UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería del Software

Análisis de la Tecnología de Mensajería MQTT en el contexto del Internet de las Cosas: un caso de estudio

Analysis of Messaging Technology MQTT in the context of the Internet of Things: a case study

Realizado por
Francisco Javier Hernández Martín

Tutorizado por
Bartolomé Rubio Muñoz
Cristian Martín Fernández

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2024

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

Análisis de la Tecnología de Mensajería MQTT en el contexto del Internet de las Cosas: un caso de estudio

Analysis of Messaging Technology MQTT in the context of the Internet of Things: a case study

Realizado por
Francisco Javier Hernández Martín

Tutorizado por
Bartolomé Rubio Muñoz
Cristian Martín Fernández

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2024

Fecha defensa: julio de 2024

Abstract

Messaging technologies have become a crucial element of the modern digital ecosystem, as they play a very important role in the communication between distributed systems, which surround us everywhere in the current age. This thesis presents a comprehensive exploration of the *Message Queuing Telemetry Transport* (MQTT) messaging protocol, focusing on its significance in the context of the *Internet of Things* (IoT). The study involves a thorough investigation into MQTT's fundamental principles, features, and its distinctive characteristics compared to other messaging technologies that are more established. The research delves into both theoretical and practical aspects, including a critical review of relevant literature, a detailed analysis of MQTT's performance through practical experiments, and an examination of real world use cases in diverse industrial scenarios.

A key aspect of the empirical investigation in this thesis is the development and evaluation of a practical test case based in an ambience monitoring system, providing insights into MQTT's capabilities, limitations and applicability in real-world IoT applications. A series of experiments carried out in this project will serve as well as a comparative analysis with another technology deeply established in the sector, unraveling the strengths and limitations of each protocol. Additionally, the research extends to scrutinizing MQTT implementations across various enterprises, identifying common challenges, and showcasing best practices adopted in real-world scenarios.

This thesis aims to serve as a valuable resource regarding the selection and implementation of messaging protocols in distributed systems, particularly in IoT environments.

Keywords: **MQTT, IoT, Messaging Technology**

Resumen

Las tecnologías de mensajería se han convertido en un elemento crucial del ecosistema digital moderno, ya que desempeñan un papel muy importante en la comunicación entre sistemas distribuidos, los cuales nos rodean en la era actual. El presente trabajo aporta una exploración exhaustiva del protocolo de mensajería *Message Queuing Telemetry Transport* (MQTT), centrándose en su importancia en el contexto del *Internet de las Cosas* (IoT - Internet of Things). El estudio implica una investigación detallada sobre los principios fundamentales de MQTT, sus características y sus rasgos distintivos en comparación con otras tecnologías de mensajería más notorias en la actualidad. La investigación aborda aspectos teóricos y prácticos, incluyendo una revisión crítica de la literatura relevante, un análisis detallado del rendimiento de MQTT a través de experimentos prácticos y un examen de casos de uso del mundo real en diversos escenarios de la industria.

Un aspecto clave de la investigación empírica en el presente trabajo es el desarrollo y evaluación de un caso de prueba práctico basado en un sistema de monitorización ambiental, proporcionando información sobre las capacidades, limitaciones y aplicabilidad de MQTT en aplicaciones IoT del mundo real. Una serie de experimentos realizados en este proyecto servirán también como un análisis comparativo con otra tecnología mucho más establecida en el sector, sacando a relucir las fortalezas y limitaciones de cada protocolo. Además, la investigación se extiende a examinar las implementaciones de MQTT en diversas empresas, identificando desafíos comunes y mostrando las mejores prácticas adoptadas en ciertos escenarios del mundo real.

El presente trabajo tiene como objetivo ser un recurso estimable en la selección e implementación de protocolos de mensajería en sistemas distribuidos, especialmente en entornos de IoT.

Palabras clave: **MQTT, IoT, Tecnología de Mensajería**

Índice

Índice de figuras

1 Introducción

1.1. Motivación

La creciente relevancia de la comunicación eficiente en el panorama actual de la tecnología, especialmente en el ámbito del Internet de las Cosas (IoT - Internet of Things), nos obliga a buscar nuevas alternativas que se adapten a esta evolución inminente. La necesidad de establecer conexiones sólidas entre dispositivos distribuidos ha propiciado una búsqueda continua de tecnologías de mensajería que sean capaces de ofrecer rendimiento óptimo, baja latencia y alta confiabilidad. En este contexto, la tecnología *Message Queuing Telemetry Transport* (MQTT) [1] emerge como una opción prometedora. La motivación fundamental del presente trabajo se centra en esclarecer las características distintivas de esta tecnología prometedora y, por tanto, el potencial reemplazo de otras tecnologías ya establecidas en el sector en favor de la eficiencia en la industria de las tecnologías IoT.

1.2. Objetivos

El núcleo central de esta investigación reside en llevar a cabo una evaluación profunda de la tecnología MQTT y su contraste con otras soluciones como el ya establecido protocolo *Hypertext Transfer Protocol* (HTTP). La meta principal es adentrarse en las particularidades de MQTT, comprender sus aplicaciones, ventajas y desafíos en comparación con alternativas establecidas. Esto será posible desglosando este objetivo general en los siguientes más concretos:

- **Estudio de MQTT como tecnología de mensajería** para describir la tecnología y profundizar en sus puntos fuertes y sus casos de uso generales.
- **Investigación sobre distintos casos de uso comerciales de MQTT** para ejemplificar el uso de la tecnología y llevarlo a la actualidad.

- **Comparativa con otras alternativas disponibles en el mercado** para analizar sus diferencias con respecto de MQTT, y debatir si serían suficientes para reemplazar a estas alternativas más comúnmente usadas.

- **Diseño y desarrollo de un caso de ejemplo de monitorización del ambiente local al dispositivo aplicando el estudio** para aportar resultados que proporcionen más detalles relevantes a la investigación.

Al lograr estos objetivos, se busca ofrecer una perspectiva integral y aplicada de MQTT, proporcionando información práctica y relevante para la implementación de tecnologías de comunicación en sistemas distribuidos y entornos IoT.

Cabe destacar que el caso de uso planteado originalmente era un proyecto de monitorización de energía que, debido principalmente a la disponibilidad de sensores adecuados, se ha optado por reemplazar a favor del nuevo caso de uso que se presenta, estando este basado en un proyecto de monitorización ambiental de manera local al dispositivo IoT. Además, el cambio ha contribuido a disponer de más información a representar y tratar, lo cual resulta interesante para el objetivo global del trabajo.

1.3. Metodología

La estrategia empleada en el proyecto es la metodología ágil, que se fundamenta en un enfoque de desarrollo iterativo e incremental. En este método, tanto los requisitos como las soluciones evolucionan con el tiempo, ajustándose a las necesidades cambiantes del proyecto. Cada ciclo de trabajo se ha extendido por un período de entre 2 y 3 semanas. En cada uno de estos ciclos de trabajo se ha comprobado con los tutores el trabajo realizado en ese período de tiempo, arreglando lo que fuera necesario o dirigiéndolo por la ruta que se ha creído conveniente entre los tres.

