TFG microprocessor benchmark

Versión 1.0

Carlos Castillo Martínez

13 de octubre de 2021

Índice

CAPÍTULO 1

Introducción

1.1 Motivación

1.2 Objetivos

1.3 Estructura del trabajo

CAPÍTULO 2

Marco teórico y estado del arte

2.1 Sistemas en tiempo real

Se dice que un sistema opera en tiempo real cuándo 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.

https://es.wikipedia.org/wiki/Sistema_de_tiempo_real

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 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.
- Gran tamaño: Suelen ocupar mucho espacio tanto hablando de hardware como de software, cuyas librerías suelen estar formadas por una gran cantidad de lineas de código.
- 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.

[1] Libro: Distributed real-time systems. Theory and practice

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 siguiente:

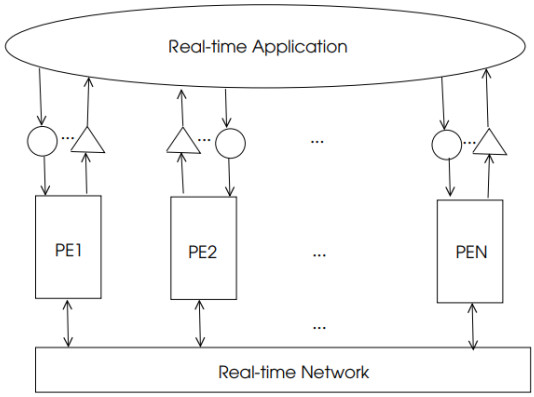

Figura 1: Arquitectura general de un sistema en tiempo real distribuido

En la anterior figura se observa como 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.

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.

https://es.wikipedia.org/wiki/Robot_Operating_System

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.

https://openwebinars.net/blog/que-es-ros/

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, Debian 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.

https://www.ros.org/history/

Figura 3: Robot con Inteligencia Artificial de Stanford (STAIR)

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 mies 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

El sistema operativo de robots 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 los próximos años, desempeñando un papel fundamental en la revolución de la industria 4.0 y el fenómeno conocido como “el internet de las cosas”.

http://docs.ros.org/en/rolling/

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 de software que más se asemeja a la empleada en el análisis es la siguiente.

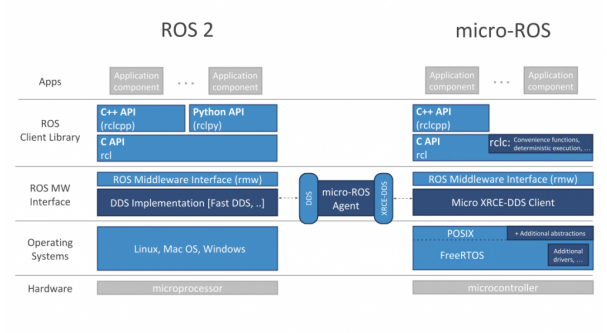

Figura 1: Comparación de la estructura de ROS 2 y micro-ROS

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.

https://docs.ros.org/en/rolling/

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.

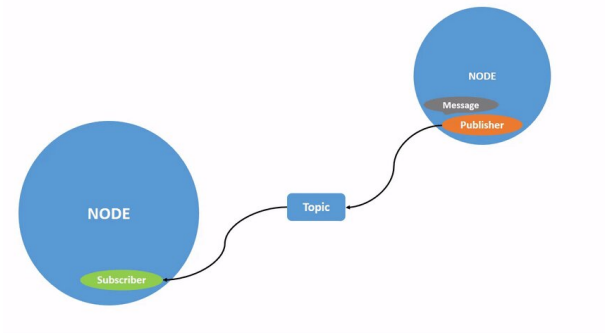

Figura 3: Funcionamiento de un topic

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 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.

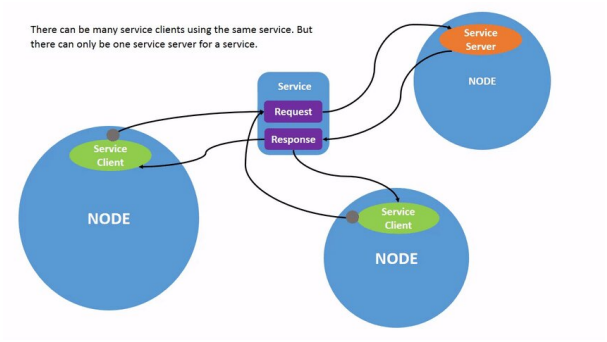

Figura 4: Funcionamiento de un servicio

Finalmente, los nodos también pueden comunicarse mediante acciones. Estas están constituidas por topics y servicios. El modelo de comunicación es similar a la 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).

