



GRADO EN INGENIERÍA DE ROBÓTICA SOFTWARE

Escuela de Ingeniería de Fuenlabrada

Curso académico 2023-2024

Trabajo Fin de Grado

Programación de flujo de
datos en multirobótica

Tutor: Julio Vega Pérez

Autor: Unai Sanz Conejo



Este trabajo se distribuye bajo los términos de la licencia internacional CC BY-NC-SA International License (Creative Commons AttributionNonCommercial-ShareAlike 4.0). Usted es libre de *(a) compartir*: copiar y redistribuir el material en cualquier medio o formato; y *(b) adaptar*: remezclar, transformar y crear a partir del material. El licenciador no puede revocar estas libertades mientras cumpla con los términos de la licencia:

- *Atribución.* Usted debe dar crédito de manera adecuada, brindar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo de la licenciatante.
- *No comercial.* Usted no puede hacer uso del material con propósitos comerciales.
- *Compartir igual.* Si remezcla, transforma o crea a partir del material, debe distribuir su contribución bajo la misma licencia del original.

Agradecimientos

Quisiera expresar mi más sincero agradecimiento a todas las personas que contribuyeron a la realización de este trabajo. En primer lugar, agradezco al equipo de ZettaScale por darme la oportunidad de realizar mis prácticas con ellos y de aprender incontables aspectos acerca de las telecomunicaciones y por su persistente ayuda. También agradezco a mi tutor de TFG por su orientación constante y su gran paciencia a lo largo de este proceso.

Además estoy profundamente agradecido a mis compañeros de clase por sus ideas y debates constructivos, que han enriquecido enormemente mi investigación.

No puedo dejar de agradecer a mis padres, hermana, amigos cercanos y pareja por sus ideas, consejos y apoyo moral e incondicional.

Por último, pero no menos importante, quiero expresar mi gratitud a todas las fuentes y recursos que consulté durante la elaboración de este trabajo, así como a cualquier institución o persona que haya contribuido de alguna manera, aunque indirecta, a este proyecto.

Sin el apoyo de todas estas personas y entidades, este trabajo no habría sido posible. Gracias de todo corazón.

*A mi abuelo;
que estaría sumamente orgulloso de mi trabajo.*

Madrid, 31 de Mayo de 2024

Unai Sanz Conejo

Resumen

La robótica es un campo amplio que ha sido desarrollado con el objetivo de mejorar la calidad de vida de las personas en numerosos ámbitos. Existe una gran variedad de robots, cada uno dedicado a un propósito específico, en el que generalmente igualan o superan el rendimiento humano, incluso trabajando continuamente sin necesidad de descanso y eliminando riesgos.

En el ámbito educativo y con la creciente tendencia a incluir la robótica en el itinerario formativo, ROS es el software mayormente utilizado pero debido a su dificultad, resulta difícil de aprender, lo que a su vez genera una brecha en la educación en este campo. Otro de los grandes problemas de este software reside en la utilización de DDS, un protocolo de comunicaciones que genera una gran cantidad de mensajes, que pueden saturar la red.

El presente trabajo pretende solucionar estos problemas, utilizando plataformas *hardware* como los robots Turtlebot 2 y 4, y herramientas *software* como Zenoh-Flow, generando un entorno de programación de flujos de datos compatible con nodos existentes de ROS2, lo que lo hace más accesible a los estudiantes de este campo.

Zenoh-Flow, la principal herramienta *software* utilizada, fue enlazada con ROS2 aprovechando la capacidad de Zenoh-bridge-DDS para traducir los mensajes bidireccionalmente entre los protocolos Zenoh y DDS.

Los objetivos mencionados fueron alcanzados a través de numerosos experimentos realizados sobre una aplicación creada siguiendo el paradigma de programación de flujos de datos con Zenoh-Flow. En ella, varios robots deben buscar y acercarse a un objeto de manera organizada, dividiéndose el mapa equitativamente y optimizando trayectorias de barrido de áreas, de modo que el robot que encuentre dicho objeto, comunique su posición al resto.

Esto se logró en simulación y, parcialmente debido a errores externos, en un entorno real de laboratorio, demostrando de esta manera la viabilidad de la programación de flujos de datos en robótica y su compatibilidad con ROS2. Asimismo fue demostrada su sencillez, requisito indispensable para su aplicación en la educación robótica preuniversitaria, ayudando a reducir la brecha educativa en este ámbito.

Abstract

Robotics is a broad field that has been developed with the aim of improving the quality of life in numerous areas. There is a wide variety of robots, each dedicated to a specific purpose, generally matching or surpassing human performance, even working continuously without the need for rest and eliminating risks.

In the educational field, with the growing trend to include robotics in the curriculum, ROS is the most widely used software, but due to its difficulty, it is hard to learn, which creates an educational gap in this field. Another major problem with this software lies in the use of DDS, a communication protocol that generates a large number of messages, which can saturate the network.

This work aims to solve these problems by using hardware platforms such as the Turtlebot 2 and 4 robots, and software tools like Zenoh-Flow, creating a data flow programming environment compatible with existing ROS2 nodes, making it more accessible to students in this field.

Zenoh-Flow, the main software tool used, was linked with ROS2 by leveraging Zenoh-bridge-DDS's capability to translate messages bidirectionally between the Zenoh and DDS protocols.

The mentioned objectives were achieved through numerous experiments conducted on an application created following the data flow programming paradigm with Zenoh-Flow. In it, several robots must search for and approach an object in an organized manner, dividing the map equitably and optimizing area sweeping trajectories, so that the robot that finds the object communicates its position to the rest.

This was achieved in simulation and, partially due to external errors, in a real laboratory environment, demonstrating the viability of data flow programming in robotics and its compatibility with ROS2. Its simplicity was also demonstrated, which is an essential requirement for its application in pre-university robotics education, helping to reduce the educational gap in this field.

Acrónimos

API *Application Programming Interface*

ARM *Advanced RISC Machine*

DDS *Data Distribution Service*

GNU *GNU's Not Unix (acrónimo recursivo)*

HSV *Hue, Saturation, Value ('espacio de color')*

IMU *Inertial Measurement Unit*

JAXA *Japan Aerospace Exploration Agency*

LED *Light Emitting Diode*

LIDAR *Light Detection and Ranging / Laser Imaging Detection and Ranging*

LTS *Long-Term Support*

NASA *National Aeronautics and Space Administration*

OOP *Object Oriented Programming*

OSRF *Open Source Robotics Foundation (Open Robotics)*

PDCA *Plan, Do, Check, Act ('ciclo de desarrollo')*

RAM *Random Access Memory*

RGB *Red, Green, Blue*

RGBD *Red, Green, Blue, Depth ('color y profundidad')*

RISC *Reduced Instruction Set Computer*

RMW *ROS Middleware Interface*

ROS *Robotic Operating System*

SDF *Simulation Description Format*

SSH *Secure Shell*

STEM *Ciencia, Tecnología, Ingeniería y Matemáticas*

UPC *Universidad Politécnica de Cataluña*

URDF *Unified Robotics Description Format*

URJC *Universidad Rey Juan Carlos*

XML *eXtensible Markup Language*

Índice general

1. Introducción	1
1.1. La robótica	1
1.2. La robótica en la ciencia	2
1.3. La robótica móvil	3
1.4. La robótica educativa	5
1.5. La robótica de bajo coste	8
1.6. La robótica colaborativa	9
1.7. Flujos de datos en robótica	11
2. Objetivos	14
2.1. Descripción del problema	14
2.1.1. El problema de la congestión de red	14
2.1.2. El problema del escalón de aprendizaje en robótica	15
2.1.3. Planteamiento de la solución	15
2.2. Requisitos	16
2.3. Competencias	17
2.4. Metodología	18
2.5. Plan de trabajo	19
3. Plataforma de desarrollo	21
3.1. Hardware	21
3.1.1. Turtlebot 2	22
3.1.2. Turtlebot 4	23
3.1.3. Ordenadores de a bordo	24
3.1.4. Ordenador principal	25
3.1.5. Router	25
3.2. Software	26
3.2.1. Sistema Operativo	26
3.2.2. Lenguaje de programación	27

3.2.3. Middleware Robótico	27
3.2.4. Simulación	28
3.2.5. Protocolos de comunicación	29
3.2.6. Visión Artificial	30
3.2.7. Software de navegación	32
3.2.8. Software matemático	32
3.2.9. Visualización de datos	33
4. Arquitectura software	35
4.1. Topología hardware	35
4.2. Topología software	36
4.2.1. Zenoh-bridge-DDS	37
4.2.2. Zenoh-Flow	37
4.2.3. Flujos de datos con Zenoh-Flow y ROS2	42
5. Experimentos	45
5.1. Bases y desarrollo del proyecto	45
5.1.1. Aplicación Swarm Object Finder	45
5.1.2. Actualización de Swarm Object Finder	46
5.1.3. Mejoras de Swarm Object Finder	47
5.2. Pruebas en simulación	50
5.3. Pruebas en un entorno real	52
5.3.1. Pruebas de telecomunicaciones	53
5.3.2. Pruebas de la aplicación	56
6. Conclusiones	63
6.1. Objetivos cumplidos	63
6.2. Competencias adquiridas	65
6.3. Líneas futuras	66
Bibliografía	67

Índice de figuras

1.1.	Rover Perseverance e Ingenuity de la NASA en Marte [Rankin, 2021]	3
1.2.	Robots Spot (frente) y Atlas (fondo) [Haridy, 2020]	4
1.3.	Placa Arduino UNO [JotaCartas, 2011]	6
1.4.	Raspberry Pi 4, modelo B [Batocera Team, 2022]	6
1.5.	Código de Arduino en Scratch [Mattruffoni, 2023]	7
1.6.	Arquitecturas de ROS y ROS2 [Auledas, 2024]	8
1.7.	Múltiples robots navegando en conjunto [Snape et al., 2021]	11
1.8.	Robots educativos Turtlebot 2 (arriba) y Turtlebot 4 (abajo)	12
1.9.	Flujo de datos de un robot para seguir códigos QR	13
1.10.	Comparación entre flujo de datos y diseño de robot	13
2.1.	Esquema del desarrollo software iterativo	18
3.1.	Modelos del robot Turtlebot [OSRF, s f]	22
3.2.	Base Kobuki [ROS ORG, s f]	23
3.3.	Turtlebot 2 [OSRF, s f]	23
3.4.	Turtlebot 4 <i>Lite</i> (izquierda) y <i>Standard</i> (derecha) [OSRF, s f]	24
3.5.	Ordenador portátil HP, modelo ProBook 450 G6 [You, 2019]	25
3.6.	Router ASUS, modelo ROG Rapture GT-AXE16000 [Ludlow, 2023] . .	26
3.7.	Simulación en Gazebo del Turtlebot 3 en un mundo virtual [Yuhong, 2022]	28
3.8.	Rendimiento de distintos protocolos [Eclipse and ZettaScale, 2023] . . .	30
3.9.	Espacio de color HSV [BuckyBall, 2006]	31
3.10.	Representación de imagen RGB como matriz [Wiki300, 2011]	32
3.11.	<i>Sliders</i> de un filtro de color con OpenCV	34
4.1.	Topología de red centralizada	36
4.2.	Flujo de datos de ejemplo en Zenoh-Flow	39
4.3.	Topología software con ROS (DDS) y Zenoh-Flow (Zenoh)	44
5.1.	Flujo de datos de la primera versión de <i>swarm-obj-finder</i>	46

5.2.	Modelos 3D del robot antes (izq.) y después (dcha.) de modificar el sensor	51
5.3.	Inferencia de las coordenadas del objeto detectado	52
5.4.	Representación de las diferentes versiones del nodo Navigator	53
5.5.	Comparación de mensajes variando la cantidad de robots	57
5.6.	Detector de círculos en simulación	59
5.7.	Detección de círculos en el laboratorio	59
5.8.	Espacio de $2 \times 2 \text{m}^2$ (aprox.)	61
5.9.	Espacio de $6 \times 6 \text{m}^2$ (aprox.)	61
5.10.	Detección en espacio de $2 \times 2 \text{m}^2$ (aprox.)	62
5.11.	Detección en espacio de $6 \times 6 \text{m}^2$ (aprox.) con rango máx. de 2m	62

Listado de códigos

3.1. Función para calcular transformadas	33
4.1. Definición de flujo de datos en Zenoh-Flow	38
4.2. Fichero de descriptor de un nodo de Zenoh-Flow	40
4.3. Fichero de código de un nodo <code>operator</code> en Zenoh-Flow	41
4.4. Funciones para serializar y deserializar mensajes de ROS en Zenoh-Flow	43
4.5. Serializador/deserializador en los input/output de un nodo Zenoh-Flow	43
5.1. Clase del objeto <code>WorldPosition</code> del módulo <code>message_utils.py</code>	48
5.2. Funciones serializadoras del módulo <code>comms_utils.py</code>	48
5.3. Funciones del módulo <code>map_utils.py</code>	49
5.4. Funciones matemáticas del módulo <code>geom_utils.py</code>	50
5.5. Configuración del <i>router</i> de Zenoh en la Raspberry Pi	54
5.6. Configuración del <i>router</i> de Zenoh en la Raspberry Pi	55
5.7. Funciones de detección de círculos de un nodo de Zenoh-Flow	60

Listado de ecuaciones

5.1.	Cambio de ejes de coordenadas de la imagen i al mundo w	49
5.2.	Obtención de cuaterniones a partir de ángulos de Euler (RPY)	50
5.3.	Obtención de ángulos de Euler (RPY) a partir de Cuaterniones	50

Índice de cuadros

5.1. Resultados del nodo detector de círculos en un espacio de $2 \times 2 \text{m}^2$	61
5.2. Resultados del nodo detector de círculos en un espacio de $6 \times 6 \text{m}^2$	61

Capítulo 1

Introducción

El éxito es la suma de pequeños esfuerzos repetidos día tras día.

Robert Collier

1.1. La robótica

La robótica es un campo multidisciplinario que se concentra en el diseño, construcción, programación y operación de robots. Estos dispositivos electromecánicos, con frecuencia modelados antropomórficamente o zoomórficamente, están destinados a realizar tareas de manera autónoma o semiautónoma en una variedad de entornos, para lo que disponen de sensores que les proveen de información del medio que les rodea, de cierta capacidad de cómputo para tomar decisiones acerca de estos datos recolectados, y de actuadores que les permiten interactuar con el mismo y llevar a cabo dichas decisiones. Siendo así, sus capacidades y limitaciones están determinadas por su *hardware*, mientras que su inteligencia reside en su *software*.

Este gran campo de estudio e investigación ha experimentado un rápido crecimiento y expansión desde sus inicios en la década de 1950, abarcando una amplia gama de aplicaciones en la industria, la medicina, el entretenimiento y la exploración espacial, entre muchas otras, y se encuentra en constante evolución, desempeñando un papel cada vez más importante en nuestra sociedad moderna, gracias en gran medida a los incesantes avances en tecnologías como la inteligencia artificial, los sensores, los procesadores y los actuadores.

1.2. La robótica en la ciencia

La robótica ha conformado un factor crucial en la exploración espacial, y esta ha impulsado la creación de nuevas tecnologías que han sido desarrolladas exclusivamente para ciertas misiones espaciales pero que luego se han aplicado en la Tierra, mejorando la calidad de vida de las personas. Ejemplos destacados incluyen los sistemas de purificación de agua, los tejidos avanzados como la viscoelástica, los pañales y los dispositivos de imágenes médicas, como la resonancia magnética, que han proporcionado a las personas acceso a agua potable en regiones remotas, a colchones y almohadas que promueven un mejor descanso, a mayores facilidades en cuanto al cuidado de los niños y mayores, y a diagnósticos médicos precisos sin radiación nociva, lo cual ha salvado innumerables vidas. De esta manera, la investigación espacial no solo expande nuestro conocimiento del universo, sino que también beneficia directamente a la humanidad en la Tierra.

En cuanto al papel de la robótica en este campo, podemos destacar ejemplos como los resistentes robots enviados a diferentes planetas y lunas del sistema solar en busca de datos científicos. Numerosas misiones científicas lo evidencian, como la misión *Mars 2020* de la NASA¹, cuyos robots se ven ilustrados en la Figura 1.1, tomada por uno de ellos, el rover Perseverance, que depositó con éxito al segundo robot sobre la superficie marciana, el helicóptero Ingenuity, que aparece más al fondo en la imagen, y que ayudó al rover a llevar a cabo su exploración y toma de muestras. Este helicóptero estaba diseñado para realizar cinco vuelos durante un mes a modo de demostrador tecnológico, pero su misión pudo alargarse hasta casi los tres años, que cumplió el pasado 25 de enero, momento en el que termina debido a la rotura de una de sus palas, sumando entonces un total de 72 vuelos.

Para el éxito de esta misión fue clave la investigación en múltiples ámbitos, como la robótica móvil y todos los campos que esta conlleva, como pueden ser la visión artificial, la navegación o la localización; áreas clave que no solo están redefiniendo los límites de la tecnología, sino que también tienen un gran impacto en cómo la utilizamos en nuestra vida diaria; como ha sucedido, por ejemplo, con las aspiradoras robóticas o la conducción autónoma, que inevitablemente ya forman parte de nuestra sociedad.

¹<https://science.nasa.gov/mission/mars-2020-perseverance/>



Figura 1.1: Rover Perseverance e Ingenuity de la NASA en Marte [Rankin, 2021]

1.3. La robótica móvil

La robótica móvil ha emergido como un campo multidisciplinario que fusiona la ingeniería, la inteligencia artificial y múltiples ramas de la robótica y la mecatrónica para crear sistemas capaces de moverse y operar en entornos dinámicos, aprovechando áreas como la robótica de campo, la creación de mapas, la localización y la navegación con ayuda de otros campos como la visión artificial o la manipulación de objetos.

Desde sus inicios, la robótica móvil ha sido impulsada por los avances tecnológicos, permitiendo su aplicación en una amplia gama de entornos, desde la exploración espacial y submarina, hasta la logística industrial y la atención médica, siendo ya parte indispensable de nuestras vidas y mejorando la calidad de las mismas. En este contexto, la investigación en robótica móvil se centra en desarrollar sistemas autónomos capaces de navegar de manera segura y eficiente en entornos conocidos o desconocidos, adaptarse a cambios imprevistos y realizar tareas complejas de manera autónoma.

Un ejemplo representativo de este tipo de robots se puede ver en la Figura 1.2,

donde se pueden observar dos de los robots más desarrollados en el ámbito móvil, ambos de la empresa Boston Dynamics², que han demostrado una gran versatilidad en una variedad de entornos para ejecutar una amplia variedad de tareas, desde abrir puertas, pasando por transportar cargas de peso o realizar trabajos manuales, hasta incluso seguir rutinas deportivas variadas, que en muchos casos iguala o incluso supera la de los humanos.



Figura 1.2: Robots Spot (frente) y Atlas (fondo) [Haridy, 2020]

En concreto, la autolocalización de los robots juega un papel importante en la robótica móvil debido a la indispensable necesidad de conocer su ubicación exacta a la hora de navegar por el entorno, que normalmente se consigue tomando puntos de referencia gracias a los sensores y creando un modelo probabilístico de la posición del robot basado en estos datos.

Por su parte, para el correcto funcionamiento de la navegación, es indispensable conocer la posición del robot durante el movimiento del mismo, para tener la capacidad de sortear obstáculos y poder desplazar el robot al objetivo.