El uso de una metodología ágil nos brinda una capacidad superior de adaptación a los cambios y circunstancias que puedan presentarse durante la realización del trabajo, de manera que puedan subsanarse sin problema en corto plazo.

1.4. Estructura del trabajo

El análisis de la tecnología presentada en el presente trabajo está dividido en dos mitades: el estudio del protocolo MQTT, y el caso de uso práctico. Específicamente, la memoria está dividida en las siguientes secciones: la sección 1, en la cual introducimos la tecnología y las motivaciones, objetivos y metodología seguidos en el trabajo. La sección 2 es donde explicamos las tecnologías usadas a lo largo del trabajo. Tras esta, encontramos la sección 3, en la que analizaremos a fondo el protocolo MQTT, el modelo Publicación-Suscripción como base de este, su evolución desde su creación hasta el presente, su estructura, y algunos casos reales de empresas que actualmente hacen uso de él. Posteriormente, se presenta el caso de uso práctico en la sección 4, en la que se describen las condiciones bajo las cuales se realizará el caso práctico, se detallan las arquitecturas hardware y software empleadas, y se realiza una prueba tanto con MQTT COMO con HTTP, tras las cuales se observan y explican los resultados obtenidos. Por último, en la sección 5 se presentan las conclusiones generales del trabajo, y específicas de los resultados obtenidos en el caso de uso, así como las líneas futuras que proponemos posteriores a la realización de este trabajo.

2
Tecnologías

2.1. Arduino IDE

El Arduino IDE (*Integrated Development Environment*, Entorno de Desarrollo Integrado) es una herramienta de software de código abierto diseñada específicamente para programar placas Arduino y otros dispositivos compatibles. Con una interfaz simple y fácil de usar, el Arduino IDE ofrece a los usuarios un entorno intuitivo para escribir, compilar y cargar código en sus dispositivos Arduino. Además de la capacidad de crear y editar *sketches* (programas) en el lenguaje de programación de Arduino, el IDE proporciona una variedad de funciones útiles, como la verificación de errores de sintaxis, la gestión de bibliotecas de software y la monitorización del puerto serial para depuración y comunicación con el dispositivo.

2.2. Arduino Language

El lenguaje de programación de Arduino, conocido como *Arduino Language* o *Wiring*, es una versión simplificada de C++ que ha sido adaptada para facilitar el desarrollo de proyectos con placas Arduino y otras plataformas de hardware compatible. Este lenguaje ofrece una sintaxis clara y accesible, lo que lo hace especialmente adecuado para principiantes y estudiantes de electrónica y programación. Permite a los usuarios interactuar con los periféricos y componentes de hardware de manera intuitiva, utilizando funciones predefinidas para leer sensores, controlar actuadores y comunicarse con otros dispositivos. Además, el entorno de desarrollo integrado (IDE) de Arduino proporciona herramientas de compilación, depuración y carga de código que simplifican el proceso de desarrollo y pruebas.

2.3. Docker

Docker [2] es una plataforma de código abierto que utiliza contenedores para simplificar el desarrollo, implementación y ejecución de aplicaciones. Los contenedores son entornos aislados que incluyen todo lo necesario para ejecutar una aplicación, lo que garantiza la consistencia en diferentes entornos. Docker facilita la portabilidad, eficiencia de recursos y gestión simplificada, lo que ha llevado a su amplia adopción en el desarrollo y despliegue de aplicaciones.

2.4. Eclipse Paho

Eclipse Paho [3] es un conjunto de bibliotecas de código abierto desarrollado por la Fundación Eclipse para implementar los protocolos MQTT y MQTT-SN (*MQTT for Sensor Networks*, MQTT para Redes Sensoriales). Estas bibliotecas proporcionan una implementación eficiente y escalable de los estándares de mensajería MQTT, facilitando la comunicación entre dispositivos conectados a Internet de manera liviana y de bajo consumo de ancho de banda. Paho se utiliza comúnmente en el desarrollo de aplicaciones y servicios IoT para la transmisión eficiente de datos entre dispositivos y servidores.

2.5. Adafruit HUZZAH32 - ESP32 Feather

El microcontrolador ESP32 [4] es una potente y versátil unidad de procesamiento diseñada especialmente para aplicaciones de IoT y proyectos embebidos, ya que incluye conectividad Wi-Fi y Bluetooth integrada. Este microcontrolador ofrece una combinación única de rendimiento, conectividad y eficiencia energética, lo que lo convierte en una opción popular para una amplia gama de aplicaciones, como dispositivos de sensorización, sistemas de control remoto, proyectos de automatización del hogar y mucho más. En concreto, usaremos el modelo *HUZZAH32 Feather* distribuido por la empresa Adafruit [5].

2.5.1. Sensor AM2302

Manufacturado por la empresa *Adafruit*, el sensor AM2302 [6] es un dispositivo que obtiene la temperatura y la humedad del ambiente del lugar donde se encuentre. Para tratar la

información recopilada por este sensor, usaremos una librería de código abierto para Arduino [7].

2.5.2. Sensor CCS811

El sensor CCS811 [8], concretamente el modelo manufacturado por la empresa *Sparkfun*, es un dispositivo que obtiene los niveles de CO2 (dióxido de carbono) y TVOC (*Total Volatile Organic Compounds* - Compuestos Orgánicos Volatiles Totales) presentes en el ambiente del lugar donde se encuentre. Para manejar la información recopilada por este sensor, usaremos una librería de código abierto para Arduino [9].

2.6. Git

Git [10] es un sistema de control de versiones distribuido que permite rastrear y gestionar cambios en el código fuente durante el desarrollo de software. Proporciona un historial de versiones, facilita la colaboración entre desarrolladores y permite ramificar y fusionar código de manera eficiente.

2.7. GitHub

GitHub [11] es una plataforma basada en la web que utiliza Git y proporciona servicios adicionales para el alojamiento de repositorios de código, colaboración en proyectos y seguimiento de problemas. GitHub facilita la colaboración en equipos distribuidos, la revisión de código y la integración continua, convirtiéndose en una herramienta fundamental para el desarrollo colaborativo de software.

2.8. Grafana

Grafana [12] es una plataforma de análisis y visualización de datos de código abierto que ofrece una forma eficiente y flexible de monitorear y analizar datos en tiempo real. Diseñada para integrarse con una amplia variedad de fuentes de datos, incluidas bases de datos, sistemas de monitorización y servicios en la nube, Grafana permite crear paneles interactivos y personalizados con gráficos, tablas y alertas para visualizar y comprender mejor los datos. Su interfaz intuitiva y altamente personalizable, junto con su soporte para la creación de paneles

dinámicos y el establecimiento de alertas, la convierten en una herramienta indispensable para profesionales de TI, operadores de sistemas y equipos de desarrollo que buscan supervisar y analizar datos críticos de manera efectiva. Además, Grafana fomenta la colaboración y el intercambio de conocimientos a través de su comunidad activa y la disponibilidad de numerosas integraciones y complementos que amplían su funcionalidad y adaptabilidad a diversos entornos y necesidades empresariales.