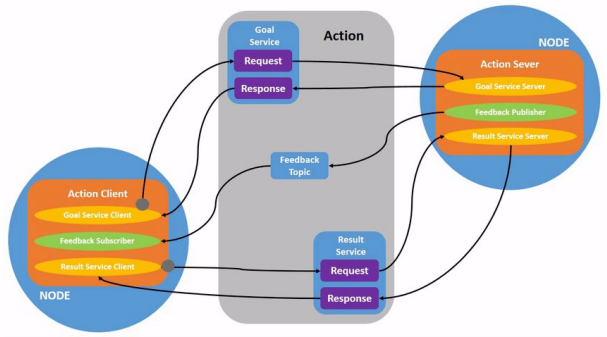

Figura 5: Funcionamiento de una acción

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 2.

https://docs.ros.org/en/foxy/Tutorials.html

3.1.2 Instalación

El proceso de instalación de ROS 2 se encuentra perfectamente explicado en la documentación oficial, en la pagina web https://docs.ros.org/en/foxy/Installation.html.

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.

En segundo lugar es necesario añadir la herramienta avanzada de paquetes (APT) de ROS 2 a nuestro sistema. A continuación hay que añadir el repositorio a nuestra lista fuente.

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 añadir 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.

```
# 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
```

https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html

3.2 Micro-ROS

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 6: 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.

https://www.youtube.com/watch?v=slMhPRnBVwM

3.2.1 Características principales

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

- 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 XRCDE-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. A parte 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.

https://micro.ros.org/docs/overview/features/

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.

```
# 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
sudo apt-get install python3-pip

# Build micro-ROS tools and source them

colcon build

source install /local_setup.bash
```

https://micro.ros.org/docs/tutorials/core/first_application_linux/

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.

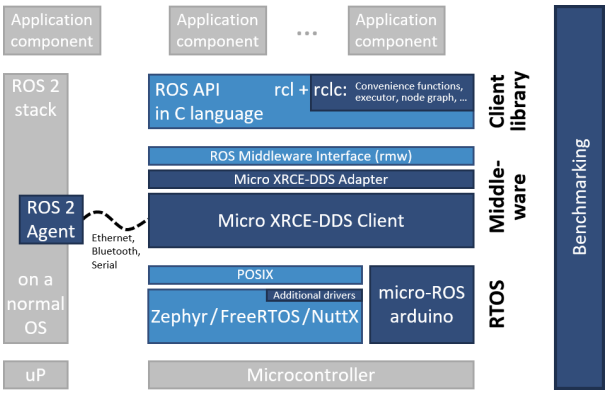

Figura 7: Estructura de micro-ROS

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 modernos ya cuentan hasta con algunos megabytes de 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:

- 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.

https://www.fiware.org/2020/06/02/two-layered-api-introducing-the-micro-ros-client-library/

Figura 8: Arquitectura de la librería del cliente

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.

Figura 9: Arquitectura del middleware

https://design.ros2.org/articles/ros_middleware_interface.html

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.

https://es.wikipedia.org/wiki/Data_Distribution_Servic

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 es la "Fast DDS" de eProsima. 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.

https://www.eprosima.com/index.php/products-all/eprosima-fast-dds

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.

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.

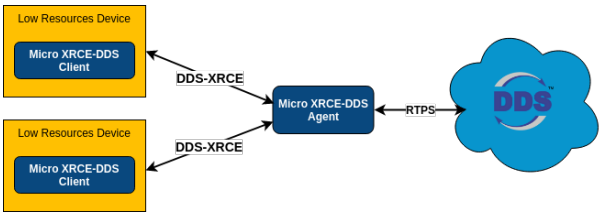

Figura 10: Arquitectura de Micro XRCDE-DDS

Dentro de las características principales de Micro XCRE-DDS, caben destacar las siguientes:

- Alto rendimiento: El cliente utiliza una librería de serialización de bajo nivel que codifica en XCDR.
- Bajo consumo de recursos: La librería del cliente está libre de memoria XRCDE-DDS 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.

https://www.digikey.es/es/articles/real-time-operating-systems-and-their-applications

Figura 11: Arquitectura del RTOS

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.

https://es.wikipedia.org/wiki/Sistema_operativo_de_tiempo_real

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.

https://micro.ros.org/docs/concepts/rtos/

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.

https://micro.ros.org/docs/tutorials/core/first_application_rtos/

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.

https://www.digikey.com/en/articles/how-to-select-the-right-rtos-and-microcontroller-platform-for-the-iot

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.

https://micro.ros.org/docs/overview/rtos/#freertos

Figura 12: 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.

https://micro.ros.org/docs/overview/hardware/

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.

https://www.digikey.es/es/supplier-centers/espressif-systems

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

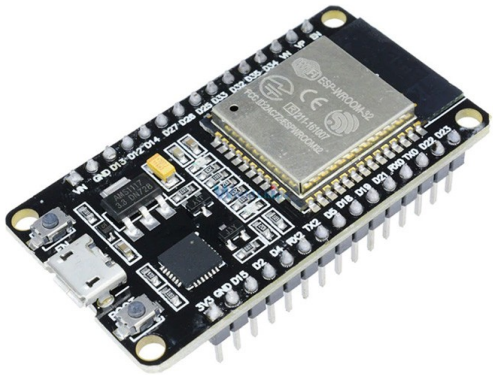

Figura 1: Placa ESP32

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 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.

https://www.espressif.com/en/products/devkits/esp32-devkitc

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.

