TRABAJO FIN DE GRADO

ESTUDIO DEL COMPORTAMIENTO DE MICRO-ROS. UN ANÁLISIS TEÓRICO Y PRÁCTICO

TRABAJO FIN DE GRADO PARA LA OBTENCIÓN DEL TÍTULO DE **GRADUADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES**

NOVIEMBRE 2021

Carlos Castillo Martínez

DIRECTOR DEL TRABAJO FIN DE GRADO:
Ricardo Sanz Bravo

INDUSTRIALES
ETSII | UPM

POLITÉCNICA

“If I have seen further it is by standing on the shoulders of Giants”

- Sir Isaac Newton

Agradecimientos

En primer lugar quería agradecer a mi familia el haber sido un apoyo incondicional para mí en todo momento por celebrar mis éxitos como si de los suyos se tratara y por evitar que me hundiese en los momentos más difíciles.

Le doy las gracias a mi tutor, Ricardo Sanz, por haberme brindado la oportunidad de realizar este proyecto y aprender de él. También quería agradecer a la doctoranda Esther Aguado su disponibilidad y ayuda durante todo el trabajo, fundamentales para el desarrollo del mismo.

Por último, quisiera recordar a todos mis amigos y compañeros que me han acompañado en este largo viaje los últimos años. Sin ellos hubiera sido imposible llegar hasta este punto.

Quiero dedicar este trabajo especialmente a mis padres, por ser mis principales referentes en todo aquello que quiero llegar a ser en mi vida, tanto personal como profesionalmente.

Glosario

- **ROS**: Sistema operativo de robots
- **IoT**: Internet de las cosas
- **RTOS**: Sistema operativo en tiemo real
- **Framework**: Entorno de trabajo
- **Firmware**: Programa que controla los circuitos electrónicos de un dispositivo
- **Middleware**: Software intermedio entre dispositivos y plataformas de software
- **Throughput**: Tasa de transferencia efectiva
- **Topic**: Tópico del mensaje
- **Publisher**: Editor de un topic
- **Subscriber**: Suscriptor de un topic
- **API**: Interfaz de programación de aplicaciones
- **DDS**: Servicio de distribución de datos
- **XRCE**: Entorno de recursos extremadamente limitados
- **UDP**: Protocolo de datagrama de usuario
- **TCP**: Protocolo de control de transmisión
- **POSIX**: Interfaz de sistema operativo portable
- **OTT**: Servicios de libre transmisión
- **UART**: Transmisor-Receptor Asíncrono Universal
- **MIMO**: Múltiple entrada, múltiple salida

Resumen Ejecutivo

Durante los últimos años, la tecnología ha ido creciendo a una velocidad exponencial en relación a etapas anteriores. Este hecho se traduce en la irrupción de la ya actual cuarta revolución industrial, también denominada industria 4.0.

Este fenómeno viene marcado por la aparición de nuevas tecnologías que están relacionadas con el tratamiento masivo de datos o "big data" y el Internet de las cosas (IoT).

Este último concepto se refiere a una interconexión digital de objetos cotidianos con Internet, lo cual genera un ecosistema de dispositivos inteligentes habilitados para recoger, enviar y actuar sobre los datos que adquieren de sus entornos.

En este sentido, la robótica constituye un sector fundamental en esta novedosa tecnología, que aprovecha y potencia las posibilidades que los robots ofrecen. Asimismo, los sistemas embebidos juegan un papel clave en la creación de estos ecosistemas, ya que permiten implementarse de manera sencilla en una gran cantidad de objetos cotidianos a un reducido coste, manteniendo unas prestaciones más que suficientes para las tareas que van a acometer.

Esto ha producido que el mundo de la robótica acelere su desarrollo a pasos agigantados en la última década, tanto a nivel de hardware como de software.

ROS es una tecnología pionera en el control de robots en tiempo real. Su crecimiento durante estos años está permitiendo a día de hoy la monitorización de muchos robots de una manera muy efectiva. Actualmente la versión en uso es ROS2. Sin embargo, recientemente se ha creado micro-ROS, una tecnología que acerca el mundo de ROS a los microcontroladores.

En este trabajo se va a estudiar el comportamiento de esta tecnología. En una primera etapa se analizará la base del funcionamiento de las comunicaciones en tiempo real. En este apartado se incluirán las restricciones que dichos sistemas requieren y las ventajas que pueden aportar.

Seguidamente se realizará un estudio teórico del funcionamiento de ROS. Esto incluye la base de programación que soporta ROS2, las distintas partes que forman la arquitectura del software y las herramientas que se utilizan para lograr integrar las funciones de ROS2 en microcontroladores.

Finalmente se plantearán una serie de análisis que simulen aplicaciones que se puedan dar en la vida real y se medirá el comportamiento que presenta micro-ROS instalado en una placa ESP-32.

Todo esto se realizará de manera autónoma sin ninguna experiencia previa en el sector, por lo que el trabajo resultante servirá también como guía de iniciación para la introducción a la programación del sistema operativo de robots.

Este trabajo ha sido realizado con Sphinx y LaTeX. Todo el código fuente empleado en el documento, tanto la memoria en LaTeX como la aplicación desarrollada, estarán disponibles en el repositorio https://github.com/aslab/ROSbenchmark.

Codigos Unesco:

120317 Informática

120323 Lenguajes de Programación

120324 Teoría de la Programación

120326 Simulación

Índice

Resumen Ejecutivo I

1. Introducción 1
- 1.1. Motivación . . . 1
- 1.2. Objetivos . . . 1
- 1.3. Estructura . . . 2
- 1.4. Metodología . . . 2

2. Marco teórico y estado del arte 3
- 2.1. Sistemas en tiempo real . . . 3
  - 2.1.1. Sistemas embebidos . . . 4
  - 2.1.2. Sistemas en tiempo real distribuido . . . 4
- 2.2. ROS . . . 5
  - 2.2.1. Definición . . . 5
  - 2.2.2. Historia . . . 6
  - 2.2.3. Proyección futura . . . 7
- 2.3. Micro-ROS . . . 7
  - 2.3.1. Definición . . . 7
  - 2.3.2. Historia . . . 8
  - 2.3.3. Proyección futura . . . 8

3. Software 11
- 3.1. ROS 2 . . . 12
  - 3.1.1. Conceptos . . . 12
  - 3.1.2. Instalación . . . 13
- 3.2. Micro-ROS . . . 15
  - 3.2.1. Características principales . . . 15
  - 3.2.2. Instalación . . . 16
  - 3.2.3. Arquitectura modular . . . 16
  - 3.2.4. Librería del cliente . . . 16
  - 3.2.5. Middleware . . . 18
  - 3.2.6. RTOS . . . 19

4. Hardware 21
- 4.1. ESP-32 . . . 21
- 4.2. Computador . . . 24

4.3. Cable micro-USB . . . 24
4.4. Router TP-Link . . . 24

5. Diseño del análisis **25**
5.1. Preparación previa . . . 25
5.2. Estructura principal del análisis . . . 26
5.3. Herramientas empleadas . . . 28

6. Resultados **33**
6.1. Latencia . . . 33
6.1.1. Distribución . . . 34
6.1.2. Histograma . . . 35
6.1.3. Conclusión . . . 37
6.2. Throughput . . . 37
6.2.1. Conclusión . . . 39
6.3. Consumo de memoria . . . 39
6.4. Influencia de interferencias . . . 39

7. Conclusiones y líneas futuras **43**
7.1. Conclusiones . . . 43
7.2. Lineas futuras . . . 44

8. Planificación temporal y presupuesto **45**
8.1. Planificación temporal . . . 45
8.2. Presupuesto . . . 46

9. Impacto social y medioambiental **47**
9.1. Impacto social . . . 47
9.2. Impacto medioambiental . . . 48

10. Anexos **49**
10.1. Anexo 1: Muestra de resultados obtenidos . . . 49
10.2. Anexo 2: Script de python para la generación de gráficas . . . 52
10.3. Anexo 3: Inciencias ocurridas . . . 54
10.3.1. Conexión Wi-Fi . . . 54
10.3.2. Fallo en la conexión con el agente de ROS 2 . . . 55

Bibliografía **61**

CAPÍTULO 1

Introducción

1.1 Motivación

El motivo que ha predominado en la elección de este tema ha sido la relación que mantiene con la tecnología del futuro. Resultaba muy interesante el hecho adentrarse y contribuir en una tecnología en auge que puede marcar la diferencia en los años venideros.

Por otra parte, enfrentarse a una tecnología tan sofisticada sin ningún conocimiento previo suponía un reto. De este modo, este trabajo plasma las habilidades de resiliencia, persistencia y duro trabajo que se han adquirido durante los últimos años.

1.2 Objetivos

La principal finalidad de este trabajo es realizar un estudio completo de ROS2 y micro-ROS, que comprenda desde el fundamento teórico e informático que los soporta, hasta los análisis de comportamiento que presentan dichos softwares frente a aplicaciones que simulan situaciones que se puedan dar en la vida real.

La idea del proyecto es iniciarse en una tecnología avanzada que se emplea en ámbitos profesionales de la que se parte sin ningún conocimiento ni experiencia, para que al final de este sea posible diseñar un experimento que muestre su comportamiento frente a aplicaciones semejantes a las que se podrían dar en el mundo real. El motivo de este estudio intensivo de dicha tecnología es ser capaces de indicar, no solo cómo responde frente a distintas cargas de trabajo, sino también explicar el porqué de dicho comportamiento desde un fundamento teórico que permita identificar las posibles debilidades y puntos de mejora.

1.3 Estructura

El trabajo se divide en dos fases claramente diferenciadas.

La primera etapa está relacionada con la teoría que soporta todo el trabajo.

En el primer capítulo se exponen conceptos generales relacionados con la comunicación de sistemas en tiempo real y se narran brevemente los inicios de la tecnología que es sujeto de este estudio.

Una vez expuestos dichos conceptos, se explica con detalle el software medido. Primero se desarrolla la base del funcionamiento de ROS. Seguidamente, se explican las especificaciones de ambos *frameworks*, incluyendo los conceptos generales, la instalación, el requerimiento de otros tipos de software para su uso y la arquitectura que los soporta.

Para concluir esta fase del trabajo, se finaliza explicando con gran detalle todo el hardware que se ha empleado durante el proyecto, incluyendo todas las especificaciones que puedan afectar a los resultados del análisis.

La segunda parte está ligada con la práctica que se ha llevado a cabo.

En el primer capítulo de esta sección se expone detalladamente la forma en la que se ha diseñado el análisis. Esto engloba los escenarios que se han formado, los parámetros que se han escogido y las herramientas que se han usado para realizar dichas mediciones.

Seguidamente se exponen en el capítulo posterior todos los resultados que se han obtenido y una discusión de estos junto a las conclusiones recogidas.

Después de discutir los resultados se han recogido en poco más de una cara todas las conclusiones adquiridas durante todo el desarrollo del trabajo, tanto relacionadas con la investigación llevada a cabo como con los resultados de los análisis prácticos.

Para concluir con el trabajo se ha añadido un apartado de incidencias ocurridas en el desarrollo del mismo, algo que podría resultar muy útil para futuras investigaciones.

1.4 Metodología

La metodología empleada se caracteriza por ser principalmente autodidacta.

La tecnología que se ha analizado está sufriendo un desarrollo constante en los últimos años, por lo que a día de hoy no existe la cantidad suficiente de información para resolver todas las complicaciones que pueden emerger al utilizarlo, sobre todo para un usuario no experimentado.

Esto se traduce en un trabajo en el que en algunos momentos ha sido necesario recurrir y preguntar directamente a desarrolladores y consultar foros que apenas llevaban creados pocos días.

En este sentido, el proyecto ha potenciado la capacidad de autosuficiencia y de resolución de problemas del autor en situaciones en las que el entorno no era el más favorable.

CAPÍTULO 2

Marco teórico y estado del arte

2.1 Sistemas en tiempo real

Se dice que un sistema opera en tiempo real cuando el tiempo que tarda en efectuarse la salida es significativo. El tiempo de respuesta puede ser relativamente flexible (tiempo real suave) o más estricto (tiempo real duro), lo que se denomina software crítico. [36]

La falta de respuesta en el tiempo establecido puede ocasionar graves consecuencias para el entorno del sistema, llegando a producir daños a la vida y a la propiedad.

Es por ello por lo que un sistema en tiempo real se diseña específicamente para la tarea que ha de acometer, utilizando un hardware y software dedicados.

El software empleado en sistemas de tiempo real cuenta con una serie de características propias que garantizan el correcto funcionamiento del sistema:

- Sistema operativo en tiempo real: Los sistemas operativos en general tienen dos principales funciones, gestionar bien los recursos que proporciona el hardware y facilitar el uso del mismo al usuario. En este caso, los objetivos del sistema operativo en tiempo real son los mismos pero enfocados a las restricciones de tiempo y ocupar un tamaño reducido para que pueda aplicarse a sistemas embebidos.
- Lenguaje de programación en tiempo real: Proporciona esquemas básicos como la comunicación y la sincronización entre tareas, el manejo de errores y la programación de funciones a realizar en tiempo real. El lenguaje C se ha utilizado ampliamente para este tipo de tareas debido a su facilidad de uso e interacción con el hardware. Sin embargo, otros lenguajes como Ada o Java se han desarrollado específicamente para este tipo de uso y cada vez tienen más peso en el sector.
- Una red en tiempo real: Un sistema en tiempo real necesita una red que sea puntual y fiable en la transferencia de mensajes, para ello cuentan con un protocolo específico para trabajar en tiempo real que proporciona una entrega puntual y garantizada de los mensajes a través de la red.

Las características principales de los sistemas en tiempo real son las siguientes:

- Cumplimiento en los plazos de ejecución: Es lo que distingue a este tipo de sistemas respecto al resto de sistemas informáticos.

- Previsibilidad: Han de ser capaces de prever cualquier tipo de orden que pueda ocurrir posteriormente para estar preparado y que no haya fallos en los tiempos de ejecución.
- Seguridad y fiabilidad: Muchos sistemas de este tipo se encargan de controlar otros sistemas peligrosos en los que es de vital importancia la precisión, ya no solo en el tiempo de la ejecución sino en los movimientos del sistema.
- Tolerancia a los fallos: Debido a la importancia del correcto funcionamiento de estos sistemas, deben estar diseñados para que un fallo en el propio hardware o software del mismo no repercuta drásticamente en el resto de componentes y operaciones que ejecute el sistema.
- Concurrencia: El sistema tiene que ser capaz de cooperar con otros sistemas que estén operando en el mismo entorno y, en determinadas ocasiones, incluso utilizar el hardware o software de dichos sistemas. [10]

2.1.1 Sistemas embebidos

La mayoría de sistemas utilizados en tiempo real son sistemas embebidos. Estos son aquellos en los que el computador se encuentra integrado en el sistema. Se caracterizan por no ser sistemas informáticos generales que se programan para distintas tareas, sino que están diseñados para cumplir un objetivo en concreto. Generalmente, un sistema embebido está constituido por un microcontrolador y una infraestructura diseñada para el propósito para el que está diseñado. El microcontrolador está constituido por una unidad central (CPU), que se encarga de realizar la mayoría de procesos, una memoria, que almacena las instrucciones y otro tipo de datos que aseguran el correcto funcionamiento del sistema; y un subsistema de entrada y salida, que suele contar con temporizadores, convertidores de señales analógicas y digitales, y canales de comunicación en serie.

2.1.2 Sistemas en tiempo real distribuido

Un sistema que trabaja en tiempo real distribuido está formado por unos nodos autónomos que se comunican entre sí a través de una red que trabaja en tiempo real y que cooperan para lograr unos objetivos comunes en unos plazos determinados.

Estos sistemas son fundamentales debido a varias razones. En primer lugar, la computación en tiempo real es esencialmente distribuida, ya que se basa en la transferencia de información entre dos extremos (nodos) a realizar en un tiempo determinado.

Seguidamente, la comunicación en tiempo real distribuido permite aislar las partes del sistema e identificar fallos en el mismo evaluando los nodos de la operación por separado. El cálculo realizado en cada nodo debe cumplir con las restricciones de tiempo de las tareas, y la red debe proporcionar un procesamiento en tiempo real con retrasos limitados en los mensajes.

Además de esto, un equilibrio entre los distintos nodos del sistema mejora el rendimiento del mismo.

Existen varios tipos de sistemas en tiempo real distribuido, sin embargo, la arquitectura general de todos ellos es similar a la figura que aparece a continuación.

En la figura se observa cómo todos los nodos están conectados entre sí a través de la red de tiempo real, y a su vez, cada uno está en contacto con distintas funciones propias que interactúan directamente con el sistema.

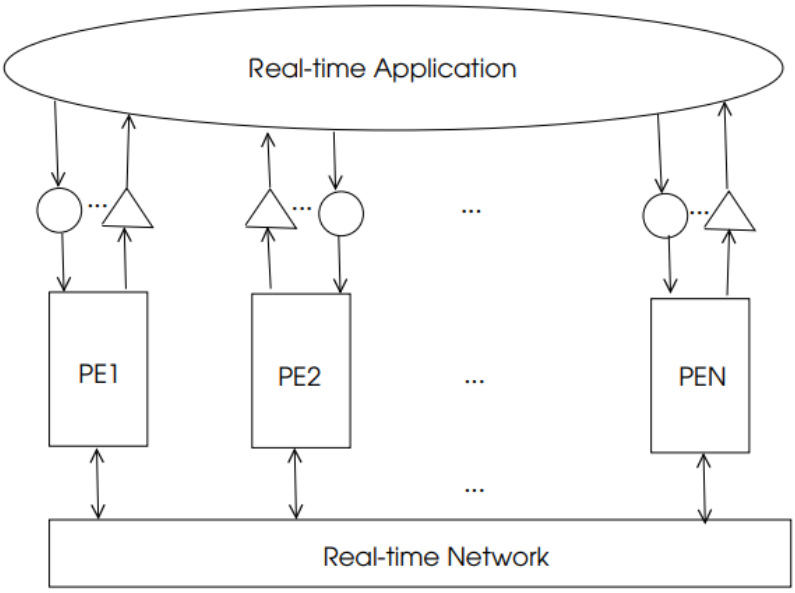

Figura 1: Arquitectura general de un sistema en tiempo real distribuido (Fuente: Distributed Real-Time Systems, 2019)

2.2 ROS

2.2.1 Definición

El ROS o Robot Operating System (sistema operativo de robots), es una colección de *frameworks* para el desarrollo de software de robots. Un *framework* es un entorno de trabajo tecnológico que se basa en módulos concretos que sirve de base para la organización y el desarrollo de software. [38]

Figura 2: Logotipo de ROS

ROS no llega a ser considerado un sistema operativo como tal, ya que necesita de un software de nivel superior para ser utilizado. Sin embargo, ROS provee los servicios básicos de uno, como son la abstracción del hardware, el control de dispositivos de bajo nivel, la implementación de funcionalidad de uso común, el paso de mensajes entre procesos y el mantenimiento de paquetes. [26]

Está basado en una arquitectura de grafos, esto es, una estructura formada por nodos, o extremos del sistema, y un conjunto de arcos que establecen las relaciones entre dichos nodos. Estas relaciones se basan en recibir, mandar y multiplexar mensajes de sensores, control, periféricos, etc.