2.9. InfluxDB

InfluxDB [13] es una base de datos de series temporales de código abierto diseñada para almacenar, consultar y visualizar datos de series temporales en tiempo real. Es altamente escalable y optimizada para el almacenamiento y recuperación eficiente de datos que cambian con el tiempo, como métricas de sistemas, registros de eventos y datos de sensores. InfluxDB utiliza un modelo de datos basado en series temporales y ofrece dos potente lenguaje de consultas llamados InfluxQL y Flux (el idioma más nuevo), así como integración con herramientas de visualización como Grafana, lo que lo convierte en una opción popular para aplicaciones de monitorización, análisis de rendimiento e IoT.

2.10. Mosquitto

Mosquitto [14] es un servidor de mensajería MQTT de código abierto que desempeña un papel crucial en la implementación y gestión eficientes de la arquitectura MQTT. Diseñado para facilitar la comunicación entre dispositivos en redes de Internet de las cosas (IoT), Mosquitto actúa como un intermediario o *broker* que permite el intercambio de mensajes entre los clientes conectados. Este software, desarrollado por la Eclipse Foundation, ofrece una solución robusta y escalable para la implementación de la tecnología MQTT, facilitando la suscripción, publicación y gestión de mensajes en entornos distribuidos. La flexibilidad de Mosquitto lo convierte en una elección popular para diversos casos de uso, desde proyectos de pequeña escala hasta despliegues empresariales más complejos, y su código abierto fomenta la participación de la comunidad en su mejora continua y adaptación a diversas necesidades.

2.11. Raspberry Pi 3B+

La Raspberry Pi 3B+ es una versátil y asequible computadora de placa única (SBC) desarrollada por la Raspberry Pi Foundation [15]. Equipada con un procesador de cuatro núcleos, junto con 1 GB de memoria RAM, la Raspberry Pi 3B+ ofrece un rendimiento sólido en un formato compacto. Esta placa también incluye conectividad inalámbrica mediante Wi-Fi y Bluetooth, ampliando sus capacidades de comunicación. Con puertos USB, HDMI, y GPIO (*General Purpose Input/Output*), la Raspberry Pi 3B+ se adapta a una amplia gama de proyectos y aplicaciones, desde la creación de estaciones de trabajo ligeras hasta la implementación de soluciones IoT. Su flexibilidad y precio accesible la han convertido en una opción popular para entusiastas, educadores y desarrolladores que buscan una plataforma de desarrollo compacta y potente para una variedad de proyectos.

2.12. Telegraf

Telegraf [16] es una herramienta de código abierto para la recopilación y envío de métricas y datos de telemetría. Diseñado para ser ligero y altamente eficiente, Telegraf puede recopilar una amplia variedad de datos, incluidas métricas de sistemas, aplicaciones y dispositivos de red, desde diversas fuentes y en diversos formatos. Utilizando una arquitectura de plugins, Telegraf es altamente adaptable y puede integrarse fácilmente con una variedad de sistemas de monitorización y almacenamiento, como InfluxDB, Prometheus, y muchos otros. Su flexibilidad y escalabilidad lo hacen una opción popular para implementaciones de monitoreo en tiempo real y análisis de rendimiento en infraestructuras de cualquier tamaño.

2.13. Visual Studio Code

Visual Studio Code [17] es un entorno de desarrollo de código abierto y gratuito desarrollado por Microsoft. Diseñado para satisfacer las necesidades de programadores de diversos lenguajes y plataformas, Visual Studio Code destaca por su interfaz sencilla y altamente personalizable. Con soporte integrado para una amplia gama de lenguajes de programación, extensiones y complementos, esta herramienta ofrece una experiencia de desarrollo eficiente y versátil. Funcionalidades como resaltado de sintaxis, depuración integrada, control de versiones y una integración fluida con Git contribuyen a su popularidad entre desarrolladores

individuales y equipos. La capacidad de extender sus funcionalidades mediante extensiones proporciona a los usuarios un control total sobre su entorno de desarrollo, convirtiendo a Visual Studio Code en una opción preferida para la creación y mantenimiento de proyectos de software de cualquier tamaño y complejidad.

3

Estudio de MQTT

En la actual era digital, donde la conectividad entre dispositivos se ha vuelto esencial, MQTT ha surgido como una alternativa bastante prometedora para las tecnologías de mensajería. A lo largo de su historia, este protocolo ha atravesado una evolución que va desde simplemente su nombre hasta el modelo de disponibilidad que sigue.

3.1. El modelo Publicación-Subscripción

Como describe Andy Stanford-Clark, investigador de IBM, en la especificación de la primera versión del protocolo [18] que analizaremos posteriormente, antes de la aparición de MQTT la situación de las comunicaciones presentaba bastantes limitaciones para escenarios que iban surgiendo en base a ciertas necesidades, sobre todo aquellos donde se necesitaba tener una comunicación en tiempo real. Hasta este entonces, las tecnologías de comunicación disponibles consistían esencialmente en líneas telefónicas y tecnologías de radio, y por ello, el método principal de intercambio de información utilizado era el de petición-respuesta. Este método, aunque es muy comúnmente utilizado incluso en la actualidad, tiene ciertas limitaciones que pueden ser clave para este ámbito de trabajo. La limitación más importante es el hecho de que este tipo de tecnologías disponibles en el momento solo soportan el envío de información en modo semidúplex, lo cual suponía la necesidad de implementar algún tipo de detección de colisiones para poder evitarlas. Además, el hecho de que la información no se recibe hasta que esta no se solicite, complica la implementación de sistemas en tiempo real ya que para obtener la información actualizada de manera constante se tendrían que hacer continuas peticiones (polling), y comparar la información recibida como respuesta con la que ya se tenía anteriormente, en busca de cualquier cambio que hubiese ocurrido.

Pero con la llegada del modelo publicación-suscripción, los sistemas en tiempo real atravesaron una evolución sin precedentes y se vieron beneficiados gracias a la estructura que lo

define. Según afirma el artículo del profesor Patrick Eugster [19], la principal característica de este modelo es que a diferencia del modelo petición-respuesta, donde la comunicación tiene únicamente dos extremos, en el modelo publicación-suscripción encontramos una figura central llamada *broker*, la cual es la encargada de gestionar la información recibiéndola y repartiéndola a los diferentes clientes. Es decir, un cliente que quiere transmitir información, no lo hace con el objetivo de mandarla a otro cliente de manera unívoca como lo haría si siguiera el modelo petición-respuesta. En cambio, lo que hará este dispositivo es publicarla a un canal, llamado tópico, del cual otros clientes podrán leer si es que se han suscrito a ese tópico previamente. Podemos observar una comparativa entre los dos modelos con un ejemplo de comunicación básico en la Figura 1. Siguiendo las afirmaciones de Eugster, con este modelo se consigue por tanto una notificación de eventos de manera totalmente asíncrona gracias al desacoplamiento alcanzado en el modelo a lo largo de las siguientes tres dimensiones:

- **Desacoplamiento espacial:** Como ya habíamos avanzado, los participantes de la comunicación no tienen por qué conocerse entre ellos para que esta se produzca. Los publicadores publican su información a los tópicos que prefieran, y a su vez los suscriptores se suscriben a los tópicos de los que deseen leer la información publicada. Ninguna de las dos partes conoce siquiera el número de participantes que hay en la parte contraria.

