

sudo docker run --rm -it --network=host --privileged -v /tmp/.X11-unix:/tmp/.X11-unix -e DISPLAY=$DISPLAY --name master_pc alejocr18/manager_multimaster_ros ./setup.sh

De esta manera, al finalizar la ejecución del archivo de lanzamiento (con "*ctrl + c*", por ejemplo), se eliminará automáticamente el contenedor.

En el PC central se puede configurar otro aspecto adicional antes de lanzar el contenedor, el cual puede ser bastante útil. Se trata sobre guardar las direcciones IP de los robots y sus respectivos hostname con los que aparecen en la red (o como se deseen guardar de manera personalizada) en el archivo *hosts* ubicado en el directorio */etc*, para que sea más fácil y rápido hacer una conexión remota a ellos por SSH editando el archivo *config* presente en la ruta */home/alejandro/.ssh/config* (cambiando "alejandro" por el usuario respectivo), como se aprecia en la Figura 7.13, o usar estos hostname en lugar de tener que poner las direcciones IP en ciertos parámetros o argumentos presentes en partes de código, como por ejemplo el argumento *robot_hosts_list* del archivo de lanzamiento del sistema de comunicación e interfaz gráfica, como se ve en la Figura 7.15.

Para editar el archivo *hosts* hay que ubicarse en el directorio */etc* y se ejecuta el siguiente comando:

sudo nano hosts

Una vez dentro, se edita de manera que quede algo como en la Figura 7.12:

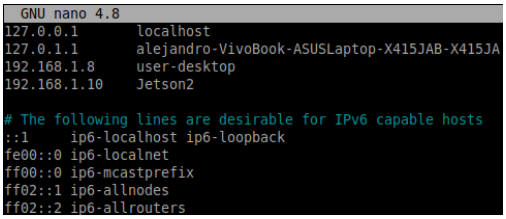

Figura 7.12: Archivo *hosts* (Fuente: Elaboración propia).

Luego de esto ya se pueden emplear los hostname para reconocer los robots.

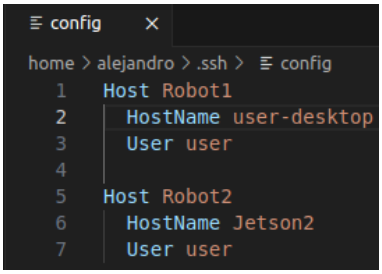

Figura 7.13: Archivo de configuración SSH (Fuente: Elaboración propia).

Se puede hacer la conexión SSH de la manera como se muestra en la Figura 7.14

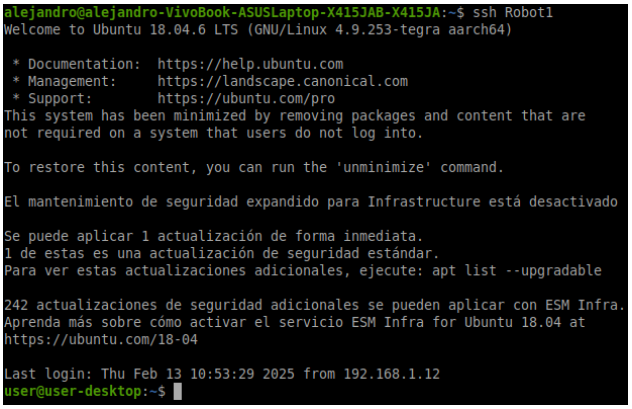

Figura 7.14: Conexión SSH con el Robot 1 (Fuente: Elaboración propia).

Es importante tener en cuenta que si las direcciones IP no son fijas en la red que usualmente se usa pueden variar en cualquier momento, por lo que se deberá editar el archivo *hosts* cada vez que esto pase para que no surjan errores.

server.launch ×

src > server_manager > launch > server.launch

```
1  <launch>
2      <arg name="robot_hosts_list" default="['user-desktop', 'Jetson2']"/>
3    <node pkg="fkie_master_discovery" type="master_discovery" name="master_discovery" output="screen">
4      <param name="log_level" value="DEBUG"/>
5      <param name="send_mcast" value="False"/>
6      <param name="listen_mcast" value="False"/>
7      <rosparam param="robot_hosts" subst_value="True">$(arg robot_hosts_list)</rosparam>
8    </node>
```

Figura 7.15: Argumento de lista de hosts (Fuente: Elaboración propia).