La librería está pensada y diseñada para ser utilizada en un sistema operativo UNIX (base del actual Linux), sin embargo, también se están lanzando versiones experimentales para otros sistemas operativos muy comunes como Mac OS X o Microsoft Windows.

ROS se divide en dos partes básicas. Por un lado, actúa como nexo entre el usuario y el hardware (más similar a un sistema operativo convencional) y, por otra parte, se comporta como una batería de paquetes desarrollados por una comunidad de usuarios. Estos paquetes implementan numerosas funcionalidades como la localización y el mapeo simultáneo, la planificación, la percepción, la simulación, etc.

2.2.2 Historia

ROS se desarrolló en 2007 bajo el nombre de switchyard por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford (STAIR) y al programa de robots personales (PR), en los cuales se crearon prototipos internos de sistemas de software destinados a la robótica. [27]

Figura 3: Robot con Inteligencia Artificial de Stanford (Fuente: Stanford University)

Desde 2008, el proyecto continuó principalmente en Willow Garage, un instituto de investigación con más de veinte instituciones colaborando en un modo de desarrollo federado, que proporcionó importantes recursos para ampliar los conceptos ya creados y crear implementaciones sometidas a varias pruebas.

El proyecto fue impulsado por una gran cantidad de investigadores con mucha experiencia en el sector que aportaron numerosas ideas tanto al núcleo central de ROS como al desarrollo de sus paquetes de software fundamentales.

En un inicio, el software fue desarrollado utilizando la licencia de código abierto BSD (Berkeley Software Distribution) y poco a poco se ha convertido en una plataforma ampliamente utilizada en la comunidad de investigación robótica.

Desde el principio, ROS ha sido desarrollado en múltiples instituciones y para numerosos tipos de robots, incluidas aquellas que recibieron los robots personales (PR2) directamente desde Willow Garage.

Cualquier persona puede iniciar su propio repositorio de código ROS en sus propios servidores, y mantienen la plena propiedad y control del mismo; además pueden poner su repositorio a disposición del público y recibir el reconocimiento y el crédito que merecen por sus logros. De esta forma también se fomenta la mejora del software ya existente con la aportación de otros profesionales del sector.

Actualmente, el ecosistema de ROS cuenta con decenas de miles de usuarios en todo el mundo, que trabajan en ámbitos que van desde proyectos personales hasta grandes sistemas de automatización industrial.

Algunos de los robots que a día de hoy utilizan ROS son el robot personal de Ken Salisbury en Stanford (PR1), el robot personal de Willow Garage (PR2), el Baxter de Rethink Robotics, el Robot de Shadow, en el cual participan universidades españolas, o el robot limpiador HERB de Intel.

2.2.3 Proyección futura

ROS ya cuenta hoy en día con una estructura muy completa que proporciona al usuario múltiples posibilidades. Algunas de las funcionalidades que engloba este software a día de hoy son la creación, destrucción y correcta distribución de nodos en la red, la publicación o suscripción de flujos de datos, la multiplexación de la información, la modificación de los parámetros del servidor y el testeo de sistemas.

A pesar de la gran cantidad de servicios que ya ofrece, se espera que en futuras versiones se incorporen algunas de las siguientes funcionalidades a las aplicaciones de ROS: identificación y seguimiento de objetos, reconocimiento facial y de gestos, la comprensión del movimiento, el agarre y la egomoción, entre muchas otras.

Como se ha podido comprobar, esta tecnología ha avanzado enormemente durante los últimos años, y se prevé que este auge se maximice en el futuro próximo, desempeñando un papel fundamental en la revolución de la industria 4.0 y el fenómeno conocido como “el internet de las cosas”. [29]

2.3 Micro-ROS

2.3.1 Definición

Micro-ROS es un *framework* que acerca las aplicaciones robóticas diseñadas para infraestructuras de gran tamaño a dispositivos con recursos limitados como son los microcontroladores. Este software lleva la interfaz de programación de ROS a estos dispositivos y permite integrarlos en los sistemas basados en ROS 2. La combinación entre ROS 2 y micro-ROS da como resultado un marco robótico que reduce las barreras de entrada al mercado, reduciendo costes y acelerando el desarrollo de robots.

Figura 4: Logotipo de micro-ROS

La contribución de micro-ROS al mundo de la robótica va más allá. El poder adaptar el sistema operativo de robots a sistemas embebidos permite la interoperabilidad que exigen los sistemas robóticos distribuidos para explotar la creciente superposición entre la robótica, los dispositivos integrados y el IoT. De este

modo, se simplifica la construcción y el diseño de aplicaciones para sistemas robóticos de gran tamaño, pudiendo dividirse estos en sistemas aislados más pequeños y sencillos capaces de conectarse entre si, dotando al sistema general de más información acerca del entorno, permitiendo que los sistemas robóticos verdaderamente distribuidos interactúen de forma aún más inteligente con el mundo que les rodea. [11]

2.3.2 Historia

Micro-ROS surgió a finales del año 2018, durante la celebración de la «ROSCon», el evento más importante para la comunidad de ROS. Durante la conferencia se habló sobre los beneficios que podría suponer la integración de ROS2 en los microcontroladores. [23]

Durante 2019, micro-ROS comenzó a utilizarse en los primeros RTOS. El primero en incorporarlo en sus librerías fue NuttX, el cual desarrolló una serie de aplicaciones de prueba en mayo de ese mismo año. [25]

En 2020, micro-ROS continuó creciendo y realizando proyectos en común con otros RTOS como FreeRTOS o Zephyr. Sin embargo, el mayor avance realizado en ese año fue el desarrollo de nuevas versiones de Micro XRCE-DDS, el agente de micro-ROS encargado de conectar el mundo de los microcontroladores con el espacio de datos de ROS. [24]

Ese mismo año se incluyó micro-ROS en la IDE de Arduino, un avance muy notable ya que se trata de uno de los entornos de desarrollo más utilizados en lo que a microcontroladores se refiere. [22]

Durante este año se ha hecho realidad la noticia que afirma que micro-ROS se integrará en el sistema operativo de Microsoft, el Microsoft Azure RTOS. El hecho de que micro-ROS se haya incorporado en un software soportado por Microsoft le otorga una estabilidad y un renombre que impulsará todavía más el desarrollo de esta tecnología. [17]

Figura 5: Logotipo de microsoft

2.3.3 Proyección futura

Recientemente ha tenido lugar la conferencia de «ROS World 2021». En ella se han mostrado los nuevos avances que tendrán lugar en el futuro cercano del universo de ROS. [32]

En lo referido a micro-ROS, se ha anunciado una nueva librería del cliente y nuevas funciones del middleware. En particular, se ha revelado el uso de la interfaz de Micro XRCE-DDS para transportes personalizados, el paquete de diagnósticos de micro-ROS y el concepto de trabajador para la gestión de la ejecución en la librería del cliente en C, rclc.

Además, se ha confirmado la posibilidad del desarrollo de aplicaciones profesionales utilizando IDEs basados en Eclipse.

Figura 6: Logotipo de ROS World 2021

Estas noticias son muestras de que micro-ROS tiene un futuro muy prometedor por delante, apoyado por una comunidad y un entorno muy favorables. La clave de esta previsión de futuro es el esfuerzo que está realizando el mundo de ROS por acercar esta tecnología a programadores inexpertos para ampliar en gran medida el alcance de la robótica.

CAPÍTULO 3

Software

Como ya se ha comentado anteriormente, el foco principal de este trabajo reside en analizar las características en tiempo real de ROS2 y micro-ROS. Sin embargo, la comunicación entre estos sistemas operativos y los sistemas reales requiere de la participación de otros softwares intermedios o "middlewares" que facilitan la comunicación con el mundo real.

En este capitulo se va a explicar el software empleado para la realización del trabajo y la instalación del mismo.

La estructura del software empleada en el análisis es la siguiente.

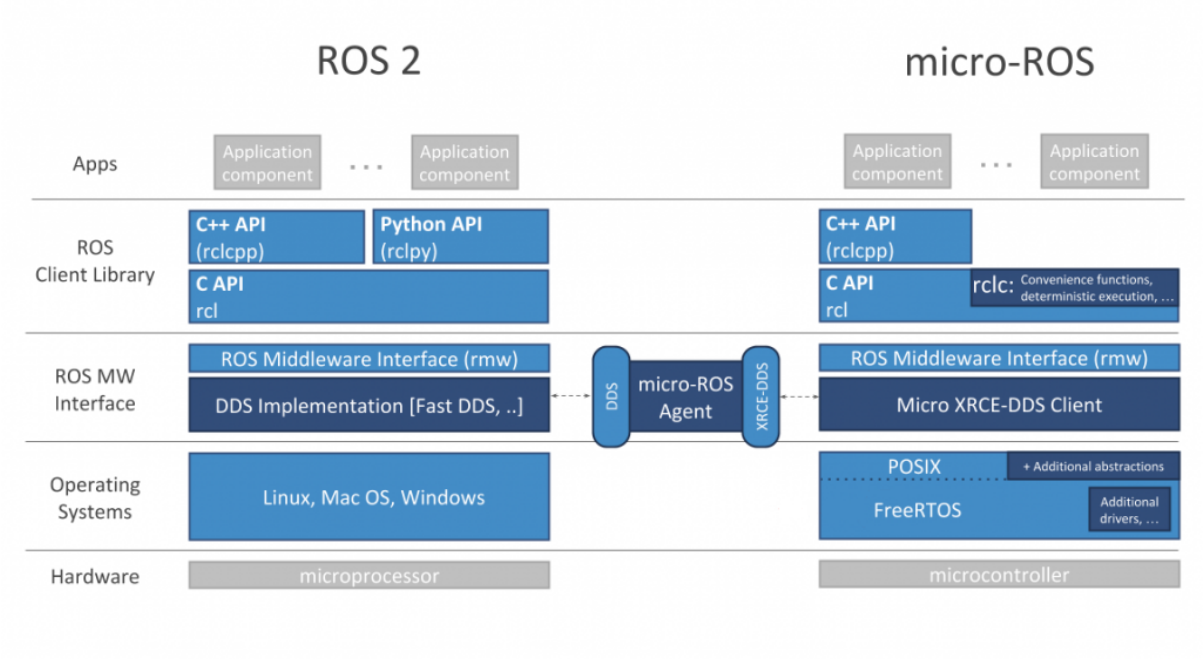

Figura 1: Comparación de la estructura de ROS 2 y micro-ROS (Fuente: freertos.org)

3.1 ROS 2

Desde que ROS comenzó en 2007, ha cambiado mucho la robótica y la propia comunidad de ROS. Con el objetivo de adaptar esos cambios, recientemente se ha lanzado una nueva versión de este software llamada ROS 2. Esta recoge todo lo bueno que tenía ROS 1 y mejora aquello que se había quedado algo obsoleto.

El software de ROS 2 se mantiene constantemente actualizado. Cada cierto tiempo se lanza una nueva distribución, esto es, un conjunto versionado de paquetes de ROS. Las actualizaciones se realizan de esta forma de modo que se permite a los desarrolladores trabajar con una base de código relativamente estable hasta que estén preparados para hacer avanzar todo su trabajo a la siguiente distribución. Por lo tanto, una vez que se libera una distribución, se trata de limitar los cambios a la corrección de errores y a las mejoras que no rompan el núcleo de los paquetes. [29]

Las dos distribuciones que están activas actualmente son "Foxy fitzroy", que fue lanzada en junio de 2020, y "Galactic Geochelone", que es la más reciente, lanzada en mayo de 2021.

Para la realización de este trabajo se ha escogido la distribución de Foxy, ya que, a pesar de no ser la más novedosa, es la distribución que da soporte al software de micro-ROS, el cual se explicará en el siguiente apartado.

Figura 2: Logotipo de la distribución «Foxy fitzroy»

Se pueden instalar los paquetes de ROS 2 Foxy Fitzroy tanto para Linux (Ubuntu), Windows o MacOS. En nuestro caso se ha escogido la distribución de Linux ya que originariamente ROS fue creada para este sistema operativo y está más optimizada.

3.1.1 Conceptos

Para comprender el análisis llevado a cabo en este trabajo es necesario conocer de una manera básica como funciona ROS 2. El sistema operativo de robots funciona como un nexo entre dos o más sistemas o nodos.

Estos nodos tienen varias formas de comunicarse. La más sencilla es mediante topics. Estos topics actúan como buses de información para intercambiar mensajes entre nodos. Los nodos pueden actuar como publicadores (publisher) o como suscriptores (subscribers). Los publicadores son los encargados de publicar mensajes al topic y los suscriptores son los que reciben esos mensajes del topic.

Otra forma de comunicación entre nodos es mediante servicios. Estos están basados en un modelo de solicitud y servicio. En este caso existe un solo nodo que actúa como servidor (server) y uno o más nodos que actúan como clientes (clients). Los clientes demandan un servicio y el servidor responde

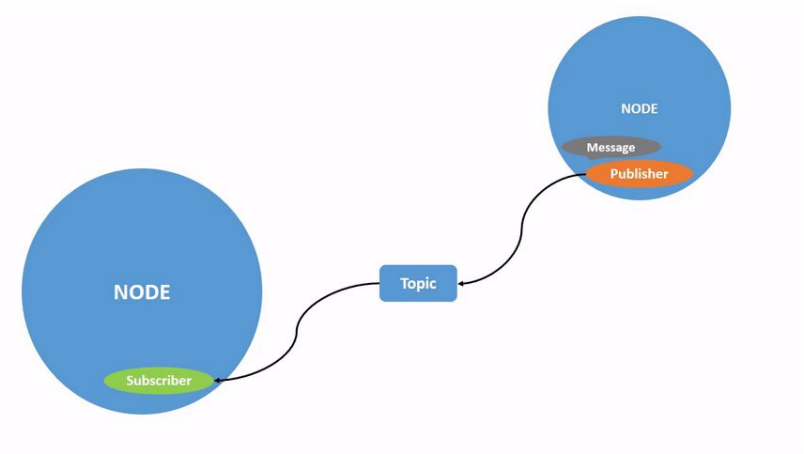

Figura 3: Funcionamiento de un topic (Fuente: ros.org)

con un mensaje. A diferencia de la comunicación mediante topics, en este caso los clientes solo envían información cuando esta ha sido pedida por otro nodo o cliente.

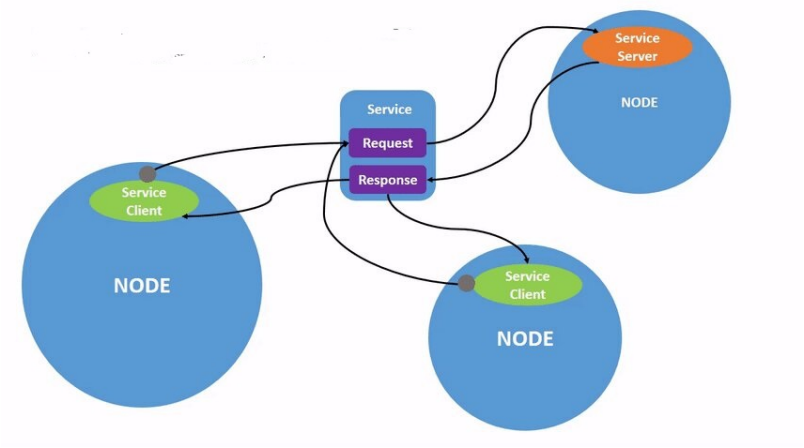

Figura 4: Funcionamiento de un servicio (Fuente: ros.org)

Finalmente, los nodos también pueden comunicarse mediante acciones. Estas están constituidas por topics y servicios. El modelo de comunicación es similar al de los servicios, con la peculiaridad de que cuentan con un tópico que actúa de feedback entre el servidor y el cliente, y dos servicios, uno para el objetivo que quiere cumplir el cliente (goal service) y otro para los resultados obtenidos (result service).

Por otro lado, también es posible modificar el estado de un nodo mediante parámetros. Estos son características propias del nodo que pueden ser modificadas por los servidores de ROS. [30]

3.1.2 Instalación

El proceso de instalación de ROS 2 se encuentra perfectamente explicado en la documentación oficial, en la pagina web [28]

Hay dos formas de instalar los paquetes de ROS 2 para Ubuntu. A continuación se explicará de forma resumida la instalación llevada acabo para la realización de este trabajo.

Se ha escogido la instalación con los paquetes Debian, debido a su sencillez y rapidez. En primer lugar es necesario asegurarse que nuestro local soporta el formato de codificación UTF-8.

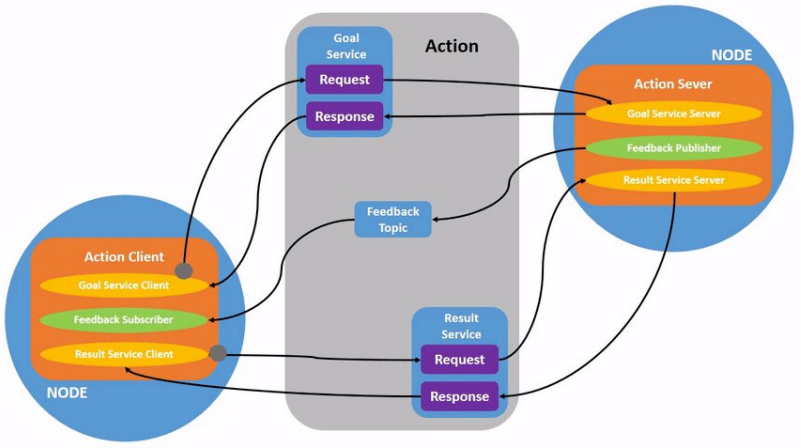

Figura 5: Funcionamiento de una acción (Fuente: ros.org)

En segundo lugar es necesario descargar la herramienta avanzada de paquetes (APT) de ROS 2.

Finalmente, se instalan los paquetes de ROS 2. Para ello hay que actualizar la caché del repositorio de la herramienta de paquetes y ya se podrá utilizar para realizar la instalación de escritorio, que contiene el ROS, demos, y tutoriales; y la instalación básica que proporciona al sistema las librerías, los paquetes con los mensajes y las herramientas de la linea de comandos.

Por último, es importante advertir que cada vez que se vaya a utilizar ROS 2 es necesario añadir el fichero "setup.bash" a la lista fuente.