- **Desacoplamiento temporal:** Los participantes no tienen por qué estar participando en la comunicación a la vez. Los publicadores pueden enviar información en algún momento en el que no haya ningún suscriptor conectado recibiendo mensajes. Los suscriptores posteriormente se pueden suscribir al tópico donde fue publicada la información y estos la recibirán, sin necesidad de que los publicadores estén conectados en este momento tampoco.

- **Desacoplamiento sincrónico:** Ni los publicadores ni los suscriptores se bloquean en el proceso de comunicación. Un publicador puede enviar información y dedicarse a otra tarea, sin esperar respuesta, y un suscriptor puede recibir la información a través de una función de *callback* mientras que está realizando otra tarea concurrentemente.

Habiendo surgido este innovador e interesante nuevo modelo de comunicaciones a la luz, comenzaron a aparecer los primeros protocolos aprovechándolo, entre ellos el que estudiamos en el presente trabajo y que hoy en día conocemos como MQTT.

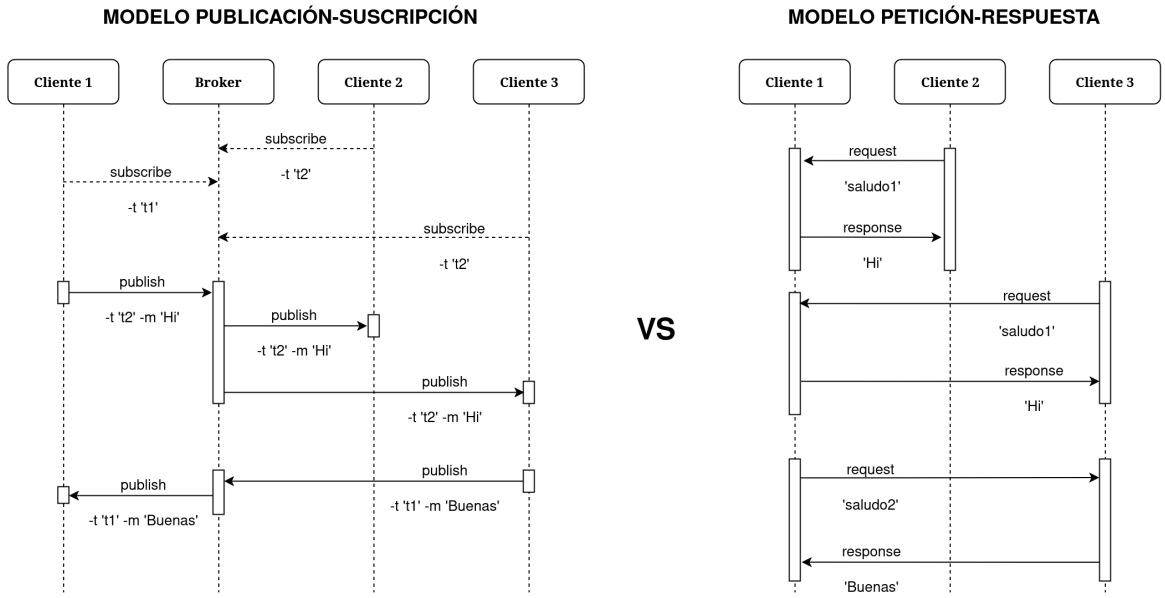

Figura 1: Comparativa de los diagramas de secuencias de los modelos Publicación-Suscripción y Petición-Respuesta.

3.2. Evolución del protocolo MQTT

3.2.1. Primeras versiones del protocolo

El 26 de marzo de 1999 se hace pública la versión 1.6 del protocolo *MQ Series Integrator Pervasive Device Protocol* [18] a manos de Andy Stanford-Clark y Arlen Nipper, presidente de Arcom Control Systems. Previamente conocido como *Argo Lightweight On The Wire Protocol* (Protocolo ligero por cable *Argo*), este protocolo se creó para proveer un método eficiente de comunicaciones en el marco del sistema de control industrial SCADA, que estaba siendo utilizado para monitorizar oleoductos. Previo a la publicación de especificación, el protocolo se creó de manera privada dentro de IBM, y esta fue la primera especificación del protocolo que vio la luz pública. Tras una serie de versiones implementando cambios a nivel pequeño y medio, se alcanza la última modificación en la versión de la especificación, la versión 2.3, versión actual de este documento de especificación. En estas versiones iniciales se define el núcleo del protocolo, el cual en su gran mayoría se mantiene hasta la versión vigente de MQTT.

El protocolo alcanzó la versión 3.0 [20] a manos de Andy Stanford-Clark en octubre del mismo año, siendo esta la última versión antes de pasar a ser autorizada completamente por

IBM como organización. Los cambios de esta versión con respecto de la versión 2.3, excepto la autoría compartida entre Stanford-Clark y IBM, son mínimos y no alteran el funcionamiento original del protocolo.

3.2.2. Versión 3.1 - Autorización de IBM

La primera y única versión del protocolo autorizada por IBM, la versión 3.1 [21], vio la luz en 2013, antes de pasar a cargo del grupo OASIS en la versión 3.1.1. El nombre del protocolo se cambió a *MQ Telemetry Transport* (MQ Transporte de Telemetría, en español), fuente original del acrónimo MQTT. En esta versión se introdujeron una serie de cambios más esenciales, entre ellos, el más importante es la introducción de la posibilidad de enviar una combinación «usuario,contraseña» en el paquete CONNECT para autentificar la conexión, si es que se activa esta configuración en el sistema. Los demás cambios incluyen nuevos códigos de respuesta para los mensajes de tipo CONNACK introducidos por motivos de seguridad y el soporte completo de codificación UTF-8 de los Strings (cadenas de carácteres) en vez de usar únicamente el subconjunto US-ASCII (carácteres ASCII americanos). En esta versión, los flujos de comunicación no se ven afectados aún, aunque se hubiera introducido la posibilidad de autentificar usuarios, proceso que en posteriores versiones si recibiría un flujo dedicado a él. Como hemos comentado previamente, la autentificación en esta versión se hace mandando la información necesaria en el paquete CONNECT, que será recibido y verificado por el servidor, y establecerá la conexión si la verificación ha sido exitosa.