4. Adición de más robots

En caso de querer añadir más robots al sistema, inicialmente se deberá configurar el nuevo robot para que funcione de igual manera que los dos robots ya presentes. Para esto, se deberá equipar el robot nuevo con los mismos componentes con los que cuentan los robots ya actualizados, lo que incluye la Jetson Nano, el LiDAR, la IMU, el Arduino UNO y el GPS. En caso de no poder implementar las mismas referencias, se deberán integrar componentes que funcionen relativamente igual y que sean compatibles, esto suponiendo que se quiere utilizar el robot de la misma manera para la aplicación de navegación autónoma. Los demás componentes, como la Power Bank, la batería NiMH, las antenas y el módulo WiFi se pueden integrar a elección. Para esta explicación sí se necesitará contar con la Jetson Nano.

Para empezar, se necesita una tarjeta micro SD, preferiblemente de 64 GB, para bootear el sistema. Esta tarjeta se flasheará con la imagen del sistema operativo a usar, la cual será la distribución Ubuntu de Linux. Para realizar este proceso se recomienda seguir la guía de NVIDIA, que puede abrir desde aquí. Si ocurre algún error y no permite flashear la tarjeta con la imagen que se presenta en la guía, se deberá intentar flashear con una versión anterior del JetPack, que puede encontrar en el centro de descargas de Jetson.

<Una vez flasheada la tarjeta micro SD y haber booteado la Jetson, lo que procede es descargar todo el software necesario. Para esto se va a necesitar una conexión a internet. En caso de no haber configurado la conexión en el proceso de arranque del sistema, hay que seguir los siguientes pasos:

- **Verificar la interfaz WiFi:** Ejecuta:
  nmcli device
  Si se reconoce el adaptador WiFi, se tendrá que ver algo como:
  wlan0 wifi disconnected
  Si no aparece, usa: *iwconfig*
  Para ver si *wlan0* u otra interfaz está disponible.

- **Conectar a la red WiFi:** Si la interfaz WiFi aparece como *wlan0*, hay que conectarse con:
  nmcli dev wifi connect "NombreDeLaRed" password Çontraseña"
  Si la conexión es exitosa, se revisa con:
  nmcli connection show

Posteriormente, se puede proceder a actualizar todos los paquetes ejecutando:

sudo apt update
sudo apt upgrade -y

Después de actualizar, se puede liberar espacio con:

sudo apt autoremove -y
sudo apt clean

Y reiniciar la Jetson:

sudo reboot

Un elemento importante para instalar después de actualizar todo es el editor de texto para la terminal *nano*:

sudo apt update
sudo apt install nano -y

Ahora se puede proceder con la instalación de NoMachine. Una forma recomendable de hacerlo es buscar la versión compatible con el sistema (ARM64) en la página oficial de NoMachine y descargar el archivo *.deb*. Esto se puede hacer desde un computador externo y luego transferir el archivo descargado a la Jetson con:

scp nomachine_8.16.1_1_arm64.deb user@ip_de_la_jetson:~

Una vez transferido, en la Jetson se ejecuta:

sudo dpkg -i nomachine_8.10.1_1_arm64.deb

De esta manera ya queda instalado. Para verificar que el servicio NoMachine ya está corriendo, se ejecuta:

sudo /etc/NX/nxserver –status

Si está activo, se debe mostrar algo como:

NX>162 Enabled service: nxserver.
NX>162 Enabled service: nxnode.
NX>162 Enabled service: nxd.

Si no está corriendo, se intenta iniciarlo manualmente con:

sudo /etc/NX/nxserver --restart

En este punto ya se debería poder conectar remotamente a la Jetson a través de NoMachine desde un PC externo, teniendo en cuenta la dirección IP asignada.

Lo siguiente a instalar será ROS en su versión Melodic, ya que esta es la versión compatible con Ubuntu 18.04 que seguramente es la distribución instalada en la Jetson. Para verificar la versión de Ubuntu se usa:

lsb_ release -a

Para instalar ROS, primero se configuran los repositorios, ejecutando lo siguiente para asegurarse de que se tienen los repositorios adecuados:

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_ release -sc) main" >
/etc/apt/sources.list.d/ros-latest.list'

Luego se agrega la clave del repositorio:

sudo apt update
sudo apt install curl -y
curl -sSL 'https://raw.githubusercontent.com/ros/rosdistro/master/ros.key' | sudo apt-key add -

Se actualiza la lista de paquetes:

sudo apt update

Y se instala ROS Melodic:

sudo apt install ros-melodic-desktop-full -y

Posteriormente se configura el entorno de ROS. Para que ROS se cargue cada vez que se abra una terminal, se agrega esto al final de *~/.bashrc*:

echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
source ~/.bashrc

También se instalan dependencias adicionales:

*sudo apt install python-rosdep python-rosinstall python-rosinstall-generator *
python-wstool build-essential -y