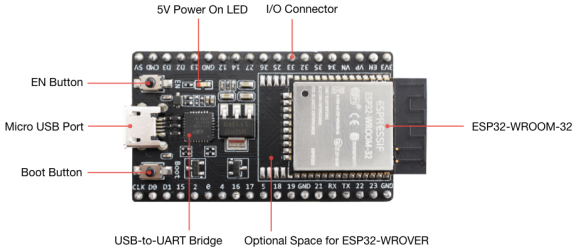

Figura 2: Componentes de la placa ESP32

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

La corriente de reposo del chip ESP32 es inferior a 5 uA, lo que lo hace 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.

Figura 3: Módulo ESP32-WROOM-32D

https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf

4.2 Computador

Todo el trabajo se ha realizado haciendo uso de un ordenador personal. 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

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.

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html#what-you-need

En primer lugar es necesario instalar los requisítos 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 primero 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.

https://micro.ros.org/docs/tutorials/core/first_application_linux/

Después de realizar unas primeras prácticas tanto con el software como con el hardware que se quiere probar, es el momento de juntarlos y realizar las primeras pruebas de micro-ROS en la placa esp32.

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.

https://link.medium.com/JFof42RUwib

En primer lugar 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 aplciación que se quiere probar y el tipo de conexión que se quiere establecer con la placa. Además, es necesario operar sobre un menu 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.

5.2 Estructura principal del análisis

5.3 Incidencias 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.

5.3.1 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
Brownout detector was triggered
```

Investigando el último mensaje de error "Brownout detector was triggered" (https://www.esp32.com/viewtopic.php?t=16299), se descubrió que la incidencia estaba relacionada con la falta de potencia en la alimentación de la placa.

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
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.

5.3.2 Fallo en la conexión del agente de micro-ROS con 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 si 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
[1633603131.807073] debug    | UDPv4AgentLinux.cpp | recv_message            ␣
↪| [==>> UDP <<==]        | client_key: 0x1EA53DF9, len: 56, data:
0000: 81 80 00 00 01 07 30 00 00 0A 00 01 01 03 00 00 21 00 00 00 00 01 A5 A5␣
↪19 00 00 00 66 72 65 65
0020: 72 74 6F 73 5F 69 6E 74 33 32 5F 70 75 62 6C 69 73 68 65 72 00 00 00 00
[1633603131.934983] info     | ProxyClient.cpp    | create_participant      ␣
↪| participant created    | client_key: 0x1EA53DF9, participant_id: 0x000(1)
[1633603131.935244] debug    | UDPv4AgentLinux.cpp | send_message            ␣
↪| [** <<UDP>> **]        | client_key: 0x1EA53DF9, len: 14, data:
0000: 81 80 00 00 05 01 06 00 00 0A 00 01 00 00
[1633603131.935307] debug    | UDPv4AgentLinux.cpp | send_message            ␣
↪| [** <<UDP>> **]        | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
[1633603132.132584] debug    | UDPv4AgentLinux.cpp | recv_message            ␣
↪| [==>> UDP <<==]        | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1633603132.132919] debug    | UDPv4AgentLinux.cpp | send_message            ␣
↪| [** <<UDP>> **]        | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
[1633603132.133873] debug    | UDPv4AgentLinux.cpp | send_message            ␣
↪| [** <<UDP>> **]        | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1633603132.149265] debug    | UDPv4AgentLinux.cpp | recv_message            ␣
↪| [==>> UDP <<==]        | client_key: 0x1EA53DF9, len: 13, data:
```

(continué en la próxima página)