3.2.3. Versión 3.1.1 - Incorporación a OASIS Open

A partir de la versión 3.1.1 [22], lanzada en octubre del 2014, IBM cedió el mantenimiento y autoría de MQTT al grupo OASIS Open [23], siendo OASIS el acrónimo de *Organization for the Advancement of Structured Information Standards* (Organización para el Avance de Estándares de Información Estructurada), aunque de la edición de las especificaciones se seguían encargando trabajadores de IBM. En concreto, los editores de la especificación de esta versión del protocolo son Andrew Banks y Rahul Gupta, de IBM. OASIS es una organización internacional que se dedica al desarrollo, adopción y mantenimiento de estándares de servicios web. Entre las diversas categorías de los estándares que tratan encontramos IoT, energía, ciberseguridad, mensajería, cloud (servicios en la nube), junto con muchos otros.

En cuanto a cambios prácticos del funcionamiento del protocolo se refiere, no encontramos muchos más allá de la estandarización de todo el contenido de la especificación anterior. En el apéndice C de la especificación de esta versión podemos ver un historial más detallado de los cambios que ha atravesado el protocolo desde la versión anterior, siendo en su mayoría resoluciones a *issues* (propuestas o puntos a resolver en un proyecto), y como comentábamos, la normalización del protocolo.

Existe una versión *secreta*, la versión 4 del protocolo. Según el artículo de Jen Deters [24], es secreta ya que nunca se llegó a publicar una versión 4 como tal, si no que la versión 3.1.1 ya fue considerada la versión 4 de MQTT. Y esto se debe a que en los mensajes enviados usando la versión 3.1.1 del protocolo, en el byte número 7 (el byte de la cabecera variable en el que se especifica la versión del protocolo), se usó siempre el valor 4 (0x04, en hexadecimal). Por tanto, la siguiente versión del protocolo MQTT fue la versión 5.

3.2.4. Versión 5.0 - Versión actual

La versión 5.0 de MQTT [25] es la versión actual (en el año 2024) del protocolo, y fue lanzada en marzo del 2019. Sigue siendo autorizada por el grupo OASIS, y es la versión que usaremos posteriormente tanto de referencia para definir sus características y partes más importantes, como en el caso de uso práctico. En esta especificación se suman a los editores de la especificación de la versión anterior (Andrew Banks y Rahul Gupta), Ed Briggs de Microsoft y Ken Borgendale de IBM.

Aunque en la sección 3.3 revisaremos los detalles más importantes de la especificación actual, podemos listar los cambios más relevantes con respecto a la versión anterior, que de nuevo se encuentran en el apéndice C de la especificación. Por tanto, los cambios más destacables son: la introducción de un sistema de vencimiento de sesión (para continuar la sesión anterior sin crear una nueva, mientras no hayamos sobrepasado el intervalo de vencimiento de sesión especificado, que empieza a contar tras una desconexión), un nuevo sistema de vencimiento de los mensajes publicados (pasado este intervalo de vencimiento de mensaje especificado, que empieza a contar tras la publicación de un mensaje, el broker dejará de publicar dicho mensaje a los suscriptores), ahora el broker es capaz de enviar un DISCONNECT para indicar la razón del cierre de la conexión, se introduce la posibilidad de especificar un formato (binario, texto) y un estilo MIME (*Multipurpose Internet Mail Extensions*, Extensiones Multipropósito de Correo

de Internet) para el payload de un mensaje (que se redirigirá a los respectivos receptores de este), la aparición de un ID de suscripción (que se envía con el SUBSCRIBE, y se devolverá con cada mensaje recibido a causa de esa suscripción) cuyo propósito es determinar qué suscripción o suscripciones causaron que cada mensaje se reciba, la creación de las suscripciones compartidas para equilibrar la carga de los consumidores de una suscripción, la mejora del sistema de autenticación (ofreciendo la posibilidad de usar un mecanismo de desafío-respuesta), entre los numerosos cambios introducidos.

3.2.5. MQTT-SN

Inicialmente conocido como MQTT-S, MQTT-SN (*MQTT for Sensorial Networks*, MQTT para Redes de Sensores) es una versión del protocolo MQTT fabricada particularmente para las redes de sensores. La versión actual de esta variante del protocolo es la versión 1.2 [26], publicada en mayo de 2011. No obstante, la primera versión del protocolo fue la versión 1.0, creada en noviembre del 2007, y luego pasó por la versión 1.1 en junio del 2008, como nos sugiere el historial de cambios de esta especificación (las especificaciones de estas dos primeras versiones no están públicas). Como nos narra la especificación, esta variante del protocolo MQTT surge debido al creciente interés en las redes de sensores inalámbricas (WSNs, de *Wireless Sensor Networks*). Las redes de sensores son redes de ordenadores muy pequeños (los nodos), equipados con sensores, que se unen para trabajar colectivamente en una o más tareas.

MQTT-SN surge concretamente debido a la aplicabilidad que presenta el modelo Publicación-Suscripción y como este solucionaría diversas limitaciones que presenta dicha familia de sistemas. La primera de las limitaciones de estos sistemas viene de que, al ser un número grande de dispositivos interconectados, se tiene que establecer una conexión inalámbrica entre ellos ya que una conexión cableada sería inviable. Esta configuración es muy dinámica por naturaleza, ya que la conexión inalámbrica puede romperse temporalmente en cualquier momento y los nodos pueden fallar y ser reemplazados con frecuencia, y es por esto que mantener el sistema tradicional de direccionamiento entre los dispositivos puede ser muy complicado de lograr. Y en muchas ocasiones, en este tipo de sistemas no se necesita conocer la dirección/identidad, si no que interesa más el contenido de los datos. Además, varias aplicaciones pueden estar interesadas en información de un mismo sensor para propósitos distintos. Adicional a este problema, observamos que los esquemas de direccionamiento en dos dispositivos pueden ser

distintos, y de esta manera, estos dispositivos no podrían nunca dirigirse entre ellos para entablar una comunicación. Aquí es donde entra el modelo Publicación-Suscripción como opción a considerar, debido a que este modelo es inherentemente orientado a los datos y presenta esos desacoplamientos espacial, temporal y sincrónicos que argumentábamos previamente. Esto significa que si usamos el modelo Publicación-Suscripción podemos agrupar y direccionar los datos en función de lo que queramos, sin tener que identificar a los participantes de la comunicación. Por ejemplo, si tenemos un sensor que mide valores energéticos en cierto lugar, y dos aplicaciones, una de monitorización energética y otra de un sistema en el que los valores energéticos son información adicional, ambos haciendo uso de la información recibida por dicho sensor, usando el modelo Publicación-Suscripción podemos agrupar toda la información saliente del sensor en un tópico al que se suscribirán ambas aplicaciones, sin necesidad de identificar ninguno de los participantes en ningún momento.

3.3. Estructura del protocolo MQTT

A lo largo de toda esta sección vamos a comentar las partes principales del protocolo MQTT, basándonos para ello en el documento de especificación de la versión 5.0 del protocolo [25].

3.3.1. Cabecera

La primera parte de la especificación nos detalla como sería la cabecera de un mensaje enviado usando MQTT. Hay dos cabeceras que se usan en un mensaje enviado por MQTT, la cabecera fija y la cabecera variable.