Y se inicializa *rosdep*:

sudo rosdep init
rosdep update

Se puede comprobar que todo haya salido bien y ROS ha sido instalado correctamente si al ejecutar el siguiente comando se inicia el master sin errores:

roscore

Lo siguiente a realizar es descargar el espacio de trabajo de ROS con el que se se puede ejecutar el sistema de comunicación con la aplicación de navegación autónoma. Para ello, primero se crea un directorio de espacio de trabajo y se accede a él:

mkdir catkin_ ws

Ahora se clona el repositorio de GitHub donde se encuentran los archivos necesarios. Primero hay que asegurarse de tener Git instalado:

sudo apt install git -y

Y luego:

git clone https://github.com/AlejandroCorreaRamos/Multimaster_ROS.git

Esto habrá creado un nuevo directorio llamado *Multimaster_ROS* en la posición en la que se encuentra en ese momento. Del contenido de este directorio únicamente es necesario el directorio *src* que se encuentra en el directorio *Robots*, así que se moverá dicho directorio al directorio del espacio de trabajo creado (*catkin_ws*) de esta manera:

mv /Multimaster_ROS/Robots/src/ /catkin_ws/

Se verifica que el directorio se haya movido correctamente y, de ser así, lo demás se puede eliminar sin problema:

rm -rf Multimaster_ROS

Por último se compila el espacio de trabajo:

cd catkin_ws catkin_make

Esta compilación muy seguramente generará errores por falta de diferentes elementos como bibliotecas, dependencias o paquetes. Siguiendo el siguiente listado de comandos se puede resolver cada posible error:

- *sudo apt update*
- *sudo apt-get install libsdl2-dev libsdl2-image-dev*
- *sudo apt install ros-melodic-variant-topic-tools*
- *sudo apt install ros-melodic-openslam-gmapping*
- *sudo apt install ros-melodic-geographic-msgs*
- *sudo apt install libgeographic-dev*
- *sudo apt install ros-melodic-tf2-sensor-msgs*
- *sudo apt install ros-melodic-move-base-msgs*
- *sudo apt install python-pip*
- *sudo apt install ros-melodic-rospy*
- *pip2 install grpcio grpcio-tools*
- *pip3 install rospkg*
- *pip install pyserial*

En principio con todo esto ya debería compilar bien el espacio de trabajo. De no ser así, se tendrá que revisar el posible error en los logs y buscar soporte en foros o la página oficial de ROS.

Adicionalmente se tendrá que otorgar permisos de ejecución a ciertos archivos que son encargados de crear algunos de los nodos del lanzamiento del sistema. A continuación se muestra la lista:

- *cd /catkin_ws/src/ros_arduino_bridge/ros_arduino_python/nodes*
  chmod +x arduino_node.py
- *cd /home/user/catkin_ws/src/ros_imu_bno055/src*
  chmod +x imu_ros.py
- *cd /home/user/catkin_ws/src/ *
  nmea_navsat_driver-2ac72c41be3a18999770d609c17e1ac97225d9b6/scripts
  chmod +x nmea_serial_driver
- *cd /home/user/catkin_ws/src/server_bridge/src*
  chmod +x cancel.py
- *cd /home/user/catkin_ws/src/multimaster_fkie-melodic-v1.2.4/fkie_master_discovery *
  /nodes
  chmod +x master_discovery
- *cd /home/user/catkin_ws/src/multimaster_fkie-melodic-v1.2.4/fkie_master_sync *
  /nodes
  chmod +x master_sync

Por último, es necesario añadir el usuario al grupo *dialout* para que se le otorguen ciertos permisos:

sudo usermod -a -G dialout $USER

Hay un paso adicional que es recomendable realizar para evitar que el sistema operativo mantenga variando el puerto asignado a cada periférico conectado, para que así se mantenga estable y no se deba estar cambiando los puertos de los sensores en el archivo de lanzamiento *full_hw_test.launch*. Este paso se trata sobre crear reglas *udev*. Las reglas udev se encuentran en */etc/udev/rules.d/*. En este directorio se crea un archivo que va a contener las reglas para los sensores o demás dispositivos que estén conectados a la Jetson, como se muestra a continuación:

sudo nano /etc/udev/rules.d/98-mi_regla.rules

En este archivo se deben escribir las reglas como se ve en la Figura 7.16

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6001", SYMLINK+="LiDAR_A1"

SUBSYSTEM=="tty", ATTRS{idVendor}=="10c4", ATTRS{idProduct}=="ea60", SYMLINK+="IMU"

SUBSYSTEM=="tty", ATTRS{idVendor}=="1546", ATTRS{idProduct}=="01a7", SYMLINK+="GPS_Ublox"
```

Figura 7.16: Reglas udev del robot 2 (Fuente: Elaboración propia).

Estas reglas se escriben de acuerdo al *Vendor ID* y *Product ID* de cada dispositivo, y para encontrar estos valores se debe ejecutar en la terminal *lsusb*, como se ve en la Figura 7.17. Desconectando y conectando cada dispositivo se puede identificar cuál línea hace referencia a cada uno.