(proviene de la página anterior)

```
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1633603132.149349] debug     | UDPv4AgentLinux.cpp | recv_message             ␣
↪| [==>> UDP <<==]         | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
[1633603132.149621] debug     | UDPv4AgentLinux.cpp | send_message             ␣
↪| [** <<UDP>> **]         | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
[1633603132.191649] debug     | UDPv4AgentLinux.cpp | recv_message             ␣
↪| [==>> UDP <<==]         | client_key: 0x1EA53DF9, len: 92, data:
0000: 81 80 01 00 01 07 52 00 00 0B 00 02 02 03 00 00 44 00 00 00 1C 00 00 00␣
↪72 74 2F 66 72 65 65 72
0020: 74 6F 73 5F 69 6E 74 33 32 5F 70 75 62 6C 69 73 68 65 72 00 00 01 0E 80␣
↪1C 00 00 00 73 74 64 5F
0040: 6D 73 67 73 3A 3A 6D 73 67 3A 3A 64 64 73 5F 3A 3A 49 6E 74 33 32 5F 00␣
↪00 01 00 00
[1633603132.191877] info      | ProxyClient.cpp     | create_topic             ␣
↪| topic created           | client_key: 0x1EA53DF9, topic_id: 0x000(2),␣
↪participant_id: 0x000(1)
[1633603132.191992] debug     | UDPv4AgentLinux.cpp | send_message             ␣
↪| [** <<UDP>> **]         | client_key: 0x1EA53DF9, len: 14, data:
0000: 81 80 01 00 05 01 06 00 00 0B 00 02 00 00
[1633603132.192054] debug     | UDPv4AgentLinux.cpp | send_message             ␣
↪| [** <<UDP>> **]         | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 02 00 00 00 80
[1633603132.220081] debug     | UDPv4AgentLinux.cpp | recv_message             ␣
↪| [==>> UDP <<==]         | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 01 00 01 00 80
[1633603132.220254] debug     | UDPv4AgentLinux.cpp | send_message             ␣
↪| [** <<UDP>> **]         | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 02 00 00 00 80
[1633603132.230947] debug     | UDPv4AgentLinux.cpp | recv_message             ␣
↪| [==>> UDP <<==]         | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 01 00 00 00 80
[1633603132.287495] debug     | UDPv4AgentLinux.cpp | recv_message             ␣
↪| [==>> UDP <<==]         | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 02 00 00 00 80
[1633603132.287570] debug     | UDPv4AgentLinux.cpp | recv_message             ␣
↪| [==>> UDP <<==]         | client_key: 0x1EA53DF9, len: 24, data:
0000: 81 80 02 00 01 07 10 00 00 0C 00 03 03 03 00 00 02 00 00 00 00 00 00 01
[1633603132.287776] info      | ProxyClient.cpp     | create_publisher         ␣
↪| publisher created       | client_key: 0x1EA53DF9, publisher_id: 0x000(3),␣
↪participant_id: 0x000(1)
[1633603132.287923] debug     | UDPv4AgentLinux.cpp | send_message             ␣
↪| [** <<UDP>> **]         | client_key: 0x1EA53DF9, len: 14, data:
0000: 81 80 02 00 05 01 06 00 00 0C 00 03 00 00
[1633603132.287978] debug     | UDPv4AgentLinux.cpp | send_message             ␣
↪| [** <<UDP>> **]         | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 03 00 00 00 80
[1633603132.327156] debug     | UDPv4AgentLinux.cpp | recv_message             ␣
↪| [==>> UDP <<==]         | client_key: 0x1EA53DF9, len: 13, data:
```

(continúe en la próxima página)

(proviene de la página anterior)

```
0000: 81 00 00 00 0A 01 05 00 03 00 00 00 80
[1633603132.349746] debug     | UDPv4AgentLinux.cpp | recv_message
↪| [==>> UDP <<==]         | client_key: 0x1EA53DF9, len: 36, data:
0000: 81 80 03 00 01 07 19 00 00 0D 00 05 05 03 00 00 0B 00 00 00 00 02 01 00
↪03 00 00 00 00 00 00 00
0020: 03 00 00 00
[1633603132.350367] info      | ProxyClient.cpp     | create_datawriter
↪| datawriter created      | client_key: 0x1EA53DF9, datawriter_id: 0x000(5),
↪publisher_id: 0x000(3)
[1633603132.350530] debug     | UDPv4AgentLinux.cpp | send_message
↪| [** <<UDP>> **]         | client_key: 0x1EA53DF9, len: 14, data:
0000: 81 80 03 00 05 01 06 00 00 0D 00 05 00 00
[1633603132.350618] debug     | UDPv4AgentLinux.cpp | send_message
↪| [** <<UDP>> **]         | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 04 00 00 00 80
[1633603132.358801] debug     | UDPv4AgentLinux.cpp | recv_message
↪| [==>> UDP <<==]         | client_key: 0x1EA53DF9, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 04 00 00 00 80
[1633603133.465127] debug     | UDPv4AgentLinux.cpp | recv_message
↪| [==>> UDP <<==]         | client_key: 0x1EA53DF9, len: 16, data:
0000: 81 80 04 00 07 01 08 00 00 0E 00 05 00 00 00 00
[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 (https://github.com/micro-ROS/micro_ros_arduino/issues/7) 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
```

CAPÍTULO 6

Resultados

CAPÍTULO 7

Conclusiones y líneas futuras

CAPÍTULO 8

Planificación temporal y presupuesto

Índice de figuras

Índice de tablas