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Byte 1	Tipo de control del paquete				Flags específicos para cada tipo de control			
Byte 2	Remaining Length (Longitud Restante)							

Tabla 1: Cabecera fija de un mensaje enviado usando el protocolo MQTT

La cabecera fija, representada en la Tabla 1, tiene un tamaño de 2 bytes: el primero es el que contiene información como el tipo de mensaje y los flags pertinentes, y el segundo está dedicado a un campo llamado *Remaining Length* (longitud restante) y es una variable entera

de 1 byte cuyo contenido representa el tamaño de la cabecera variable sumado con el tamaño del payload. En el primer byte, los bits del 7 al 4 están dedicados al tipo de mensaje, y de entre los posibles tipos destacamos:

- **CONNECT:** Elemento de la enumeración 1. Un cliente solicita conectarse al broker.
- **CONNACK:** Elemento de la enumeración 2. Confirmación de la solicitud de conexión al broker
- **PUBLISH:** Elemento de la enumeración 3. Publicación de un mensaje.
- **PUBREC:** Elemento de la enumeración 5. Publicación de mensaje recibida (seguro de entrega 1).
- **PUBREL:** Elemento de la enumeración 6. Publicación de mensaje liberada (seguro de entrega 2).
- **PUBCOMP:** Elemento de la enumeración 7. Publicación de mensaje completado (seguro de entrega 3).
- **SUBSCRIBE:** Elemento de la enumeración 8. Petición de suscripción de un cliente.
- **UNSUBSCRIBE:** Elemento de la enumeración 10. Petición de baja de una suscripción de un cliente.
- **DISCONNECT:** Elemento de la enumeración 14. Un cliente notifica su desconexión del broker.
- **AUTH:** Elemento de la enumeración 15. Intercambio de autenticación.

Los bits restantes del primer byte están dedicados a los flags. En concreto, los flags que podemos activar en un mensaje enviado por MQTT son DUP (envío duplicado) en el bit 3, QoS (*Quality of Service*, calidad de servicio) en los bits del 2 al 1, y RETAIN (retener) en el bit 0. El flag DUP se activará si el mensaje es un mensaje de publicación (enviado por un cliente o por el broker) que ya se había intentado enviar previamente, y además el flag de QoS tiene un valor mayor a 0 (que como veremos a continuación, significa que se proporcione al menos el nivel mínimo de confirmación en los mensajes en la comunicación). Los bits que forman el flag QoS

sirven para decidir el nivel de seguridad del envío de mensajes de publicación. Hay 4 posibles valores para este flag, representados por los dos bits que ocupa, como podemos observar en la Tabla 2.

Valor QoS	Bit 2	Bit 1	Descripción
0	0	0	como máximo 1 (<=1). *Fire and Forget*, disparar y olvidar.
1	0	1	como mínimo 1 (>=1). *Acknowledged delivery*, entrega reconocida.
2	1	0	exactamente 1 (==1). *Assured delivery*, entrega asegurada.
3	1	1	no se usa QoS.

Tabla 2: Posibles valores del flag QoS con sus respectivos bits y descripciones.

Usando el valor 0, la entrega se realiza siguiendo los mejores esfuerzos que el protocolo TCP/IP puede ofrecer, sin esperar ninguna respuesta de confirmación por parte del receptor y sin usar ningún mecanismo de reintento en caso de fallo. Es decir, usando este valor de QoS, el mensaje puede llegar 0 o 1 vez (como máximo 1). Con el valor 1, el mensaje se coloca en una cola de mensajes, y la entrega de este se reconoce haciendo uso de un mensaje de tipo PUBACK. En el caso de error, o de que haya pasado un tiempo y no se haya recibido el reconocimiento de la entrega, el emisor reenviaría el mensaje con el flag DUP activado, dejando constancia de que este mensaje es un mensaje duplicado de un mensaje que se envió anteriormente. Así, con este valor de QoS, el mensaje llegaría 1 o más veces si es que este se duplica (como mínimo 1). En cambio, haciendo uso del valor 2 conseguimos que el mensaje se entregue sin hacer uso del mecanismo de duplicación, a través de otros flujos del protocolo que sacrifican un precio a pagar en términos de tráfico de red a cambio del mayor nivel de garantía que el protocolo puede ofrecer en cuanto a la recepción de mensajes se refiere.

La especificación del protocolo nos muestra también el uso de una cabecera variable. La cabecera variable es una segunda cabecera adicional a la cabecera fija, en la que se incluyen más parámetros de la comunicación como el nombre y versión del prótocolo, el temporizador de *Keep Alive* (mantener vivo) de un mensaje, el nombre del tópico donde se publica un mensaje, etc. En el flujo de comunicación del protocolo, veremos la cabecera variable que puede presentar cada tipo de mensaje.

3.3.2. Payload

El payload de un mensaje es el verdadero contenido del mensaje, lo que el emisor quiere transmitir al receptor. Este payload puede tener cualquier formato, desde texto plano hasta JSON (*JavaScript Object Notation*, notación de objecto de JavaScript), y por tanto el protocolo MQTT lo tratará como un BLOB (*Binary Large Object*, objeto binario grande) sin presentar ninguna suposición sobre la naturaleza o contenido de este. Los tipos de mensajes que usan un payload son CONNECT (este contendrá información que identifica al cliente), SUBSCRIBE (el cual incluirá una lista de tópicos a los que el cliente se desea suscribir, y el respectivo flag QoS para cada uno de ellos), SUBACK (este proporcionará una lista de niveles QoS admitidos con respecto a la petición realizada previamente con un mensaje SUBSCRIBE), y PUBLISH (llevando el mensaje que se quiere publicar).

3.3.3. Flujo de comunicación

Los distintos flujos de comunicación del protocolo son:

- **CONNECT - CONNACK:** Flujo cuyo propósito es la conexión de un cliente al broker. Este se inicia cuando el cliente manda un paquete de tipo CONNECT y el servidor le responde con un paquete de tipo CONNACK. Solo se debe mandar un paquete de tipo CONNECT desde un cliente al broker en una conexión de red. En caso de recibir un segundo paquete CONNECT desde el mismo cliente en la misma conexión de red, el broker deberá asumirlo como un error de protocolo y cerrar la conexión. En este paquete, en la cabecera fija solo veremos el tipo de mensaje (1 - CONNECT) y el campo *Remaining Length* (tamaño de la cabecera variable y tamaño del payload sumados), y el resto de flags serán reservados de usarse. En la cabecera variable encontramos los campos referentes al nombre del protocolo ("MQTT", como un String codificado en UTF-8), la versión del protocolo (5, enviado en forma de byte sin signo), los flags de CONNECT (usuario, contraseña, los parámetros del *Will* o voluntad, el flag *Clean Start* o Inicio de Cero y el flag reserved, cuyo valor tiene que ser 0 o de lo contrario denotaría un paquete malformado), los 2 bytes de *Keep Alive* (intervalo de tiempo máximo, medido en segundos, que está permitido que transcurra desde que el cliente termina de transmitir un paquete de control de MQTT hasta que empieza a mandar el siguiente), y las propie-