La robótica móvil también forma parte de campos más grandes y complejos, e incluso sienta las bases de algunos de ellos, como sucede con la multirobótica, que representa un paso adelante en la complejidad y la escala de los sistemas robóticos individuales, y permite realizar tareas que un solo robot no es capaz de hacer, o realizarlas de manera mucho más rápida o eficiente. Como ejemplo de esta ventaja, son notables ciertos trabajos realizados sobre localización con múltiples robots, como en el artículo [Trawny et al., 2009], en el que se ponen a prueba las mismas técnicas utilizadas para

²<https://bostondynamics.com/>

un solo robot y evalúan su viabilidad en un entorno unidimensional.

También se han realizado trabajos enfocados a entornos tridimensionales, como se observa en el artículo [Fox et al., 2000], en el cuál se logra una localización basada en sensores, teniendo en cuenta la posición de los demás robots y aumentando de este modo la precisión de la localización del propio robot en cuestión.

Además, este campo supone una gran oportunidad para el proceso de enseñanza-aprendizaje, como se verá en la Sección 1.4, ya que juega un papel crucial en el desarrollo de habilidades tecnológicas a la vez que en la motivación de los estudiantes, los cuales obtienen una gran sensación de realización y entusiasmo por aprender, al poder visualizar los resultados en movimiento de manera autónoma.

1.4. La robótica educativa

Debido a la creciente participación de los robots móviles en nuestras vidas diarias, como se ha hecho notar en el caso de las aspiradoras robóticas o el ensamblaje automático de piezas en las cadenas de montaje, la relevancia de la robótica en el ámbito educativo ha ido ganando terreno en los últimos años, tanto en la comunidad europea como en otros muchos países. Esto ha dado lugar a un aumento en la implementación de programas educativos las cuales incluyen actividades prácticas de robótica en escuelas de educación primaria y secundaria, promoviendo la creatividad, el pensamiento crítico y las habilidades tecnológicas entre los estudiantes, además de fomentar el trabajo en equipo y la resolución de problemas complejos, preparándolos de este modo para futuros grados o carreras relacionadas con la tecnología, cuya demanda aumenta año tras año.

En el caso de España, desde la década de los 90, se han implementado programas piloto y competiciones robóticas, como el programa Robolot³ (1992), desarrollado por la UPC, las Olimpiadas de Informática⁴ (1993), que incorporaron desafíos relacionados con la programación de robots, así como la Robocampeones⁵ (1999), organizada en sus orígenes por la URJC y en la que cada año participan más institutos o la competición RoboCupJunior⁶ (2000), ofreciendo a los estudiantes la oportunidad de diseñar, cons-

³<https://www.robolot.online/>

⁴<https://olimpiada-informatica.org/>

⁵<https://tv.urjc.es/video/579f2bd5d68b1420378b50a2>

⁶<https://junior.robocup.org/>

truir y programar robots para competir en diferentes categorías.

Desde alrededor de 2014, dependiendo de la comunidad autónoma de España, se han introducido programas y asignaturas que incluyen la robótica como parte esencial del plan de estudios, en concreto en la Comunidad de Madrid, se implantaron asignaturas relacionadas con la robótica en el itinerario formativo en la educación entre 2014 y 2015. Concretamente en la Educación Primaria en 2014, según el Decreto [Madrid, 2014], se añadió la asignatura llamada *Tecnología y recursos digitales para la mejora del aprendizaje*, en cuyo anexo III se describen los contenidos, que incluyen la programación en *Scratch*⁷. Más tarde, en 2015, conforme al Decreto [Madrid, 2015], se añadió una asignatura correspondiente al periodo de Educación Secundaria con el nombre *Tecnología, Programación y Robótica*, en cuyo anexo III se establecen los contenidos, también basados en *Scratch*, *Arduino*⁸ y la impresión 3D. Toda esta información se puede ver resumida en el apartado 2.12 del artículo [Ministerio de Educación, 2018].

Al ser necesario un contexto simple y barato para la introducción de la robótica, en educación se buscan herramientas como las placas *Arduino* (Figura 1.3) o *Raspberry Pi*⁹ (Figura 1.4), que presentan una amplia compatibilidad con distintos sensores y actuadores y brinda un entorno sencillo para aquellos que se están introduciendo en este campo y plataformas como *Scratch* para simplificar la programación gracias a su interfaz de bloques y la asociación de ideas a colores, como se muestra en la Figura 1.5.



Figura 1.3: Placa Arduino UNO
[JotaCartas, 2011]



Figura 1.4: Raspberry Pi 4, modelo B
[Batocera Team, 2022]

Las placas utilizadas en el ámbito educativo, mencionadas anteriormente, resultan

⁷<https://scratch.mit.edu/>

⁸<https://www.arduino.cc/>

⁹<https://www.raspberrypi.com/>

ideales para estos propósitos debido a su coste y simplicidad, sin embargo, también imponen ciertas limitaciones en las capacidades del robot y en la posibilidad de añadir *hardware* externo más complejo y potente (sobre todo en el caso de Arduino), como cámaras o LIDARs, siendo estos últimos sensores dedicados a la medición de distancias mediante la emisión y recepción de un láser, o actuadores como motores más potentes que a menudo requieren de mayor alimentación eléctrica de la que estas placas pueden brindar. Esto genera trabas a la propia originalidad y aprendizaje de los estudiantes, restringiendo así su creatividad, innovación y potencial de creación, una vez se han obtenido unos conocimientos básicos.

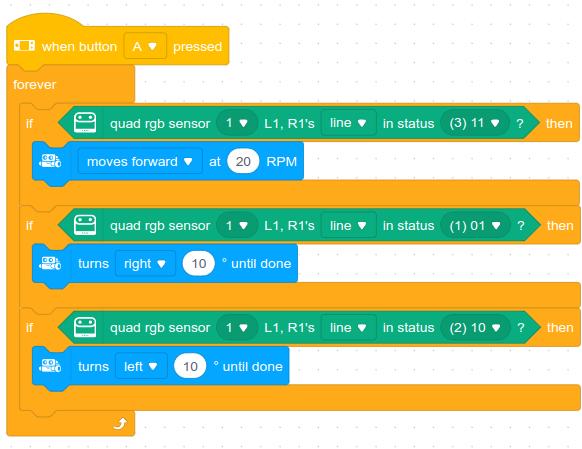


Figura 1.5: Código de Arduino en Scratch [Mattruffoni, 2023]

Para paliar las limitaciones de lenguajes básicos de programación como Scratch, existe ROS [Stanford Artificial Intelligence Laboratory, 2018], cuyas siglas significan, en inglés, sistema operativo de robots, aunque nada tiene que ver con ningún sistema operativo, siendo por el contrario el *middleware* estándar por excelencia en robótica, encargado de abstraer del hardware al programador, y permitiéndolo, por tanto, un desarrollo más ameno y compatible con una amplia gama de robots, cuya arquitectura interna puede verse apreciada en la Figura 1.6, en la que se ve comparada con la de su sucesor o versión posterior, ROS2 [Macenski et al., 2022].

Este proceso implica un considerable escalón de aprendizaje, ya que no solo se debe dominar un lenguaje de programación más complejo, sino que también se debe comprender el entorno que rodea a esta plataforma, en el que se incluyen campos de la robótica como son las comunicaciones, la arquitectura *software*, la programación modular y orientada a objetos, algoritmos y estructuras de datos, entre otros muchos,

y que suelen conllevar decenas de asignaturas con identidad propia en cualquier grado universitario. La teoría de la existencia de una brecha educativa en este ámbito se ve respaldada por trabajos como la tesis doctoral de [Vega, 2018], en cuyas secciones A.3 y A.4, se analiza esta misma perspectiva y se propone una solución respectivamente.

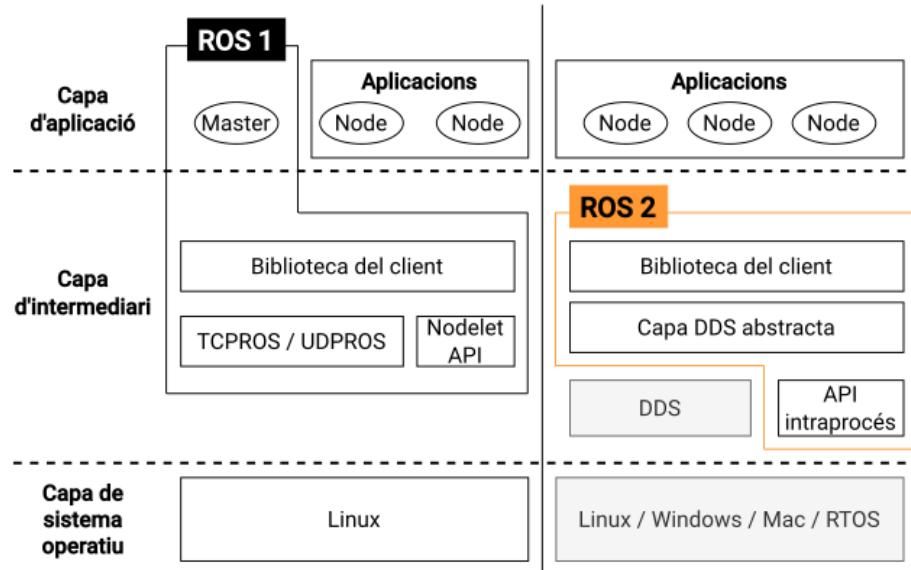


Figura 1.6: Arquitecturas de ROS y ROS2 [Auledas, 2024]

Por este motivo, resulta evidente la necesidad de un paso intermedio que pueda actuar como puente entre estos dos niveles de aprendizaje, el cual podría ser incorporado, por ejemplo, en el programa educativo de la etapa de Educación Secundaria o Bachillerato. Este nivel intermedio facilitaría la transición entre esta etapa educativa y la universitaria, concretamente dentro del ámbito científico-tecnológico.

1.5. La robótica de bajo coste

La robótica de bajo coste se refiere al desarrollo e implementación de sistemas robóticos, como los descritos anteriormente, utilizando componentes y recursos económicos, con el objetivo de hacer la tecnología robótica más accesible y asequible para una amplia gama de aplicaciones y usuarios. Este enfoque busca reducir los costes asociados con la construcción y operación de robots, empleando materiales económicos, hardware de bajo coste y técnicas de fabricación eficientes.

En el contexto de la robótica móvil, los sistemas de bajo coste pueden ofrecer soluciones viables para aplicaciones con presupuestos limitados o despliegues a gran escala,

abarcando un papel crucial en áreas como la educación, la investigación académica, la asistencia social y la exploración de entornos remotos o peligrosos. Además de su utilidad práctica, la robótica de bajo coste también promueve la innovación y el desarrollo de nuevas tecnologías al proporcionar una plataforma accesible para la experimentación y la creatividad abierta a una amplia comunidad.

Un ejemplo destacado de este tipo de robótica es el robot Sora-Q¹⁰, enviado a la Luna recientemente por la JAXA¹¹ y desarrollado por la juguetería japonesa Takara Tomy¹², que se mueve por la superficie lunar arrastrándose al rotar sus dos piezas semiesféricas, lo que le da el aspecto y tamaño de una bola de béisbol. Dicho robot, tras completar su misión, fue comercializado por 150€, hito que ilustra cómo la tecnología robótica puede volverse accesible para un público más amplio, incluso después de su participación en misiones espaciales.

De entre las distintas áreas en que se aplica este enfoque, cabe destacar la robótica educativa, que suele basarse en este tipo de sistemas de bajo coste, ya que las instituciones educativas enfrentan limitaciones presupuestarias que dificultan la adquisición de sistemas más costosos, por el presupuesto limitado y por el gran número de alumnos, a los que no podrían proveer de sistemas de este calibre de otra manera; o no al menos de forma individual.

La robótica de bajo coste no solo ha democratizado el acceso a la tecnología robótica, sino que también ha revolucionado la forma en que se enseña la robótica en las escuelas. Este enfoque económico ha permitido a las instituciones educativas superar las limitaciones presupuestarias y proporcionar a un mayor número de estudiantes la oportunidad de involucrarse en actividades prácticas de robótica, allanando el camino para que los estudiantes se sumerjan en áreas más avanzadas de este área, como pueden ser la robótica móvil o campos estrechamente relacionados.

1.6. La robótica colaborativa

La multirobótica es un campo de investigación que estudia y desarrolla sistemas robóticos compuestos por múltiples robots que trabajan en conjunto para realizar una amplia variedad de tareas complejas, y que ha sido ampliamente estudiada en artículos

¹⁰<https://www.takaratomy.co.jp/english/products/sora-q/>

¹¹<https://global.jaxa.jp/>

¹²<https://www.takaratomy.co.jp/english/>

como [Verma and Ranga, 2021], en los que se relatan todos los aspectos de la misma, poniendo en contexto este novedoso campo.

Estos sistemas pueden dividir sus tareas, como son la exploración de entornos desconocidos o la búsqueda y rescate en áreas de difícil acceso, por lo que la colaboración entre ellos es de vital importancia, e implica aspectos como el establecimiento de comunicaciones para compartir información entre ellos y entender de un mejor modo el mundo y contexto que les rodea, la creación de mapas del entorno para poder localizarse y navegar por el mismo de manera controlada o la manipulación de objetos, muchas veces necesaria para completar el objetivo propuesto para estos sistemas. Todo ello puede verse descrito en el trabajo de [Parker, 2003], en el que se relatan con más detalle los avances logrados en estos aspectos.

La coordinación entre estos sistemas puede suponer la diferencia entre el éxito o el fracaso de su misión, por lo que también es de suma importancia, y por ello se han realizado múltiples trabajos acerca de este tema. En estos términos, los equipos de robots pueden operar eficientemente asignando roles y responsabilidades como expone el artículo [Alami et al., 1998], en el que se desarrolla un sistema de control con la menor centralización posible para estudiar la cooperación multirobot en el proyecto MARTHA¹³.

A pesar de intentar crear sistemas con la mayor descentralización posible, el trabajo anterior sigue siendo un sistema centralizado, donde un robot puede asumir roles específicos. La tendencia actual se inclina hacia sistemas directamente o casi totalmente descentralizados, donde la coordinación y la optimización son fundamentales, como se hace notar en el trabajo de [Sheng et al., 2006], en el que todos los robots actualizan su propio mapa local con la información de los demás, sin que ninguno de ellos adquiera un mayor protagonismo o importancia.

Gracias a este tipo de trabajos, se ha conseguido mejorar la eficiencia y la robustez de los sistemas robóticos, y la multirobótica se ha convertido en un campo importante de investigación. Los principios y problemas técnicos en este campo se exploran en diversos contextos, como se ilustra en la Figura 1.7, en la cual, la flota de robots está realizando pruebas de localización y navegación conjunta, actualizando sus mapas

¹³MARTHA: European ESPRIT III Project No 6668 “Multiple Autonomous Robots for Transport and Handling Applications”

utilizando la información de todos los robots.

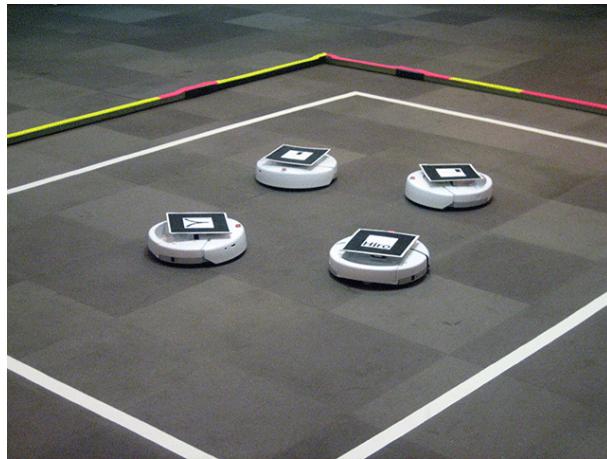


Figura 1.7: Múltiples robots navegando en conjunto [Snape et al., 2021]

En el ámbito educativo, la multirobótica ofrece una oportunidad única para involucrar a los estudiantes en actividades prácticas y colaborativas. Al trabajar con sistemas multirobot, los estudiantes no solo adquieren conocimientos sobre programación, control y mecánica de robots, sino que también exploran conceptos como son las telecomunicaciones entre robots, la coordinación, la planificación y asignación de tareas con o sin prioridades, la localización y navegación conjunta, como se ha explicado en los trabajos citados en la Sección 1.3, así como la seguridad que se requiere para evitar colisiones entre ellos, adquiriendo a su vez habilidades útiles para el trabajo en equipo. Además, este campo proporciona un entorno de aprendizaje dinámico y estimulante que despierta aún más la curiosidad y la creatividad de los estudiantes, preparándolos para enfrentar los desafíos del mundo tecnológico en constante evolución.

Un ejemplo representativo de los robots educativos, en este caso del Laboratorio de Robótica de la Universidad Rey Juan Carlos, aparece en la Figura 1.8, en la que se pueden diferenciar dos modelos distintos de robots: Turtlebot 2, en la parte superior de la imagen; y Turtlebot 4, más modernos, en la parte inferior de la misma.

1.7. Flujos de datos en robótica

Las mencionadas telecomunicaciones entre robots son fundamentales en la multirobótica, ya que garantizan una comunicación rápida, óptima, eficiente y ordenada entre



Figura 1.8: Robots educativos Turtlebot 2 (arriba) y Turtlebot 4 (abajo)

los distintos dispositivos, necesaria para un correcto desempeño de la funcionalidad en cuestión. Sin embargo, este proceso puede enfrentarse a desafíos, como la congestión de la red, y la consecuente pérdida de mensajes, que pueden ser críticos para el correcto funcionamiento de los robots. Por este motivo, resulta crucial gestionar cuidadosamente la cantidad de robots y mensajes generados, intentando minimizarlos para optimizar el rendimiento del sistema y evitando de esta manera el problema conocido como *cuello de botella*, siendo este el objetivo principal del estudio de los flujos de datos en Robótica.

Un flujo de datos consiste en un grafo dirigido de los datos que fluyen entre operaciones. Mantener un flujo de datos correcto es fundamental para solventar los problemas de telecomunicaciones mencionados en el desarrollo de sistemas de multirobótica. Además, simplifica el desarrollo del software al proporcionar una clara visión de la dirección, el sentido, el origen y el destino de los datos en cada momento. Esto permite dividir el programa en partes claramente diferenciadas, normalmente llamadas nodos, modularizándolo y dando lugar a la división del problema último en varios problemas más simples y fáciles de atajar, creando un paradigma que modela el programa como un flujo de datos. Como resultado, el desarrollo se vuelve un proceso más sostenible y escalable, y por tanto, más fácil de llevar a cabo por los estudiantes.

Este proceso se puede ver ilustrado en la Figura 1.9, donde se ilustra, de manera simplificada, el flujo de datos de un robot programado para detectar y seguir códigos QR¹⁴, en la que se puede observar cómo los datos siguen un esquema de nodos dirigido, en este caso unidireccional, pasando de su origen en el robot a su procesamiento en otra

¹⁴https://github.com/USanz/follow_beacon

máquina y acabando en su posterior vuelta al robot en forma de órdenes de movimiento, pudiendo saber en todo momento en qué proceso se encuentran dichos datos.