```
user@Jetson2:~$ lsusb
Bus 002 Device 003: ID 0bda:0411 Realtek Semiconductor Corp.
Bus 002 Device 002: ID 0bda:0411 Realtek Semiconductor Corp.
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 003: ID 8087:0a2b Intel Corp.
Bus 001 Device 009: ID 1546:01a7 U-Blox AG
Bus 001 Device 008: ID 1a86:7523 QinHeng Electronics HL-340 USB-Serial adapter
Bus 001 Device 007: ID 0bda:5411 Realtek Semiconductor Corp.
Bus 001 Device 006: ID 10c4:ea60 Cygnal Integrated Products, Inc. CP210x UART Bridge / myAVR mySmartUSB light
Bus 001 Device 005: ID 2341:0043 Arduino SA Uno R3 (CDC ACM)
Bus 001 Device 004: ID 0403:6001 Future Technology Devices International, Ltd FT232 USB-Serial (UART) IC
Bus 001 Device 002: ID 0bda:5411 Realtek Semiconductor Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Figura 7.17: Dispositivos presentes en el robot 2 (Fuente: Elaboración propia).

Después de crear el archivo se deben recargar las reglas con:

sudo udevadm control --reload-rules sudo udevadm trigger

Una vez creadas estas reglas, cuando se ejecute el comando *ls -l /dev |grep ttyUSB* o el comando *ls -l /dev |grep ttyACM* se mostrarán los puertos asignados a cada periférico, pero que siempre van a mantener el mismo nombre que se les estableció en las reglas, sea cual sea el puerto. Se verá algo como lo que se aprecia en la Figura 7.18.

```
user@Jetson2:~$ ls -l /dev |grep ttyUSB
lrwxrwxrwx  1 root root           7 feb 13 11:58 gps2 -> ttyUSB2
lrwxrwxrwx  1 root root           7 feb 13 11:58 IMU -> ttyUSB2
lrwxrwxrwx  1 root root           7 feb 13 11:58 LiDAR_A1 -> ttyUSB0
crw-rw----  1 root dialout 188,   0 feb 13 11:58 ttyUSB0
crw-rw----  1 root dialout 188,   1 feb 13 11:58 ttyUSB1
crw-rw----  1 root dialout 188,   2 feb 13 11:58 ttyUSB2
user@Jetson2:~$ ls -l /dev |grep ttyACM
lrwxrwxrwx  1 root root           7 feb 13 11:58 gps1 -> ttyACM1
lrwxrwxrwx  1 root root           7 feb 13 11:58 GPS_Ublox -> ttyACM1
crw-rw----  1 root dialout 166,   0 feb 13 11:58 ttyACM0
crw-rw----  1 root dialout 166,   1 feb 13 11:58 ttyACM1
```

Figura 7.18: Puertos asignados (Fuente: Elaboración propia).

El principal propósito de hacer esto es no tener que hacer cambios constantemente en el archivo de lanzamiento anteriormente mencionado, sino que con solo editarlo una vez con los nombres que hemos establecido se mantendrá así permanentemente. En la Figura 7.19 se ve el ejemplo del robot 2 y cómo se define cada parámetro relacionado con puertos.

Una adición importante que hay que hacerle al archivo *~/.bashrc* es la de agregarle la configuración de las variables de entorno para que se inicialice el espacio de trabajo cada que se abra una terminal del sistema. Además, hay que añadir una línea donde se exporte la variable "*ROBOT_ID*" con el