Se muestran a continuación los comandos necesarios para ejecutar dichas acciones. [31]

```
# Set locale

locale  # check for UTF-8
sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
locale  # verify settings

# Setup Sources

sudo apt update && sudo apt install curl gnupg2 lsb-release
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key  -o /usr/share/
↪keyrings/ros-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.
↪gpg] http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" | sudo tee /etc/apt/
↪sources.list.d/ros2.list > /dev/null

# Install ROS 2 packages

sudo apt update
sudo apt install ros-foxy-desktop
sudo apt install ros-foxy-ros-base

#Environment setup

source /opt/ros/foxy/setup.bash
```

3.2 Micro-ROS

3.2.1 Características principales

Micro-ROS posee siete características claves que lo convierten en un software optimizado para microcontroladores: [21]

- Una API adaptada para microcontroladores que incluye todos los conceptos principales de ROS: Este *framework* adaptado cuenta con las mismas prestaciones principales que ROS 2, como son la publicación y suscripción a mensajes de un tópico por parte de nodos, la mecánica de cliente/servicio, el ciclo de vida y el gráfico de nodos. Esta API se basa en la biblioteca estándar de soporte de clientes de ROS 2 (rcl) y un conjunto de extensiones (rclc), que se explicarán posteriormente.
- Integración perfecta con ROS 2: El agente de micro-ROS se conecta con los nodos de los microcontroladores a través de sistemas ROS 2 estándar. Esto permite acceder a los nodos micro-ROS con las herramientas y APIs conocidas de ROS 2 como si se trataran de nodos suyos.
- Un middleware con recursos muy limitados pero de gran flexibilidad: Micro-ROS utiliza Micro XRCE-DDS de eProsima como middleware para sistemas embebidos. Este software es el nuevo estándar de DDS para entornos con recursos limitados, el cual se explicará en el siguiente capítulo. Para la integración con la interfaz del middleware de ROS (rmw) en la pila de micro-ROS, se introdujeron herramientas de memoria estática para evitar asignaciones de memoria dinámica en tiempo de ejecución.
- Soporte de varios sistemas operativos en tiempo real con un sistema de compilación genérico: Otro de los softwares requeridos para la ejecución de programas en sistemas de tiempo real es un sistema operativo en tiempo real, el cual se explicará más adelante. Micro-ROS soporta tres populares sistemas operativos en tiempo real (a partir de ahora RTOS) de código abierto: FreeRTOS, Zephyr y Nuttx. Además puede ser portado a cualquier RTOS que tenga una interfaz POSIX. Los sistemas de compilación específicos de RTOS están integrados en algunos scripts de configuración genéricos, que se proporcionan como un paquete de ROS 2. Además, micro-ROS proporciona herramientas específicas para algunos de estos RTOS.
- Software de licencia permisiva: Micro-ROS se encuentra bajo la misma licencia que ROS 2, "Apache License 2.0". Esto se aplica a la biblioteca del cliente de micro-ROS, la capa de middleware y las herramientas.
- Comunidad y ecosistema muy activos: Micro-ROS ha sido desarrollado por una comunidad autoorganizada y en constante crecimiento, respaldada por el "Embedded Working Group", un grupo serio de trabajo de ROS 2. Esta comunidad proporciona apoyo a través de GitHub y comparte tutoriales de nivel básico. Aparte de eso, también crea herramientas en torno a micro-ROS para optimizar las aplicaciones ya creadas al hardware del microcontrolador. Estas permiten comprobar el uso de la memoria, el consumo de tiempo de la CPU y el rendimiento general.
- Mantenibilidad e interoperabilidad a largo plazo: Micro-ROS está formado por varios componentes independientes. Varios RTOSes de código abierto con cierto renombre, un middleware estandarizado y la biblioteca estándar de soporte de clientes ROS 2 (rcl). De este modo se minimiza la cantidad de código específico de micro-ROS para su mantenimiento a largo plazo. Al mismo tiempo, la pila de micro-ROS conserva la modularidad de la pila estándar de ROS 2. Esto se traduce en que el software de micro-ROS no depende de si mismo para garantizar un buen mantenimiento, sino que está respaldado por otros componentes con más soporte detrás y que podrían ser sustituibles.

3.2.2 Instalación

Después de instalar ROS 2, es necesario crear un espacio de trabajo para micro-ROS. Una vez creado, se clona el repositorio de github que contiene las herramientas y los ficheros para instalar micro-ROS. Finalmente, se compilan todos los ficheros y se obtendrían las herramientas principales de micro-ROS. [14]

```
# Source the ROS 2 installation

source /opt/ros/ $ROS_DISTRO /setup.bash

# Create a workspace and download the micro-ROS tools

mkdir microros_ws

cd microros_ws

git clone -b $ROS_DISTRO https://github.com/micro-ROS/micro_ros_setup.git src/micro_ros_setup

# Update dependencies using rosdep

sudo apt update  && rosdep update

rosdep install --from-path src --ignore-src -y

# Install pip

sudo apt-get install python3-pip

# Build micro-ROS tools and source them

colcon build

source install /local_setup.bash
```

3.2.3 Arquitectura modular

Micro-ROS sigue la arquitectura de ROS 2, y aprovecha su capacidad de conexión del middleware para utilizar el DDS para microcontroladores (DDS-XRCE). Además utiliza los RTOS basados en POSIX en lugar de Linux.

A continuación se procederá a explicar los componentes que forman la arquitectura de Micro-ROS divididos en tres grupos: librería del cliente, middleware y RTOS.

3.2.4 Librería del cliente

El objetivo general de esta librería es proporcionar todos los conceptos relevantes de ROS 2 en implementaciones adecuadas para microcontroladores y posteriormente lograr la compatibilidad de la API con ROS 2 para facilitar la portabilidad. Para minimizar el coste de mantenimiento a largo plazo, se trata de utilizar las estructuras de datos y los algoritmos existentes de la pila de ROS 2, o bien introducir los cambios necesarios en la pila principal. Esto genera una preocupación por la dudosa aplicabilidad de las capas existentes de ROS 2 en los microcontroladores en términos de eficiencia en tiempo de ejecución, la portabilidad a diferentes RTOS, la gestión de memoria dinámica, etc.

C es el lenguaje de programación dominante en los microcontroladores. Sin embargo, existe una clara tendencia a utilizar lenguajes de alto nivel, especialmente C++, debido a que los microcontroladores más

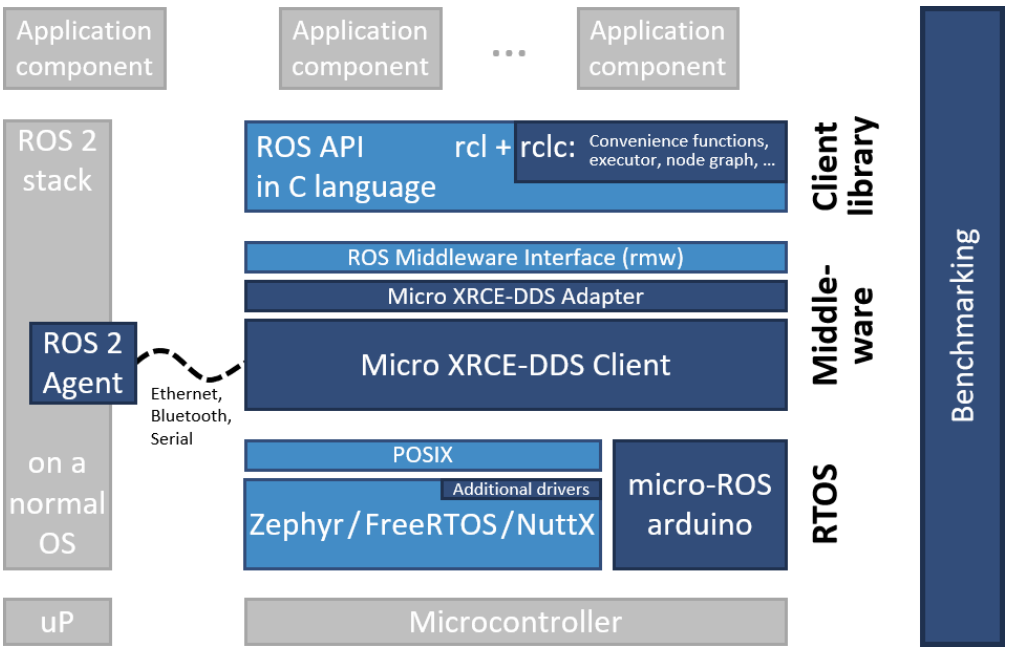

Figura 6: Estructura de micro-ROS (Fuente: micro-ROS.org)

modernos ya cuentan hasta con una mayor memoria RAM. Es por ello por lo que micro-ROS pretende ofrecer y soportar dos APIs.

- La API en C basada en la librería de soporte de ROS 2 (rcl): Esta API está formada principalmente por paquetes modulares para el diagnóstico, la gestión de la ejecución y los parámetros.
- La API en C++ basada en la rclcpp de ROS 2: Esta API en cambio, requiere primero de la aptitud de rclcpp para su uso en microcontroladores, en particular cuando se trata de la memoria, el consumo de CPU y la gestión de la memoria dinámica. Esta incluye las estructuras de datos relacionadas con la generación de mensajes como pueden ser los topics, los servicios y las acciones.

Dentro de estas APIs existen paquetes diseñados específicamente para micro-ROS. La librería rclc cuenta con numerosas extensiones dedicadas a microcontroladores. Cuenta con funciones como temporizadores, logging, gráficos específicos, modificación de parámetros, etc.

Además de estas aplicaciones, se han desarrollado varios conceptos avanzados en el contexto de la librería del cliente. En general, estos conceptos se desarrollan primero para el rclcpp estándar antes de implementar una versión en C adaptada. Estas funciones son las siguientes: [12]

- Ejecutor en tiempo real: El objetivo de este módulo consiste en aportar mecanismos de tiempo real prácticos y fáciles de usar que proporcionen soluciones para garantizar los requisitos de tiempo demandados. También pretende integrar funcionalidades de tiempo real o no real en una plataforma de ejecución y soporte específico para RTOS y microcontroladores.
- Ciclo de vida y modos del sistema: En micro-ROS se ha detectado que el entrelazamiento de la gestión de tareas, la gestión de imprevistos y la gestión de errores del sistema, que se manejan en la capa de deliberación, generalmente conduce a la alta complejidad del flujo de control, algo que podría reducirse introduciendo abstracciones adecuadas para las llamadas y notificaciones orientadas al sistema. El objetivo de esta funcionalidad reside en proporcionar abstracciones y funciones marco adecuadas para la configuración del tiempo de ejecución del sistema y el diagnóstico de errores y contingencias del sistema.
- Transformación integrada: El gráfico de transformación es una herramienta que, desde su lanzamiento, ha sido fundamental para los marcos de trabajo de robótica. Sin embargo, un problema

persistente ha sido su alto consumo de recusos. Micro-ROS ejecuta el árbol de transformación dinámico en un dispositivo integrado, manteniendo el uso de recursos al mínimo, basándose en un análisis de los detalles espaciales y temporales que realmente necesitan.

Figura 7: Arquitectura de la librería del cliente (Fuente: fiware.com)

3.2.5 Middleware

La principal característica de los softwares de robots es la comunicación entre distintos nodos que permita el intercambio de información con unas características determinadas.

Para implementar todos esos conceptos de comunicación, en ROS 2 se decidió hacer uso de un middleware ya existente llamado DDS. De esta forma, ROS 2 puede aprovechar una implementación enfocada en ese sector ya existente y bien desarrollada. [2]

Figura 8: Arquitectura del middleware (Fuente: fiware.com)

DDS son las siglas de Data Distribution Service. Es un servicio de distribución de datos que sirve como estándar de comunicación de sistemas en tiempo real para los middlewares de tipo publish/subscribe, como puede ser ROS. Fue creado debido a la necesidad de estandarizar los sistemas centrados en datos. [34]

Existen numerosas implementaciones distintas de DDS y cada una tiene sus ventajas y sus desventajas en términos de plataformas soportadas, rendimiento, licencias, dependencias y huellas de memoria. Es por ello por lo que ROS pretende soportar múltiples implementaciones DDS a pesar de que cada una de ellas difiera ligeramente en su API. Para abstraerse de dichas especificaciones, se ha introducido una interfaz abstracta que puede ser implementada para diferentes DDS. Esta interfaz de middleware define la API entre la librería del cliente de ROS y cualquier implementación específica.

Como ya se ha comentado en el anterior párrafo, ROS 2 da soporte a varias DDS. La más utilizada y considerada la DDS por defecto en la distribución «Foxy fitzroy» es la "Fast DDS" de eProsima. [9] Esta implementación está diseñada en C++ e implementa el protocolo RTPS (Real Time Publish Subscribe), el cual permite comunicaciones a través de distintos medios como el protocolo de datagrama de usuario (UDP), un protocolo ligero de transporte de datos que funciona sobre IP. [1]

Para adaptar todo este mecanismo de comunicación a Micro-ROS, eProsima ha desarrollado "Micro XRCE-DDS". Esta adaptación permite comunicar entornos con recursos extremadamente limitados (eXtremely Resource Constrained Environments, XRCE) con una red existente de DDS. La librería Micro XRCE-DDS implementa un protocolo de cliente/servidor que permite a los microcontroladores participar en comunicaciones de DDS. El agente de Micro XRCE-DDS actúa como un puente entre el cliente

y el espacio de datos de DDS y permite a estos dispositivos actuar como publicadores y suscriptores o como clientes y servidores.

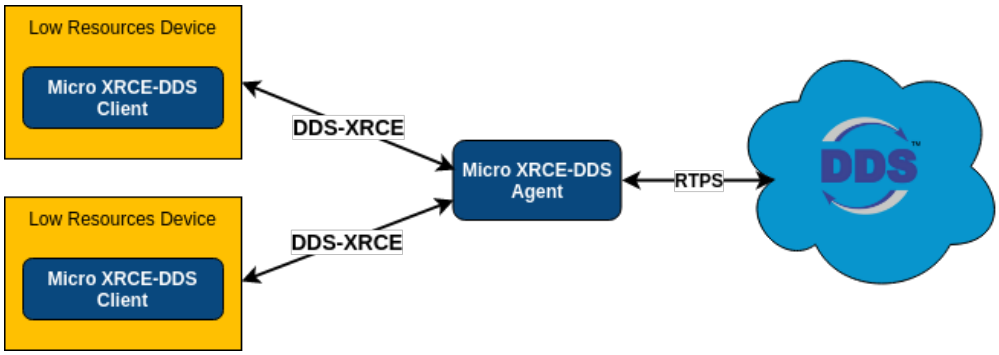

Figura 9: Arquitectura de Micro XRCE-DDS (Fuente: freertos.org)

Dentro de las características principales de Micro XRCE-DDS, cabe destacar las siguientes:

- Alto rendimiento: El cliente utiliza una librería de serialización de bajo nivel que codifica en XCDR.
- Bajo consumo de recursos: El cliente de la librería XRCE está libre de memoria dinámica y estática, por lo que la única huella de memoria se debe al crecimiento de la pila. Puede gestionar un emisor/suscriptor simple con menos de 2 kB de RAM. Además el cliente está construido según un concepto de perfiles, lo que permite añadir o eliminar funcionalidades a la librería al mismo tiempo que modifica su tamaño.
- Multiplataforma: Las dependencias del sistema operativo son módulos aditivos, por lo que los usuarios pueden implementar los módulos específicos de cada plataforma a la librería del cliente. Por defecto, el sistema permite trabajar con los sistemas operativos estándar Windows y Linux, y con los RTOS Nuttx, FreeRTOS y Zephyr.
- Multitransporte: A diferencia de otros middlewares de transferencia de datos, XRCE-DDS soporta múltiples protocolos de transporte de forma nativa. En concreto, es posible utilizar los protocolos UDP, TCP o un protocolo de transporte en serie personalizado.
- De código abierto: La librería del cliente, el ejecutable del agente, la herramienta de compilación y otras dependencias internas son libres y de código abierto.
- Dos modos de funcionamiento: Micro XRCE-DDS soporta dos modos de funcionamiento. El modo *best-effort* implementa una comunicación rápida y ligera, mientras que el modo *reliable* asegura la fiabilidad independientemente de la capa de transporte utilizada.

3.2.6 RTOS

Como ya se ha explicado previamente, RTOS significa sistema operativo en tiempo real. Esto es un sistema operativo ligero que se emplea para facilitar la multitarea y la integración de tareas en sistemas con recursos y tiempo limitados. La clave de un RTOS es la previsibilidad y el determinismo en el tiempo de ejecución más que la inmediatez, ya que lo fundamental en un sistema que opera de este modo es que realice una serie de tareas en un tiempo determinado, y no necesariamente lo más rápido posible. [5]

Un sistema operativo de este tipo cuenta con las siguientes características: no utiliza gran cantidad de memoria, es susceptible de actuar tras eventos realizados en el soporte físico, un tiempo de respuesta predecible, fiabilidad y multi-arquitectura (esto es la posibilidad de portar el código a cualquier tipo de CPU). [37]

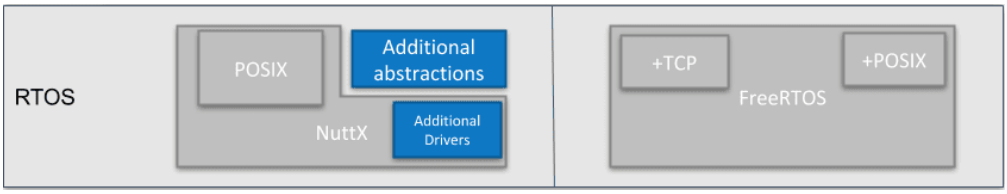

Figura 10: Arquitectura del RTOS (Fuente: fiware.com)

Los RTOS suelen utilizar capas de abstracción de hardware que facilitan el uso de recursos del hardware, como temporizadores y buses de comunicación, aligerando el desarrollo y permitiendo la reutilización de código. Además, ofrecen entidades de hilos y tareas que proporcionan las herramientas necesarias para implementar el determinismo en las aplicaciones. La programación consta de diferentes algoritmos, entre los que mejor se adapten a sus aplicaciones.

Debido a todos los beneficios que ofrecen estos sistemas operativos, micro-ROS los integra en su pila de software. Esto mejora las capacidades de micro-ROS Y permite reutilizar todas las herramientas y funciones proporcionadas por estos.