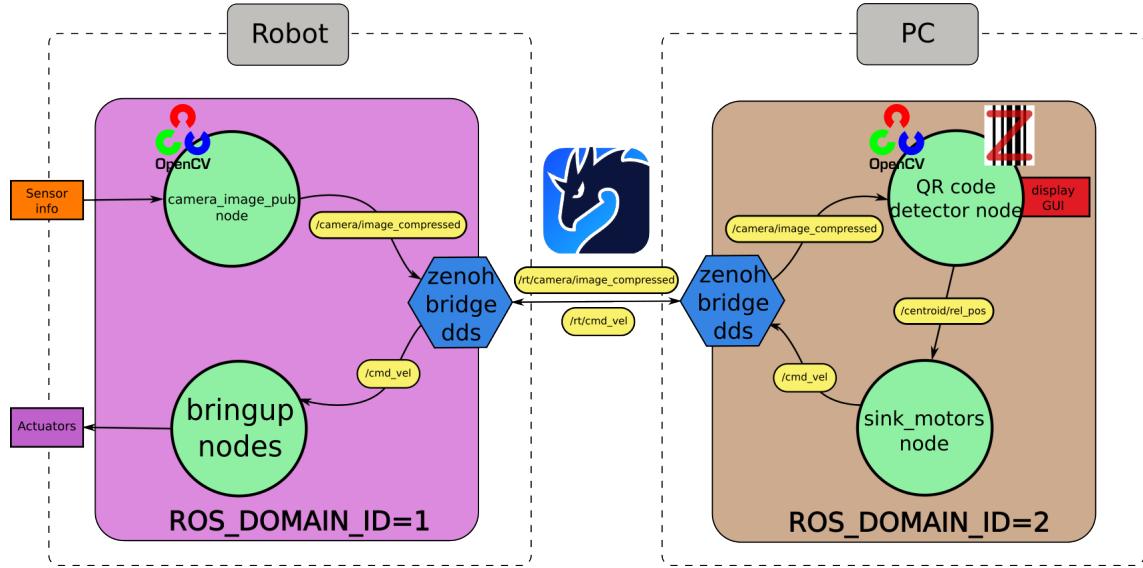


Figura 1.9: Flujo de datos de un robot para seguir códigos QR

Este proceso es equiparable a la forma de programación de un robot reactivo, ya que, como se observa en la Figura 1.10, en los flujos de datos existen tres tipos esenciales de nodos: unos, en los que se originan los datos; otros, donde se computan; y otros, donde los datos llegan a su final y que encuentran correspondencia en los nodos encargados de la percepción de un robot, del cómputo de estos datos obtenidos y de los nodos encargados de la actuación del robot, respectivamente¹⁵. Además, este tipo de programación de los robots de manera reactiva es el más simple y el primero que se suele aprender, por lo que genera una sinergia con el área educativa en este ámbito.

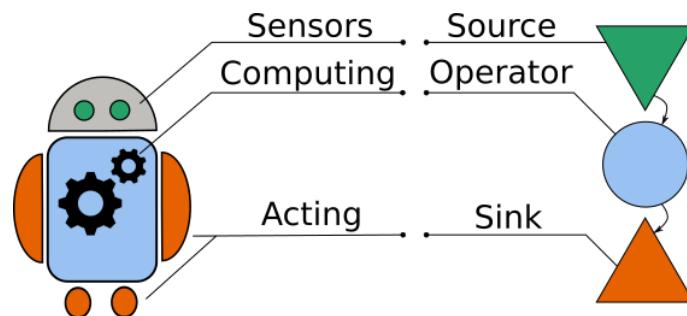


Figura 1.10: Comparación entre flujo de datos y diseño de robot

¹⁵[Ponencia conf. ROSCON Madrid 2023] <https://www.youtube.com/watch?v=ZgFHCvEFU0I>

El próximo capítulo detalla los objetivos planteados y logrados en este trabajo, ofreciendo una perspectiva más completa de su finalidad.

Capítulo 2

Objetivos

Dame seis horas para talar un árbol y pasaré las primeras cuatro afilando el hacha

Abraham Lincoln

Una vez establecido el marco contextual de este proyecto, se procederá a presentar una descripción del problema a abordar, así como el proceso creativo e intelectual que guiará el desarrollo del mismo, lo que incluirá los requisitos del proyecto, la metodología empleada y el plan de trabajo detallado.

2.1. Descripción del problema

Este proyecto surge como respuesta a la escasa investigación acerca del paradigma de programación de flujos de datos empleados en conjunto con ROS2, ofreciendo a su vez un entorno propicio para la creación sencilla de distintas aplicaciones robóticas basadas en dichos flujos de datos, que pueden replicar de manera versátil y sencilla los comportamientos reactivos de los robots, eludiendo la complejidad inherente de este *middleware* robótico (ROS2) y dando lugar, por tanto, a un nuevo entorno de programación más simple, a la vez que solucionando varios problemas expuestos a continuación.

2.1.1. El problema de la congestión de red

Con el uso de ROS2, que por defecto funciona sobre el protocolo DDS, existe un problema de congestión de red: los nodos de ROS2 generan una gran cantidad de mensajes de manera iterativa, que a su vez se transmiten a través de DDS, siendo este un protocolo de comunicaciones que, por su naturaleza, origina muchos mensajes de *Discovery*, lo que en conjunto conlleva a la congestión de la red, y dificulta de esta

manera la programación de aplicaciones multirobot.

Una solución muy sencilla a este problema consistirá en cambiar el protocolo que lo provoca por otro con mejores prestaciones. En nuestro caso se utilizará un protocolo llamado Zenoh, como ya se expondrá más adelante, en el Capítulo 3.

2.1.2. El problema del escalón de aprendizaje en robótica

La educación en robótica se basa principalmente en robots de bajo coste, como se expuso en la Sección 1.4, lo que suele limitar la capacidad del robot en cuestión y merma la cantidad de *hardware* externo compatible, así como su calidad, y consecuentemente limita la creatividad y el aprendizaje de los estudiantes. Además, siendo ROS el estándar en robótica para la programación de robots, se genera un escalón de aprendizaje, previamente expuesto en la sección referenciada, debido al gran cambio desde la programación de placas como Arduino, a la utilización tanto de *hardware*, como de *software* mucho más complejos.

Esto conlleva a una gran diferencia entre la robótica que se enseña en las etapas de la educación primaria y secundaria respecto a la etapa universitaria, y es debido precisamente, a la complejidad de código y enseñanza de ROS, para los cuales, se requiere incluso de varias asignaturas en esta última etapa. Es por este motivo, que este trabajo pretende incorporar un paso intermedio en la enseñanza, simplificando el desarrollo del *software* en ROS2, y dando la posibilidad de crear aplicaciones robóticas más complejas de una manera más simple para robots más completos, y que a su vez permanecen dentro de la categoría de robots de bajo coste, siendo asequibles para instituciones como colegios o institutos.

2.1.3. Planteamiento de la solución

La simplicidad del código de ROS2 se conseguirá gracias al uso de un *framework* llamado Zenoh-Flow, que funciona sobre el protocolo mencionado en la Sección 2.1.1, y el cuál le da nombre. Este *framework* está pensado para la programación de flujos de datos, por lo que se requerirá la previa definición de un flujo de datos a seguir por parte de los nodos, los cuales activarán su iteración al momento de recibir un dato, y generarán —en su conjunto— un flujo de datos transmitiendo los datos de nodo a nodo únicamente cuando es necesario, cualidad que reducirá en gran cantidad los mensajes generados.

La implementación conjunta con ROS2 será posible gracias a que Zenoh-Flow permite serializar los datos que se quieren enviar, y existe un *bridge* que los traduce de Zenoh a DDS y viceversa, para que los nodos de ROS2 y de Zenoh-Flow puedan entender la información que se envían. Es por este motivo que se serializarán los mensajes de la misma manera que se hace internamente en ROS2, y así se podrá seguir utilizando nodos de ROS2 existentes, como pueden ser los relativos a la navegación, lo que evitara la complejidad de su programación.

Este trabajo pretende, por tanto, solucionar el problema del escalón de aprendizaje, suponiendo un paso intermedio en la enseñanza de la robótica, así como el problema de la congestión de red generada por DDS, suponiendo una posible solución a la misma, además de presentar un entorno en el que se puede aplicar el paradigma de programación de flujos de datos en conjunto con ROS2.

Con el fin de resolver estos problemas, y cumplir los objetivos principales descritos, se detallan a continuación los siguientes subobjetivos:

1. Desarrollo de un entorno de programación que posibilite la programación de flujos de datos.
2. La solución anterior debe ser funcional en conjunto con nodos de ROS2, permitiendo la mutua comunicación entre ambos *softwares* de manera bidireccional.
3. El desarrollo de una aplicación robótica en simulación utilizando el anterior entorno de desarrollo de flujos de datos.
4. El traslado de la anterior aplicación a un entorno con robots reales.
5. Demostrar la simplicidad de código y la viabilidad de su aplicación en entornos educativos, solventando el escalón de aprendizaje descrito.
6. Disminuir la congestión de red originada por DDS.

2.2. Requisitos

Para solucionar los problemas descritos, además de cumplir los subobjetivos marcados, este trabajo deberá cumplir los siguientes requisitos:

1. Se utilizará *GNU/Linux*, con la distribución *Ubuntu 22.04 LTS* como sistema operativo en todos los *hardwares*.

2. Las herramientas *software* utilizadas deben ser compatibles para funcionar correctamente en conjunto.
3. Las aplicaciones demostrativas que se desarrolle deben ser fácilmente reproducibles y desplegables tanto en un entorno simulado como en un ambiente educativo real o de laboratorio.
4. El desarrollo del *software* debe ser lo suficientemente sencillo para poder ser llevado a cabo por alumnos preuniversitarios.
5. El *hardware* utilizado debe ser suficientemente económico para ser adquirido por cualquier institución educativa, independientemente del presupuesto del que dispongan.

2.3. Competencias

Las competencias generales que se cumplen con la realización de este trabajo de fin de grado, según la guía docente de la asignatura, son las siguientes:

1. *CB2*. Que los estudiantes sepan aplicar sus conocimientos a su trabajo o vocación de una forma profesional y posean las competencias que suelen demostrarse por medio de la elaboración y defensa de argumentos y la resolución de problemas dentro de su área de estudio. Esta competencia se cumple con la realización de la parte del *software* de este trabajo, en la que se aplican distintos conocimientos adquiridos durante el grado.
2. *CB4*. Que los estudiantes puedan transmitir información, ideas, problemas y soluciones a un público tanto especializado como no especializado. Esta competencia se adquiere al detallar todo el complejo proceso consecuente a este trabajo de manera clara y comprensible en el presente documento.
3. *CB5*. Que los estudiantes hayan desarrollado aquellas habilidades de aprendizaje necesarias para emprender estudios posteriores con un alto grado de autonomía. Esta competencia queda cumplida al adquirir los conocimientos suficientes para el desarrollo de este trabajo de manera completamente autónoma, a base de distintas pruebas y consultas en distintas fuentes: publicaciones científicas, foros de desarrollo en la web, etc.

La competencia específica *CE28* de la asignatura detalla lo siguiente: Desarrollo de las capacidades adecuadas para realizar un ejercicio original individual (o excepcionalmente colectivo), presentarlo y defenderlo ante un tribunal universitario, consistente

en un proyecto en el ámbito de las tecnologías específicas del campo de la Robótica de naturaleza profesional en el que se sinteticen e integren las competencias adquiridas en las enseñanzas. Esta última competencia se cumple con el desarrollo de este proyecto, la investigación del estado del arte en el que se enmarca, la confección de su memoria, y su defensa ante un tribunal.

2.4. Metodología

La metodología utilizada sigue pautas de investigación sobre el estado del arte previo al trabajo, y posteriormente sobre el *software* utilizado, siempre evaluando de antemano la compatibilidad con el *hardware* disponible, así como realizando pruebas pertinentes sobre su correcto funcionamiento en los distintos entornos, incluyendo la simulación y el laboratorio.

En relación con el desarrollo del *software* demostrativo se siguió un ciclo de desarrollo *software* iterativo, que consiste en la planificación del *software*, el desarrollo del mismo, su consecuente revisión mediante pruebas y su corrección, todo ello de manera periódica, generando en cada una de las iteraciones un resultado ejecutable mejor que el anterior, hasta conseguir al final una versión completamente funcional, como se ve reflejado en la Figura 2.1. Este proceso de desarrollo puede verse alineado con los principios de mejora continua del ciclo de desarrollo PDCA (*Plan, Do, Check, Act*).

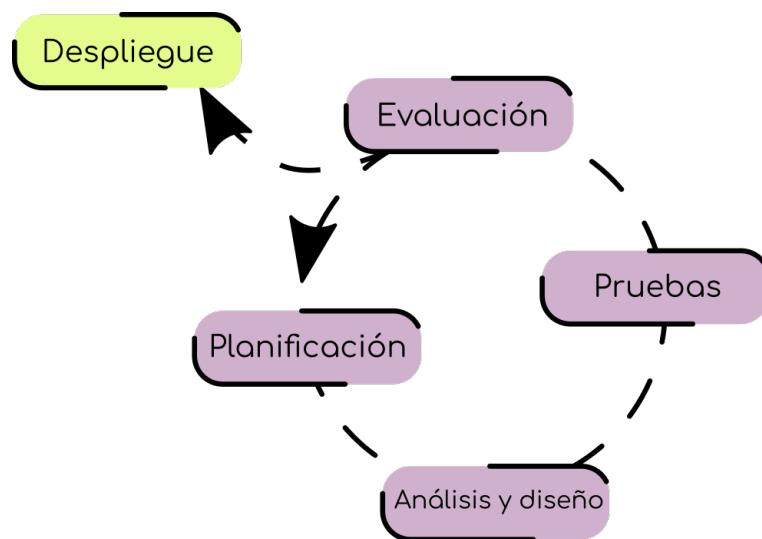


Figura 2.1: Esquema del desarrollo software iterativo

Este desarrollo implicó pruebas periódicas en simulación con el fin de identificar errores y perfeccionar los valores de los parámetros, para posteriormente evaluar su funcionamiento en un entorno real, concretamente en el Laboratorio de Robótica del Aulario 3 de la Universidad Rey Juan Carlos.

2.5. Plan de trabajo

El desarrollo del proyecto ha comprendido varias etapas, que incluyen la investigación del *software* a utilizar, la investigación del estado del arte, la implementación de una arquitectura *software* funcional en los distintos entornos y el desarrollo del *software* demostrativo o de ejemplo, comprendiendo un periodo de tiempo superior a un año, comenzando en febrero de 2023 y finalizando en junio de 2024.

1. *Investigación del software a utilizar.* Periodo de febrero a mayo de 2023 durante las prácticas de empresa, en las que estuve aprendiendo el funcionamiento de *softwares* como Zenoh, Zenoh-Flow, Zenoh-Bridge-DDS, CycloneDDS, y acerca de las telecomunicaciones entre robots, 35 horas semanales durante 4 meses.
2. *Investigación del estado del arte.* Periodo de junio a agosto de 2023, en el que se investigó acerca de los trabajos previos relacionados, y sobre la viabilidad y compatibilidad del proyecto.
3. *Implementación de una arquitectura software funcional en simulación.* Periodo de junio a agosto de 2023, en el que se consiguió un correcto funcionamiento del *software* en simulación.
4. *Desarrollo de software demostrativo.* Periodo de agosto a noviembre de 2023 en el que se migró el *software* desarrollado durante las prácticas a versiones posteriores.
5. *Implementación de una arquitectura software funcional en un entorno real.* Periodo de noviembre de 2023 a enero de 2024 en el que se consiguió un correcto funcionamiento del *software* en el laboratorio.
6. *Pruebas del software desarrollado en el laboratorio.* Periodo de noviembre de 2023 a abril de 2024 en el que se realizaron las pruebas y cambios necesarios para un correcto funcionamiento del *software* demostrativo en el entorno real del laboratorio.
7. *Escritura de la memoria.* Periodo de marzo a junio de 2024 en el que se elaboró el presente documento, así como la presentación para su defensa.

Durante los periodos de desarrollo de este proyecto fuera de las prácticas de empresa, se dedicaban aproximadamente de 30 a 40 horas semanales, manteniendo reuniones con el tutor, que generalmente se llevaban a cabo semanalmente, y ocasionalmente cada dos semanas.

Todo el proceso de trabajo se ha ido alojando en un repositorio público de GitHub¹. Asimismo, el trabajo diario se ha ido documentando detalladamente en la Wiki² de dicho repositorio, haciendo las veces de cuaderno de bitácora, donde quedan reflejados todos los contratiempos, soluciones y pruebas realizadas.

¹<https://github.com/RoboticsURJC/tfg-unai>

²<https://github.com/RoboticsURJC/tfg-unai/wiki>

Capítulo 3

Plataforma de desarrollo

Un buen artesano no se separa nunca de sus herramientas, las conoce y las elige con sabiduría.

Leonardo da Vinci

En este capítulo describe la infraestructura, tanto de hardware como de software, utilizada como base para el desarrollo y ejecución de los sistemas robóticos que se explicarán más adelante.

3.1. Hardware

Como se ha detallado en el Capítulo 1, el *hardware* constituye la infraestructura fundamental de los robots, definiendo sus capacidades operativas. Estas capacidades están intrínsecamente ligadas a los componentes físicos del robot, lo que implica que la disponibilidad y calidad del *hardware* son críticas para ejecutar cualquier tarea. Por ejemplo, la ausencia de un sistema de agarre adecuado o la limitación en su fuerza o precisión pueden comprometer la ejecución correcta de una tarea de *pick and place*.

En nuestro caso, hemos utilizado plataformas robóticas como los famosos robots de la familia Turtlebot¹, en concreto los modelos 2 y 4 *Lite*, que se pueden apreciar en la Figura 3.1, debido a su naturaleza móvil y económica, y a su disponibilidad en el laboratorio.

¹<https://www.turtlebot.com/about/>

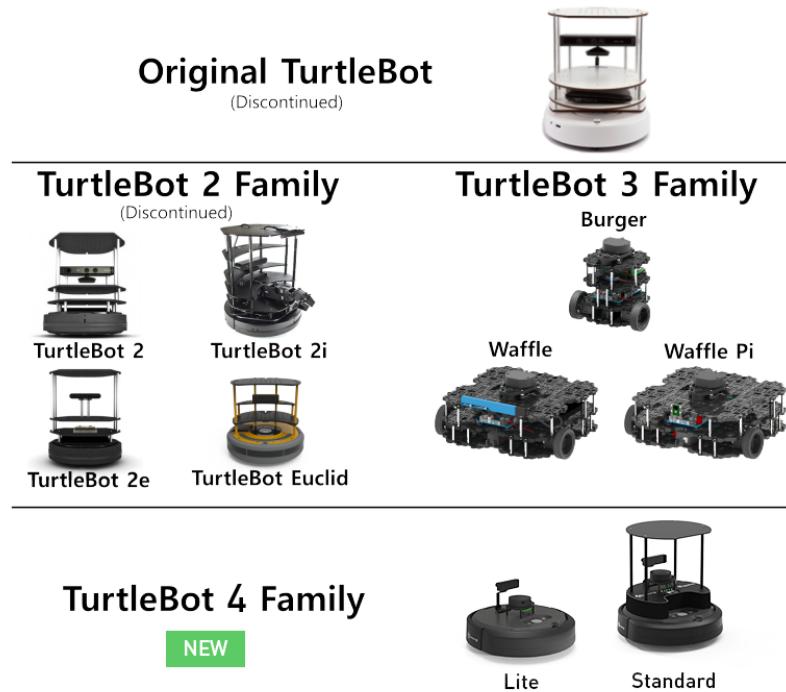


Figura 3.1: Modelos del robot Turtlebot [OSRF, s f]

3.1.1. Turtlebot 2

En concreto, el robot Turtlebot 2² se fundamenta en la base *Kobuki*, que puede verse en la Figura 3.2, y que dispone de los sensores necesarios para una navegación robusta, como puede ser la IMU o Unidad de Medida Inercial, para las correcciones en los datos de odometría y un *bumper* o sensor de colisiones. Esta base *Kobuki* también dispone de una batería que dota al robot de cierta autonomía y de actuadores que le permiten llevar a cabo dicha navegación, como son los motores y otros que le permiten transmitir información de manera visual o auditiva, como los LEDs, y el *buzzer*, con capacidad de reproducir varios sonidos.