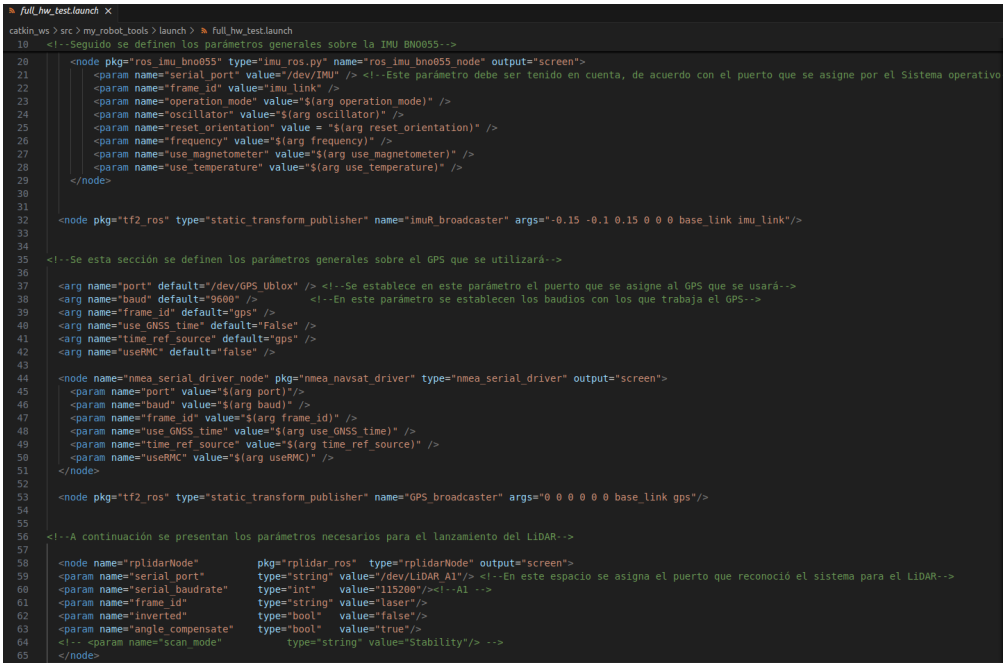

Figura 7.19: Archivo de lanzamiento de HW del robot 2 (Fuente: Elaboración propia).

valor correspondiente al número del robot, ya que esta variable es usada como parámetro en un par de códigos para diferenciar a cada robot. En la Figura 7.20 se puede ver cómo se hace esto tomando de ejemplo el robot 2.

```
source /opt/ros/melodic/setup.bash
source /home/user/catkin_ws/devel/setup.bash

export ROBOT_ID=2
```

Figura 7.20: Configuración de variables de entorno y parámetro *ROBOT_ID* (Fuente: Elaboración propia).

En cuanto al PC central y la imagen del contenedor también se deben realizar algunos cambios debido a la adición de más robots. Estos cambios se listan a continuación:

- **Agregar robot al archivo *hosts*:** Como se mencionó en la anterior subsección, el añadir los robots a este archivo facilita la forma de reconocer a cada uno ya que se usa su correspondiente hostname para realizar la conexión remota o la comunicación con los nodos del paquete multimaster.
- **Agregar hostname al argumento *robot_hosts_list*:** Una vez realizado el anterior ítem, se puede añadir el hostname del nuevo robot a este argumento que se encuentra en el archivo de lanzamiento *server.launch*, para que así el PC pueda descubrir el master del robot en la

red multimaster y sincronizar los nodos necesarios.

- **Añadir frames y demás elementos necesarios a la GUI:** Al incluir más robots en el sistema, la GUI no tendría manera de permitir controlarlos ya que está diseñada actualmente únicamente para los dos robots disponibles. Es por eso que si se quiere incluir algún otro robot al sistema de comunicación y navegación en conjunto, se deberá adaptar la GUI para darle control a más robots. El archivo de edición de la GUI por medio de la herramienta PyQt5 se encuentra en el repositorio de GitHub del proyecto, en la ruta *PC/Archivo de edición de la GUI*.

- **Añadir robot en el archivo *ui_main_window.py*:** Una vez editado el archivo de diseño de la GUI, se puede descargar la versión de dicha interfaz en forma de código de Python. De este script resultante se hace la importación de la interfáz gráfica en el archivo *manager.py*, el cual es el que genera el nodo de la interfaz gráfica. Normalmente el nombre de este archivo es *ui_main_window.py*. Al final de este archivo se encuentra algo como lo que se ve en la Figura 7.21.

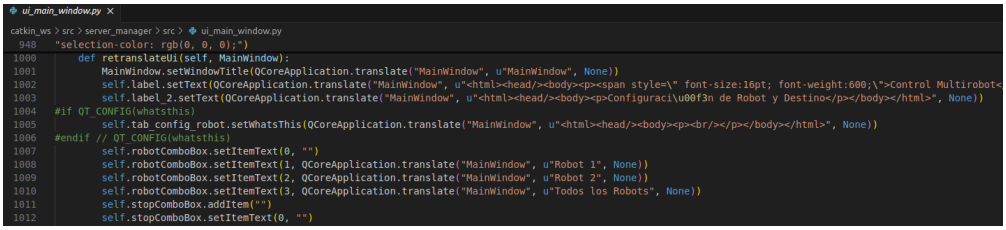

Figura 7.21: Archivo *ui_main_window.py* (Fuente: Elaboración propia).