Al igual que los sistemas operativos convencionales, los RTOS también tienen diferentes soportes para las interfaces estándar. Esto se establece en una familia de estándares denominada POSIX. Este está basado en Linux, el sistema operativo nativo de ROS 2, por lo que la portabilidad de gran parte del código de este a micro-ROS se facilita empleando los RTOS de este grupo. Tanto NuttX como Zephyr cumplen en buena medida con los estándares POSIX, haciendo que el esfuerzo de portabilidad sea mínimo, mientras que FreeRTOS proporciona un plugin, FreeRTOS+POSIX, gracias al cual una aplicación existente que cumpla con POSIX puede ser fácilmente portada al ecosistema FreeRTOS. [19]

A pesar de que todos utilizan el mismo código base de micro-ROS y que sus herramientas han sido integradas en el sistema de compilación de ROS 2, existen notables diferencias en sus características. [15]

A la hora de escoger un RTOS aparecen varios factores a tener en cuenta: La responsabilidad y exposición legal, el rendimiento, las características técnicas, el coste, el ecosistema, el middleware a emplear, el proveedor y la preferencia de ingeniería. [4]

FreeRTOS ha sido el sistema operativo en tiempo real escogido para la realización de este análisis, debido a que es el que mejor se adapta a la placa que se usará en el mismo. Este es distribuido bajo la licencia MIT. Las propiedades clave de este RTOS son las herramientas de gestión de memoria que contiene, los recursos de transporte que ofrece, TCP/IP y IwIP, las tareas estándar y ociosas disponibles con prioridades asignables, la disponibilidad de la extensión POSIX y el tamaño tan reducido que ocupa, permitiendo ser utilizada en prácticamente cualquier microcontrolador. [20]

Figura 11: Logotipo de FreeRTOS

CAPÍTULO 4

Hardware

Micro-ROS tiene como objetivo llevar ROS 2 a un amplio conjunto de microcontroladores para conseguir tener entidades de ROS 2 de primera clase en el mundo embebido. Los principales objetivos de micro-ROS son las familias de microcontroladores de gama media de 32 bits. Normalmente, los requisitos mínimos para ejecutar micro-ROS en una plataforma embebida son las restricciones de memoria. En general, micro-ROS necesitará microcontroladores que contengan decenas de kilobytes de memoria RAM y periféricos de comunicación que permitan la comunicación entre el cliente y el agente de micro-ROS.

El soporte de hardware de micro-ROS se divide en dos categorías, las placas con soporte oficial y las placas soportadas por la comunidad. Dentro de la gran cantidad de gamas de placas que poseen soporte directo de micro-ROS, encontramos dispositivos de proveedores con cierto renombre como Renesas, Espressif, Arduino, Raspberry, ROBOTICS, Teensy, ST, Olimex, etc. [18]

4.1 ESP-32

La placa que se ha utilizado para la medición de los tiempos de respuesta ha sido la "Espressif ESP32". Esta posee numerosas cualidades positivas que se explicarán a continuación. Sin embargo, las razones principales de esta elección han sido su bajo consumo, la posibilidad de conexión vía WIFI y la activa comunidad y soporte que ofrece micro-ROS a Espressif.

Espressif es una empresa pionera en el mundo del internet de las cosas (IoT). Son un equipo de especialistas en creación de chips y desarrollo de software. Una particularidad de esta empresa es el apoyo que proporcionan a sus clientes para construir sus propias soluciones y conectar con otros socios del mundo IoT. Los productos de Espressif se han implementado principalmente en el mercado de placas, cajas OTT (servicios de libre transmisión), cámaras e IoT. [3]

El modelo concreto que se ha utilizado es la "ESP32-DevKitC V4". Forma parte de las placas de desarrollo o "DevKits", dispositivos de reducido tamaño y accesibles para programadores inexpertos diseñadas para facilitar el prototipado. Estas están alimentadas por un módulo que les suministra la mayoría de funcionalidades. La disposición de los pines de entrada/salida están repartidos en ambos lados para facilitar la interconexión. Los desarrolladores pueden conectar los periféricos con cables puente o montar

Figura 1: Placa ESP32

la DevkitC en una protoboard. Una de las particularidades que más destacan de estos modelos son la posibilidad de conexión vía Wi-Fi y Bluetooth. Esto permite realizar prototipos inalámbricos que simulan un entorno con muchas posibilidades que se asemeja más a la idea original del IoT. [6]

La DevKitc V4 que se ha utilizado cuenta con los siguientes componentes:

- Un módulo ESP32-WROOM-32D
- Botón de reseteo "EN"
- Botón de descarga "Boot": Presionando el botón "Boot" y después pulsando el botón "EN", inicia la descarga del firmware a través del puerto en serie.
- Puente de USB a UART: Permite la transferencia de datos desde un puerto de tipo USB a un puerto UART.
- Puerto Micro USB: Sirve tanto como fuente de alimentación como de interfaz de comunicación entre el ordenador y el módulo ESP32-WROOM-32D.
- Led de encendido de 5V: Se enciende cuando el USB u otra fuente de alimentación está conectada a la placa.
- Entradas/salidas: La mayoría de los pines del módulo están repartidos en los cabezales de los pines de la placa. A través de ellos se pueden programar múltiples funciones como PWM, ADC, DAC, I2C, I2S, SPI, etc.

El ESP32-WROOM-32D es un módulo de microcontrolador genérico que se dirigen a una amplia gama variedad de aplicaciones, que van desde redes de sensores de baja potencia hasta otras tareas de mayor exigencia, como codificación de voz, transmisión de música y decodificación de MP3.

El núcleo de este módulo es el chip ESP32-D0WD. El chip está diseñado para ser escalable y adaptable. Existen dos núcleos de CPU que pueden ser controlados individualmente y la frecuencia de reloj es ajustable de 80 MHz a 240 MHz. El chip cuenta con un coprocesador de bajo consumo que puede utilizarse en lugar de la CPU para ahorrar energía en tareas que no requieren mucha potencia de cálculo, como la monitorización de periféricos. ESP32 cuenta con un amplio conjunto de periféricos integrables, que van desde sensores táctiles capacitivos, sensores Hall, interfaz de tarjeta SD, Ethernet, SPI de alta velocidad, UART, I2S e I2C.

La integración de Bluetooth y Wi-Fi garantiza que se pueda abordar una amplia gama de aplicaciones y una gran polivalencia del módulo. El uso de Wi-Fi permite un gran alcance físico y la conexión directa a Internet a través de un router, mientras que el uso de Bluetooth permite al usuario conectarse

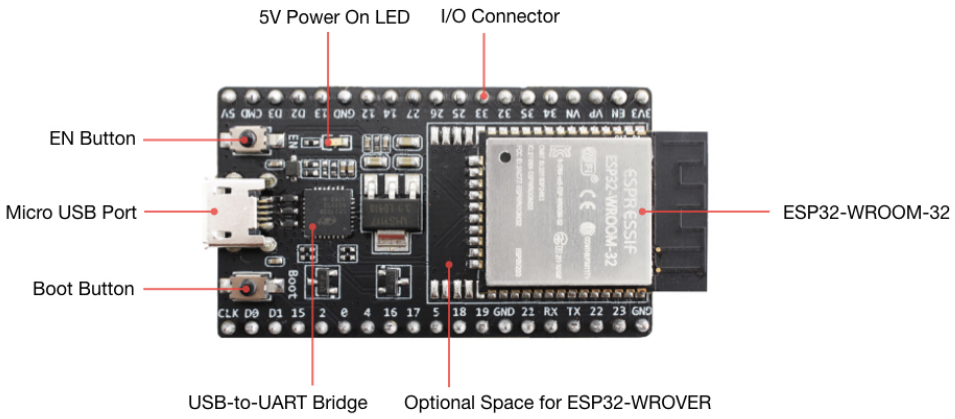

Figura 2: Componentes de la placa ESP32 (Fuente: Espressif)

cómodamente al teléfono o emitir balizas de baja energía para su detección.

La corriente de reposo del chip ESP32 es inferior a 5 uA, hecho que lo convierte adecuado para aplicaciones alimentadas por batería y de electrónica portátil. El módulo admite una velocidad de datos de hasta 150 Mbps y una potencia de salida de 20 dBm en la antena para garantizar el mayor alcance físico posible.

El sistema operativo elegido para ESP32 es freeRTOS con LwIP, aunque también se ha incorporado TLS 1.2 con aceleración por hardware. [7]

Figura 3: Módulo ESP32-WROOM-32D

4.2 Computador

Todo el trabajo se ha realizado haciendo uso de un ordenador personal, de unos 4 años de uso que se encuentra en perfecto estado. Este es un Asus UX340. Este ordenador portátil cuenta con 16 GB de memoria RAM, 256 GB de almacenamiento SSD, arquitectura de 64 bits y un microprocesador Intel i5.

Figura 4: Asus UX430U

Se ha utilizado el sistema operativo Linux, en la distribución Ubuntu 20.04.3 LTS.

4.3 Cable micro-USB

En el transcurso del proyecto se han utilizado dos cables. En primer lugar se utilizó un cable estándar, sin embargo, no permitía entregar toda la potencia requerida por la placa. Seguidamente se sustituyó por un cable de calidad superior.

Figura 5: Cable micro-USB

4.4 Router TP-Link

Para obtener unas medidas que no se vean alteradas por el tráfico que pueda existir en la red doméstica, se ha adquirido un router adicional. Este es un Archer 80 de la marca TP-Link. Cuenta con 1300 Mbps en la banda de 5 GHz y 600 Mbps en la banda de 2.4 GHz. Tiene una tecnología MIMO 3x3 con cobertura de Wi-Fi potenciada. Este se conectará por cable a una red doméstica de 1Gbps de descarga.

Figura 6: Router TP-Link

CAPÍTULO 5

Diseño del análisis

5.1 Preparación previa

Para realizar todas las mediciones de este trabajo, ha sido preciso un estudio previo del entorno y una preparación para la utilización del software a probar.

En primer lugar, es necesario conocer el funcionamiento de la placa. Desde la propia página de Espressif es posible encontrar un documento con todos los pasos detallados para iniciarse en la programación de la placa. [8]

Al principio se necesita instalar los requisitos de la aplicación en función del sistema operativo en el que se opere. Después, hay que instalar una serie de librerías proporcionadas por Espressif denominadas ESP-IDF. Posteriormente, se instalará una serie de herramientas y se configurarán las variables de entorno. Una vez realizados estos pasos, ya será posible crear un proyecto para la placa para comprobar que funciona correctamente.

Una vez comprobado el correcto funcionamiento de la placa, es necesario instalar el software de micro-ROS y realizar una serie de pruebas. Es posible realizar primero una serie de prácticas con clientes creados dentro del propio Linux. Para ello hay que instalar y compilar el firmware adecuado, crear un agente de micro-ROS y ejecutar la aplicación. Si todo funciona correctamente, será posible observar una serie de mensajes publicados en el topic en cuestión. [16]

Después de realizar unas primeras prácticas tanto con el software como con el hardware que se quiere probar, es el momento de unir ambas partes y realizar las primeras pruebas de micro-ROS en la placa.

Para ello hay que seguir un tutorial similar al anterior en el que se explica como realizar una primera aplicación de micro-ROS con conexión vía Wi-Fi. [33]

Inicialmente, hay que crear y configurar un nuevo firmware de trabajo. En este momento hay que escoger el RTOS sobre el que se va a trabajar y descargar sus herramientas y librerías propias. Posteriormente es necesario configurar dicho firmware, especificando la aplicación que se quiere probar y el tipo de conexión que se quiere establecer con la placa. Además, es necesario operar sobre un menú de la propia placa en el que se pueden modificar numerosos aspectos de la misma, como las variables de entorno o las especificaciones de la conexión (p.e. SSID y contraseña Wi-Fi).

Una vez configurado, se compilará y se flasheará a la placa. En este momento se envía la aplicación a la placa vía USB y esta se ejecuta. Sin embargo, es necesario crear un agente en Linux para que la placa pueda conectarse a un espacio de datos y publicar los mensajes. Tras realizar dicha acción, podrá observarse cierta información en el agente creado, confirmando el establecimiento de conexión entre el agente y el cliente. Este será el momento en el que podremos comprobar que todo funciona correctamente. Mediante el comando «ros2 topic list» se mostrará el topic creado y con `ros2 topic echo /[project name]` podremos suscribirnos y observar los mensajes enviados por el cliente.

Después de realizar unas pruebas con las demos que proporciona el sistema operativo, es recomendable realizar una serie de tutoriales más avanzados que proporciona la propia página de micro-ROS.[1] En estos se enseña cómo diseñar tu propia aplicación, incluyendo cómo crear tu propio nodo, tus publishers y subscribers, un temporizador o incluso seleccionar la calidad de la comunicación.

5.2 Estructura principal del análisis

Lo primero que hay que hacer en el momento de diseñar un test de comportamiento de un software es definir los parámetros que se van a medir y en qué condiciones. Existen multitud de variables que se ven afectadas a la hora de realizar una medición que aportan distintos tipos de información. Sin embargo, por distintas razones, no es posible analizar todas ellas y es recomendable centrarse en un número limitado de ellas para indagar más a fondo y obtener unas conclusiones más concisas.

En este experimento se van a analizar la latencia global del sistema operativo en tiempo real, el "throughput" o tasa de transferencia efectiva, el consumo de memoria del sistema y la influencia de una interferencia externa en la red de ROS.

A continuación se explicará detalladamente en que consisten estos parámetros y de que manera pueden afectar a un sistema en tiempo real.

La latencia es el retraso entre los eventos generados por un hardware y la transmisión efectiva de datos. En otras palabras, la latencia es el tiempo que tarda en ejecutarse una tarea desde el momento en el que es ordenada. Esto en un sistema en tiempo real es crucial, ya que es uno de los principales responsables de que se cumplan o no los tiempos que deben de cumplir los sistemas. [35]

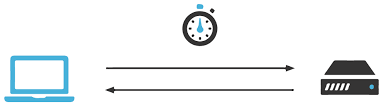

Figura 1: Latencia (Fuente: Inube)

Generalmente, un evento en un sistema de este estilo está formado por distintos eventos más pequeños que dan lugar a la realización del evento principal. Por este motivo, la latencia general del evento se descompone en varias latencias más pequeñas que sumadas dan lugar a la latencia general del evento. En nuestro caso, el evento comprende desde el momento en el que la aplicación ordena el envío del mensaje hasta que este es publicado en el DDS, momento en el que este podrá ser enviado a otros clientes que estén suscritos al topic. Aquí se pueden encontrar distintas latencias. En primer lugar, el tiempo de respuesta que emplea el microcontrolador en reaccionar al envío del mensaje. Seguidamente, cabe recalcar el tiempo que tarda esta información en enviarse desde el cliente al agente al que está conectado. Finalmente, es importante el tiempo que emplea el agente en publicar los mensajes en el DDS.

[1] micro-ROS. Micro-ROS programming tutorial. URL: https://micro.ros.org/docs/tutorials/programming_rcl_rclc/overview.

Este último es el parámetro que se ha escogido para representar en un análisis, ya que el sistema operativo en el que se lanza el agente nos proporciona herramientas que nos indican las latencias que ocurren en el sistema con alta precisión.

El throughput es el segundo parámetro que se va a medir en el test de comportamiento. Esta variable muestra la capacidad de transmisión del sistema. Esta puede verse limitada por distintos aspectos, tanto correspondientes al software tanto como al hardware. En este ámbito, el factor más limitante va a residir en el microcontrolador, ya que es una placa diseñada para operar con recursos muy limitados. En este sentido resultará muy interesante comprobar el momento en el que se produzca la saturación de la placa para determinar las limitaciones del sistema y para qué aplicaciones podría emplearse el microcontrolador.

Figura 2: Throughput (Fuente: Corporate Finance Institute)

Seguidamente, se procederá a estudiar el consumo se memoria del sistema. En un principio, micro-ROS es un software diseñado para microcontroladores, por lo que el efecto de las acciones realizadas por este en el sistema global no deberían ser notables. En este sentido, será interesante comprobar si realmente se trata de un sistema que economiza los recursos y hasta qué punto.

Por último, se va a someter al sistema a una perturbación externa. Se creará un topic adicional y será el propio ordenador el que actue como cliente. Se ha decidido escoger la demo «ping_pong» para este propósito, ya que es una de las demos que trae micro-ROS más completa, ya que crea dos nodos, un publisher y un subscriber y crea una conexión constante entre ellos dos. Una vez añadida esta interferencia en la red, se repetirán las mediciones de la latencia para comprobar si esta se ve afectada y en qué manera.

Los parámetros previamente mencionados aportarán información de gran interés de cara a formalizar una idea general del rendimiento del software y del hardware en valores absolutos. Sin embargo, al no conocerse un estudio semejante, resulta difícil otorgarle un valor relativo a dichos resultados frente a otros sistemas. Es por ello por lo que se han escogido varios escenarios para la realización de pruebas. De este modo sera posible obtener unas conclusiones que expresen tanto un sentido absoluto como relativo.

Se han diseñado cuatro escenarios para la obtención de datos. Como ya se ha comentado previamente, la placa ESP32 cuenta con la peculiaridad de ofrecer conexión vía Wi-Fi, algo poco habitual en placas de este estilo, además de una conexión en serie más convencional. De otro modo, ya se ha explicado en el apartado de «software» que micro-ROS cuenta con dos modos de comunicación para el envío de información. Estos son el modo *reliable*, que requiere de una señal de confirmación por parte del receptor, priorizando la fiabilidad de la comunicación; y el modo *best-effort*, que trata de enviar la mayor cantidad de mensajes a la mayor velocidad, aunque en redes poco robustas puede resultar poco fiable.

De este modo, combinando los dos tipos de conexión y los dos tipos de comunicación se han formado cuatro situaciones que mostraran distintos resultados de los que extraer conclusiones tras ser comparados.

5.3 Herramientas empleadas

La herramienta principal de la que nos serviremos durante la totalidad de la evaluación será una aplicación que se ha diseñado con el propósito exclusivo de someter al sistema a distintas situaciones en las que, mediante otras herramientas, recopilar datos.

La aplicación está programada en C e incluye las librerías de freeRTOS que proporcionan las funciones propias de micro-ROS así como los tipos de mensajes que se van a utilizar.

Esta aplicación será añadida al firmware, compilada y enviada a la placa, donde se ejecutará periódicamente.