A parte de los componentes de la base, el robot cuenta con una estructura de madera soportada por barras de aluminio que permite apoyar un ordenador portátil a modo de ordenador de a bordo, y sensores externos, cuyos modelos pueden variar dependiendo del robot utilizado en cada momento, ya que se disponen de varias configuraciones de los mismos en el laboratorio, sin afectar a la compatibilidad. Dichos sensores son: un LIDAR para la detección de obstáculos (modelos RPLIDAR³ A1 o A2), y una cámara

²<https://www.turtlebot.com/turtlebot2/>

³<https://www.slamtec.ai/product/slamtec-rplidar-s2/>



Figura 3.2: Base Kobuki [ROS ORG, s f]

RGBD, de color RGB y profundidad (modelos Orbbec⁴ Astra o Asus⁵ Xtion). Este último sensor aporta un gran potencial debido a su versatilidad en cuanto a la detección se refiere. Se puede ver una imagen del modelo en la Figura 3.3.



Figura 3.3: Turtlebot 2 [OSRF, s f]

3.1.2. Turtlebot 4

Por su parte, el robot Turtlebot 4⁶ *Lite* utilizado es bastante parecido al anterior en cuanto a los sensores de la base se refiere, correspondiendo esta vez con el modelo

⁴<https://www.orbbec.com/>

⁵<https://www.asus.com/es/>

⁶<https://www.turtlebot.com/>

Create 3 de iRobot⁷ que, a efectos prácticos, es una aspiradora robótica sin el aspirador ni las escobillas que las caracterizan, como se puede observar en la Figura 3.4.



Figura 3.4: Turtlebot 4 *Lite* (izquierda) y *Standard* (derecha) [OSRF, s f]

Este robot también añade un sensor LIDAR y una cámara RGBD, que, en el caso de nuestro laboratorio, son los modelos *RPLIDAR-S2* y *Oak-D-Pro* respectivamente, sustituyendo a los modelos originales *RPLIDAR-A1* y *Oak-D-Lite*. A diferencia de su versión estándar y del anterior robot, este no tiene una estructura de soporte, por lo que el ordenador de a bordo es una Raspberry Pi 4 Model B⁸, como la mostrada en la anterior Figura 1.4, que en nuestro caso tiene 8Gb de RAM, aunque en el modelo original es de 4Gb.

3.1.3. Ordenadores de a bordo

Como ya se ha presentado en la Sección 3.1.1, para comandar órdenes al robot se necesita un ordenador de a bordo. En nuestro caso se ha utilizado un ordenador portátil, de la marca HP, modelo *ProBook 450 G6*⁹, mostrado en la Figura 3.5.

Para algunas de las pruebas realizadas también se añadió una Raspberry Pi 4 Model B, de 4Gb de RAM, como ordenador de a bordo de uno de los robots Turtlebot 2.

⁷https://www.irobot.es/es_ES/roomba.html

⁸<https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>

⁹<https://support.hp.com/es-es/document/c06195762>



Figura 3.5: Ordenador portátil HP, modelo ProBook 450 G6 [You, 2019]

3.1.4. Ordenador principal

Para llevar a cabo este trabajo se precisó de la utilización de un ordenador con la suficiente capacidad tanto de desarrollar el *software*, como de llevar a cabo las pruebas del mismo, ejecutando los resultados tanto en un entorno simulado como en uno real. Este ordenador también se corresponde con el portátil HP mencionado en la Sección 3.1.3.

En cuanto a las pruebas realizadas con robots reales, el uso de este mismo componente a modo de ordenador principal o centralizado también fue necesario. Este ordenador portátil es el *hardware* en el que se albergó toda la capacidad de cómputo acerca de la toma de decisiones, así como el lugar de ejecución del *software* utilizado para la ejecución de la aplicación robótica creada.

3.1.5. Router

La comunicación entre los distintos robots es una parte indispensable de este trabajo, por lo que surge la necesidad de añadir un *hardware* con la capacidad de comunicar múltiples robots, en nuestro caso un router de la marca ASUS, Modelo ROG Rapture GT-AXE16000¹⁰ sin necesidad de conexión a Internet, ya que se utiliza únicamente para la comunicación local entre los distintos robots, y que se puede observar en la Figura 3.6.

¹⁰<https://rog.asus.com/es/networking/rog-rapture-gt-axe16000-model/>



Figura 3.6: Router ASUS, modelo ROG Rapture GT-AXE16000 [Ludlow, 2023]

3.2. Software

El *software* desempeña un papel crucial al convertir las capacidades del *hardware* en acciones concretas. Es el encargado de emitir las instrucciones necesarias para que el robot, en conjunto, funcione como un sistema inteligente capaz de llevar a cabo la tarea definida. En ausencia de una programación inteligente, el robot permanecerá como un dispositivo inerte, con un potencial latente pero incapaz de realizar tareas de manera autónoma.

A continuación se explicarán las distintas herramientas *software* utilizadas, desde el sistema operativo, pasando por el lenguaje de programación principal, hasta el *middleware* robótico.

3.2.1. Sistema Operativo

El sistema operativo utilizado es Ubuntu en su versión 22.04 LTS (Long Term Support o soporte a largo plazo). Además, es una distribución basada en Debian GNU/Linux, que a su vez se basa en el *software* libre y de código abierto, lo que permite una mayor participación de cualquier persona en su desarrollo, aumentando su robustez y mejorando su soporte.

Esta elección se fundamenta principalmente en la facilidad de uso y programación con el mismo, así como la compatibilidad con el resto de programas utilizados. La versión del sistema operativo utilizada coincide con la más nueva de soporte a largo plazo en la fecha de realización de este proyecto.

3.2.2. Lenguaje de programación

El lenguaje de programación utilizado para el desarrollo del *software* de este trabajo, es **Python**, ya que debido a su sencillez de programación y a su legibilidad de código, da lugar a un buen entorno para el desarrollo de *software* educativo, permitiendo una gran facilidad a la hora de aprender, además de ser de código abierto, con las respectivas ventajas que esto supone, explicadas en la Sección 3.2.1.

Python es un lenguaje de programación interpretado, lo que evita tener que ser compilado, que suele conllevar un proceso tedioso para programadores principiantes. Además, se define como lenguaje de alto nivel, ya que soporta la programación orientada a objetos (OOP), lo que también permite su escala en complejidad, si así se requiriese.

Este lenguaje ha tomado un gran impulso en los últimos años debido a su popularidad, aunque cabe destacar que es un lenguaje sumamente lento en comparación a otros más simples, y generalmente de más bajo nivel, como **C**. En concreto utilizaremos la versión 3.10.12 de este lenguaje debido a su compatibilidad y a su relevancia actual.

Este lenguaje de programación será usado directamente a la hora de programar nuestro propio *software*. Además, otros lenguajes serán utilizados indirectamente por aplicaciones o nodos externos. Por ejemplo, **C++** será utilizado en algunos nodos de ROS2, mientras que **Rust**, un lenguaje con una creciente popularidad y conocido por su seguridad y rendimiento, además de su excelente gestión de la memoria, estará presente en los programas relacionados con Zenoh.

3.2.3. Middleware Robótico

ROS2 es un *middleware* robótico creado para abstraer al programador del *hardware* del robot, permitiendo una mayor atención en el desarrollo *software* del mismo. Por todo ello, simplifica este proceso, sirviendo de herramienta tanto para la programación o desarrollo, como para el uso o la ejecución del *software* en cuestión.

En este *middleware*, el código se divide en nodos ejecutables de manera iterativa a una determinada frecuencia, que pueden comunicarse a través del protocolo DDS, ge-

neralmente con la utilización de *topics*, ya que dicho protocolo basa su funcionamiento en un modelo de publicador/suscriptor, donde los componentes del sistema se dividen en publicadores, responsables de enviar datos, y suscriptores, encargados de recibirlas. La comunicación se facilita mediante un *broker* o intermediario, que coordina la trasferencia de mensajes entre los nodos, permitiendo una comunicación flexible y organizada en temas anteriormente llamados *topics*, y sin necesidad requerir que los publicadores y suscriptores se conozcan directamente. La librería cliente de ROS2 permite programar los nodos en Python o C++, aunque la amplia comunidad ha desarrollado extraoficialmente el equivalente en lenguajes como Java, C, Rust o incluso Android, entre otros.

3.2.4. Simulación

En cuanto a la simulación, el *software* utilizado es Gazebo, un simulador muy popular que permite la visualización de los robots y de los entornos, gracias a su previa descripción, generalmente mediante un archivo con formato XML (eXtensible Markup Language) o derivado, que pueden ser URDF¹¹ (Unified Robotics Description Format) y SDF¹² (Simulation Description Format) respectivamente. Puede verse un ejemplo de simulación en la Figura 3.7.

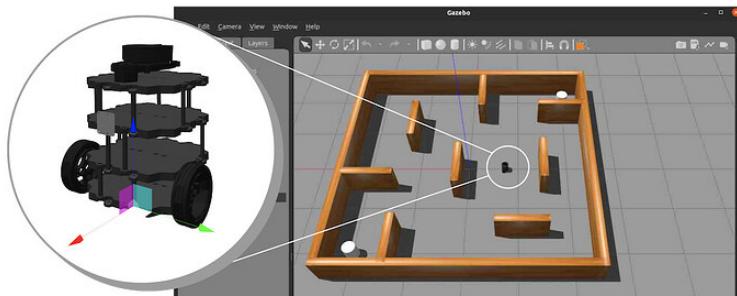


Figura 3.7: Simulación en Gazebo del Turtlebot 3 en un mundo virtual [Yuhong, 2022]

Este simulador también permite la visualización de ciertos datos de sus sensores si así se requiere, o de las partes o componentes de un robot gracias al archivo de descripción del mismo, mencionado en el párrafo anterior. Además permite el uso de *plugins* que pueden añadir las funcionalidades que sean necesarias.

¹¹<https://wiki.ros.org/urdf>

¹²<http://sdformat.org/>

3.2.5. Protocolos de comunicación

En este ámbito, una de las empresas más importantes es ZettaScale Technology¹³, que ha tomado mucha relevancia en los últimos años, ya que es responsable de una de las implementaciones de *middleware* de telecomunicaciones de DDS utilizado en ROS2, llamado CycloneDDS¹⁴, y son los desarrolladores de un reciente protocolo de comunicaciones llamado Zenoh¹⁵ [Eclipse Foundation, 2020], por sus siglas en inglés, Zero Overhead Network Protocol, que ya cuenta con importantes clientes, como la NASA, debido a las prestaciones que este ofrece, superando en la mayoría de casos a los protocolos de su misma índole y que ya ha sido oficialmente seleccionado como el próximo RMW (ROS *Middleware* o *Middleware* de comunicación de ROS) de ROS2¹⁶. El protocolo Zenoh, está basado en un modelo de publicador/suscriptor, así como DDS, por lo que hacen una buena sinergia.

Además de esto, poseen un potente *framework* para la programación de flujos de datos aún en desarrollo, llamado Zenoh-Flow¹⁷, [Eclipse Foundation, 2021] así como un puente llamado Zenoh-Bridge-DDS¹⁸ que hace las veces de traductor entre los protocolos DDS y Zenoh, lo que permite la comunicación entre estos flujos de datos con nodos de ROS2, haciendo posible la creación de aplicaciones robóticas en forma de flujos de datos, utilizando nodos de ROS2 existentes, y fusionando de esta manera ambos campos.

Es notable destacar el gran soporte que brinda esta empresa, que ayuda en gran medida a resolver los problemas o dudas que puedan surgir durante el desarrollo o despliegue de las aplicaciones.

El protocolo de comunicaciones utilizado en el *software* desarrollado en este trabajo es Zenoh, que reduce en gran medida los bytes de la cabecera de los mensajes, y ofrece mejores prestaciones en comparación con otros protocolos de su misma índole, como se puede apreciar en la gráfica de la Figura 3.8, del artículo [Liang et al., 2023] en el que se hicieron distintas pruebas, una de ellas en varias máquinas, con los protocolos Kafka, MQTT, CycloneDDS y distintas variantes de Zenoh, siendo estas últimas las

¹³<https://www.zettascale.tech/>

¹⁴<https://cyclonedds.io/>

¹⁵<https://zenoh.io/>

¹⁶<https://discourse.ros.org/t/ros-2-alternative-middleware-report/33771>

¹⁷<https://zenoh.io/blog/2023-02-10-zenoh-flow/>

¹⁸<https://github.com/eclipse-zenoh/zenoh-plugin-dds>

que mejores resultados obtuvieron, en esta como en las demás pruebas.

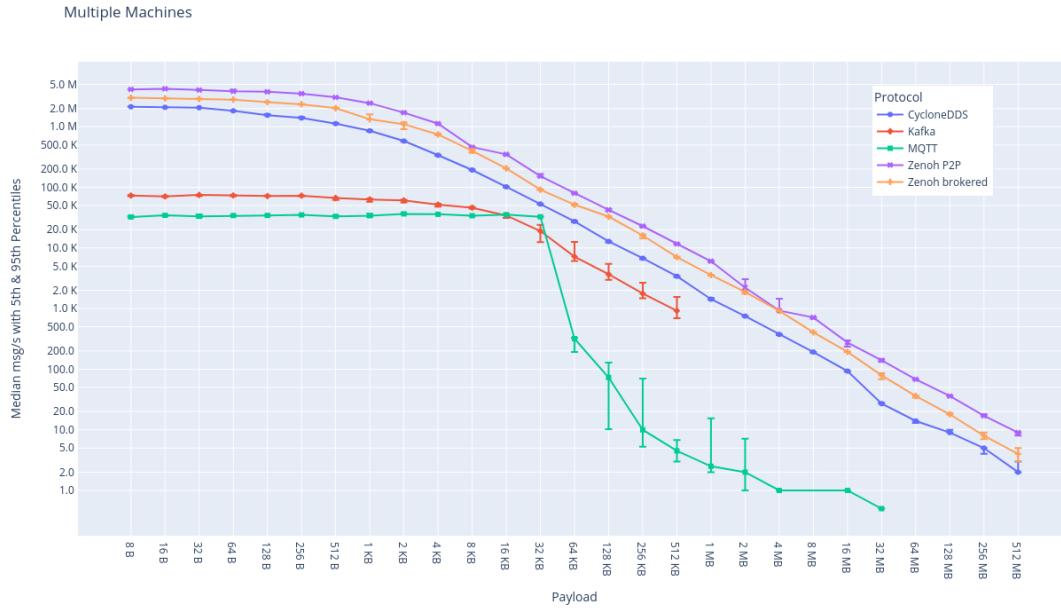


Figura 3.8: Rendimiento de distintos protocolos [Eclipse and ZettaScale, 2023]

En nuestro caso utilizaremos Zenoh-Flow para la programación de flujos de datos, ya que tiene una API (Application Programming Interface) en Python, lo que permite programarlos de manera sencilla, como se ha explicado en la Sección 3.2.2.

También se utilizará DDS indirectamente en los nodos de ROS2 de los que se hará uso, para los que utilizaremos la RMW de CycloneDDS, ya que es el que mejor funciona con el software utilizado.

Además, enlazaremos dichos nodos de ROS2 con nuestro flujo de datos mediante el uso del Zenoh-Bridge-DDS, que nos permitirá traducir los mensajes de un protocolo a otro en ambos sentidos, ya que Zenoh-Flow utiliza Zenoh para las comunicaciones, mientras que ROS2 se comunica mediante DDS.

3.2.6. Visión Artificial

Para la detección de objetos a partir de imágenes, ya sean generadas por una cámara o creadas a partir de los datos de otros sensores, se ha utilizado una reconocida

librería de procesamiento de imágenes llamada OpenCV, disponible tanto en Python como en C++.

Este software permite, entre otras muchas cosas, el cambio entre distintos espacios de color, entre los que se encuentran los formatos RGB, GreyScale o escala de grises, que permite ahorrar memoria o detectar bordes o HSV (Hue, Saturation, Value), que permite entre otros muchos usos, trabajar con los colores en distintas intensidades de luz, brindando así mejores resultados que con otros espacios de color. Este espacio de color se ve representado en la Figura 3.9.

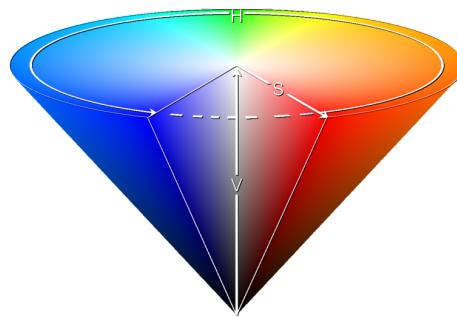


Figura 3.9: Espacio de color HSV [BuckyBall, 2006]

Esta librería también ofrece distintas funciones de detección, como pueden ser la detección de rostros, bordes o contornos, o la detección de objetos mediante aprendizaje profundo. En nuestro caso se han utilizado, tanto la detección de objetos mediante el uso de un filtro de color aplicado a la imagen obtenida desde la cámara, como la detección de bordes en conjunto con la posterior detección de círculos, en una imagen generada a partir de los datos posicionales de un sensor LIDAR, como se explicará en la Sección 5.3.2.

Esta librería también se apoya en otra, llamada Numpy, encargada de la representación matemática eficiente de matrices, así como de operaciones matemáticas entre ellas, ya que las imágenes a color, generalmente son matrices tridimensionales, que albergan tres canales (RGB o HSV), cuyas columnas tienen el tamaño de la longitud vertical de la imagen en píxeles, así como el tamaño de las filas, corresponde con el ancho de la imagen, como se puede ver en la Figura 3.10.

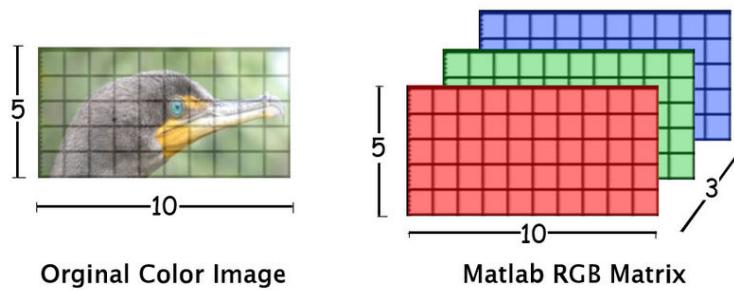


Figura 3.10: Representación de imagen RGB como matriz [Wiki300, 2011]

3.2.7. Software de navegación

El *software* de navegación utilizado para trasladar los robots de una posición a otra en el espacio se basa en los paquetes del conocido *stack* de Nav2¹⁹, que brinda herramientas que solucionan la navegación del robot de manera bastante robusta y precisa, gracias a un método de localización probabilístico basado en el algoritmo de Montecarlo. Este algoritmo consiste en distribuir una gran cantidad de puntos a lo largo del mapa del entorno, que representan posibles ubicaciones del robot, y por tanto se moverán de acuerdo al movimiento del mismo. Además, tienen una probabilidad asociada basada en los datos de los sensores, alrededor de la cual se desarrollará una nueva generación de partículas: se partirá de las más probables añadiendo una variable de aleatoriedad y se eliminarán las de menor probabilidad, acercándose de esta manera en cada iteración a la posición real del robot, incluso en entornos altamente simétricos.

3.2.8. Software matemático

Para suprir la necesidad de realizar operaciones matemáticas de todo tipo, como pueden ser las transformaciones entre espacios de coordenadas o entre ejes de coordenadas, operaciones simples o complejas con grandes cantidades de números, como matrices o imágenes, o cualquier otro tipo de operaciones, se han utilizado tanto la librería Numpy como Math, dependiendo del tipo de operación, ya que ambas son muy eficientes.