En esta parte del código se debe añadir otra línea, como por ejemplo:
self.robotComboBox.setItemText(3, QCoreApplication.translate("MainWindow", \ uRobot 3", None))
De esta manera se añade un tercer robot al ComboBox de selección de robot en la GUI. Esto se debe hacer sucesivamente para cada robot que se vaya incluyendo en el sistema. Asímismo se debe ir incrementando el número de la línea de "Todos los Robots"por cada robot, es decir, en este caso pasaría de 3 a 4, para que la lista quede en orden.

- **Adaptar script *manager.py*:** Este script es el que contiene toda la lógica detrás de la GUI. En varias partes a lo largo del script se emplea la lógica solamente para los dos robots disponibles en el momento, así que si se quieren incluir más robots se deberá amoldar la lógica para más robots. Específicamente estas partes son:
- La inicialización de los widgets correspondientes al nuevo robot en la GUI.
- Conectar la señal de los botones del nuevo robot a las funciones correspondientes.
- Inicializar la variable GPS del nuevo robot.
- Inicializar el estado de navegación del nuevo robot.

- Definir los servicios *launch* y *cancel* del nuevo robot.
- Definir el *rospy.Subscriber* correspondiente al nuevo robot.
- Definir el método para obtener la ubicación actual del nuevo robot.
- Añadir los cambios en los frames de la interfaz al incluir el nuevo robot en el método *selectionChanged()*.
- Añadir la lógica para mandar una orden de navegación al nuevo robot y a todos en conjunto en el método *send_ data()*.
- Añadir la lógica de finalización del hilo del nuevo robot en el método *handle_ thread_ finish(thread)*.
- Añadir la lógica de cancelar la navegación para el nuevo robot y para todos en conjunto en el método *stop_ thread_ navigation()*.
- Añadir la lógica para comparar distancias incluyendo al nuevo robot en el método *mas_ cercano()*.
- Incluir la lógica para añadir un nuevo lugar de acuerdo a la posición del nuevo robot y la línea para agregar los lugares añadidos al ComboBox de lugares del nuevo robot en el método *add_ lugar(robot)*.
- Incluir el método para remover lugares correspondiente al nuevo robot, como podría serlo *remove_ lugar_ 3()*.

Con todos estos cambios se adaptaría el script y la GUI para funcionar con más robots. Parece demandar mucho, pero en realidad la gran mayoría es copiar y pegar partes del mismo código, solo que configurándolas para corresponder al nuevo robot.

- **Adaptar script *index.html*:** De manera muy similar al anterior punto, se debe añadir la lógica correspondiente a un nuevo robot en este archivo, el cual es el encargado de generar el mapa y los elementos que lo incluyen, como lo son los marcadores y las trayectorias. Incluir esta lógica también es fácil ya que solo se debe reutilizar el mismo código presente pero con identificación de un nuevo robot.

Finalmente después de realizar todos estos cambios, el sistema de comunicación para la aplicación de navegación conjunta estará listo para funcionar con más robots.

Bibliografía

[1] Leaflet. (2024) Leaflet: an open-source JavaScript library for mobile-friendly interactive maps. Accessed: 2024-09-16. [Online]. Available: https://leafletjs.com/

[2] IMB. (2024) ¿Qué es LiDAR? Accessed: 2024-09-16. [Online]. Available: https://www.ibm.com/mx-es/topics/lidar

[3] CloudFlare. (2024) ¿Qué es el modelo OSI? Accessed: 2024-09-16. [Online]. Available: https://www.cloudflare.com/es-es/learning/ddos/glossary/open-systems-interconnection-model-osi/

[4] U. O. de catalunya (UOC), "Datagrama udp," https://cv.uoc.edu/UOC/a/moduls/90/90_329/web/imagenes/f6_17.gif, 2024, accedido el 16 de septiembre de 2024.

[5] U. O. de Catalunya (UOC), "Segmento tcp," https://cv.uoc.edu/UOC/a/moduls/90/90_329/web/imagenes/f6_18.gif, 2024, accedido el 16 de septiembre de 2024.

[6] S. Spain, "Servicios ROS," https://149591262.v2.pressablecdn.com/wp-content/uploads/2022/06/ROS2-1024x468.png, 2024, accedido el 16 de septiembre de 2024.

[7] R. Wiki, "actionlibDetailedDescription," http://wiki.ros.org/actionlib/DetailedDescription, 2024, accedido el 16 de septiembre de 2024.

[8] Oracle, "¿Qué es Docker?" https://www.oracle.com/co/cloud/cloud-native/container-registry/what-is-docker/, 2024, imagen tomada del artículo.