A continuación se muestra el código de la aplicación y posteriormente se procederá a explicar los aspectos más reseñables del mismo.

```
#include <stdio.h>
#include <unistd.h>

#include <rcl/rcl.h>
#include <rcl/error_handling.h>
#include <std_msgs/msg/string.h>

#include <rclc/rclc.h>
#include <rclc/executor.h>

#define ARRAY_LEN 1024

#ifdef ESP_PLATFORM
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#endif

#define RCCHECK(fn) {
    rcl_ret_t temp_rc = fn;
    if((temp_rc != RCL_RET_OK)){
        printf("Failed status on line %d: %d. Aborting.\n",__LINE__,(int)temp_rc);
        vTaskDelete(NULL);
    }
}
#define RCSOFTCHECK(fn) {
    rcl_ret_t temp_rc = fn;
    if((temp_rc != RCL_RET_OK)){
        printf("Failed status on line %d: %d. Continuing.\n",__LINE__,(int)temp_rc);
    }
}

rcl_publisher_t publisher;
std_msgs__msg__String msg;

void timer_callback(rcl_timer_t * timer, int64_t last_call_time)
{
    RCLC_UNUSED(last_call_time);
    if (timer != NULL) {
        RCSOFTCHECK(rcl_publish(&publisher, &msg, NULL));
    }
}

void appMain(void * arg)
{
    rcl_allocator_t allocator = rcl_get_default_allocator();
    rclc_support_t support;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    // create init_options
    RCCHECK(rclc_support_init(&support, 0, NULL, &allocator));

    // create node
    rcl_node_t node;
    RCCHECK(rclc_node_init_default(&node, "my_test_app_publisher", "", &support));

    // create publisher
    RCCHECK(rclc_publisher_init_default(
        &publisher,
        &node,
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, String),
        "my_custom_publisher"));

    // create timer,
    rcl_timer_t timer;
    const unsigned int timer_period = 1;
    RCCHECK(rclc_timer_init_default(
        &timer,
        &support,
        RCL_MS_TO_NS(timer_period),
        timer_callback));

    // create executor
    rclc_executor_t executor;
    RCCHECK(rclc_executor_init(&executor, &support.context, 1, &allocator));
    RCCHECK(rclc_executor_add_timer(&executor, &timer));

    msg.data.data = (char *) malloc (ARRAY_LEN * sizeof(char));
    msg.data.size = 0;
    msg.data.capacity = ARRAY_LEN;

    memset(msg.data.data,'1',1024);
    msg.data.size = 1024;

    while(1){
        rclc_executor_spin_some(&executor, RCL_MS_TO_NS(1000));
    }

    // free resources
    RCCHECK(rcl_publisher_fini(&publisher, &node));
    RCCHECK(rcl_node_fini(&node));

    vTaskDelete(NULL);
}
```

En primer lugar, se añaden todas las librerías que se utilizarán y se definen las funciones «RCCHECK» y «RCSOFTCHECK». Estas serán de gran utilidad durante toda la ejecución, ya que se llamarán en el momento de utilizar cualquier otra función para asegurar su correcto funcionamiento en un tiempo establecido. De no ser así se generarán distintos mensajes de error e incluso se forzará la detención de la aplicación en función de la gravedad del fallo. Esto resulta crucial en aplicaciones de este tipo, ya que un pequeño error en los tiempos puede resultar muy significativo en sistemas de tiempo real.

Posteriormente, se crea la función «timer_callback», que se ejecutará cada vez que el timer llegue a cero. En ella simplemente se publica un mensaje siempre que el timer siga contando.

Seguidamente, se crean el nodo y el publisher. En la creación del publisher es en la que se determina tanto la calidad de la comunicación como el tipo de mensaje que este enviará. En este caso se utiliza la función «rclc_publisher_init_default», lo que creará un publisher que actuará bajo el modo *reliable*. Para

cambiar al modo *best-effort*, sería necesario sustituir esta función por «rclc_publisher_init_best_effort», manteniendo iguales los parámetros de la misma. Como se puede observar, el tipo de mensaje escogido ha sido una cadena de caracteres o «string». Esto es debido a la simplicidad que existe para modificar su tamaño y la facilidad de uso.

A continuación, se crean el timer y el executor. Al timer se le asigna el periodo en la variable «timer_period». Esta viene determinada en milisegundos, por lo que en este caso el periodo sería de 1 milisegundo y la frecuencia de 1000 Hz. El executor es el encargado de que cuando el temporizador baje a 0 se ejecute la función «timer callback».

Consecutivamente se completa la cadena de caracteres. Primero se reserva el espacio en memoria que se pretende utilizar y después se rellenan todos esos caracteres con la función memset. En este caso se han reservado y rellenado 1024 caracteres, lo que equivale a 1 kilobyte.

Finalmente, se lanza un bucle infinito en el que simplemente se llama a la función «rclc_spin_some», que llamará al executor cada vez que el contador del timer finalice. Se le ha asignado un «wake up time» de 1000 milisegundos para asegurarse que siempre se ejecute a pesar de que pueda existir un pequeño delay en el sistema.

Esta aplicación será lanzada numerosas veces, asignando en cada ocasión los parámetros que se quieran analizar. Cada vez que se modifique la aplicación será necesario recompilar el firmware.

Una vez diseñada la aplicación es momento de configurar el firmware.

Para ello lo primero que hay que hacer es declarar el modo de conexión que se quiere establecer. Este se realiza mediante los siguientes comandos.

```
ros2 run micro_ros_setup configure_firmware.sh my_test_app -t serial

ros2 run micro_ros_setup configure_firmware.sh my_test_app -t udp -i [IP] -p [port ID]
```

Mediante el primer comando se establecerá una conexión en serie. En el segundo comando se configura una conexión vía Wi-Fi, en el que será necesario añadir la ip de la conexión y el número de puerto que se pretende utilizar, normalmente el 8888.

Si se ha seleccionado la conexión inalámbrica se empleará el siguiente comando para añadir el SSID y la contraseña de nuestra red.

```
ros2 run micro_ros_setup build_firmware.sh menuconfig
```

Finalmente se compilará el firmware completo y se enviará a la placa con los dos siguientes comandos.

```
ros2 run micro_ros_setup build_firmware.sh

ros2 run micro_ros_setup flash_firmware.sh
```

En este momento será necesario lanzar el agente de micro-ROS desde nuestra máquina. En función de si hemos optado por una conexión en serie o inalámbrica emplearemos uno de los dos siguientes comandos:

```
ros2 run micro_ros_agent micro_ros_agent serial --dev [device ID]

ros2 run micro_ros_agent micro_ros_agent udp --port [port ID]
```

El device ID es la identificación de nuestro dispositivo, la cual se puede averiguar escribiendo `ls /dev/serial/by-id/*` en la línea de comandos, y el port ID debe ser el mismo que el seleccionado en la configuración del hardware.

De este modo ya se ejecutará la aplicación y se enviarán los datos al espacio DDS.

Para medir la latencia es imprescindible escoger y conocer una herramienta muy precisa. En este caso se va a utilizar Cyclictest, una herramienta de benchmarking para sistemas en tiempo real. En concreto, sirve para medir la latencia del sistema. [13]

Un análisis de la latencia puede ser muy distinto de otro dependiendo de varios factores y las condiciones en las que se quiera realizar el test. Es por ello por lo que es fundamental configurar bien la herramienta antes de ser utilizada para obtener unos datos fiables.

En este caso se ha utilizado la siguiente configuración:

```
cyclictest -D 1 --verbose -i 100 -p 95
```

El parámetro D indica la duración del test, en este caso de un segundo. «Verbose» expresa que se produzca una salida detallada de la latencia. La opción i muestra el tamaño del intervalo entre medidas, en este caso de 100 micro segundos, por lo que se realizarán un total de 10000 medidas. Finalmente, p indica la prioridad porcentual de los procesos que ocurran en tiempo real, en este caso de máxima prioridad.

Estos resultados han sido volcados a un fichero para analizarlos posteriormente.

Se ha lanzado un análisis por cada escenario, estableciendo la frecuencia en 1000 Hz y el tamaño del mensaje en 1 kilobyte. De este modo, la placa trabajará bajo una gran demanda, sometiéndola a una situación límite. De esta forma, podremos observar la evolución de la latencia cuando la placa utiliza todos sus recursos.

Para medir el throughput se ha utilizado el propio agente de micro-ROS. Añadiendo la opción -v5 después de ejecutar el agente, se muestra por pantalla los mensajes publicados en el DDS. Se ha decidido volcar la salida por pantalla en un fichero.

En este ámbito se han realizado 24 mediciones, 6 por cada escenario. En ellas se ha modificado la frecuencia del envío de mensajes manteniendo el tamaño del mismo.

La recopilación de cada análisis, ha sido de unos 15 segundos, tiempo más que suficiente para generar una muestra amplia del número de mensajes que se ha llegado a publicar por segundo. En la salida del agente también se muestra el tiempo exacto de la publicación de los mensajes por lo que simplemente ha sido necesario realizar una media del número de mensajes publicados por segundo y multiplicarlos por el tamaño del mensaje.

Seguidamente, la medición de la memoria empleada se ha producido utilizando el comando `htop` de Ubuntu, en el que se muestra el consumo de la memoria de cada tarea llevada acabo en cada momento.

La influencia de la perturbación en la red se medirá del mismo modo en el que se ha medido la latencia, pero en este caso solo se utilizaran los escenarios en los que se emplea la conexión Wi-Fi, ya que el ordenador empleado solo puede conectarse al agente creado con un puerto.

Finalmente, cabe destacar que se ha utilizado Jupyter Notebook para realizar las gráficas y los análisis estadísticos.

Figura 3: Logotipo de Jupyter notebook

CAPÍTULO 6

Resultados

A continuación, se muestran los resultados de los análisis y una discusión de estos, además de las conclusiones obtenidas teniendo en cuenta diferentes factores.

6.1 Latencia

Seguidamente, se muestran las gráficas de los datos expresadas como densidades de latencias y un histograma con los datos discretos de todas las latencias.

Observando los resultados durante la recopilación de datos, ha resultado llamativo el hecho de que más del 90 % de las latencias obtenidas se comprenden entre los 3 y los 4 microsegundos. Sin embargo, es un resultado esperado, ya que la frecuencia con la que se deberían enviar los mensajes es de 1000 Hz (periodo de 1 milisegundo), y el intervalo de medida de la latencia es de 100 microsegundos (10 veces menor), por lo que cabe esperar que 9 de cada 10 veces, las latencias producidas sean de eventos que realiza el sistema simplemente por permanecer encendido.

Esto se ha diseñado de esta manera a propósito, puesto que tras realizar ensayos previos se ha detectado que las latencias producidas en el sistema oscilaban entre los 5 y los 50 microsegundos, por lo que con un periodo inferior cabía la posibilidad de que el delay en el sistema fuera superior al intervalo y la medición de esta finalizase antes que el propio retraso. Por otra parte, si se aumentaba en gran medida el intervalo, se corría el riesgo de no registrar las latencias producidas por algunos eventos.

Es por ello por lo que se ha decidido suprimir en las gráficas las mediciones de 3 microsegundos, ya que no son relevantes a la hora de analizar el sistema e impiden observar con claridad el comportamiento de este.

6.1.1 Distribución

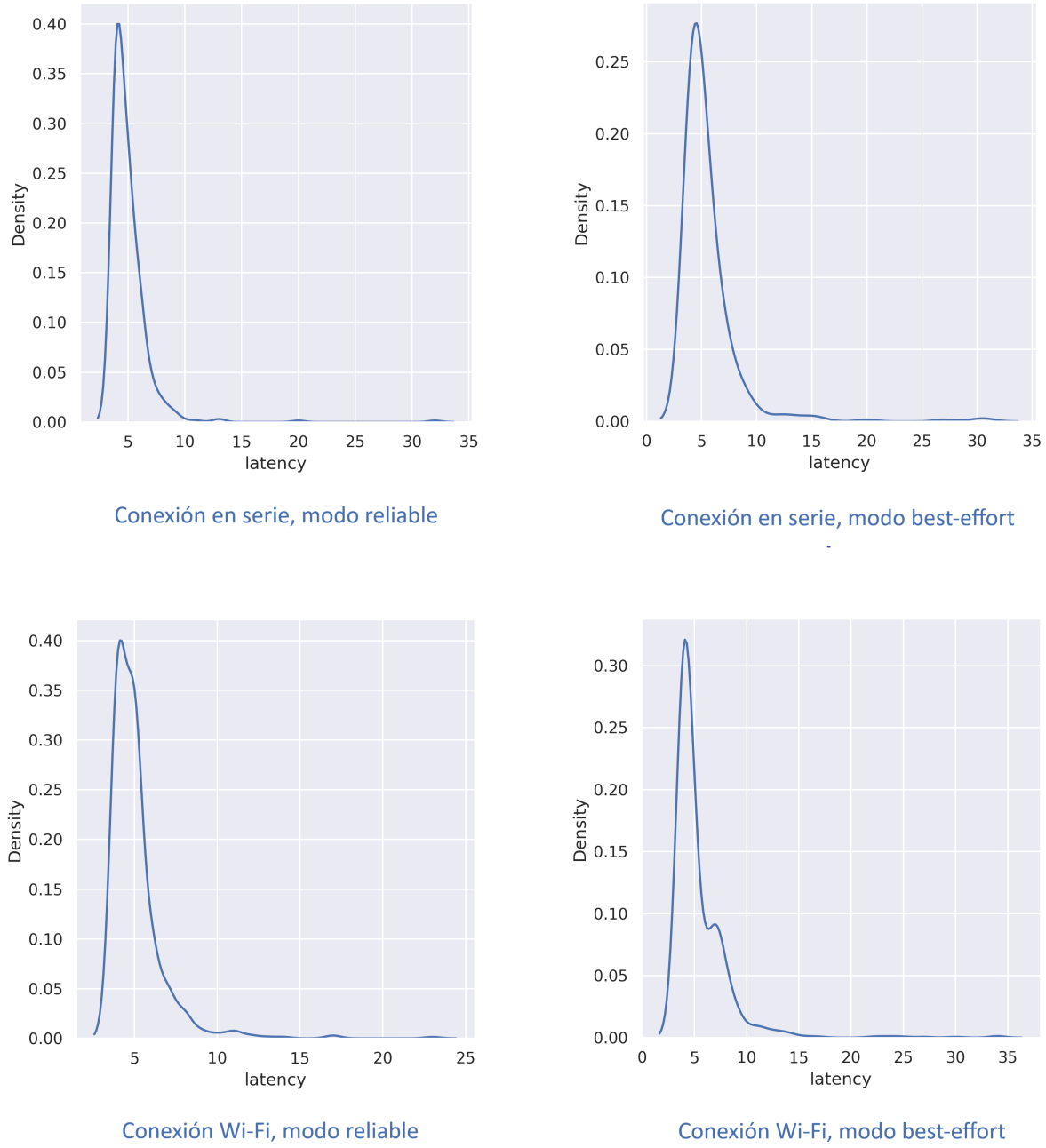

Figura 1: Distribución de latencias

Comparando las cuatro gráficas de las distribuciones de las latencias, lo primero que resalta a simple vista es el hecho de las gráficas del mismo modo de conexión tienen una morfología bastante similar.

Las distribuciones de densidades obtenidas mediante la conexión en serie siguen una curva especialmente regular, con un máximo en una latencia ligeramente inferior a 5 microsegundos, desde donde la función desciende con una pendiente prácticamente constante hasta los 7 microsegundos, donde comienza una curva considerable que produce que la densidad se haga aproximadamente nula a los 10 microsegundos.

Por el contrario, las gráficas resultantes de la conexión Wi-Fi contienen una forma peculiarmente irregular. Es cierto que en ambas se sigue alcanzando el máximo global en una latencia muy similar a las

gráficas anteriores. Sin embargo, se puede observar cómo la pendiente descendente del análisis realizado con conexión Wi-Fi y en modo *reliable* está dividida en dos tramos: una primera pendiente muy moderada hasta los 5 microsegundos seguida de una mucho más brusca que desciende la función desde el 35 % hasta el 7 % de la densidad de la latencia. De la misma manera, el modo *best-effort* también presenta un patrón poco reconocible, debido al cambio de tendencia que existe en la función después de una fuerte bajada. Este punto de inflexión ocurre en torno a los 6 microsegundos y genera un mínimo relativo para que la curva experimente un ligero remonte hasta los 7.5 microsegundos.

Estas diferencias en las morfologías de las gráficas pueden explicarse entendiendo que la conexión en serie da lugar a una transmisión de datos más regular que la conexión inalámbrica.

Por otro lado, también resulta llamativo otro aspecto relacionado con la calidad del servicio escogida. Las gráficas obtenidas de los experimentos en modo *reliable* tienen el 40 % de las latencias obtenidas entre los 4 y 5 microsegundos.

Sin embargo, la distribución de las latencias resultantes de las pruebas realizadas en modo *best-effort* es mucho más homogénea, teniendo estas su máximo en torno al 27 % y el 32 %. Además, la anchura del pico de las bajas latencias es notablemente superior, finalizando este prácticamente en los 10 microsegundos, frente a los 7 microsegundos que presentan las gráficas de modo *reliable*.

Estos resultados son muy comprensibles. El modo *best-effort* permite una comunicación más fluida, lo que indica que bajo la gran demanda que estaba trabajando la placa, se envíen más mensajes en el mismo tiempo que en el modo *reliable*. Al llegar más datos en la misma cantidad de tiempo, el agente tiene una mayor carga de trabajo y realiza más operaciones, ya que tiene que publicar más mensajes en el DDS.

6.1.2 Histograma

El histograma de latencias muestra una información similar a la distribución previamente estudiada.

Sin embargo, cabe destacar que en estos diagramas se puede apreciar más fácilmente cómo la gráfica correspondiente a la conexión en serie con modo *reliable* presenta el patrón más reconocible, al tratarse de una comunicación más estable y menos saturada; y cómo la gráfica correspondiente a la conexión vía Wi-Fi y en modo *best-effort* forma una secuencia mucho más irregular, resultado de una conexión más inestable y de mayor volumen.

Tras analizar estos resultados se llega a la conclusión de que, como era de esperar, la manera en la que se comunican el cliente y el agente repercute notablemente en la manera en la que responde el sistema en cierto real.

Es cierto que, a pesar de que un pequeño retraso en el cumplimiento de una tarea asignada puede ser crítico en el desempeño general del sistema, los retrasos producidos entre la orden y el cumplimiento del evento en este sistema son prácticamente despreciables en cualquiera de los cuatro escenarios formados, ya que en su mayoría no sobrepasan los 15 microsegundos.

Esto es causado principalmente porque el sistema está compuesto por dos componentes con una diferencia abismal de capacidad de procesamiento. Por un lado, la placa ESP32 es un dispositivo que cuenta con unos recursos extremadamente limitados, cuyo objetivo no es trabajar con un volumen elevado de datos que puedan suponer un problema en el rendimiento del sistema, sino establecer una conexión rápida y fiable con un agente para realizar pequeñas operaciones a una alta velocidad. Por otro lado, se está utilizando como agente un ordenador de última generación que cuenta con un procesador Intel de cuatro núcleos, al que una transferencia de datos, que en el caso más optimista rondaría el megabyte por segundo, no debería afectar en gran medida a su rendimiento.