Para operaciones de transformadas (TFs) de ROS2 entre dos *frames*, utilizaremos

¹⁹<https://github.com/ros-navigation/navigation2>

las propias herramientas de ROS2, aunque no podremos utilizar cualquiera, ya que la mayoría tienen la necesidad de correr en un nodo de ROS2 y no funcionan fuera del mismo, por lo que concretamente utilizaremos la función del Código 3.1. Es por lo que se necesitará haber creado una suscripción previamente al *topic* de TFs del robot en cuestión.

```
from builtin_interfaces.msg import Duration
import tf2_ros, rclpy

buffer_core = tf2_ros.BufferCore(Duration(sec=1, nanosec=0))
buffer_core.lookup_transform_core(
    frame_id_1, frame_id_2, rclpy.time.Time()
)
```

Código 3.1: Función para calcular transformadas (TFs)

3.2.9. Visualización de datos

Para representar de manera simplificada y visual todos los datos, tanto de los sensores como de los actuadores de los robots, se utiliza RViz2²⁰, una herramienta integrada en ROS2 que permite visualizar imágenes procedentes de las cámaras, así como el propio modelo del robot en movimiento, o la posición probabilística del mismo derivada de un algoritmo de localización, como el mencionado en la Sección 3.2.7.

Además, a la hora de hacer pruebas, también se han usado otras herramientas, como la misma librería de OpenCV, para crear ventanas que muestran una imagen con *sliders* que permiten modificar variables, generando de esta manera una mayor interactividad y fluidez a la hora de encontrar los valores óptimos de ciertos parámetros, como pueden ser los valores máximos y mínimos de un filtro de color, permitiendo ver el resultado en la imagen de manera dinámica, como se puede observar en la Figura 3.11.

Asimismo, se ha utilizado software externo, como PlotJuggler²¹, para mostrar gráficas sobre datos provenientes de *topics* en directo.

²⁰<https://github.com/ros2/rviz>

²¹<https://plotjuggler.io/>

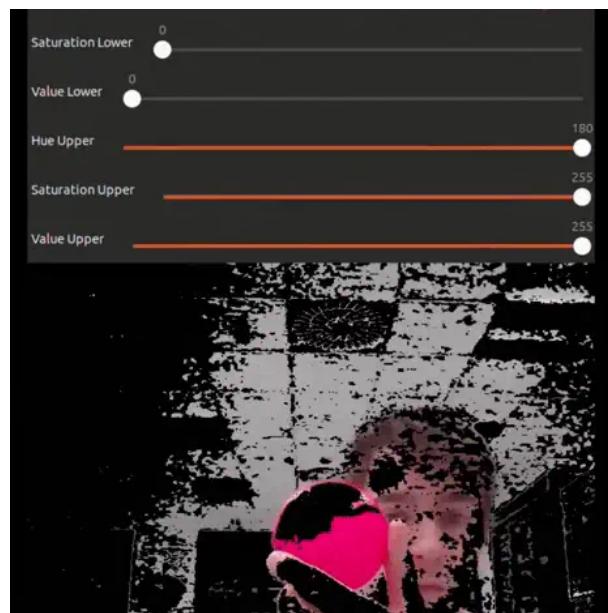


Figura 3.11: *Sliders* de un filtro de color con OpenCV

Capítulo 4

Arquitectura software

La práctica es el único camino para la maestría.

Stephen R. Covey, *Los 7 hábitos de la gente altamente efectiva*

Este capítulo describe la topología de red utilizada, tanto a nivel de *hardware* como a nivel de *software*. También se describe la manera en la que los distintos elementos *software* se complementan, detallando los aspectos más importantes de cada uno de ellos, y explicando cómo funcionan juntos.

4.1. Topología hardware

La topología de red utilizada comprende varias máquinas, debido a la naturaleza multirobótica de este trabajo, correspondiendo todas ellas, a los ordenadores de a bordo de los robots utilizados, a excepción del *router*, que será el componente encargado de establecer comunicaciones entre el resto de máquinas y encaminar así los datos que se envían entre ellas.

Comenzando con el *router*, podemos remarcar que no se ha necesitado una conexión a internet, ya que las comunicaciones se han realizado de manera local, es decir, que todos los ordenadores de abordo en los robots se comunican directamente con dicho *router*, sin necesidad de mandar datos más allá de esta red local, dado que todos se encuentran en la misma sala, al alcance de un único *router*. Asimismo este debe estar configurado de manera que permita la comunicación en ambos sentidos (entrante y saliente), y debe permitirla a través de los protocolos DDS, para los nodos de ROS2; y Zenoh, para los nodos de Zenoh-Flow.

Los ordenadores de a bordo deben tener una mínima capacidad de cómputo como para no saturarse a la hora de enviar o recibir una gran cantidad de mensajes. En nuestro caso se ha utilizado el ordenador portátil mencionado en la Sección 3.1.3. Puesto

que se necesitan más ordenadores para realizar los experimentos, y solo se dispone de un ordenador portátil, también se han utilizado las máquinas del modelo Raspberry Pi 4B mencionadas en la sección referenciada.

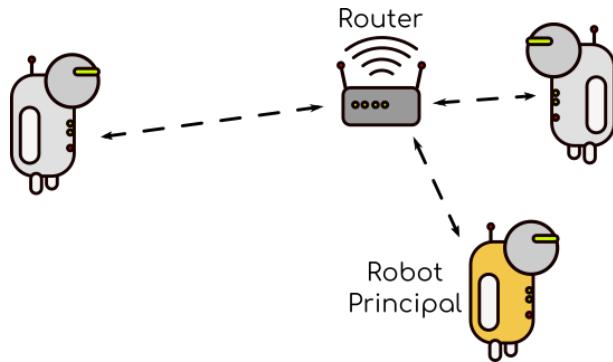


Figura 4.1: Topología de red centralizada

El ordenador portátil se ha utilizado en conjunto con el robot Turtlebot 2, mostrado en la Sección 3.1.1, mientras que las Raspberry Pi 4B es utilizada en los propios robots Turtlebot 4, mostrados en la Sección 3.1.2, a modo de ordenador de a bordo interno.

Dado que el portátil es el ordenador más potente utilizado, este deberá correr todo el software relativo a Zenoh-Flow, además de estar conectado al robot, como se puede ver representado en la Figura 4.1.

4.2. Topología software

A nivel de *software*, aparte de la configuración del *router* mencionada anteriormente, se ha necesitado ejecutar un *router* de Zenoh y un *bridge* de Zenoh en cada robot, permitiendo de esta manera las comunicaciones en ambos sentidos entre ellos y a su vez la utilización de los protocolos Zenoh y DDS al mismo tiempo. Las máquinas utilizadas también necesitan una configuración de red para poder operar con estas herramientas *software*.

Además, nuestra topología será centralizada, lo que quiere decir que una de las máquinas, llamada ordenador principal o centralizado, correrá todo el *software* desarrollado en Zenoh-Flow, por lo que deberá tener instalado dicho *software* además

del *router* y del bridge de Zenoh. La máquina elegida para este propósito será consecuentemente la que mayor capacidad de cómputo tiene, que en este caso es el portátil, como se ha mencionado en la Sección 4.1.

4.2.1. Zenoh-bridge-DDS

Como ya se ha mencionado previamente, este *software* permite traducir las comunicaciones entre Zenoh y DDS, en ambas direcciones, estableciendo un puente entre ellas, como su nombre indica. Gracias a esta propiedad del *bridge* y a la posibilidad de serializar los mensajes en el interior de los nodos de Zenoh-Flow, utilizando la misma función que se usa internamente en ROS2 para serializarlos, se puede establecer una comunicación directa entre nodos de Zenoh-Flow y de ROS2 en ambas direcciones y a pesar de utilizar distintos protocolos.

Esto nos permite seguir utilizando las funcionalidades existentes programadas en ROS2, a la vez que desarrollar código compatible en Zenoh-Flow que utilice dichas funcionalidades sin ningún impedimento, como se verá más adelante en la Sección 4.2.2.

4.2.2. Zenoh-Flow

Zenoh-Flow es un *framework* diseñado para la programación de flujos de datos, que funciona sobre el protocolo Zenoh, y que se puede comunicar a través de DDS con nodos de ROS gracias al Zenoh-bridge-DDS, como queda explicado en la Sección 4.2.1. Esta capacidad nos permite hacer uso de nodos de ROS en los flujos de datos, así como realizar aplicaciones robóticas directamente en conjunto con nodos de ROS, como se explica más adelante en esta sección.

Para la programación de flujos de datos con este *framework* es necesario la definición de un flujo de datos, en un archivo en formato `yaml`, como el que podemos ver en el ejemplo del Código 4.1 y que puede encontrarse en el repositorio oficial de ejemplos de Zenoh-Flow¹.

En el contenido de este archivo pueden verse varios apartados importantes, como son: las variables de configuración que recibirán los nodos, denominada con la etiqueta `vars`; la definición de los distintos nodos con las etiquetas `source`, `operator` o `sink`,

¹<https://github.com/ZettaScaleLabs/zenoh-flow-examples/blob/master/getting-started>

```

flow: getting-started
vars:
  BASE_DIR: "/path/to/zenoh-flow-examples/getting-started"

sources:
  - id: zenoh-sub
    configuration:
      key-expressions:
        out: zf/getting-started/hello
      descriptor: "builtin://zenoh"
operators:
  - id: greetings-maker
    descriptor: "file://{{ BASE_DIR
      }}/nodes/python/greetings-maker/greetings-maker.yaml"
sinks:
  - id: file-writer
    descriptor: "file://{{ BASE_DIR
      }}/nodes/python/file-writer/file-writer.yaml"
  - id: zenoh-writer
    configuration:
      key-expressions:
        in: zf/getting-started/greeting
    descriptor: "builtin://zenoh"

links:
  - from:
      node: zenoh-sub
      output: out
    to:
      node: greetings-maker
      input: name
  - from:
      node: greetings-maker
      output: greeting
    to:
      node: file-writer
      input: in
  - from:
      node: greetings-maker
      output: greeting
    to:
      node: zenoh-writer
      input: in

```

Código 4.1: Definición de flujo de datos en Zenoh-Flow

donde se definen las rutas a los archivos `yaml` que los describen, incluyendo sus entradas, salidas o variables de configuración, como se explicarán a continuación; y las conexiones entre ellos con la etiqueta `links`, en cuyo interior se definen las conexiones

entre las entradas y salidas de cada nodo.

Además, se puede añadir la etiqueta `mapping`, donde se especifica en qué máquina correrá cada nodo, aunque en nuestro caso no será necesario debido a que todos los nodos correrán en el mismo ordenador (el más potente), para liberar capacidad de cómputo a los demás. De esta manera, nuestra topología será centralizada.

En este ejemplo se puede ver cómo los nodos `zenoh-sub` y `zenoh-writer` contienen la línea `descriptor`: "builtin://zenoh", lo que indica que no necesitan un fichero que describa dónde se encuentra el código de dicho nodo, ya que este se encuentra directamente integrado en Zenoh-Flow, y lo que harán será recibir información de la *key-expression* `zf/geting-started/hello` y publicar información a la *key-expression* `zf/geting-started/greeting` respectivamente, como puede observarse resumidamente en la Figura 4.2.

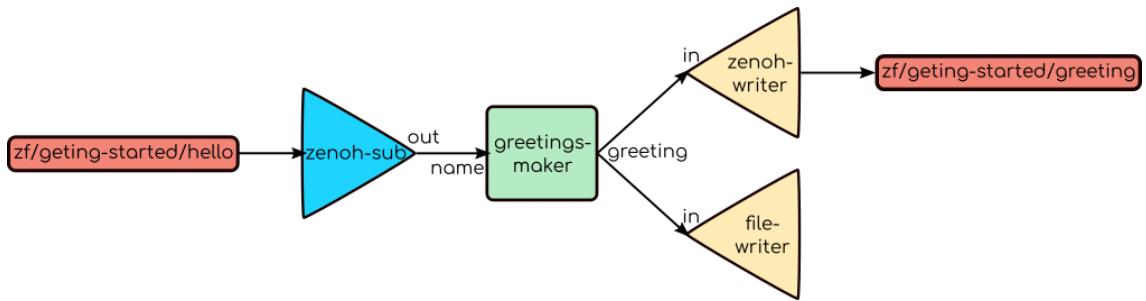


Figura 4.2: Flujo de datos de ejemplo en Zenoh-Flow

El resto de nodos deben ser programados, para lo que existe un fichero descriptor, que indicará el nombre del nodo, las variables de configuración, la ruta al archivo de código del propio nodo y los nombres de sus *inputs* y *outputs*. Continuando con el mismo ejemplo, se muestran en el Código 4.2 las propiedades mencionadas, en este caso del nodo *operator* con el nombre `greetings-maker`, que consecuentemente se encontrará en la ruta declarada en su etiqueta `descriptor`, en el anterior Código 4.1.

Las distintas partes del código de este nodo se muestran en el Código 4.3, en el que puede verse cómo se reduce a una sencilla clase que hereda de la clase del nodo en cuestión, en este caso de la clase `Operator`. Además debe existir una función externa `register()` que debe devolver la clase creada para este nodo, en nuestro caso llamada

```

id: greetings-maker
vars:
  BASE_DIR: "/path/to/zenoh-flow-examples/getting-started"
  uri: "file://{{ BASE_DIR
}}/nodes/python/greetings-maker/greetings-maker.py"
inputs: [name]
outputs: [greeting]
```

Código 4.2: Fichero de descriptor de un nodo de Zenoh-Flow

GreetingsMaker.

La programación de los nodos de Zenoh-Flow, realizada en Python, es muy parecida entre los distintos nodos, ya que su funcionamiento es muy similar: los nodos *source* solo comprenden salidas de datos, los nodos *sink* solo comprenden entradas, y los nodos *operator* comprenden ambas, ya que pueden recibir información de otros nodos *source* u *operator* y pueden enviar datos a otros nodos *operator* o *sink*, como podemos ver en el flujo de datos representado en la anterior Figura 4.2 o en la Figura 1.9 de la Sección 1.7.

La línea `outputs.take("greeting", str, lambda s: bytes(s, "utf-8"))`, en la función constructor de esta clase (`__init__()`), toma como argumentos el nombre del *output*, el tipo de mensaje, y la función que utilizará para serializar el mensaje en el momento de enviarlo, devolviendo el *output* correspondiente. Lo mismo sucede con la línea siguiente, aunque al tratarse de un *input*, esta toma como tercer argumento una función que utilizará para deserializar el mensaje en el momento en el que sea recibido. Esta línea devolverá el *input* que corresponda. Son estas funciones las que se deberán sustituir por las análogas de ROS, si se quiere enviar o recibir información de los nodos de ROS con ayuda del *bridge*, como ya se ha explicado anteriormente.

Además, en esta como en cualquier otra clase de Python se pueden declarar las variables necesarias a la hora de desarrollar el *software*. Teniendo en cuenta que la variable `configuration` es un diccionario, las distintas variables de configuración especificadas en los archivos anteriores, pueden ser obtenidas con una línea como `configuration.get(conf_var_name, default)`, en la que se deberá especificar el nombre de la variable en cuestión en forma de cadena de caracteres o `str` en el primer argumento, y será devuelto el valor asociado a dicha clave.

Dentro de esta clase existe una función llamada `finalize()` que se ejecutará antes

```

from zenoh_flow.interfaces import Operator
from zenoh_flow import Inputs, Outputs
from zenoh_flow.types import Context
from typing import Dict, Any

class GreetingsMaker(Operator):
    def __init__(
        self,
        context: Context,
        configuration: Dict[str, Any],
        inputs: Inputs,
        outputs: Outputs,
    ):
        self.output = outputs.take("greeting", str, lambda s: bytes(s,
            "utf-8"))
        self.in_stream = inputs.take("name", str, lambda buf:
            buf.decode("utf-8"))
        if self.in_stream is None:
            raise ValueError("No input 'name' found")
        if self.output is None:
            raise ValueError("No output 'greeting' found")

    def finalize(self) -> None:
        return None

    async def iteration(self) -> None:
        message = await self.in_stream.recv()
        name = message.get_data()
        if name is not None:
            greetings = self.generate_greetings(name)
            await self.output.send(greetings)
        return None

    def generate_greetings(self, name: str) -> str:
        greetings_dict = {
            "Sofia": "Ciao, {}!\n",
            "Gabriele": "Ciao, PaaS manager!\n",
        }
        greet = greetings_dict.get(name, "Hello, {}!\n")
        return greet.format(name)

    def register():
        return GreetingsMaker

```

Código 4.3: Fichero de código de un nodo operator de Zenoh-Flow

de destruir el nodo, por lo que dentro de esta función se deberá liberar memoria, cerrar archivos o ventanas que hayan sido abiertas o creadas durante la ejecución del nodo o finalizar cualquier otra funcionalidad que así lo requiera.

Por último, tenemos la función más importante, llamada `iteration()`, que deberá ser asíncrona para poder recibir mensajes, y lo hará ejecutando el método `recv()` del `input` en cuestión como se ve en la línea `await self.in_stream.recv()`, lo cual deserializará el dato y lo devolverá cuando este llegue. Dentro de esta parte del código se deberán realizar los cálculos y tomar las decisiones que se necesiten, para mandar los datos resultantes de manera similar, esta vez usando el método `send()` del `output` por el que se deseé enviar, como sucede con la línea `await self.output.send(greetings)`, con lo que, finalmente, podemos deducir la función que realizará este nodo: publicará una frase de bienvenida que variará en función del nombre que reciba. Esta función se repetirá iterativamente hasta finalizar el flujo de datos.

4.2.3. Flujos de datos con Zenoh-Flow y ROS2

Como se ha visto en la Sección 4.2.2, Zenoh-Flow divide sus posibles nodos en tres tipos: `source`, `operator` y `sink`, que podríamos trasladar a las palabras en español: `origen`, `operador` y `final`, pudiendo ver la relación de estos nodos con un grafo direccionalizado, como es un flujo de datos, en el que los datos se obtienen de los nodos origen o `source`, pasan por el operador, en el que son computados o tratados, hasta llegar al final o `sink`, donde los datos dejan de viajar entre nodos para terminar convirtiéndose en una acción o decisión del sistema. Ya se ha visto representada esta misma comparación en la Figura 1.10 de la Sección 1.7.

Para poder complementar los flujos de datos en Zenoh-Flow con las acciones de los robots, que se ejecutan en nodos de ROS2, se deben crear las piezas faltantes, para anclar ambos sistemas, uniendo sus entradas y salidas. Entre estas piezas se encuentran el serializador y el deserializador, que como se ha explicado previamente, debe ser el mismo utilizado internamente en ROS.