[9] E. Castaño and J. S. Solarte, "Actualización del sistema de navegación para exteriores de un robot móvil terrestre usando robot operating system," Colombia, Cali, 2022.

[10] SLAMTEC, "RPLIDAR A1," https://www.slamtec.com/en/Lidar/A1/, 2024, fecha de Acceso: 2024/09/10.

[11] T. V. Fargo, "Power Bank Carga Rapida Tipo C 20000 Mah Reales Qc 3.0 45 W," https://www.tiendavirtualfargo.com/MCO-881740815-power-bank-carga-rapida-tipo-c-20000-mah-reales-qc-30-45-w-_JM, 2024, fecha de Acceso: 2024/08/12.

[12] Y. U. Cao, A. S. Fukunaga, and A. B. Kahng, "Cooperative mobile robotics: Antecedents and directions," *Autonomous Robots*, vol. 4, no. 1, pp. 7–27, 1997.

[13] W. Ren and R. W. Beard, *Distributed Consensus in Multi-Vehicle Cooperative Control*. London: Springer-Verlag, 2008.

[14] J. Orr and A. Dutta, "Multi-agent deep reinforcement learning for multi-robot applications: A survey," *Sensors*, vol. 23, no. 7, 2023.

[15] G. Kyprianou, L. Doitsidis, and S. A. Chatzichristofis, "Towards the achievement of path planning with multi-robot systems in dynamic environments," *Journal of Intelligent Robotic Systems*, vol. 104, no. 15, 2021.

[16] G. Ferrer, A. Garrell, and A. Sanfeliu, "Robot companion: A social-force based approach with human awareness-navigation in crowded environments," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2013, pp. 1688–1694.

[17] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *IEEE Communications Magazine*, vol. 40, no. 8, pp. 102–114, 2002.

[18] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, and et al., "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, vol. 3, no. 3.2, 2009, p. 5.

[19] J. Bohren and S. Cousins, "The SMACH high-level executive [ROS news]," *IEEE Robotics Automation Magazine*, vol. 17, no. 4, pp. 18–20, 2010.

[20] M. Coppola, P. Cataldi, G. Manes, and A. Mecocci, "Merging ros-based multi-robot systems and multi-agent systems: a case study in inspection robotics," *Robotics and Autonomous Systems*, vol. 100, pp. 101–113, 2018.

[21] T. Niemueller, G. Lakemeyer, and S. Srinivasa, "A generic robot database and its application in fault analysis and performance evaluation," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2012, pp. 364–369.

[22] M. Intelligence. (2024) Mordor intelligence. [Online]. Available: https://www.mordorintelligence.com/industry-reports/robotics-market

[23] J. J. Roldán-Gómez, J. de León Rivas, P. García-Aunón, and A. Barrientos, "Una revisión de los sistemas multi-robot: Desafíos actuales para los operadores y nuevos desarrollos de interfaces," *RIAI. Revista Iberoamericana de Automática e Informática Industrial*, vol. 17, no. 3, pp. 294–305, 2020. [Online]. Available: https://doi.org/10.4995/riai.2020.13100

[24] C. Juárez. (2021) The logistic world. [Online]. Available: https://thelogisticsworld.com/logistica-y-distribucion/centros-de-distribucion-de-amazon-vistazo-a-su-logistica-en-un-recorrido-virtual/

[25] M. Intelligence. (2021) Agricultural robots market - growth, trends, and forecasts.

[26] J. Banfi, A. Q. Li, N. Basilico, I. Rekleitis, and F. Amigoni, "Multirobot online construction of communication maps," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 2577–2583.

[27] S. Zaman, N. U. Haq, M. I. Gul, and A. Habib, "Robotic navigation based on logic-based planning," in *2017 International Conference on Communication, Computing and Digital Systems (C-CODE)*, 2017, pp. 396–401.

[28] Z. Ma, L. Zhu, P. Wang, and Y. Zhao, "Ros-based multi-robot system simulator," in *2019 Chinese Automation Congress (CAC)*, 2019, pp. 4228–4232.

[29] D. Bozhinoski, E. Aguado, M. G. Oviedo, C. Hernandez, R. Sanz, and A. Wąsowski, "A modeling tool for reconfigurable skills in ros," in *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*, 2021, pp. 25–28.

[30] A. Dutta, S. Roy, O. P. Kreidl, and L. Bölöni, "Multi-robot information gathering for precision agriculture: Current state, scope, and challenges," *IEEE Access*, vol. 9, pp. 161 416–161 430, 2021.