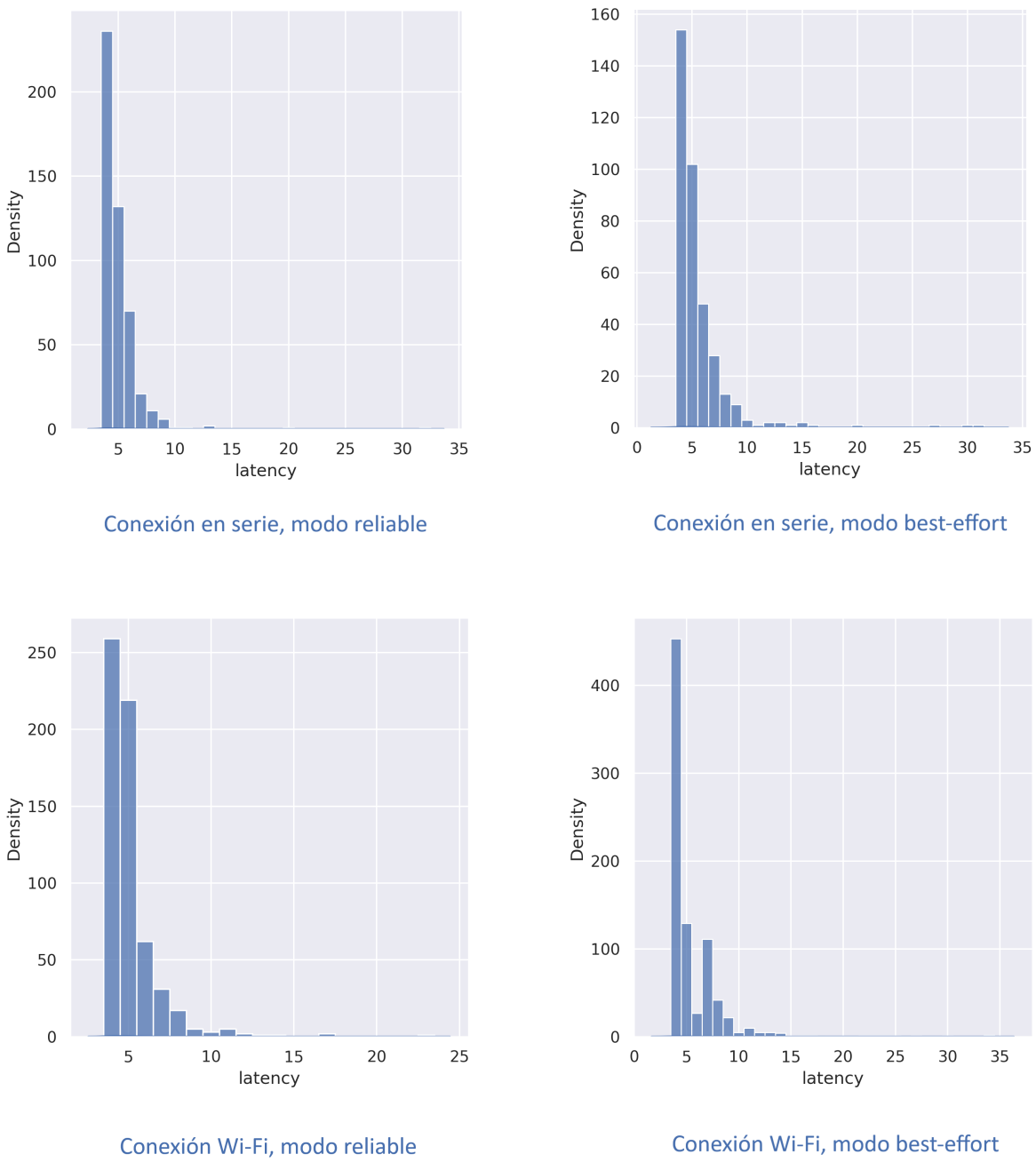

Figura 2: Histogramas de latencias

6.1.3 Conclusión

Estas conclusiones muestran las posibilidades que ofrecen este tipo de sistemas en la vida real. Por una parte, sería muy posible reducir en gran medida la calidad del dispositivo en la mayoría de aplicaciones, obteniendo igualmente unos resultados sobresalientes. Por otro lado, también resultaría muy interesante un sistema similar en el que un solo agente pudiera controlar una gran cantidad de clientes sin verse afectado el rendimiento.

En cualquiera de los escenarios propuestos, los resultados que se han obtenido en las mediciones de la latencia son fácilmente extrapolables y proporcionan una idea de las ventajas y limitaciones que tendrían en función de la configuración del mismo.

6.2 Throughput

La capacidad de transferencia es un dato muy relevante en estos sistemas, ya que suponen un indicativo de peso para delimitar las aplicaciones que pueden llevarse a cabo.

En primer lugar, se puede observar cómo todas las gráficas obtenidas tienden hacia un valor de throughput. Este valor indica la capacidad de transferencia de datos de nuestro sistema cuando este se satura, es decir, la teórica máxima tasa de transferencia.

Sin embargo, como se verá a continuación, esto no siempre es así y existen excepciones.

Para estimar los valores en los que podía darse el punto de saturación, se ha utilizado un método de prueba y error en los que se ha repetido el experimento numerosas veces hasta establecer un intervalo en el que la tasa de transferencia no aumentaba. De este modo se han establecido unos valores, distintos en cada caso, para los que tomar datos y observar con precisión el momento de saturación.

A continuación, se va a realizar un análisis detallado de las gráficas obtenidas.

Como ya ocurrió en el análisis de la latencia, un primer aspecto que llama la atención es la diferencia de morfología que se observa entre las gráficas de las pruebas realizadas bajo el modo *reliable* y las realizadas con el modo *best-effort*.

Primeramente, se puede ver cómo en las gráficas del modo *reliable*, la curva generada es bastante suave y desde un primer momento mantiene una curvatura más o menos constante.

Sin embargo, en los dos casos del modo *best-effort* se ve claramente cómo en un determinado punto, existe un pico que rompe con la continuidad de la gráfica. Esto sucede en concreto en un punto que, atendiendo a la teoría, debería estar en saturación pero que, como se puede comprobar, transmite una mayor cantidad de datos de lo previsto.

Esto solo ocurre en una frecuencia ligeramente superior a la que en teoría sería la frecuencia que produzca la saturación, ya que en la siguiente recopilación de datos, la tasa de transferencia efectiva disminuye hasta el punto que corresponde con el límite al que tiende la función.

Estos resultados resultan muy interesantes ya que indican que el modo *best-effort* no cuenta con una gran ventaja frente a altos volúmenes de demanda, sino que, en puntos cercanos al límite del throughput, existe un intervalo en el que la comunicación funciona a un mejor nivel que cuando el sistema se satura.

Esto puede resultar muy conveniente en aplicaciones diseñadas para que el sistema funcione cerca de su punto de saturación, ya que si en un momento determinado se produce un pico en la demanda, este modo de comunicación otorga una mayor garantía de que esta se pueda satisfacer.

Por otro lado, atendiendo al método de conexión, existe otra notable diferencia. El orden de magnitud del throughput es completamente distinto. En la conexión en serie el límite de envío de datos por segundo

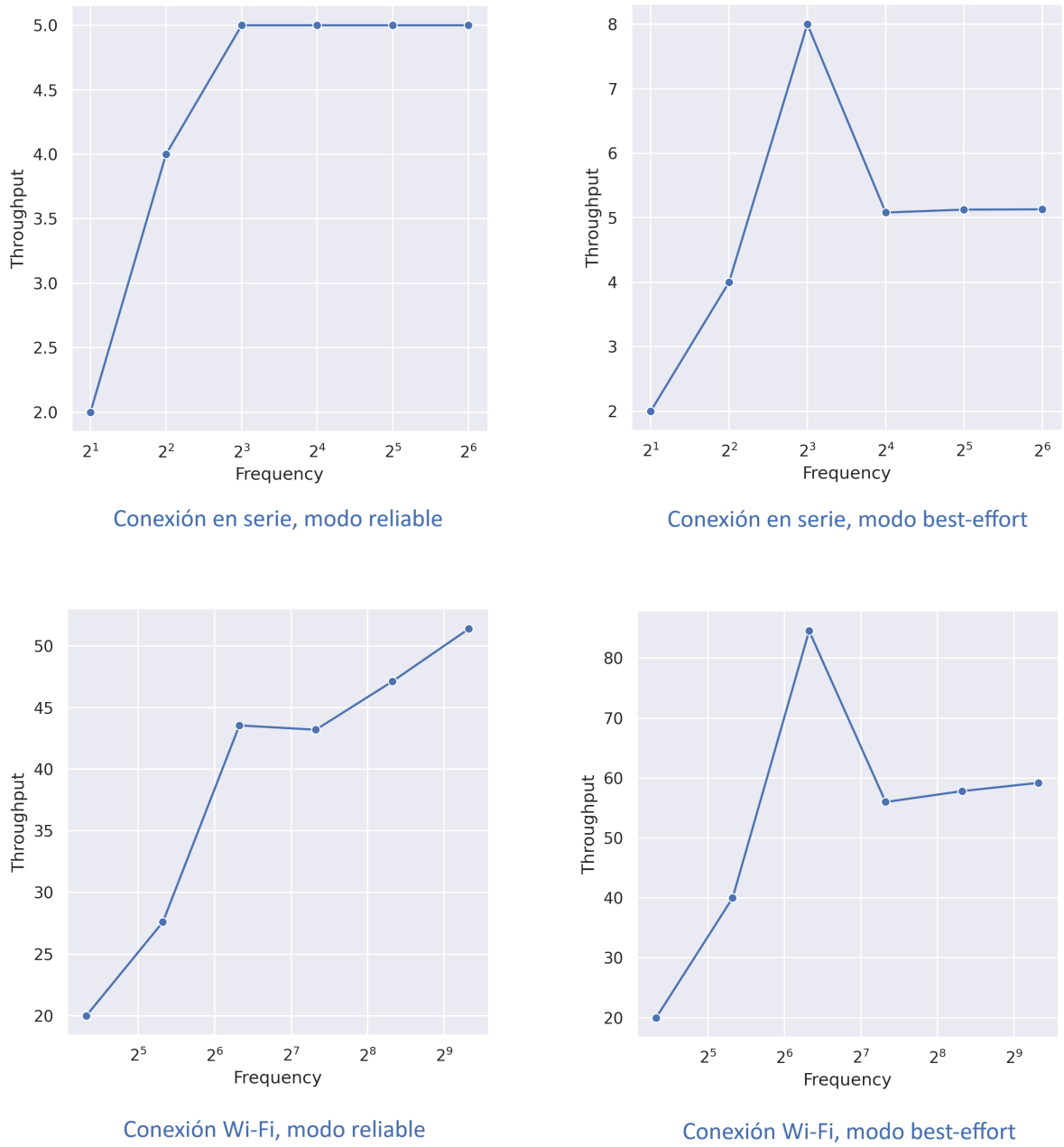

Figura 3: Evolución del throughput

tiende en ambos casos a 5 kilobytes por segundo. Por el contrario, en conexión vía Wi-Fi, esta cifra aumenta hasta casi los 60 kilobytes por segundo. Esto supone una diferencia abismal entre ambos métodos de conexión, la cual puede limitar enormemente las aplicaciones en un sistema conectado en serie.

Esto puede resultar anti-intuitivo ya que se puede llegar a pensar que una conexión en serie siempre va a proporcionar más garantías que una conexión inalámbrica, como ocurre por ejemplo con la conexión a internet vía cable Ethernet frente a conexión Wi-Fi. Sin embargo, en este caso los métodos de conexión son más independientes y, teniendo en cuenta que la conexión vía Wi-Fi es uno de los principales distintivos que proporciona la placa escogida, esta resulta estar más optimizada que una conexión en serie convencional.

En relación con el modo de comunicación, también existe una ligera diferencia en el límite del throughput, sin embargo esta es apenas apreciable y no es significativa frente a la diferencia existente con el método de conexión.

6.2.1 Conclusión

En definitiva, las tasas de transferencia obtenidas son de unos 60 kilobytes por segundo para conexiones vía Wi-Fi y de unos 5 kilobytes por segundo con una conexión en serie. Esto supone una diferencia significativa, sin embargo, en ninguno de los dos casos se trata de una tasa muy elevada teniendo en cuenta la capacidad de transferencia que permite la tecnología de hoy en día.

Esto remarca una vez más el objetivo principal del software que se está estudiando, aportar una forma sencilla de conectar y programar varios dispositivos con recursos limitados para que realicen tareas básicas.

6.3 Consumo de memoria

Tras haber realizado ya las mediciones de la latencia del sistema y del throughput, una de las conclusiones obtenidas indica que el verdadero propósito del sistema no reside en realizar tareas que requieran mucha potencia sino la fiabilidad y la velocidad de realizar tareas más modestas.

El consumo de memoria que se ha medido no hace más que reforzar dicha conclusión ya que, en los cuatro escenarios el consumo de la memoria apenas se ha notado.

En todos las pruebas hemos obtenido un consumo de la memoria de tan solo un 2 %, algo que se considera despreciable frente al consumo de memoria de la mayoría de procesos que realiza el ordenador empleado.

6.4 Influencia de interferencias

Lo primero que hay que tener en cuenta es el hecho de que en una aplicación real es muy probable que existan perturbaciones en la red que alteren el comportamiento de nuestro sistema, por lo que es importante que el diseño de este tenga en cuenta dichos escenarios.

Inicialmente, lo que más llama la atención al observar las gráficas es que estas forman un pico bastante más ancho que las obtenidas previamente. Esto es un resultado esperado ya que es lógico deducir que al añadirle carga al sistema este tarde más en realizar todo tipo de tareas.

Seguidamente, lo que vemos es que las gráficas de ambos modos tienen una irregularidad similar al experimento realizado sin interferencias, pero aún más pronunciada. Al estar corriendo otro proceso en

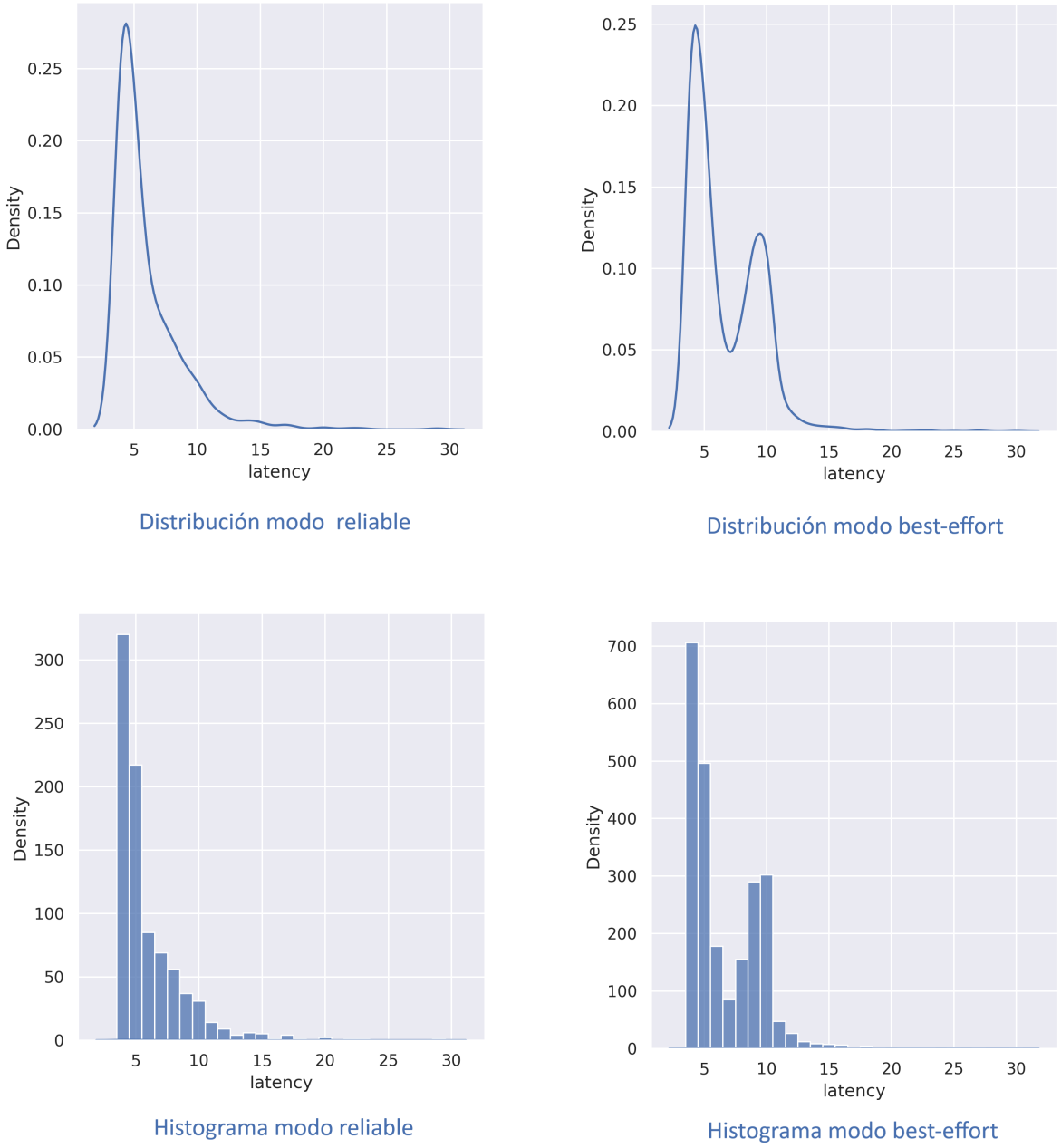

Figura 4: Influencia de las interferencias

paralelo es entendible que ambos procesos no esten sincronizados y se prolongue la irregularidad que ya presentaba la conexión vía Wi-Fi.

Resulta interesante el hecho de que la morfología de las gráficas sea muy parecida a las obtenidas en el experimento sin perturbaciones. Esto dota al análisis de una mayor fiabilidad y permite realizar predicciones de cómo el sistema podría responder en otras situaciones.

Por último, es preciso señalar que, a pesar que no se vea reflejado en las gráficas, la cantidad de veces que la latencia ha sido 3 en este último experimento (tomada como latencia referencia del sistema) ha resultado superior en ambos casos frente a las gráficas sin perturbaciones, un dato que nos confirma la mayor demanda de recursos que se prevía.

De todos modos, la latencia resultante sigue siendo poco significativa en el sistema, teniendo aún un margen bastante amplio para añadir clientes al mismo agente.

CAPÍTULO 7

Conclusiones y líneas futuras

7.1 Conclusiones

Tras varios meses realizando el trabajo se han obtenido numerosas conclusiones, tanto durante en el proceso de aprendizaje como en el momento de la realización de análisis y la discusión de los resultados.