Esto resulta en el Código 4.4, en el que se pueden observar dos funciones: la primera, `ser_ros2_msg()`, en la que se serializa el mensaje en su primer argumento; y la segunda, `get_ros2_deserializer()`, que devuelve una función `lambda` creada en el momento y que deserializará el mensaje en función de su tipo, especificado en su primer argumento. Esto debe hacerse de esta manera ya que no podemos saber el tipo de mensaje que se recibirá, ya que estará serializado en bytes, a diferencia de cuando se envía, momento en el que se tiene el tipo de mensaje, con la ayuda de la función

`type()` de Python.

```
from rclpy.impl.implementation_singleton import rclpy_implementation as
    _rclpy
from rclpy.type_support import check_for_type_support

def ser_ros2_msg(ros2_msg: Any) -> bytes:
    check_for_type_support(type(ros2_msg))
    return _rclpy.rclpy_serialize(ros2_msg, type(ros2_msg))

def get_ros2_deserializer(ros2_type: Any): -> Callable[[bytes], Any]
    check_for_type_support(ros2_type)
    # This returns a function
    return lambda ser_obj: _rclpy.rclpy_deserialize(ser_obj, ros2_type)
```

Código 4.4: Funciones para serializar y deserializar mensajes de ROS en Zenoh-Flow

Estas dos funciones serán especificadas en los argumentos de nuestros nodos de Zenoh-Flow a la hora de obtener las entradas y salidas, definiendo de esta manera al sistema el método a seguir para serializar y deserializar los mensajes de ROS, como se puede ver en el Código 4.5.

```
from zenoh_flow.interfaces import Operator
from zenoh_flow import Input, Output
from zenoh_flow.types import Context
from typing import Dict, Any
from geometry_msgs.msg import PoseStamped
from tf2_msgs.msg import TFMessage

class Navigator(Operator):
    def __init__(
        self,
        context: Context,
        configuration: Dict[str, Any],
        inputs: Dict[str, Input],
        outputs: Dict[str, Output],
    ):
        inputs.take("TF1", TFMessage,
                   deserializer=get_ros2_deserializer(TFMessage))
        outputs.take("RobotPose1", PoseStamped, serializer=ser_ros2_msg)
        ...
    ...
```

Código 4.5: Serializador/deserializador en el input/output de un nodo Zenoh-Flow

La siguiente pieza del puzzle, consiste en configurar el *bridge* de Zenoh, para que

únicamente traduzca el tráfico deseado, lo que conseguiremos con un argumento en la línea de comandos al ejecutarlo, o bien utilizando `--allow/-a` seguido de un *string* que contenga los topics deseados (o partes de los mismos), separados por el carácter “|”, o bien utilizando `--deny/-d` de la misma manera, cuyo efecto será el contrario, traducirá todos los topics excepto los que contengan los *strings* especificados. Un ejemplo del comando de ejecución con este argumento es el siguiente:

```
. ./zenoh-bridge-dds -e tcp/<ip>:7447 --allow
    "rt/robot1/goal_pose|rt/chatter"
```

siendo el argumento `-e` y el siguiente, la configuración de conexión del *bridge*, debiéndose sustituir `<ip>` por la dirección IP a la que se quiera conectar, como se verá más adelante en el Capítulo 5. Un esquema explicativo de toda esta sección puede verse en la Figura 4.3.

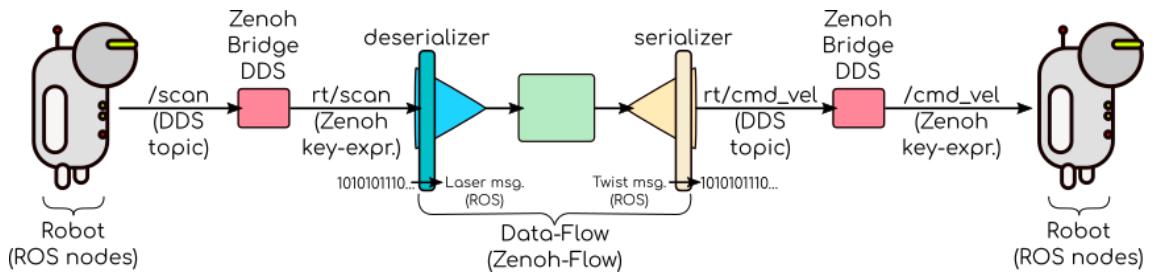


Figura 4.3: Topología software con ROS (DDS) y Zenoh-Flow (Zenoh)

Además, como podemos ver en la Figura 4.3, para que las *key-expressions* sean traducidas a topics de ROS, deben incluir al principio de las mismas la partícula `/rt/` en el campo *key-expression* del archivo de definición de flujo de datos, como el mostrado en el anterior Código 4.1.

Capítulo 5

Experimentos

No dejes que la teoría te detenga. Actúa y observa qué sucede.

Richard Branson

En este capítulo se tratarán los diferentes experimentos realizados para llevar a cabo este proyecto, desde las pruebas más básicas, hasta las más elaboradas, cuyo éxito da lugar al *software* desarrollado y a los resultados finales. Así como en las páginas o apartados de la Wiki¹, este proyecto se ha apoyado en plataformas de documentos gráficos, como YouTube, en las que se han ido actualizando vídeos de las pruebas explicadas a continuación, reunidos en una lista de reproducción².

5.1. Bases y desarrollo del proyecto

Una de las partes del periodo de prácticas de empresa consistió en probar la arquitectura *software* del proyecto que ha sido explicada en la Sección 4.2, es por ello que se parte de una base sólida y robusta por la que empezar a hacer pruebas y mejoras. Este proyecto base se aloja en un repositorio de GitHub³. El presente trabajo es una continuación de dicho trabajo.

5.1.1. Aplicación Swarm Object Finder

Esta aplicación consiste en la búsqueda y acercamiento a un objeto por múltiples robots, para demostrar la posibilidad del uso y programación de flujos de datos con Zenoh-Flow y ROS2 en conjunto.

¹<https://github.com/RoboticsURJC/tfg-unai/wiki>

²<https://www.youtube.com/watch?v=x8VAZFSAH1w&list=PL3OZgGkAPYdkHaZFa2naBTB4b5aglx1h4>

El funcionamiento interno puede verse resumido en la Figura 5.1, en la que se aprecian las distintas conexiones entre los nodos de Zenoh-Flow. Los robots comienzan dividiéndose el mapa (nodo *Paths Planner*) para recorrerlo punto por punto de manera controlada y sincronizada (nodo *Navigator*). Durante este proceso, se obtiene la información intrínseca de las cámaras, así como sus imágenes, con las que se detecta el objeto en cuestión (nodo *Object Detector*) y se infiere su posición (nodo *Object Position Inferring*), para ser enviada al nodo *Navigator*, que se encarga de hacer que los robots dejen de seguir sus rutas predeterminadas para comenzar a acercarse al objeto. Todo ello queda explicado en profundidad en la Wiki⁴.

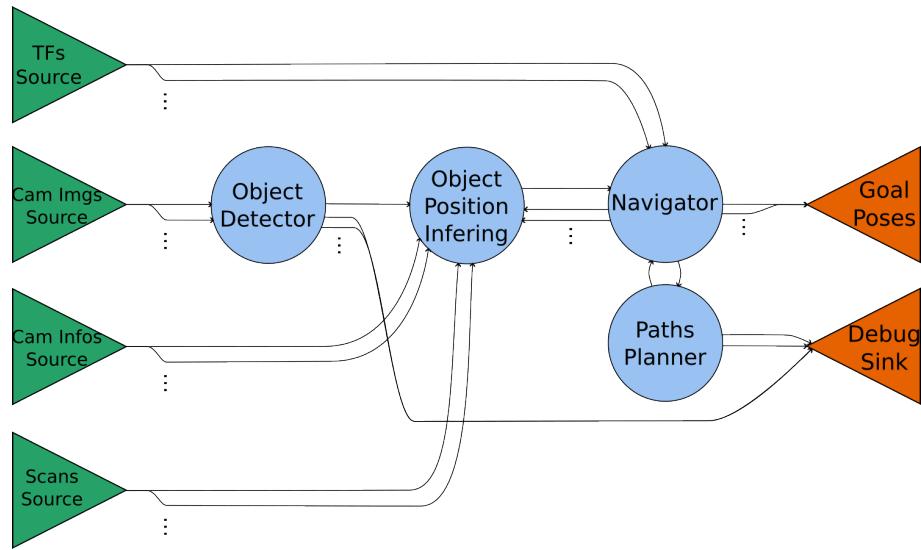


Figura 5.1: Flujo de datos de la primera versión de *swarm_obj_finder*

5.1.2. Actualización de Swarm Object Finder

Aunque el funcionamiento de esta arquitectura en simulación fue verificado durante el periodo de prácticas de empresa, el *software* en conjunto no funcionaba correctamente al completo, siendo el punto de fallo el *stack* de navegación, que momentáneamente fallaba a la hora de mover uno de los robots. Para solucionar este problema se realizaron distintas pruebas, llegando a contactar con los desarrolladores del *stack* mediante la apertura de un *Issue* en su repositorio de GitHub⁵.

Al no obtener resultados de esta manera se siguió probando hasta que finalmente se llegó a la solución, actualizando Nav2, el *stack* de navegación, de su rama “Foxy”

⁴<https://github.com/RoboticsURJC/tfg-unai/wiki/10.-Migracion/funcionamiento>

⁵<https://github.com/ros-navigation/navigation2/issues/3521>

a “Humble”, y consecuentemente actualizando ROS a esta misma distribución, lo que a su vez implicaba actualizar el sistema operativo Ubuntu de la versión 20.04 a la 22.04.

Además, se actualizó la versión de Zenoh-Flow utilizada en el repositorio original a una posterior (0.7.2-rc), en la que se solucionaban varios errores y *bugs*, además de mejorar varios apartados de la API en relación con las entradas y salidas, así como en la definición del flujo de datos, que queda explicado más profundamente en la Wiki⁶. Consecuentemente, se actualizaron la versión de Zenoh y del *bridge* de Zenoh, así como de la API de Python, a versiones compatibles con esta nueva versión de Zenoh-Flow.

5.1.3. Mejoras de Swarm Object Finder

Una vez se tuvo la aplicación funcionando de nuevo con versiones posteriores de las herramientas *software* utilizadas, se procedió a su optimización, eliminando posibles puntos de fallo y enriqueciendo su funcionalidad, aportando de esta manera mejoras a sus nodos, lo cual también queda detallado en la Wiki⁷.

La primera mejora aplicada va en relación con la creación de distintas clases con varios objetos, para modularizar el código y mejorar así las comunicaciones entre los nodos, facilitando la serialización de sus mensajes, lo que da lugar al módulo `message_utils.py`, que se puede observar en el Código 5.1. En este fragmento observamos la clase `WorldPosition`, que se utiliza entre los nodos `Navigator` y `PathsPlanner` para informarse de las posiciones objetivo de los robots en el mundo o para que el nodo `ObjPosInfer` envíe la posición en el mundo del objeto detectado al nodo `Navigator`. Así como esta clase, se crearon otras, como la llamada `CentroidMessage`, para que el nodo `ObjDetector` envíe la posición del objeto detectado al nodo `ObjPosInfer`.

Consecuentemente, se desarrolló otro módulo encargado de la serialización de estos y otros objetos para poder ser enviados entre los nodos, y que además alberga las funciones de serialización de mensajes de ROS mencionadas en la Sección 4.2.3. Este módulo se llama `comms_utils.py` y, como puede verse en el Código 5.2, serializa los mensajes en función de la longitud del `string` del nombre del emisor (“robot1”, “robot2”, etc.), variable que será un parámetro fijado en el flujo de datos, determinando esta longitud máxima para todos los nombres.

⁶https://github.com/RoboticsURJC/tfg-unai/wiki/10.-Migracion/actualizacion-swarm_obj_finder

⁷https://github.com/RoboticsURJC/tfg-unai/wiki/11.-Mejoras-y-correcciones-swarm_obj_finder

```

...
class WorldPosition:
    def __init__(self, world_pos:PoseStamped=PoseStamped(), name:str="") -> None:
        self.world_position = world_pos
        self.sender = name
    def set_world_position(self, world_pos:PoseStamped=PoseStamped()) -> None:
        self.world_position = world_pos
    def get_world_position(self) -> PoseStamped:
        return self.world_position
    def set_sender(self, name:str="") -> None:
        self.sender = name
    def get_sender(self) -> str:
        return self.sender
...

```

Código 5.1: Clase del objeto `WorldPosition` del módulo `message_utils.py`

```

...
def get_world_pos_msg_serializer(str_bytes_length: int): # Returns a function
    return lambda obj: ser_world_pos_msg(obj, str_bytes_length)
def ser_world_pos_msg(world_pos_msg: WorldPosition,
                      str_bytes_length: int) -> bytes:
    ser_sender = ser_string(world_pos_msg.sender, str_bytes_length)
    ser_world_pos = ser_ros2_msg(world_pos_msg.world_position)
    return ser_sender + ser_world_pos

def get_world_pos_msg_deserializer(str_bytes_length: int): # Returns a function
    return lambda obj: deser_world_pos_msg(obj, str_bytes_length)
def deser_world_pos_msg(world_pos_msg: bytes,
                       str_bytes_length: int) -> WorldPosition:
    ser_sender = world_pos_msg[:str_bytes_length]
    sender = deser_string(ser_sender)
    ser_world_pos = world_pos_msg[str_bytes_length:]
    world_pos = get_ros2_deserializer(PoseStamped)(ser_world_pos)
    return WorldPosition(world_pos, sender)
...

```

Código 5.2: Funciones serializadoras del módulo `comms_utils.py`

Además de estos, otros módulos fueron creados, como el módulo `map_utils.py`, el cual ayuda a simplificar y mejorar la conversión entre los distintos ejes de coordenadas, como los del mapa y los del mundo, cuyas fórmulas pueden verse en la Ecuación 5.1.

$$\begin{aligned}x_w &= x_i * r + o_x \\y_w &= (h_i - y_i) * r + o_y\end{aligned}\tag{5.1}$$

Ecuación 5.1: Cambio de ejes de coordenadas de la imagen i al mundo w

Este módulo, en el nodo *Paths Planner*, alberga funciones relacionadas con la división del mapa para la obtención de las rutas formadas por puntos objetivo en el mundo para cada robot. Algunas de estas funciones se muestran en el Código 5.3.

```

...
def get_squarest_distribution(factors: list) -> list:
    distribution = [1, 1]
    for f in reversed(factors): # Starts from the highest to the lowest.
        if distribution[0] < distribution[1]:
            # The factor is multiplied by the lowest member of the
            # distribution:
            distribution[0] *= f
        else:
            distribution[1] *= f
    return distribution
...
def img2world(img_pix: tuple, img_shape: tuple,
             res: float, origin: tuple) -> tuple:
    # Image needed info:
    xi, yi = img_pix
    hi, _ = img_shape
    # World needed info:
    ox, oy, _ = origin
    # Image pixel to world position conversion:
    xw = xi * res + ox
    yw = (hi - yi) * res + oy # Y axis needs to be inverted.
    return (xw, yw)
...

```

Código 5.3: Funciones del módulo `map_utils.py`

El módulo `geom_utils.py` se encarga de funciones matemáticas relacionadas con la geometría, albergando todos los cálculos relacionados con cuaterniones, como las que pueden verse en la Ecuación 5.2. En esta ecuación de ejemplo se está infiriendo la posición del objeto en el nodo `ObjPosInfer`, lo cual se muestra en el código 5.4. Asimismo, su operación inversa se representa en la Ecuación 5.3.

$$\begin{aligned}
 q_x &= \sin \frac{r}{2} * \cos \frac{p}{2} * \cos \frac{y}{2} - \cos \frac{r}{2} * \sin \frac{p}{2} * \sin \frac{y}{2} \\
 q_y &= \cos \frac{r}{2} * \sin \frac{p}{2} * \cos \frac{y}{2} + \sin \frac{r}{2} * \cos \frac{p}{2} * \sin \frac{y}{2} \\
 q_z &= \cos \frac{r}{2} * \cos \frac{p}{2} * \sin \frac{y}{2} - \sin \frac{r}{2} * \sin \frac{p}{2} * \cos \frac{y}{2} \\
 q_w &= \cos \frac{r}{2} * \cos \frac{p}{2} * \cos \frac{y}{2} + \sin \frac{r}{2} * \sin \frac{p}{2} * \sin \frac{y}{2}
 \end{aligned} \tag{5.2}$$

Ecuación 5.2: Obtención de cuaterniones a partir de ángulos de Euler (RPY)

```

...
def euler2quat(rpy: tuple) -> Quaternion:
    roll, pitch, yaw = rpy
    qx = math.sin(roll/2) * math.cos(pitch/2) * math.cos(yaw/2) -
        math.cos(roll/2) * math.sin(pitch/2) * math.sin(yaw/2)
    qy = math.cos(roll/2) * math.sin(pitch/2) * math.cos(yaw/2) +
        math.sin(roll/2) * math.cos(pitch/2) * math.sin(yaw/2)
    qz = math.cos(roll/2) * math.cos(pitch/2) * math.sin(yaw/2) -
        math.sin(roll/2) * math.sin(pitch/2) * math.cos(yaw/2)
    qw = math.cos(roll/2) * math.cos(pitch/2) * math.cos(yaw/2) +
        math.sin(roll/2) * math.sin(pitch/2) * math.sin(yaw/2)
    return Quaternion(x=qx, y=qy, z=qz, w=qw)

def quat2euler(q: Quaternion) -> tuple:
    x = math.atan2(2 * (q.w*q.x + q.y*q.z), (1 - 2 * (q.x**2 + q.y**2)))
    y = -math.pi/2 + 2* math.atan2(math.sqrt(1+2*(q.w*q.y-q.x*q.z)),
        math.sqrt(1-2*(q.w*q.y-q.x*q.z)))
    z = math.atan2(2*(q.w*q.z + q.x*q.y), 1-2*(q.y**2 + q.z**2))
    return (x, y, z)
...

```

Código 5.4: Funciones matemáticas del módulo `geom_utils.py`

$$\begin{aligned}
 x &= \tan \frac{2 * (q_w * q_x + q_y * q_z)}{1 - 2 * (q_x^2 + q_y^2)} \\
 y &= -\frac{\pi}{2} * \tan \sqrt{1 + 2 * (q_w * q_y - q_x * q_z)} \sqrt{1 - 2 * (q_w * q_y - q_x * q_z)} \\
 z &= \tan \frac{2 * (q_w * q_z + q_x * q_y)}{1 - 2 * (q_y^2 + q_z^2)}
 \end{aligned} \tag{5.3}$$

Ecuación 5.3: Obtención de ángulos de Euler (RPY) a partir de Cuaterniones

5.2. Pruebas en simulación

Además de los cambios anteriores, también se modificó el modelo del Turtlebot 3 *waffle* simulado, en el repositorio del *stack* de navegación Nav2, ya que la posición de su sensor LIDAR virtual no era la misma que la de su modelo 3D y, al estar situado

más alto, no permitía la detección mutua entre los robots. En la Figura 5.2 se aprecia el problema descrito: los rayos emitidos por el sensor, en la parte izquierda de la figura no corresponden con el propio sensor, mientras que sí lo hacen correctamente en la parte derecha de la figura, tras haber modificado su posición. Este cambio también fue explicado en la Wiki⁸.

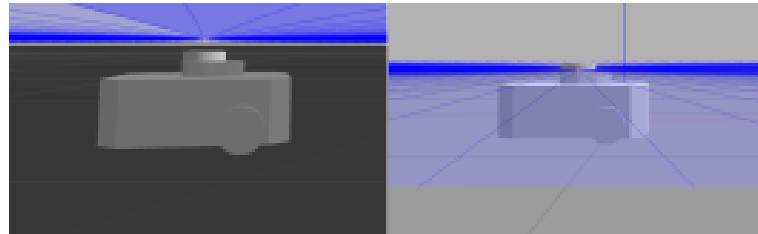


Figura 5.2: Modelos 3D del robot antes (izq.) y después (dcha.) de modificar el sensor

Así como la corrección de varios *bugs* y mejoras de la información mostrada por la interfaz, también se modificó la forma de obtención de los datos de profundidad en el momento de detección del objeto, ya que en el repositorio original eran obtenidos utilizando las medidas del sensor LIDAR, al haber sido desarrollado en un principio para un robot Turtlebot 3 modelo *burger* real, el cual no tenía cámara de profundidad.

Puesto que ahora se utilizan robots Turtlebot 3 modelo *waffle* simulados, que sí incluyen una cámara RGBD, y por tanto son capaces de detectar profundidad en sus imágenes, ya no se necesitarán las medidas del láser para este propósito. El nuevo modelo de obtención de profundidad, únicamente a partir de los datos de la cámara, se puede observar en la Figura 5.3, mediante la aplicación de cambios de ejes de coordenadas y trigonometría, y que queda explicado en la Wiki⁹.