dades de CONNECT (como el intervalo de expiración de la sesión y el tamaño máximo de paquete permitido entre otras propiedades). El payload de los mensajes de este flujo tendrá un tamaño variable, determinado por la cantidad de información que se quiere incluir, indicada previamente en la cabecera en las propiedades de identificación del cliente, propiedades/tópicos/payload de voluntad, y el usuario y contraseña a usar por el cliente, como ya hemos indicado antes. Por último, el servidor mandará al cliente un paquete de tipo CONNACK en respuesta al paquete de tipo CONNECT del cliente. Este paquete tiene en su cabecera fija el tipo de mensaje (2 - CONNACK), y el campo *Remaining Length* de la misma manera que en el paquete CONNECT. En la cabecera variable se incluyen bastante menos cosas que las que utiliza CONNECT. En este caso solo se usan el flag de reconocimiento de la conexión, el código de la razón de conexión (0x00 si la conexión es aceptada y satisfactoria, 0x80-0x9F si ha ocurrido algún error), y las propiedades (entre ellas la propiedad *Maximum QoS*, QoS Máximo, que indica el nivel máximo de QoS posible a usar durante la comunicación, donde si no se indica nada se usa el nivel 2).

- **PUBLISH - PUBACK/PUBREC/PUBREL/PUBCOMP:** Flujo cuyo objetivo es el de transportar un paquete cuyo contenido será un mensaje de la aplicación, ya sea desde un cliente hacia el broker, o desde el broker hacia a un cliente. Este flujo lo inicia el emisor de un mensaje al enviar un paquete de tipo PUBLISH. En este paquete, en la cabecera fija veremos el tipo de mensaje (3 - PUBLISH), el bit DUP (a 0 si es la primera vez que se manda el paquete, o a 1 si es otro intento de envío del mensaje, y por tanto es un paquete duplicado, y además los bits de QoS denotan un nivel de QoS, distinto de 0 y de 3, de entre los posibles valores explicados previamente), el bit RETAIN (si el valor es 1 el servidor reemplazará cualquier mensaje retenido en el tópico por este mensaje entrante, y si es 0 no se llevará a cabo ningún reemplazo, si no que se almacenarán tanto los anteriores como el entrante), y el campo *Remaining Length*. En la cabecera variable de un paquete PUBLISH lo primero que encontraremos es el nombre del tópico donde se pretende publicar el mensaje. Luego, en su contenido veremos campos como el identificador de paquete (si el nivel de QoS es 1 o 2) y las propiedades de PUBLISH (como el formato del payload, el intervalo de expiración del mensaje, el topic de respuesta donde se publicará la respuesta, el identificador de suscripción o el tipo de contenido que se

transporta). El payload en un paquete PUBLISH es directamente el mensaje de aplicación que se desea publicar. La respuesta recibida a un paquete PUBLISH variará según el nivel de QoS definido para la comunicación. Si el nivel definido es 0, no se esperará ninguna respuesta. En cambio, si el nivel establecido es 1, se esperará un paquete de confirmación PUBACK, y si es 2, se usa un mecanismo de seguro de entrega de paquetes mediante el cual se esperan recibir tres mensajes de confirmación, PUBREC, PUBREL y PUBCOMP, en este orden. En los paquetes de tipo PUBACK y PUBREC, se manda un código de respuesta en la cabecera variable, indicando la respuesta del servidor (entre los posibles códigos de respuesta encontramos *Success* o éxito, *Unspecified error* o error no especificado, *Not Authorized* o no autorizado, y otros), mientras que PUBREL y PUBCOMP solo tiene dos códigos de respuesta posible en su cabecera variable (*Success* o éxito, o *Packet Identifier not found* o identificador de paquete no encontrado) y sirven para confirmar el paquete anterior (PUBCOMP confirma un paquete PUBREL, PUBREL confirma un paquete PUBREC, y PUBREC confirma un paquete PUBLISH).

- **SUBSCRIBE/UNSUBSCRIBE - SUBACK/UNSUBACK:** Flujo que se inicia desde un cliente hacia el servidor y que consiste en crear o eliminar una o varias suscripciones de dicho cliente a los tópicos pertinentes del sistema. Al crearse una suscripción, el servidor manda al cliente los mensajes del tópico al que se ha suscrito por medio de paquetes PUBLISH. Este flujo se inicia cuando el cliente manda un paquete de tipo SUBSCRIBE (si se quiere crear las suscripciones) o UNSUBSCRIBE (si se quieren eliminar las suscripciones) y el servidor le responde con un paquete de tipo SUBACK o UNSUBACK respectivamente. En este paquete, en la cabecera fija solo veremos el tipo de mensaje (8 - SUBSCRIBE) y el campo *Remaining Length*, y el resto de flags serán reservados de usarse. En la cabecera variable encontraremos las propiedades necesarias para un paquete SUBSCRIBE (la longitud de las propiedades, el identificador de la suscripción, y las propiedades de usuario). En el payload de un paquete SUBSCRIBE veremos una lista de filtros de tópicos indicando los tópicos a los que se quiere suscribir el cliente, que tienen que estar en formato string codificado en UTF-8, y a cada filtro de tópico le acompañará un byte de opciones de la suscripción. En respuesta a este paquete, el servidor mandará un paquete SUBACK, combinando todos en uno si hay varios tópicos en el paquete SUBSCRIBE. En un paquete SUBACK, en la cabecera fija solo veremos el tipo de men-

saje (9 - SUBACK) y el campo *Remaining Length*, y el resto de flags serán reservados de usarse. En la cabecera variable encontraremos las propiedades de un paquete SUBACK (la longitud de las propiedades, la razón de la respuesta para diagnósticos y pruebas, y las propiedades de usuario). En el payload de un paquete SUBACK encontraremos una lista de códigos de respuesta (una para cada tópico al que se ha suscrito el cliente con el paquete SUBSCRIBE), siendo algunos de ellos *Granted QoS 0/1/2* (QoS concedido 0/1/2, si la suscripción se ha aceptado y se ha concedido un nivel de QoS 0/1/2), *Unspecified error* (error no especificado), o *Topic Filter invalid* (filtro de tópico inválido, si el tópico no existe o el cliente no esta autorizado).

- **UNSUBSCRIBE - UNSUBACK:** El flujo de UNSUBSCRIBE - UNSUBACK permite al cliente eliminar su suscripción a uno o varios tópicos a los que estuviera suscrito inicialmente. Este flujo y el contenido de sus paquetes son prácticamente iguales que los del flujo SUBSCRIBE - SUBACK, siendo la principal diferencia entre ambos el objetivo que tiene flujo, como ya hemos comentado. Los únicos cambios que existen como tal son los posibles códigos de respuesta del UNSUBACK.

- **AUTH:** Flujo que se utiliza para entablar un intercambio de autenticación entre un cliente y el servidor, como puede ser una autenticación por desafío-respuesta. En este paquete, en la cabecera fija solo veremos el tipo de mensaje (15 - AUTH) y el campo *Remaining Length*, y el resto de flags serán reservados de usarse. En cuanto a la cabecera variable, encontramos el código de razón (que puede ser *Success* -o éxito- viniendo del servidor hacia el cliente, *Continue authentication* -o continuar autenticación- en cualquiera de las direcciones, o *Re-authenticate* -o re-autenticarse- desde el cliente hacia el servidor), y las propiedades de AUTH (como el método de autenticación designado, los datos de autenticación o las propiedades del usuario). Los paquetes AUTH no hacen uso de ningún payload.