Lo primero, que resulta llamativo a la hora de introducirse en la programación de sistemas que operen en tiempo real es la complejidad que conlleva diseñar un software de este tipo. A la hora de diseñar la arquitectura del sistema, es crucial que los componentes del software mantengan una comunicación fiable que no se vea afectada en gran medida por las perturbaciones que puedan existir en el entorno del sistema y que la transmisión de información se ejecute de una manera regular. Esto se consigue dividiendo la arquitectura general en distintos niveles que se encargan del funcionamiento de una sección específica de la comunicación.

En este sentido, ROS consigue unificar todas las partes que tratan la información en un solo software para permitir al usuario centrarse en el desarrollo de aplicaciones.

Por otro lado es reseñable la evolución que ha experimentado este sector en los últimos años. Su comienzo se dio hace poco más de una década y durante este tiempo se han realizado numerosos proyectos y se ha logrado desarrollar una segunda versión que incorpora una gran cantidad de mejoras y que se actualiza constantemente con nuevas distribuciones.

Esto muestra claramente el apoyo que existe hacia el sector y las enormes previsiones que este posee de cara a un futuro cercano.

En este sentido micro-ROS supone un gran avance para el sector. El desarrollo de una variante del sistema operativo de robots que permite operar con dispositivos de tan reducido coste se traduce en oportunidades para permitir que cualquier persona pueda introducirse en el mundo de la programación de robots sin necesidad de un gran presupuesto. Asimismo, resulta fascinante el modo en el que, a pesar de no existir aún una gran cantidad de información en la web sobre el desarrollo de aplicaciones en micro-ROS, este cuenta con una comunidad muy activa que favorece a los usuarios menos experimentados. Durante la realización del trabajo han ocurrido situaciones de bloqueo en las que ha sido necesario utilizar información que apenas llevaba horas publicada en la red. Esto muestra una idea del potencial y del margen de mejora que tiene a día de hoy.

En cuanto a la actuación del software, ha resultado muy interesante el hecho de poder obtener unos resultados tan concluyentes empleando una aplicación relativamente sencilla y una placa que actúa con unos recursos tan limitados.

De este modo, los análisis que se han discutido previamente muestran una idea general del alcance de este software. Es cierto que numéricamente hablando pueda parecer que no cuenta con la potencia necesaria para soportar aplicaciones que vayan a ser realizadas por robots en tiempo real.

Sin embargo, el gran potencial de estos sistemas no es el poder desarrollar complejas aplicaciones para la monitorización de un gran robot, sino la facilidad de poder fragmentar un gran sistema en distintos apartados que estén controlados por dispositivos más modestos que sean diseñados específicamente para la tarea que vayan a desempeñar.

Teniendo esto en cuenta, el proyecto realizado muestra unos resultados satisfactorios que se resumen en cómo la complejidad de una tecnología tan avanzada se consigue simplificar de tal forma, manteniendo unas prestaciones más que suficientes para los objetivos para los que están diseñados la mayoría de microcontroladores.

7.2 Lineas futuras

Para los análisis que se han realizado en este trabajo se han seleccionado unos parámetros y distintos escenarios. Estos nos muestran una idea bastante acertada de cómo se comporta la placa. Sin embargo, existen muchas otras variables que no se han medido que aportarían bastante información, como puede ser la latencia que se produce en el transporte del mensaje desde el cliente al agente, la CPU que se emplea en el proceso, etc.

Asimismo, sería muy interesante repetir los experimentos que se han realizado teniendo varias placas conectadas como suscriptoras del topic. De este modo, se podría comprobar las características de la conexión en un entorno más cercano a un posible escenario de la vida real.

Otro posible análisis podría centrarse en la comparación de resultados entre distintos tipos de hardware o distintos RTOS. Esto resultaría muy útil a la hora de escoger tanto la placa como el sistema operativo que la soporta.

Finalmente, cabe recordar que la ciencia que se ha estudiado está en pleno desarrollo y sufre cambios constantemente, por lo que no sería de extrañar que de aquí a unos pocos años, o incluso meses, se optimice la calidad de la conexión de algunos de los componentes que forman tanto el software como el hardware. En cualquier caso, es muy probable que en un breve espacio de tiempo surjan numerosas mejoras.

CAPÍTULO 8

Planificación temporal y presupuesto

8.1 Planificación temporal

A continuación se muestra el diagrama de Gantt de las actividades llevadas a cabo durante la realización del trabajo. Las fechas tienen una fecha y duración aproximada.

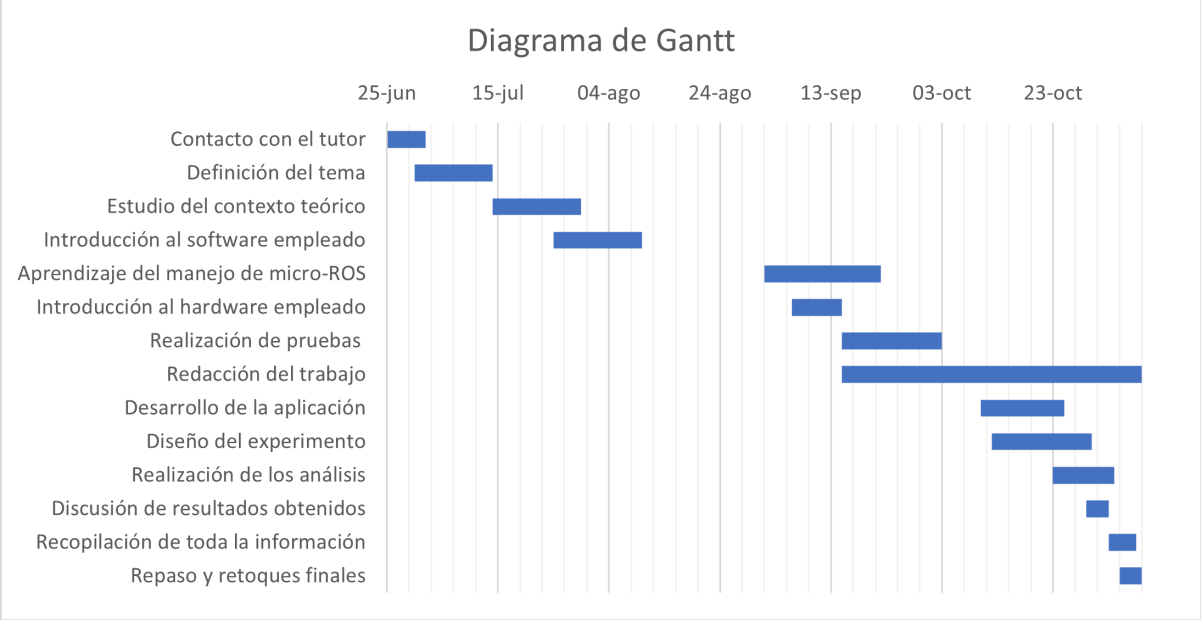

Figura 1: Diagrama de Gantt

Aunque no es apreciable en el diagrama, la carga de trabajo ha sido muy variable durante toda la duración del ejercicio, siendo muy superior durante los meses de septiembre y octubre frente a julio y agosto.

8.2 Presupuesto

En la siguiente tabla se muestra una estimación del coste total que ha supuesto el trabajo. En él se ha estimado que cada hora empleada por el tutor se valora en 40 euros y cada hora empleada por el alumno tiene un valor de 15 euros. Además se ha incluido un coste de los equipos informáticos utilizados, puesto que aunque ya se tuvieran antes del comienzo del trabajo, durante el mismo se ha producido un desgaste debido a la gran cantidad de horas que se han utilizado.

	COSTE UNITARIO	CANTIDAD	IMPORTE TOTAL
	COSTE HUMANO		
Tutor	40 €/h	20 h	800 €
Estudiante	15 €/h	300 h	4.500 €
Total coste humano			**5.300 €**
	COSTE MATERIAL		
ESP32	15 €/ud	1 ud	15 €
Equipo informático	100 €/ud	1 ud	100 €
Router TP - Link	56 €/ud	1 ud	56 €
Total coste material			**171 €**
TOTAL			**5.471 €**

Figura 2: Presupuesto

En este sentido, el valor total que se le asigna a la realizacion del trabajo es de 5.471 euros. Esta cifra es un valor aproximado pero que muestra una buena representación de lo que podría equivaler en relación con una investigación profesional.

CAPÍTULO 9

Impacto social y medioambiental

En primer lugar, es necesario diferenciar entre impacto social directo e indirecto. El impacto directo se relaciona con la manera en la que los resultados o las conclusiones obtenidas pueden afectar directamente a la sociedad o al medioambiente, mientras que el impacto indirecto tiene que ver con los efectos que se puedan mostrar a largo plazo relacionados con los avances tecnológicos a los que pueda contribuir este trabajo.

9.1 Impacto social

El impacto social directo de la totalidad del proyecto no es muy elevado, ya que directamente no proporciona ninguna herramienta que sea de gran utilidad para un usuario medio. Sin embargo, los efectos de este trabajo de investigación si pueden resultar relevantes para las entedidas que participen en el desarrollo de software de control de robots en tiempo real o en fabricantes de placas de desarrollo, pudiendo utilizar varios resultados de los obtenidos como referencia a la hora de establecer sus especificaciones.

Este trabajo contribuye al avance en el desarrollo de la tecnología relacionada con la industria 4.0 y el IoT. En este sentido, el impacto social indirecto que es capaz de producir este proyecto puede convertirse en algo muy notable, ya que, la revolución tecnológica que este fenómeno va a producir, va a afectar radicalmente en la vida de las personas. Estos cambios supondran un aumento general en la calidad de vida de la sociedad, reflejado en la automatización de numerosas tareas rutinarias, el ahorro de recursos y una nueva manera de trabajar mucho más eficiente.

9.2 Impacto medioambiental

La cantidad de recursos consumidos durante la realización de pruebas y la ejecución de los análisis es bastante reducida. Esto es así ya que el sistema que se ha formado está formado simplemente por un ordenador personal de última generación, que no consume gran cantidad de energía, y una placa que está diseñada expresamente de modo que economice los recursos al máximo, por lo que su consumo de energía es prácticamente irrelevante.

En lo relacionado con el impacto medioambiental indirecto, como ya se ha visto en la sección anterior, este trabajo supone un avance en la cuarta revolución industrial. A día de hoy es complicado saber exactamente el modo en el que este fenómeno afectará al medioambiente, sin embargo, es seguro que esta industria traerá consigo numerosas tecnologías que contribuyan en el ahorro energético. De este modo, el potencial impacto indirecto que puede provocar este trabajo resultaría muy positivo.

CAPÍTULO 10

Anexos

10.1 Anexo 1: Muestra de resultados obtenidos

A continuación se expone una muestra de los resultados obtenidos de los análisis realizados.

Latencia:

```
Thread 0 Interval: 600
    0:        0:       12
    0:        1:        8
    0:        2:        7
    0:        3:        7
    0:        4:       10
    0:        5:        8
    0:        6:        8
    0:        7:        9
    0:        8:        7
    0:        9:        7
    0:       10:        7
    0:       11:        7
    0:       12:        7
    0:       13:        9
    0:       14:        7
    0:       15:        7
    0:       16:        9
    0:       17:        7
    0:       18:        7
    0:       19:        7
    0:       20:       10
    0:       21:        8
    0:       22:        7
    0:       23:        7
    0:       24:        7
    0:       25:        7
    0:       26:        9
    0:       27:        7
    0:       28:        7
    0:       29:        6
```

Throughput:

```
^[[35m[1635685759.599018]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37msend_message
↪          ^[[m | ^[[33m[** <<SER>> **]^[[m        | client_key: 0x262A35A2, len: 13
^[[35m[1635685760.051717]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37mrecv_message
↪          ^[[m | ^[[33m[==>> SER <<==]^[[m        | client_key: 0x262A35A2, len: 508
^[[35m[1635685760.051991]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37msend_message
↪          ^[[m | ^[[33m[** <<SER>> **]^[[m        | client_key: 0x262A35A2, len: 13
^[[35m[1635685760.095816]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37mrecv_message
↪          ^[[m | ^[[33m[==>> SER <<==]^[[m        | client_key: 0x262A35A2, len: 508
^[[35m[1635685760.096039]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37msend_message
↪          ^[[m | ^[[33m[** <<SER>> **]^[[m        | client_key: 0x262A35A2, len: 13
^[[35m[1635685760.097978]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37mrecv_message
↪          ^[[m | ^[[33m[==>> SER <<==]^[[m        | client_key: 0x262A35A2, len: 52
^[[35m[1635685760.098142]^[[m debug    | ^[[34mDataWriter.cpp    ^[[m | ^[[37mwrite
↪        ^[[m | ^[[33m[** <<DDS>> **]^[[m        | client_key: 0x00000000, len: 1033
^[[35m[1635685760.098277]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37msend_message
↪          ^[[m | ^[[33m[** <<SER>> **]^[[m        | client_key: 0x262A35A2, len: 13
^[[35m[1635685760.552689]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37mrecv_message
↪          ^[[m | ^[[33m[==>> SER <<==]^[[m        | client_key: 0x262A35A2, len: 508
^[[35m[1635685760.552845]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37msend_message
↪          ^[[m | ^[[33m[** <<SER>> **]^[[m        | client_key: 0x262A35A2, len: 13
^[[35m[1635685760.596692]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37mrecv_message
↪          ^[[m | ^[[33m[==>> SER <<==]^[[m        | client_key: 0x262A35A2, len: 508
^[[35m[1635685760.596904]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37msend_message
↪          ^[[m | ^[[33m[** <<SER>> **]^[[m        | client_key: 0x262A35A2, len: 13
^[[35m[1635685760.599166]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37mrecv_message
↪          ^[[m | ^[[33m[==>> SER <<==]^[[m        | client_key: 0x262A35A2, len: 52
^[[35m[1635685760.599352]^[[m debug    | ^[[34mDataWriter.cpp    ^[[m | ^[[37mwrite
↪        ^[[m | ^[[33m[** <<DDS>> **]^[[m        | client_key: 0x00000000, len: 1033
^[[35m[1635685760.599486]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37msend_message
↪          ^[[m | ^[[33m[** <<SER>> **]^[[m        | client_key: 0x262A35A2, len: 13
^[[35m[1635685761.057626]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37mrecv_message
↪          ^[[m | ^[[33m[==>> SER <<==]^[[m        | client_key: 0x262A35A2, len: 508
^[[35m[1635685761.057882]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37msend_message
↪          ^[[m | ^[[33m[** <<SER>> **]^[[m        | client_key: 0x262A35A2, len: 13
^[[35m[1635685761.102167]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37mrecv_message
↪          ^[[m | ^[[33m[==>> SER <<==]^[[m        | client_key: 0x262A35A2, len: 508
^[[35m[1635685761.102401]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37msend_message
↪          ^[[m | ^[[33m[** <<SER>> **]^[[m        | client_key: 0x262A35A2, len: 13
^[[35m[1635685761.104181]^[[m debug    | ^[[34mSerialAgentLinux.cpp^[[m | ^[[37mrecv_message
↪          ^[[m | ^[[33m[==>> SER <<==]^[[m        | client_key: 0x262A35A2, len: 52
^[[35m[1635685761.104368]^[[m debug    | ^[[34mDataWriter.cpp    ^[[m | ^[[37mwrite
↪        ^[[m | ^[[33m[** <<DDS>> **]^[[m        | client_key: 0x00000000, len: 1033
```

Consumo de memoria:

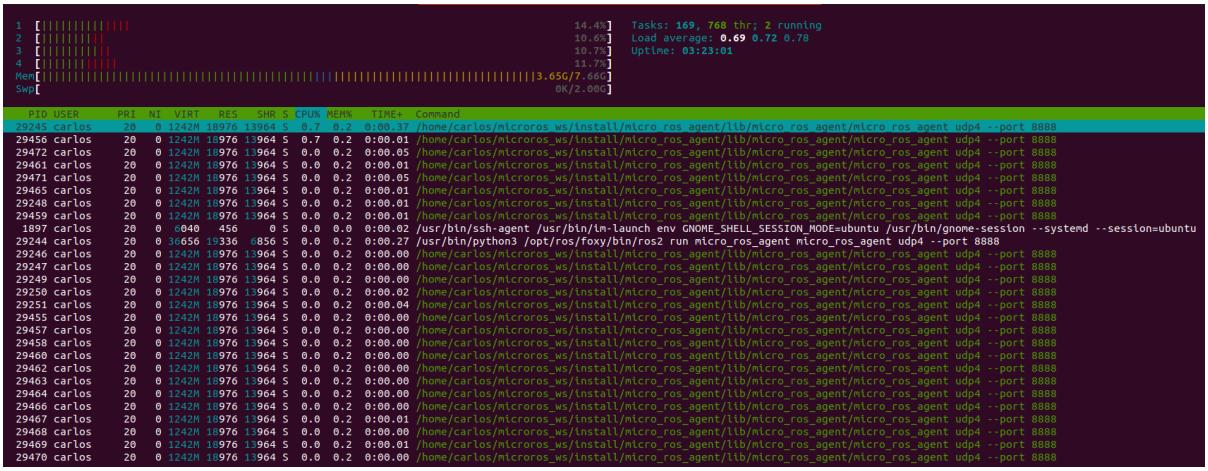

Figura 1: Resultados de «htop» al ejecutar la aplicación

10.2 Anexo 2: Script de python para la generación de gráficas

```
from gc import collect
import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

sns.set_theme(style="darkgrid")


def gen_plot_for_latency(data_file):
    df = pd.read_csv(
        data_file + ".dat",
        skiprows=3,
        delimiter=r":\s+",
        header=None,
        usecols=[2],
        names=["latency"],
        engine="python",
    )

    latency_df = df.drop(df[df.latency < 4].index)

    sns.displot(data=latency_df, x="latency", kind="kde")
    plt.savefig(data_file + "_dist_kde.png", dpi=300)

    sns.histplot(data=latency_df, x="latency", discrete=True)
    plt.savefig(data_file + "_hist.png", dpi=300)


def gen_plot_for_tp(data_info, axes):
    data_file, description, fil, col = data_info
    df = pd.read_csv(data_file + ".dat", delimiter=r"\s+")
    rel = sns.relplot(
        ax=axes[fil, col],
        x="Frequency",
        y="Throughput",
        data=df,
        kind="line",
        marker="o",
    )
    plt.xscale("log", base=2)
    plt.savefig(data_file + ".png", dpi=300)


data_files = [
    "latency_serial_best-effort",
    "latency_serial_reliable",
    "latency_wifi_best-effort",
    "latency_wifi_reliable",
    "latency_best_effort_Wi-Fi_interference",
    "latency_reliable_Wi-Fi_interference",
]


for data_file in data_files:
    gen_plot_for_latency(data_file)

data_infos = [
    ("throughput_serial_best_effort", "Conexión en serie, modo best-effort", 0, 0),
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ("throughput_serial_reliable", "Conexión en serie, modo reliable", 0, 1),
    ("throughput_wifi_best_effort", "Conexión Wi-Fi, modo best-effort", 1, 0),
    ("throughput_wifi_reliable", "Conexión Wi-Fi, modo reliable", 1, 1),
]


for data_info in data_infos:
    fig, axes = plt.subplots(2, 2)
    gen_plot_for_tp(data_info, axes)
```

10.3 Anexo 3: Inciencias ocurridas

Durante la preparación de los análisis ha sido necesario realizar numerosas pruebas intermedias que asegurasen el correcto funcionamiento del hardware y de los middlewares. En la ejecución de estas pruebas se han encontrado varias incidencias que han ralentizado la realización del ejercicio.