En cuanto al módulo de navegación, se realizaron los cambios necesarios para sustituir el comportamiento caótico de los robots al detectar el objeto, ya que todos los que lo detectaban ordenaban al resto moverse a una posición cercana al objeto, a una distancia de seguridad. Dicho cambio también queda reflejado en la Wiki¹⁰. Al ser detectado desde posiciones o puntos de vista distintos, las posiciones seguras inferidas diferían ligeramente dependiendo de la posición del robot que lo detectase y, al ser

⁸<https://github.com/RoboticsURJC/tfg-unai/wiki/8.-Solucion-problema-deteccion>

⁹<https://github.com/RoboticsURJC/tfg-unai/wiki/11.-Mejoras/#cambios-nodo-detector>

¹⁰<https://github.com/RoboticsURJC/tfg-unai/wiki/11.-Mejoras/#cambio-algoritmo-acercamiento>

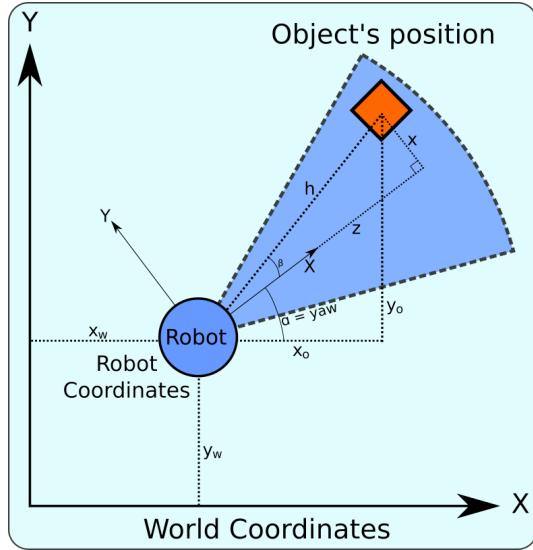


Figura 5.3: Inferencia de las coordenadas del objeto detectado

enviadas continuamente, todos los robots recalcularan su ruta al objeto por cada vez que uno de ellos enviaba su posición.

Este problema se soluciona con la implementación de un sistema jerárquico o de preferencias, en el que se asigna el rol de máster al primer robot que detecta el objeto, y este será el único que podrá enviar la posición del objeto desde su punto de vista. El rol de máster solo se cambiará, siendo otorgado a otro robot, en caso de que el robot con dicho rol pierda de vista al objeto el suficiente tiempo como para considerar que lo ha perdido, siendo este sistema robusto a cambios de posición del objeto y a occlusiones del mismo durante la ruta. Las diferencias entre estos dos métodos puede verse representada en la Figura 5.4 y, así como en la Wiki, también se han actualizado documentos gráficos en YouTube¹¹, consigiéndose así una prueba gráfica de los resultados descritos.

5.3. Pruebas en un entorno real

En esta sección se explicarán las pruebas y modificaciones realizadas para trasladar los anteriores resultados conseguidos en simulación a un entorno real de laboratorio con robots físicos.

¹¹<https://www.youtube.com/watch?v=pQUbj2aZCQs>

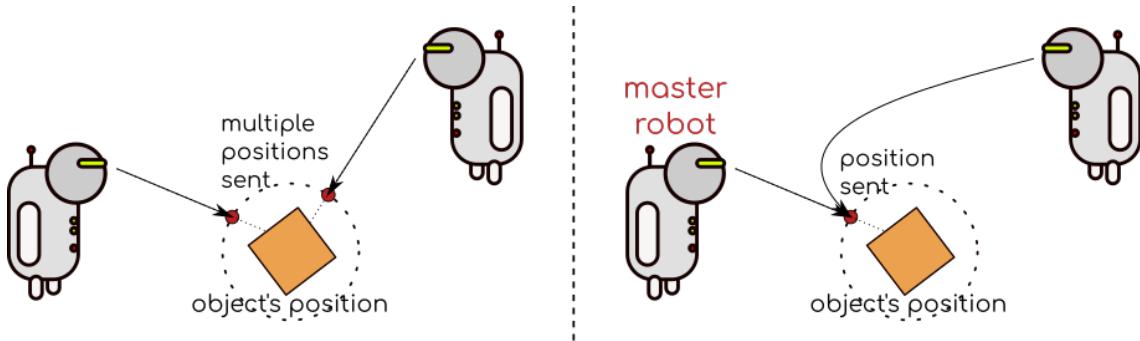


Figura 5.4: Representación de las diferentes versiones del nodo Navigator

5.3.1. Pruebas de telecomunicaciones

En primer lugar se debió asegurar que, tanto el entorno *software* como el entorno *hardware* era apto para el despliegue de la topología *software* descrita en la Sección 4.2, con lo que se llevaron a cabo pruebas de conexión de las distintas máquinas mediante el uso de *routers* de Zenoh y sus *bridges*, lo cual queda descrito en la Wiki¹².

Las pruebas mencionadas a nivel de comunicaciones básicas se llevaron a cabo en distintos pasos, el primero de estos fue probar las comunicaciones básicas entre el ordenador portátil principal con una Raspberry Pi mediante SSH (Secure Shell, protocolo que permite acceder remotamente a una máquina), para configurar el *firewall* del portátil (ya que la Raspberry Pi lo tiene desactivado por defecto) y que las comunicaciones en ambas direcciones sean posibles.

Una vez accedido a una terminal de la Raspberry Pi desde el ordenador principal, se pueden lanzar comandos para instalar el software necesario y cambiar su RMW a CycloneDDS, para lo que se realizaron pruebas únicamente utilizando nodos de ROS2 mediante los siguientes comandos, uno en cada máquina:

```
RMW_IMPLEMENTATION=rmw_cyclonedds_cpp ros2 run demo_nodes_py talker
RMW_IMPLEMENTATION=rmw_cyclonedds_cpp ros2 run demo_nodes_py listener
```

Teniendo este ejemplo funcionando correctamente, se hizo la prueba para conectar estos nodos a través del bridge de Zenoh, lo que se consiguió ejecutando en una de las máquinas los siguientes comandos:

```
./zenoh-bridge-dds -d 1 -l tcp/0.0.0.0:7447
ROS_DOMAIN_ID=1 ros2 run turtlesim turtlesim_node
```

¹²<https://github.com/RoboticsURJC/tfg-unai/wiki/12.-Instalacion-software-Raspberry-Pi>

Mientras que en la otra máquina se ejecutan los mismos comandos pero con el nodo emisor de mensajes:

```
./zenoh-bridge-dds -d 2 -e tcp/192.168.1.138:7447
ROS_DOMAIN_ID=2 ros2 run turtlesim turtle_teleop_key
```

El argumento `-d 2`, así como la variable global `ROS_DOMAIN_ID=2`, permiten correr el nodo y el *bridge* en un dominio distinto al de la anterior máquina, para asegurarnos que no se conectan mediante DDS, sino que pasan por el *bridge* para ser traducido y enviado usando Zenoh. Por supuesto, estas pruebas se realizaron también en sentido contrario, permitiendo de esta manera la comunicación bidireccional.

Añadiremos ahora Zenoh-Flow a la ecuación, para lo que necesitaremos correr además del *bridge*, un *router* de Zenoh, lo que se consigue ejecutando en la Raspberry Pi los siguientes comandos:

```
zenohd -c ./zenoh_nodes/zenoh_rpi_config.json
./zenoh-bridge-dds-0.7.2-rc -e tcp/<ip_portatil>:7447
ros2 run demo_nodes_py listener
```

Siendo la configuración mencionada la mostrada en el Código 5.5, con la debida sustitución de la etiqueta `<ip_portatil>` por la IP del portátil en la subred en cuestión.

```
{
  "listen": {
    "endpoints": ["tcp/0.0.0.0:7447"]
  },
  "connect": {
    "endpoints": ["tcp/<ip_portatil>:7447"]
  }
}
```

Código 5.5: Configuración del *router* de Zenoh en la Raspberry Pi

Además, se deberán ejecutar los siguientes comandos en el portátil:

```
RUST_LOG=zenoh_flow=debug zenohd -c ./zenoh_nodes/zenoh_laptop_config.json
zf_uuid=$(zectl launch ./zenoh_nodes/zenoh_flow_tests/data-flow.yaml)
```

Siendo esta configuración del portátil la mostrada en el Código 5.6, sustituyendo las etiquetas `<rpi_ip1>` y `<rpi_ip2>` por las IPs de las distintas máquinas Raspberry Pi:

```
{
  "listen": {"endpoints": ["tcp/0.0.0.0:7447"]},
  "connect": {"endpoints": ["tcp/<rpi_ip1>:7447", "tcp/<rpi_ip2>:7447"]},
  "plugins_search_dirs": ["/usr/lib/"],
  "plugins": {
    "storage_manager": {
      "required": true,
      "storages": {
        "zfrpc": {
          "key_expr": "zf/runtime/**",
          "volume": "memory"
        },
        "zf": {
          "key_expr": "zenoh-flow/**",
          "volume": "memory"
        }
      }
    },
    "zenoh_flow": {
      "required": true,
      "name": "computer",
      "path": "/etc/zenoh-flow",
      "pid_file": "/var/zenoh-flow/runtime.pid",
      "extensions": "/etc/zenoh-flow/extensions.d",
      "worker_pool_size": 4,
      "use_shm": false
    }
  }
}
```

Código 5.6: Configuración del *router* de Zenoh en la Raspberry Pi

Con estas pruebas funcionando correctamente, se pueden realizar pruebas en las que publicar desde Zenoh-Flow una posición objetivo a los nodos de ROS de navegación de los robots para verlos en movimiento. Esto puede hacerse creando un data-flow muy sencillo que albergue un nodo **operator**, que creará y serializará un mensaje, enviándolo a un nodo **sink**, que tendrá como key-expression “rt/robot1/goal_pose”, por lo que será entendida y traducida por el **bridge** y, por tanto, enviada a los nodos de ROS mencionados mediante DDS, como ya se ha explicado en las Secciones 4.2.2 y 4.2.3 del capítulo anterior.

Los resultados de esta prueba pueden verse en vídeo¹³, en el que se observa cómo cada robot se dirige a un punto del mapa distinto, uniendo en esta prueba todas las

¹³<https://www.youtube.com/watch?v=upLP5MzOfKo>

herramientas **software** usadas en este trabajo y demostrando por tanto el funcionamiento de la topología **software** explicada en la Sección 4.2.

La prueba mencionada se llevó a cabo con dos robots distintos (Turtlebot 2 y 4), ya que al realizar la prueba con dos robots Turtlebot 4, sus ordenadores de a bordo (Raspberry Pi) no tenían capacidad suficiente para procesar la gran cantidad de mensajes generados, y por ello dejaban de responder en cuanto se iniciaban los nodos de navegación de ROS en ambas máquinas y, consecuentemente, los nodos receptores de estos mensajes dejaban de recibir información.

Para consolidar esta teoría se realizó una prueba¹⁴ con dichas máquinas, en la que se lanzaron los nodos mencionados y se supervisó su envío de mensajes con la herramienta *software* PlotJuggler, que permitió visualizar los mensajes recibidos de los robots, dando lugar a la gráfica mostrada en la Figura 5.4. En concreto, los mensajes de esta gráfica representan la distancia medida con su sensor láser, que varía debido a la rotación continua del robot, y que comienza a ralentizarse cuando la navegación de uno de los robots es lanzada, como puede verse en la gráfica azul, donde se aprecia una menor cantidad de mensajes recibidos por unidad de tiempo, e incluso dejan de ser recibidos cuando se activa la navegación del segundo robot, como puede verse en la gráfica roja.

5.3.2. Pruebas de la aplicación

Una vez realizadas las pruebas anteriores se prosiguió con las pruebas de la aplicación *swarm_obj_finder* en los robots reales que, como se ha explicado en la Sección 5.1, debe dividir la sala y enviar a cada robot por una ruta de búsqueda en ese área en busca de un objeto. Todo este proceso queda documentado en la Wiki¹⁵.

Primero se llevó a cabo la prueba sin la detección de objetos¹⁶, para evitar posibles errores o saturación de red, por lo que los robots únicamente debían seguir la ruta proporcionada por el nodo de Zenoh-Flow PathsPlanner. Dicha prueba, tras algunos cambios menores fue llevada a cabo sin mayor problema, como también puede verse en

¹⁴<https://www.youtube.com/watch?v=OQEbhof4IFQ>

¹⁵<https://github.com/RoboticsURJC/tfg-unai/wiki/14.-Pruebas-de-laboratorio>

¹⁶<https://www.youtube.com/watch?v=M8LTSM-3-Es>

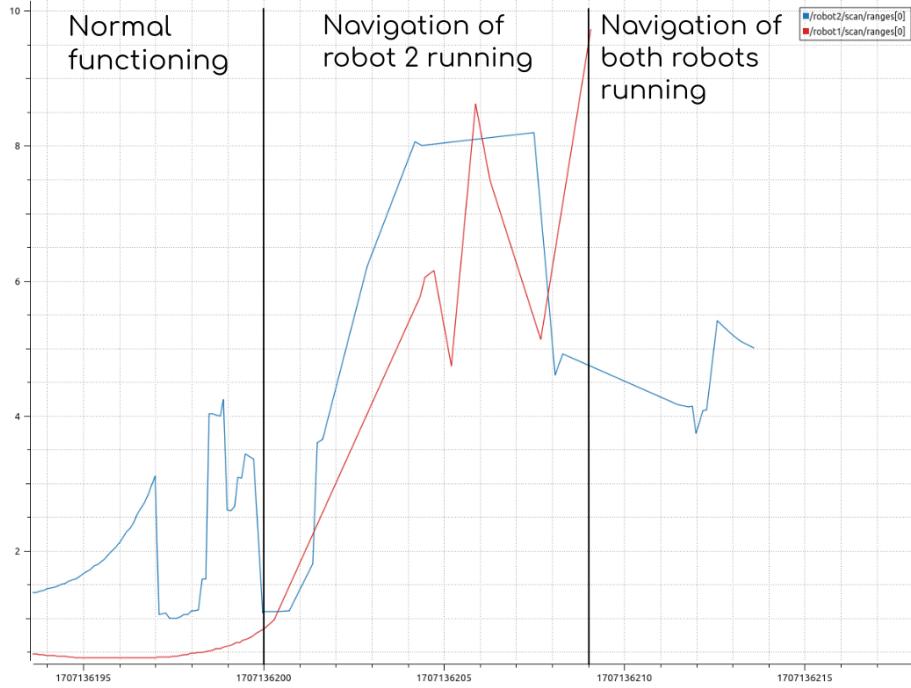


Figura 5.5: Comparación de mensajes variando la cantidad de robots

el vídeo enlazado. Esta prueba también demuestra la fácil modularización de Zenoh-Flow, en cuya estructura podemos cambiar unos nodos por otros sencillamente, o como en este caso, eliminarlos del flujo de datos.

Para añadir la detección de objetos a nuestra prueba se debieron probar las distintas cámaras, Orbbec Astra y Asus Xtion, en ambos robots, Turtlebot 2 y 4. Adicionalmente, se añadió la cámara Oak-D a las pruebas del robot Turtlebot 4, ya que viene integrada en el mismo.

En cuanto al robot Turtlebot 4, con la cámara Oak-D no se consiguió hacer funcionar la profundidad; es decir, no se pudo obtener los datos de distancias de la cámara RGBD, aún habiendo probado distintas configuraciones de la misma con distintos comandos de los paquetes oficiales `depthai_ros_driver` y `turtlebot4_bringup`, debido a diferentes errores, probablemente provocados porque los robots no fueron adquiridos al completo, sino que se adquirieron por separado sus distintos componentes *hardware*, lo que podría implicar que el *software* no esté funcionando completamente ya que la cámara original es un modelo superior (Oak-D-Pro) o por la falta de instalación de algún driver o paquete, aunque no se puede saber el motivo real, ya que tampoco se encontró más información o documentación acerca de los errores obtenidos.

Con la cámara Orbbec Astra se tuvo problemas de compilación que hicieron imposible la obtención de ningún dato, ya que el paquete tardaba en compilar lo suficiente como para que la Raspberry Pi se quedase sin memoria. También se intentó compilar, aunque sin éxito, aumentando la memoria *swap*, usando parte de la memoria no volátil como memoria RAM, y utilizando un solo núcleo de manera secuencial a la hora de compilar, para no saturar la capacidad de cómputo de la máquina y evitar bloqueos.

Respecto a la tercera de las cámaras, la Asus Xtion, tras arreglar varios problemas de compilación relacionados con paquetes que no habían sido desarrollados para la arquitectura de la Raspberry Pi (ARM), esta acabó funcionando; obteniendo así las medidas de profundidad junto con las imágenes a color. Pero el retraso de los datos recibidos superaba los dos segundos, debido a la congestión y cantidad de datos enviados.

Por otra parte, con el robot Turtlebot 2 se consiguieron resultados similares, ya que tanto con la cámara Astra como con la Xtion se recibían los mensajes con varios segundos de retraso.

Tras la realización de numerosas pruebas, este problema nos llevó a evitar el uso de cámaras para la aplicación y, en busca de alternativas, probamos a detectar objetos usando únicamente el sensor LIDAR, ya que no se puede utilizar este sensor para detectar la profundidad del objeto debido a que este puede tener un altitud distinta a la del robot y por tanto podría no ser detectado en todos los casos. Además es muy complicado obtener precisión con este método de sensores combinados debido al propio movimiento del robot que, sumado a la latencia de obtención de los datos del LIDAR, pueden resultar en medidas erróneas en cuanto al ángulo de rotación, pudiendo suponer una diferencia de varios grados, lo que puede traducirse finalmente en varios metros de distancia.

Puesto que este sensor nos proporciona distancias alrededor del robot, únicamente podremos detectar las formas de los objetos en vez de sus colores, por lo que se desarrolló un nodo de Zenoh-Flow de prueba para la detección de objetos circulares mediante el uso del este sensor.

Este procedimiento se realizó en dos pasos. El primero consistió en la obtención de una imagen a partir de los datos del LIDAR, así como se representan los diferentes puntos del sensor en la herramienta de visualización de datos RViz2. Dicho funcionamiento

se puede ver en las funciones `get_bg_img()`, `get_cart_points()` y `draw_points()`, en las que respectivamente se crea una imagen de fondo negro, se pasan los puntos de coordenadas polares a cartesianas, y se dibujan en la imagen creada para, en el segundo paso, usar la herramienta `cv2.HoughCircles()` de detección de círculos de la librería de OpenCV sobre dicha imagen, como puede verse en la función `detect_circles()` del Código 5.7, dibujando sobre ella los círculos detectados, siendo el círculo dibujado de color blanco, más probable que el resto dibujados en color gris.

El primer resultado probado en simulación fue nulo, ya que no se detectó ningún círculo, por lo que se procedió a modificar la función de dibujado de los puntos para unirlos entre ellos, con lo que se detectaron esta vez demasiados círculos aunque, tras ajustar los parámetros de la función `cv2.HoughCircles()` y filtrando los círculos por su radio, se consiguió disminuir dicha cantidad, obteniendo un resultado como el mostrado en la Figura 5.6, en el que se pueden ver varios círculos detectados; aunque existen falsos positivos y falsos negativos, ya que existen formas circulares que no son detectadas y hay otras formas que son detectadas como círculos pero no lo son.

Una vez comprobado su funcionamiento, se llevó al entorno real, donde se probaron las distintas versiones del detector, para mayor seguridad, aunque como se puede ver en la Figura 5.7, la detección en un espacio tan abierto era imposible, debido a la gran cantidad de puntos detectados, por las patas de las sillas y mesas del laboratorio.