[31] I. of Electrical and E. E. (IEEE), "IEEE Taxonomy: A Subset Hierarchical Display of IEEE Thesaurus Terms," 2024, disponible en: https://www.ieee.org/content/dam/ieee-org/ieee/web/org/pubs/ieee-taxonomy.pdf.

[32] A. A. Abdulla, H. Liu, N. Stoll, and K. Thurow, "A backbone-floyd hybrid path planning method for mobile robot transportation in multi-floor life science laboratories," in *2016 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, 2016, pp. 406–411.

[33] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 7th ed. Pearson, 2017.

[34] X. V. Guillén, M. F. Rosselló, E. L. Ochoa, and R. S. i Gracià, *Estructura de Redes de Computadoras*. Barcelona: Editorial UOC, 2012.

[35] "Master," http://wiki.ros.org/Master, Robot Operating System, 2024, accedido el 16 de septiembre de 2024.

[36] "Master API," http://wiki.ros.org/ROS/Master_API, Robot Operating System, 2024, accedido el 16 de septiembre de 2024.

[37] "srv," http://wiki.ros.org/srv, Robot Operating System, 2024, accedido el 16 de septiembre de 2024.

[38] "Services," http://wiki.ros.org/Services, Robot Operating System, 2024, accedido el 16 de septiembre de 2024.

[39] "Comprendiendo Servicios (Services) y Parámetros (Parameters) ROS," http://wiki.ros.org/es/ROS/Tutoriales/ComprendiendoServiciosParametros, Robot Operating System, 2024, accedido el 16 de septiembre de 2024.

[40] "rosservice," http://wiki.ros.org/rosservice, Robot Operating System, 2024, accedido el 16 de septiembre de 2024.

[41] "actionlib," https://wiki.ros.org/actionlib, Robot Operating System, 2024, accedido el 16 de septiembre de 2024.

[42] R. PyPI, "Pyqt5 5.15.11," https://pypi.org/project/PyQt5/, 2024, accedido el 16 de septiembre de 2024.

[43] "What is Docker?" https://docs.docker.com/get-started/docker-overview/, dockerdocs, 2024, accedido el 16 de septiembre de 2024.

[44] S. Susnjara and I. Smalley. (2024, junio) ¿Qué es Docker? Accedido el 17 de septiembre de 2024. [Online]. Available: https://www.ibm.com/mx-es/topics/docker

[45] S. J. Vaughan-Nichols. (2024, junio) ¿Qué es Docker y por qué es tan popular? Accedido el 17 de septiembre de 2024. [Online]. Available: https://instructorbenyblanco.wordpress.com/que-es-docker-y-por-que-es-tan-popular/

[46] P. Anggraeni, M. Mrabet, M. Defoort, and M. Djemai, "Development of a wireless communication platform for multiple-mobile robots using ros," in *2018 6th International Conference on Control Engineering Information Technology (CEIT)*, 2018, pp. 1–6.

[47] T.Alexander. (2020) Setup a ros master synchronization. [Online]. Available: https://wiki.ros.org/multimaster_fkie/Tutorials/Setup%20a%20ROS%20master%20synchronization

[48] K. Zhao and L. Ning, "Hybrid navigation method for multiple robots facing dynamic obstacles," *Tsinghua Science and Technology*, vol. 27, no. 6, pp. 894–901, 2022.

[49] R. Morita and K. Matsubara, "Dynamic binding a proper dds implementation for optimizing inter-node communication in ros2," in *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2018, pp. 246–247.

[50] V. F. Raña, "Sistema de comunicaciones DDS entre robots en ROS 2," *Universitat Autònoma de Barcelona*, 2023.

[51] T. Contributors, "Programming Multiple Robots with ROS 2," https://github.com/osrf/ros2multirobotbook, 2023, fecha de Acceso: 2023-05-30.

[52] "ROS melodic morenia," http://wiki.ros.org/melodic, WIKI_ROS, 2024, fecha de Acceso: 2024/09/10.

[53] "Ubuntu install of ROS Melodic," https://wiki.ros.org/melodic/Installation/Ubuntu, WIKI_ROS, 2024, fecha de Acceso: 2024/08/12.

[54] F. Ayres, *Teoría y problemas de Trigonometría plana y esférica*, ser. Serie de compendios Schaum. Mexico: McGraw - Hill „ 1987, 022045 PERDIDO/99.

[55] A. Correa and E. Ramírez, "Multimaster ROS," 2025, accessed: 2025-02-11. [Online]. Available: https://github.com/AlejandroCorreaRamos/Multimaster_ROS

[56] A. Correa and E. Ramírez, "Manager Multimaster ROS Docker Image," 2025, accessed: 2025-02-11. [Online]. Available: https://hub.docker.com/r/alejocr18/manager_multimaster_ros/tags