10.3.1 Conexión Wi-Fi

En primer lugar, surgió un percance durante el intento de realizar una primera conexión vía Wi-Fi. Una vez configurado el firmware para que se conectase a la IP de la red elegida, tras flashear el firmware en el dispositivo, el punto de acceso Wi-Fi no lo detectaba. Se utilizó entonces el siguiente comando para depurar el problema:

```
ros2 run micro_ros_setup build_firmware.sh monitor
```

Obteniendo la siguiente información relevante:

```
I (642) wifi station: ESP_WIFI_MODE_STA
I (662) wifi:wifi driver task: 3ffc4398, prio:23, stack:6656, core=0
I (662) system_api: Base MAC address is not set, read default base MAC address from BLK0 of␣
↪EFUSE
I (662) system_api: Base MAC address is not set, read default base MAC address from BLK0 of␣
↪EFUSE
I (692) wifi:wifi firmware version: 3ea4c76
I (692) wifi:config NVS flash: enabled
I (692) wifi:config nano formating: disabled
I (692) wifi:Init dynamic tx buffer num: 32
I (702) wifi:Init data frame dynamic rx buffer num: 32
I (702) wifi:Init management frame dynamic rx buffer num: 32
I (712) wifi:Init management short buffer num: 32
I (712) wifi:Init static rx buffer size: 1600
I (712) wifi:Init static rx buffer num: 10
I (722) wifi:Init dynamic rx buffer num: 32

Brownout detector was triggered
```

Investigando el último mensaje de error “Brownout detector was triggered”, se descubrió que la incidencia estaba relacionada con la falta de potencia en la alimentación de la placa.[1]

En una primera instancia se trató de modificar la fuente de alimentación, cambiando en primer lugar de puerto en el ordenador y, posteriormente, conectando la placa directamente a la red de alimentación doméstica. En ambos casos no se consiguió establecer la conexión Wi-Fi, manteniéndose el mismo error en la salida del terminal. Posteriormente se detectó que la incidencia residía en el cable micro-USB escogido inicialmente. Este no conseguía aportar toda la potencia que requiere la placa para establecer una conexión Wi-Fi, ya que esta función demanda una mayor cantidad de energía frente a otras como puede ser la conexión en serie.

Finalmente, se escogió un cable micro USB de calidad superior y se volvió a utilizar el mimo comando para comprobar la conexión, obteniendo la siguiente salida:

```
I (642) wifi station: ESP_WIFI_MODE_STA
I (662) wifi:wifi driver task: 3ffc4398, prio:23, stack:6656, core=0
I (662) system_api: Base MAC address is not set, read default base MAC address from BLK0 of␣
↪EFUSE
```

(continué en la próxima página)

[1] Expressif. Brownout detector was triggered. URL: (https://www.esp32.com/viewtopic.php?t=16299).

(proviene de la página anterior)

```
I (662) system_api: Base MAC address is not set, read default base MAC address from BLK0 of␣
↪EFUSE
I (692) wifi:wifi firmware version: 3ea4c76
I (692) wifi:config NVS flash: enabled
I (692) wifi:config nano formating: disabled
I (692) wifi:Init dynamic tx buffer num: 32
I (702) wifi:Init data frame dynamic rx buffer num: 32
I (702) wifi:Init management frame dynamic rx buffer num: 32
I (712) wifi:Init management short buffer num: 32
I (712) wifi:Init static rx buffer size: 1600
I (712) wifi:Init static rx buffer num: 10
I (722) wifi:Init dynamic rx buffer num: 32
I (822) phy: phy_version: 4180, cb3948e, Sep 12 2019, 16:39:13, 0, 0
I (822) wifi:mode : sta (e8:68:e7:30:2e:5c)
I (822) wifi station: wifi_init_sta finished.
I (942) wifi:new:<6,0>, old:<1,0>, ap:<255,255>, sta:<6,0>, prof:1
I (952) wifi:state: init -> auth (b0)
I (952) wifi:state: auth -> assoc (0)
I (962) wifi:state: assoc -> run (10)
I (1002) wifi:connected with iPhone de Carlos, aid = 1, channel 6, BW20, bssid =␣
↪42:47:22:d6:7a:e9
I (1012) wifi:security: WPA2-PSK, phy: bgn, rssi: -43
I (1012) wifi:pm start, type: 1

I (1102) wifi:AP's beacon interval = 102400 us, DTIM period = 1
I (1642) esp_netif_handlers: sta ip: 172.20.10.12, mask: 255.255.255.240, gw: 172.20.10.1
I (1642) wifi station: got ip:172.20.10.12
I (1642) wifi station: connected to ap SSID:iPhone de Carlos
```

Como se puede observar, la información del firmware nos confirma que el dispositivo se encuentra conectado al punto de acceso Wi-Fi “iPhone de Carlos”. Adicionalmente, desde el propio punto Wi-Fi se puede observar como en el momento de realizar el flash del firmware en el dispositivo, se aumenta el número de dispositivos conectados a la red en 1.

10.3.2 Fallo en la conexión con el agente de ROS 2

Una vez establecida la conexión Wi-Fi, se trató de suscribirse al topic en el que debía de estar publicando mensajes el cliente ya conectado a la red. Tras ejecutar el comando:

```
ros2 topic list
```

Se obtuvo la siguiente salida.

```
carlos@carlos-UX430UA:~/microros_ws$ ros2 topic list
/parameter_events
/rosout
carlos@carlos-UX430UA:~/microros_ws$
```

En el terminal solo se observan los topic de ROS 2 por defecto, y no se muestra el topic por el cual debería de estar publicando mensajes la placa.

En primer lugar, se comprobó si la placa funcionaba correctamente. Para ello se siguieron los siguientes tutoriales para el testeo de la placa en “Visual Studio Code”:

- https://github.com/espressif/vscode-esp-idf-extension/blob/master/docs/tutorial/install.md
- https://github.com/espressif/vscode-esp-idf-extension/blob/master/docs/tutorial/basic_use.md

Tras la instalación y la prueba de un proyecto básico en la placa, se confirmó el correcto funcionamiento de la misma.

Una vez descartado el posible error de funcionamiento de la placa, se comprobó si el cliente establecía conexión con el agente de mico-ROS y si existía intercambio de información. En primer lugar, se utilizó un agente de Docker para depurar el problema. Esto es una capa de software de adicional que proporciona abstracción y la virtualización de aplicaciones. De este modo, era posible probar la aplicación del cliente en un espacio que no fuera ROS 2.

El siguiente comando ejecuta un agente en Docker.

```
docker run -it --rm --net=host microros/micro-ros-agent:foxy udp4 --port 8888 -v6
```

En otro terminal se ejecuta el siguiente comando para entrar en la imagen del Docker:

```
docker run -it osrf/ros:eloquent-desktop
```

Se descargará una imagen más nueva del Docker. Una vez inicializada y con el agente Docker activo se comprueba si el topic es visible de nuevo con el comando "ros2 topic list". Se observa la siguiente salida:

```
root@a4032df86129:/# ros2 topic list
/freertos_int32_publisher
/parameter_events
/rosout
```

Como se puede observar, utilizando el Docker sí que se reconoce el topic de la aplicación de FreeRTOS que se había instalado en la placa.

De este modo, fue posible deducir que el problema residía en la conexión del agente de micro-ROS con el espacio de ROS 2. Se utilizó el siguiente comando para ejecutar un agente de micro-ROS que mostrara información sobre la conexión:

```
ros2 run micro_ros_agent micro_ros_agent udp4 --port  8888 -v6
```

En el agente se observa la siguiente salida:

```
carlos@carlos-UX430UA:~/microros_ws$ ros2 run micro_ros_agent micro_ros_agent udp4 --port 8888␣
↪-v6

[1633603125.726950] info     | UDPv4AgentLinux.cpp | init                     | running...    ␣
↪        | port: 8888

[1633603125.727267] info     | Root.cpp           | set_verbose_level        | logger setup  ␣
↪       | verbose_level: 6

[1633603131.602949] debug    | UDPv4AgentLinux.cpp | recv_message             | [==>> UDP <
↪<==]        | client_key: 0x00000000, len: 24, data:
0000: 80 00 00 00 00 01 10 00 58 52 43 45 01 00 01 0F 1E A5 3D F9 81 00 FC 01

[1633603131.603248] info     | Root.cpp           | create_client            | create        ␣
↪       | client_key: 0x1EA53DF9, session_id: 0x81

[1633603131.603400] info     | SessionManager.hpp | establish_session        | session␣
↪established    | client_key: 0x1EA53DF9, address: 172.20.10.12:26313

[1633603131.603645] debug    | UDPv4AgentLinux.cpp | send_message             | [** <<UDP>>␣
↪**]        | client_key: 0x1EA53DF9, len: 19, data:
0000: 81 00 00 00 04 01 0B 00 00 00 58 52 43 45 01 00 01 0F 00
```

(continué en la próxima página)

(proviene de la página anterior)

```
[1633603131.934983] info     | ProxyClient.cpp    | create_participant       | participant␣
↪created    | client_key: 0x1EA53DF9, participant_id: 0x000(1)
0000: 81 80 00 00 05 01 06 00 00 0A 00 01 00 00

[1633603132.191877] info     | ProxyClient.cpp    | create_topic             | topic created  ␣
↪       | client_key: 0x1EA53DF9, topic_id: 0x000(2), participant_id: 0x000(1)

[1633603132.287776] info     | ProxyClient.cpp    | create_publisher         | publisher␣
↪created      | client_key: 0x1EA53DF9, publisher_id: 0x000(3), participant_id: 0x000(1)

[1633603132.350367] info     | ProxyClient.cpp    | create_datawriter        | datawriter␣
↪created     | client_key: 0x1EA53DF9, datawriter_id: 0x000(5), publisher_id: 0x000(3)

[1633603133.465362] debug    | DataWriter.cpp     | write                    | [** <<DDS>> **]␣
↪       | client_key: 0x00000000, len: 4, data: 0000: 00 00 00 00
```

La información más relevante reside en comprobar que el agente y el cliente establecen una conexión y, aun más importante, que el agente de micro-ROS publica los mensajes en el DDS. De este modo era complicado averiguar el hecho de que, publicándose mensajes en la red de ROS 2, estos no eran reconocidos desde la computadora. Se investigó este fallo a través de fuentes externas[2] y se averiguó que el problema residía en el dominio de ROS escogido previamente.

Este se puede escoger a través de una variable del entorno denominada "ROS_DOMAIN_ID". En uno de los tutoriales realizados para el aprendizaje del manejo de ROS 2, era necesario establecer esta variable en el fichero .bashrc. Sin embargo, en las aplicaciones que ofrecen los RTOS, este no es el dominio empleado, por lo cuál no es posible observar los mensajes que se publican en el espacio DDS. Una vez suprimida esta línea de código en el fichero .bashrc, se volvió a ejecutar todo el proceso (flasheo del firmware y creación del agente). Finalmente, tras conectar el cliente con el agente ya era posible observar tanto los nodos como los topic a los que estaba conectada la placa.

```
carlos@carlos-UX430UA:~/microros_ws$ ros2 topic list
/freertos_int32_publisher
/parameter_events
/rosout
carlos@carlos-UX430UA:~/microros_ws$ ros2 node list
/freertos_int32_publisher
```

[2] micro-ROS. No communication between micro-ROS and ROS2. URL: (https://github.com/micro-ROS/micro_ros_arduino/issues/7).

Índice de figuras

1. Arquitectura general de un sistema en tiempo real distribuido (Fuente: Distributed Real-Time Systems, 2019) . . . 5
2. Logotipo de ROS . . . 5
3. Robot con Inteligencia Artificial de Stanford (Fuente: Stanford University) . . . 6
4. Logotipo de micro-ROS . . . 7
5. Logotipo de microsoft . . . 8
6. Logotipo de ROS World 2021 . . . 9

1. Comparación de la estructura de ROS 2 y micro-ROS (Fuente: freertos.org) . . . 11
2. Logotipo de la distribución «Foxy fitzroy» . . . 12
3. Funcionamiento de un topic (Fuente: ros.org) . . . 13
4. Funcionamiento de un servicio (Fuente: ros.org) . . . 13
5. Funcionamiento de una acción (Fuente: ros.org) . . . 14
6. Estructura de micro-ROS (Fuente: micro-ROS.org) . . . 17
7. Arquitectura de la librería del cliente (Fuente: fiware.com) . . . 18
8. Arquitectura del middleware (Fuente: fiware.com) . . . 18
9. Arquitectura de Micro XRCE-DDS (Fuente: freertos.org) . . . 19
10. Arquitectura del RTOS (Fuente: fiware.com) . . . 20
11. Logotipo de FreeRTOS . . . 20

1. Placa ESP32 . . . 22
2. Componentes de la placa ESP32 (Fuente: Espressif) . . . 23
3. Módulo ESP32-WROOM-32D . . . 23
4. Asus UX430U . . . 24
5. Cable micro-USB . . . 24
6. Router TP-Link . . . 24

1. Latencia (Fuente: Inube) . . . 26
2. Throughput (Fuente: Corporate Finance Institute) . . . 27
3. Logotipo de Jupyter notebook . . . 31

1. Distribución de latencias . . . 34
2. Histogramas de latencias . . . 36
3. Evolución del throughput . . . 38
4. Influencia de las interferencias . . . 40

1. Diagrama de Gantt 45
2. Presupuesto 46

1. Resultados de «htop» al ejecutar la aplicación 51

Bibliografía

[1] Khan Accademy. User datagram protocol. URL: https://es.khanacademy.org/computing/ap-computer-science-principles/the-internet/x2d2f703b37b450a3:transporting-packets/a/user-datagram-protocol-udp#:~:text=El%20Protocolo%20de%20datagrama%20de,o%20llegan%20fuera%20de%20orden.

[2] ROS2 Design. Ros middleware interface. URL: https://design.ros2.org/articles/ros_middleware_interface.html.

[3] Digi-Key. Espressif Systems. URL: https://www.digikey.es/es/supplier-centers/espressif-systems.

[4] Digi-Key. How to select the right RTOS and Microcontroller Platform for the IoT. URL: https://www.digikey.com/en/articles/how-to-select-the-right-rtos-and-microcontroller-platform-for-the-iot.

[5] Digi-Key. Real time operating system (RTOS) and their applications. URL: https://www.digikey.es/es/articles/real-time-operating-systems-and-their-applications.

[6] Digi-Key. ESP32-DevKitC. URL: https://www.espressif.com/en/products/devkits/esp32-devkitc.

[7] Digi-Key. ESP32WROOM32 Datasheet. URL: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf.

[8] Digi-Key. ESP32 Get Started. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html#what-you-need.

[9] ePRoxima. ePRoxima fast DDS. URL: https://www.eprosima.com/index.php/products-all/eprosima-fast-dds.

[10] K. Erciyes. *Distributed Real-Time Systems*. Springer, 2019.

[11] Fiware. Imicro-ros puts the Robot Operating System on Microcontroller. URL: https://www.youtube.com/watch?v=slMhPRnBVwM.

[12] Fiware. Micro-ROS client library. URL: https://www.fiware.org/2020/06/02/two-layered-api-introducing-the-micro-ros-client-library/.

[13] Linux Foundation. Cyclictest. URL: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start.

[14] micro-ROS. First Linux application. URL: https://micro.ros.org/docs/tutorials/core/first_application_linux/.

[15] micro-ROS. First micro-ROS application on a RTOS. URL: https://micro.ros.org/docs/tutorials/core/first_application_rtos/.

[16] micro-ROS. First micro-ROS Application on Linux. URL: https://micro.ros.org/docs/tutorials/core/first_application_linux/.

[17] micro-ROS. Microsoft Azure RTOS. URL: https://micro.ros.org/blog/2021/09/08/MicrosoftAzureRTOS/.

[18] micro-ROS. Supported Hardware. URL: https://micro.ros.org/docs/overview/hardware/.

[19] micro-ROS. Why a real time operating system. URL: https://micro.ros.org/docs/concepts/rtos/.

[20] micro-ROS. FreeRTOS. URL: https://micro.ros.org/docs/overview/rtos/#freertos.

[21] micro-ROS. Micro-ROS features. URL: https://micro.ros.org/docs/overview/features/.

[22] micro-ROS. Micro-ROS on Arduino. URL: https://micro.ros.org/blog/2020/11/24/Arduino/.

[23] micro-ROS. ROSCon2018. URL: https://micro.ros.org/blog/2018/09/30/roscon/.

[24] micro-ROS. XRCE_DDS 1.20 release. URL: https://micro.ros.org/blog/2020/05/12/xrce-dds_120_release/.

[25] micro-ROS. micro-ROS NuttX v0.0.2-alpha and Apps v0.0.3 release. URL: https://micro.ros.org/blog/2019/06/03/NuttX_micro-ROS-v0.0.2_relase/.

[26] Openwebinars. Qué es ROS. URL: https://openwebinars.net/blog/que-es-ros.

[27] ROS. History. URL: https://www.ros.org/history/.

[28] ROS. Installation. URL: https://docs.ros.org/en/foxy/Installation.html.

[29] ROS. Rolling. URL: https://docs.ros.org/en/rolling/.

[30] ROS. Tutorial. URL: https://docs.ros.org/en/foxy/Tutorials.html.

[31] ROS. Ubuntu installation. URL: https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html.

[32] RosCon. ROS World 2021. URL: https://roscon.ros.org/world/2021/.

[33] Sameer Tuteja. Connect ESP32 to ROS2 Foxy. URL: https://link.medium.com/JFof42RUwib.

[34] Wikipedia. Data distribution service. URL: https://es.wikipedia.org/wiki/Data_Distribution_Service.

[35] Wikipedia. Latency. URL: https://en.wikipedia.org/wiki/Latency_(engineering).

[36] Wikipedia. Sistema de tiempo real. URL: https://es.wikipedia.org/wiki/Sistema_de_tiempo_real.

[37] Wikipedia. Sistema operativo de tiempo real. URL: https://es.wikipedia.org/wiki/Sistema_operativo_de_tiempo_real.

[38] Wikipedia. Robot Operating System. URL: https://es.wikipedia.org/wiki/Robot_Operating_System.