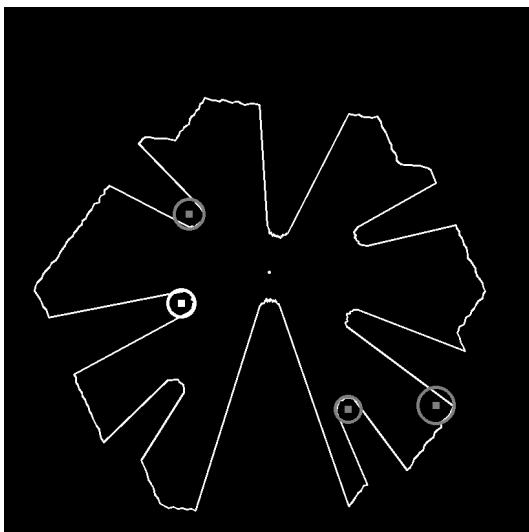


Figura 5.6: Detector de círculos en simulación

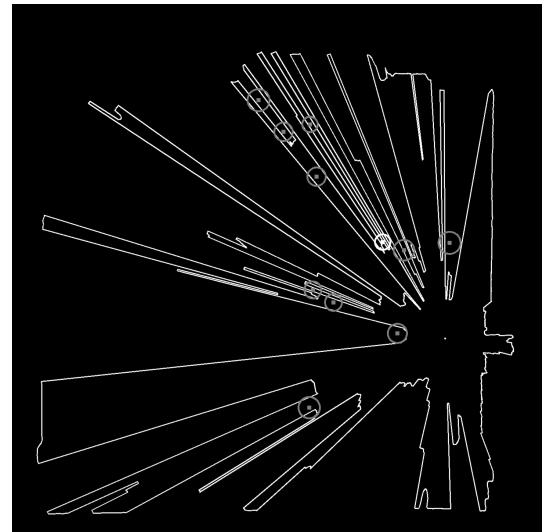


Figura 5.7: Detección de círculos en el laboratorio

```

...
def detect_circles(self, image: np.ndarray) -> list:
    blurred_img = cv2.GaussianBlur(image, (5, 5), 0) # before (sim)
    :image, (5, 5), 2

    circles = cv2.HoughCircles(blurred_img, cv2.HOUGH_GRADIENT, dp=1,
        minDist=40, param1=30, param2=25, minRadius=10, maxRadius=100)

    # If circles are found, draw them
    if circles is None:
        return blurred_img
    circles = np.round(circles[0, :]).astype("int")
    color = WHITE_COLOR_PIXEL
    for (x, y, r) in circles:
        cv2.circle(image, (x, y), r, color, 2) # Draw the circle outline
        cv2.circle(image, (x, y), 2, color, 3) # Draw the center of the
            circle
    color = GREY_COLOR_PIXEL # most probable circle will be drawn in
        white
    return image
...

async def iteration(self) -> None:
    ser_msg = await self.input.recv()
    laser_msg = ser_msg.get_data()
    if self.inverted:
        laser_msg.ranges.reverse()
    non_inf_ranges = [r for r in laser_msg.ranges if not np.isinf(r)]
    max_measured_range = max(non_inf_ranges)
    img_width = self.meter2pixel(MAX_RANGE * 2)

    bg_img = self.get_bg_img(img_width)
    cart_points = self.get_cart_points(laser_msg) # in [m]
    laser_img = self.draw_points(bg_img, cart_points)
    debug_img = self.detect_circles(laser_img)
    debug_img_msg = self.arr2img(debug_img)
    await self.output.send(debug_img_msg)
    return None
...

```

Código 5.7: Funciones de detección de círculos de un nodo de Zenoh-Flow

Es por este motivo que se probó en un espacio más reducido, de aproximadamente $2 \times 2 \text{m}^2$, como se puede ver en la Figura 5.8, ya que cuanto más grande sea el espacio, más grande deberá ser la resolución de la imagen generada, lo que ralentizará la detección. Con esta nueva configuración se obtuvo un mejor resultado, como se puede ver en el Cuadro 5.1.

Tamaño [px ²]	Resolución [px/m]	Latencia [s]	Detección [%]
350x350	75	0.7	33
450x450	100	0.8	100
560x560	125	1.1	100
880x880	200	1.3	93

Cuadro 5.1: Resultados del nodo detector de círculos en un espacio de 2x2m²Figura 5.8: Espacio de 2x2m² (aprox.)Figura 5.9: Espacio de 6x6m² (aprox.)

Dadas las mejoras en la detección, se decidió ampliar el espacio a 6x6m² aproximadamente, como puede verse en la Figura 5.9, para poder realizar pruebas más robustas, en las que comparar el anterior algoritmo con uno nuevo, que limitaba la distancia máxima detectada, creando siempre una imagen igual o más pequeña, aunque limitando su rango de detección. Estos resultados pueden verse en el Cuadro 5.2, en el que se han probado ambos algoritmos con la misma resolución (100 px/m).

Tamaño [px ²]	Res. [px/m]	Rango máx. [m]	Lat. [s]	Detección [%]
888x888	100	-	0.8	62
672x672	75	-	0.5	53
425x425	100	2	0.2	35
425x425	100	1.5	0.1	57

Cuadro 5.2: Resultados del nodo detector de círculos en un espacio de 6x6m²

Las detecciones obtenidas en ambos espacios, 2x2 y 6x6 m², pueden apreciarse en las Figuras 5.10 y 5.11, respectivamente.

A pesar de haber obtenido mejores resultados en cuanto a latencia, se obtuvieron peores resultados en cuanto a la propia detección del círculo. Además, es en este

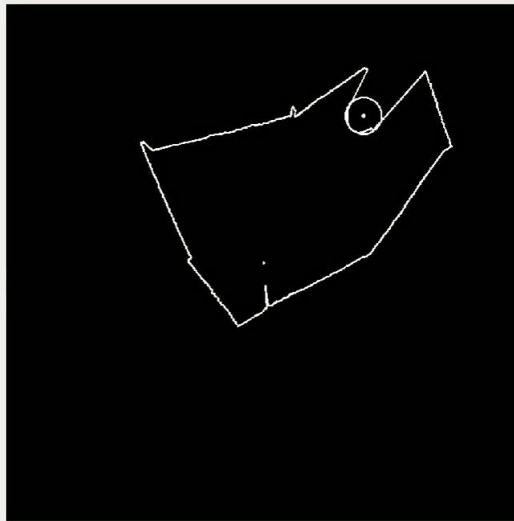


Figura 5.10: Detección en espacio de $2 \times 2 \text{m}^2$ (aprox.)

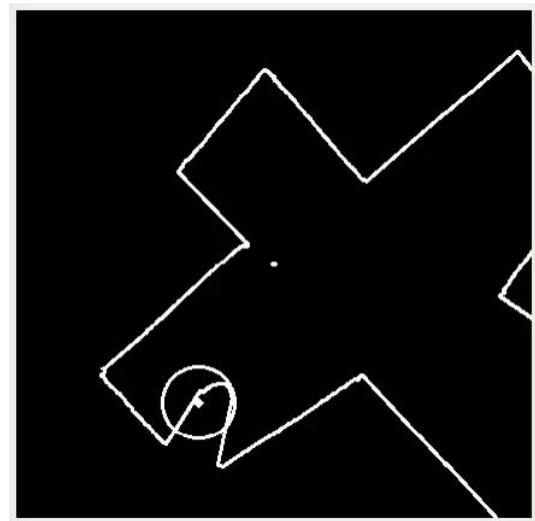


Figura 5.11: Detección en espacio de $6 \times 6 \text{m}^2$ (aprox.) con rango máx. de 2m

momento cuando notamos que las imágenes generadas no están siendo actualizadas correctamente: imprimiendo por pantalla las marcas de tiempo de los mensajes recibidos del LIDAR notamos que se estaban recibiendo únicamente los primeros mensajes del sensor en bucle, sin recibir nuevos.

Haciendo pruebas con distintos robots, sensores LIDARs, e incluso corriendo el detector sin Zenoh-Flow o ningún otro *software*, llegamos a la conclusión de que era problema de Zenoh-Flow, por lo que contactamos con la empresa ZettaScale para comentar el problema, y nos respondieron que el suscriptor de Zenoh, creado cuando se lanza el flujo de datos de Zenoh-Flow, comienza a almacenar mensajes en este momento, llenando su *buffer*, y provocando que los nuevos mensajes se pierdan, y probablemente este no se vacía lo suficientemente rápido como para considerar los mensajes como nuevos, dando lugar a una gran latencia.

Capítulo 6

Conclusiones

El éxito es más un viaje que un destino. El esfuerzo en sí mismo es el premio.

Arthur Ashe

En este último capítulo se detallan los objetivos cumplidos, presentando además las conclusiones derivadas, así como las habilidades, competencias y conocimientos adquiridos durante el desarrollo del proyecto, incluyendo futuras líneas de desarrollo del mismo.

6.1. Objetivos cumplidos

El objetivo principal de este proyecto consiste en desarrollar una forma de programación y aplicación de flujos de datos en conjunto con ROS2.

El segundo objetivo implica el desarrollo de una aplicación robótica utilizando la anterior solución para demostrar su viabilidad y funcionamiento.

Ambos objetivos fueron alcanzados, el primero mediante el uso de Zenoh-Flow y otras tecnologías como Zenoh o Zenoh-bridge-DDS, integradas en conjunto con ROS2. Esto dio lugar a un entorno de desarrollo para la programación de aplicaciones en conjunto con nodos de ROS2. En cuanto al segundo objetivo, se creó una aplicación robótica para la búsqueda de objetos, funcional en simulación y parcialmente en un entorno real debido a limitaciones en el uso de ciertos sensores. Como resultado, solo parte de la aplicación simulada fue trasladada al entorno real.

Como objetivos secundarios se estableció la sencillez de código y comprensión del entorno para su aplicación en entornos educativos. Este objetivo ha sido cumplido mediante el requisito de una única clase como mínimo para la creación de una aplicación

robótica simple. Además, dicha aplicación puede ser programada en Python, un lenguaje conocido por su naturaleza enfocada principalmente a la sencillez de sintaxis y programación, siendo una opción popular para el aprendizaje en este ámbito y proporcionando una sólida base para comprender la programación orientada a objetos.

Este enfoque también aborda la brecha de aprendizaje en este campo, ya que el nivel de programación necesario para comprender este entorno es ligeramente superior a los fundamentos de la programación. Por lo tanto, podría considerarse como una continuación de una asignatura de programación o robótica en la educación secundaria, o incluso como una asignatura independiente, que sienta las bases para comprender ROS2 antes de la universidad.

En cuanto a los subobjetivos cumplidos podemos detallar los siguientes:

- Se ha desarrollado un entorno de programación que permite la programación de flujos de datos en conjunto con nodos de ROS2, haciendo posible su comunicación de manera bidireccional, cumpliendo simultáneamente los dos primeros subobjetivos marcados.
- Se ha desarrollado una aplicación robótica en simulación utilizando el entorno de programación de flujos de datos mencionado anteriormente.
- Se ha demostrado la simplicidad de código y la viabilidad de su aplicación en entornos educativos, ofreciendo una solución al problema de la brecha de aprendizaje.

A pesar de haber logrado reducir el tráfico de mensajes originado por DDS mediante el uso de Zenoh, al seguir utilizando nodos complejos de ROS2 que generan una gran cantidad de mensajes, esta reducción no ha sido suficiente como para eliminar por completo el problema de la congestión de red.

Aunque no se ha logrado trasladar completamente la aplicación a un entorno real, la mayoría de los nodos de la aplicación sí se han trasladado con éxito, demostrando así su viabilidad en este entorno, por lo que es posible el desarrollo de tareas más simples de manera completamente funcional, lo que la hace adecuada para propósitos educativos.

Asimismo se han cumplido todos los requisitos establecidos, como son el uso del mismo sistema operativo en todas las máquinas (Ubuntu 22.04 LTS), la compatibilidad de las herramientas *software* utilizadas, la facilidad de desarrollo y despliegue de las aplicaciones, así como la viabilidad económica del *hardware* utilizado.

6.2. Competencias adquiridas

Además de las competencias numeradas en el Capítulo 2, durante la elaboración de este proyecto, se han obtenido numerosos aprendizajes y habilidades, entre los que se incluyen los siguientes:

- Conocimiento profundo de algunos aspectos del funcionamiento interno de ROS2.
- Conocimiento profundo del *framework* de Zenoh-Flow, así como del protocolo Zenoh y herramientas como Zenoh-bridge-DDS.
- Conocimiento de las comunicaciones entre ROS2 y Zenoh-Flow, así como de las similitudes y distinciones de protocolos como DDS y Zenoh.
- Descubrimiento de nuevos protocolos con mejores prestaciones como Zenoh.
- Conocimiento más extenso del protocolo SSH y sus métodos de autenticación.
- Conocimiento de nuevos comandos, variables globales, y otros aspectos relacionados con el sistema operativo, relativos a ROS2 y a las comunicaciones de red.
- Conocimiento a fondo del paradigma de programación de flujos de datos.
- Adquisición de mayor experiencia, así como el descubrimiento de nuevas librerías y funciones relativas a la programación en Python.
- Conocimiento de nuevos lenguajes de programación como Rust.
- Conocimiento de herramientas de visualización como PlotJuggler y de distintas herramientas de grabación y edición de vídeo.
- Mayor experiencia con herramientas de creación de imágenes vectoriales como Inkscape.
- Desarrollo de nuevas habilidades y conocimientos respecto a LaTex.

- Adquisición de conocimientos y posibilidad de tener charlas interesantes con otras personas del sector a través de la asistencia y presentación del trabajo en ponencias.

6.3. Líneas futuras

Las herramientas *software* utilizadas relacionadas con Zenoh, como pueden ser Zenoh-bridge-DDS o el propio Zenoh-Flow, han pasado por muchos cambios durante la realización de este trabajo debido a su estado de desarrollo, o incluso de fase beta, como es el caso de este último, por lo que una innegable línea futura pasa por la actualización de estos *softwares* a versiones estables, mejorando la experiencia de usuario así como la posible solución de *bugs* o incompatibilidades. Además existen nuevos *softwares*, como el reciente RMW de Zenoh, que podrían ser probados en este entorno, ofreciendo posibles mejoras al mismo.

Además es posible, como segunda línea futura, el empaquetamiento de estas herramientas *software* en una aplicación que brinde una interfaz gráfica, así como herramientas interactivas a este entorno, facilitando el despliegue y la simplicidad de uso en entornos educativos y ahorrando tiempo tanto a alumnos, como al profesorado. En esta hipotética aplicación, la instalación y compilación de las herramientas *software* utilizadas se realizaría automáticamente.

Bibliografía

- [Alami et al., 1998] Alami, R., Fleury, S., Herrb, M., Ingrand, F., and Robert, F. (1998). Multi-robot cooperation in the martha project. *IEEE Robotics & Automation Magazine*, 5(1):36–47.
- [Arai and Parker, 2003] Arai, T. and Parker, L. (2003). Editorial: Advances in multi-robot systems.
- [Auledas, 2024] Auledas (2024). Arquitectura de ros 1 i ros 2, basada en l'article “exploring the performance of ros2” (doi: 10.1145/2968478.2968502).
- [Batocera Team, 2022] Batocera Team (2022). File:raspberry pi 4 b.png.
- [BuckyBall, 2006] BuckyBall (2006). File:hsv cone.png.
- [Chaimowicz et al., 2001] Chaimowicz, L., Sugar, T., Kumar, V., and Campos, M. (2001). An architecture for tightly coupled multi-robot cooperation. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, volume 3, pages 2992–2997 vol.3.
- [Eclipse and ZettaScale, 2023] Eclipse and ZettaScale (2023). Comparing the performance of zenoh, mqtt, kafka, and dds.
- [Eclipse Foundation, 2020] Eclipse Foundation (2020). Zero overhead network protocol (zenoh).
- [Eclipse Foundation, 2021] Eclipse Foundation (2021). Zenoh-flow.
- [Fox et al., 2000] Fox, D., Burgard, W., Kruppa, H., and Thrun, S. (2000). A probabilistic approach to collaborative multi-robot localization. *Autonomous Robots*, 8(3):325–344.
- [Haridy, 2020] Haridy, R. (2020). Entire boston dynamics robot line-up dances in the new year. Section: Robotics.

- [JotaCartas, 2011] JotaCartas (2011). A arduino uno board.
- [Liang et al., 2023] Liang, W.-Y., Yuan, Y., and Lin, H.-J. (2023). A performance study on the throughput and latency of zenoh, mqtt, kafka, and dds. *arXiv preprint arXiv:2303.09419*.
- [Ludlow, 2023] Ludlow, D. (2023). Asus rog rapture gt-axe1600 review.
- [Macenski et al., 2022] Macenski, S., Foote, T., Gerkey, B., Lalancette, C., and Woodall, W. (2022). Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074.
- [Madrid, 2014] Madrid, C. (2014). Decreto 89/2014, de 24 de julio, por el que se establece para la comunidad de madrid el currículo de la educación primaria. Boletín Oficial de la Comunidad de Madrid, 175, de 25 de Julio de 2014.
- [Madrid, 2015] Madrid, C. (2015). Decreto 48/2015, de 14 de mayo, por el que se establece para la comunidad de madrid el currículo de la educación secundaria obligatoria. Boletín Oficial de la Comunidad de Madrid, 118, de 20 de mayo de 2015.
- [Mattruffoni, 2023] Mattruffoni (2023). Mbot2 mblock raw code line follower.
- [Ministerio de Educación, 2018] Ministerio de Educación, Formación Profesional y Deportes, I. N. d. T. E. y. d. F. d. P. (2018). Programación, robótica y pensamiento computacional en el aula. situación en españa y propuesta normativa.
- [OSRF, s f] OSRF (s. f.). Turtlebot.
- [Parker, 2003] Parker, L. E. (2003). Current research in multirobot systems. *Artificial Life and Robotics*, 7(1):1–5.
- [Rankin, 2021] Rankin, S. (2021). Perseverance and ingenuity.
- [ROS ORG, s f] ROS ORG (s. f.). Kobuki.
- [Sheng et al., 2006] Sheng, W., Yang, Q., Tan, J., and Xi, N. (2006). Distributed multi-robot coordination in area exploration. *Robotics and Autonomous Systems*, 54(12):945–955.
- [Snape et al., 2021] Snape, J., van den Berg, J., J. Guy, S., and Manocha, D. (2021). The hybrid reciprocal velocity obstacle (hrvo).

- [Stanford Artificial Intelligence Laboratory, 2018] Stanford Artificial Intelligence Laboratory (2018). Robotic operating system.
- [Trawny et al., 2009] Trawny, N., Roumeliotis, S. I., and Giannakis, G. B. (2009). Cooperative multi-robot localization under communication constraints. In *2009 IEEE International Conference on Robotics and Automation*, pages 4394–4400.
- [Vega, 2018] Vega, J. M. (2018). *Educational framework using robots with vision for constructivist teaching Robotics to pre-university students*. Doctoral thesis on computer science and artificial intelligence, University of Alicante.
- [Verma and Ranga, 2021] Verma, J. K. and Ranga, V. (2021). Multi-robot coordination analysis, taxonomy, challenges and future scope. *Journal of Intelligent & Robotic Systems*, 102(1):10.
- [Wiki300, 2011] Wiki300 (2011). File:matl.jpg. Page Version ID: 449974156.
- [You, 2019] You, H. S. (2019). Hp probook 440 g6 laptop.
- [Yuhong, 2022] Yuhong, L. (2022). New ros online courses: Gazebo simulator, intermediate ros2, tf ros2. Section: Training & Education.