- **DISCONNECT:** Flujo cuyo propósito es el cierre de la conexión entre un cliente y le server. Este flujo únicamente consta de un paquete de tipo DISCONNECT mandado por cualquiera de las dos partes, sin obtener respuesta del otro. El receptor, al recibir este mensaje, cierra la conexión automáticamente sin enviar ninguna respuesta más, siendo por tanto el paquete DISCONNECT el último mensaje de una conexión. En este paque-

te, en la cabecera fija solo veremos el tipo de mensaje (14 - DISCONNECT) y el campo *Remaining Length* (tamaño de la cabecera variable y tamaño del payload sumados), y el resto de flags serán reservados de usarse. En la cabecera variable encontramos los campos referentes al código de la razón de la desconexión (entre ellos el de la desconexión normal, el de error no especificado, el de servidor ocupado, el de expiración de Keep Alive, y muchos más), y las propiedades de DISCONNECT (como el intervalo de expiración de la sesión, la cadena de la razón de la desconexión, las propiedades del usuario, y la referencia a otro servidor en caso de que se quiera redirigir al cliente a este al cerrarse la conexión). Los paquetes DISCONNECT no tienen ningún payload.

3.3.4. Tópicos y Wildcards

Los tópicos, una de las partes más perceptibles y esenciales del protocolo, son el medio de publicación de los mensajes en MQTT. Un suscriptor recibirá los mensajes que se publiquen en los tópicos para los que creó una suscripción previamente, y un publicador publicará los mensajes que desee en los tópicos que desee. Tanto los suscriptores como los publicadores pueden suscribirse o publicar al tópico que quieran, teniendo este el nombre que tenga. Es decir, en este protocolo no hay una lista de tópicos definidos, si no que un cliente puede suscribirse o publicar a un tópico que no se hubiera definido nunca antes en el sistema. Además, los tópicos son multi-nivel, estando cada nivel separado del siguiente por una barra (/). Esto da lugar a tópicos como 'casa', 'casa/planta1', 'casa/planta1/habitacion1', y así sucesivamente.

Y eso no es todo, ya que el protocolo presenta una mayor flexibilidad todavía en los tópicos al introducir el uso de *Wildcards* (comodines en español). Los wildcards nos permite referirnos a múltiples tópicos a la vez cuyos nombres siguen una estructura común (un patrón). Existen dos tipos de wildcards: la almohadilla (#), que va al final de un tópico y puede 'rellenar' uno o varios niveles en el nombre del tópico, y el más (+), que va entre dos niveles del tópico y puede 'rellenar' un único nivel (el que está entre los dos niveles donde hemos colocado el símbolo). Vamos a ver unos ejemplos para entenderlo mejor (los tópicos usados son inventados y podríamos usar cualquier nombre en cada nivel):

- **casa/planta1/#:** Este tópico podría rellenarse con 'casa/planta1/habitación1', 'casa/planta1/salon', 'casa/planta1/habitacion2/baño', etcétera. Es decir, en este ejemplo el uso

del wildcard # nos resultaría en la posibilidad de referirnos a todas las habitaciones (y sub-habitaciones) de la planta 1 en una casa.

- **alumnos/+/puntuacion:** Este tópico podría rellenarse con 'alumnos/a1/puntuacion', 'alumnos/a2/puntuacion', 'alumnos/a3/puntuacion', etcétera. En este caso, el uso del wildcard + nos permitiría dirigirnos a la puntuación de cada uno de los alumnos.

3.4. Casos de uso reales - Shelly

En el primer caso de uso real de MQTT en empresas ponemos el foco en Shelly [27], una empresa que se dedica a lograr un alto nivel de automatización en el hogar inteligente por medio de dispositivos IoT de varios tipos. Los dispositivos de Shelly permiten a los usuarios realizar tareas como encender/apagar las luces, subir/bajar las cortinas, activar/desactivar otros aparatos electrónicos (entre otras muchas más) desde un solo dispositivo a través de Wi-Fi y Bluetooth. En cuanto a la tecnología de comunicaciones, Shelly usa en sus dispositivos el método RPC (*Remote Procedure Call*, o Llamada a Procedimiento Remoto en español), un método conocido en computación distribuida que se usa para ejecutar un procedimiento (o subrutina) en remoto desde otro programa, de forma que el programador no distingue si la subrutina está siendo ejecutada de manera local o remota al programa desde donde se ejecuta. En este caso, Shelly usa RPC para enviar un JSON al dispositivo con diversos campos, entre ellos el comando principal, y para posteriormente recibir la respuesta en JSON también.

3.4.1. Payload

El payload usado en la mensajería en Shelly es común a todas las tecniologías e integracióones que ofrecen, siendo lo único que cambia la forma de transmitirlo. La estructura de este es el descrito en el listing 1.

```
{
   "id": id,
   "src": src,
   "dst": dst,
   "method": method,
   "params": {
```

```
      "ts": x,
      "component": {
         "param1": a,
         "param2": b,
         .
         .
         .
         "paramN": N
      }
   }
}
```

Listing 1: Payload usado en los mensajes MQTT de Shelly, en formato JSON

Donde:

- **src** es el origen del mensaje.
- **dst** es el destino del mensaje.
- **method** es el método que se invoca con el mensaje. Por ejemplo: *Mqtt.GetStatus*, *Switch.Set*, etcétera).
- **params** son los parámetros necesarios del mensaje, que son:
  - **ts** es el *timestamp* de Unix en UTC (Tiempo Universal Coordinado, el estándar de tiempo por el cual se regulan los relojes del mundo).
  - **component** es el objeto del estado (con sus parámetros necesarios). La cadena 'component' no es el nombre que define este objeto, sino que es variable y puede ser un nombre solamente (wifi, cloud, bluetooth, etc) o un nombre con un identificador si hay varios ejemplares del componente usando *nombre:id*. Por ejemplo, como pueden haber varios botones, se puede usar *switch:0* o *switch:1* para los botones identificados por 0 y 1, respectivamente. Dentro de este objeto habrán tantos parámetros como sea necesario para el mensaje.

3.4.2. Tópicos - Principales

En este apartado analizaremos los tópicos más importantes usados en una integración con MQTT en un sistema con dispositivos Shelly.

1.- *shelly_id*/events/rpc

Este tópico notifica los eventos ocurridos en el dispositivo con id *shelly_id*. Un ejemplo de notificación en un sistema de estas características, sería el encendido/apagado de un dispositivo, que notificaría a quien estuviera interesado en conocer esta información (los suscriptores del tópico).

Para ejemplificar mejor cada situación, haremos unas aclaraciones sobre el payload de Shelly descrito anteriormente con respecto a las notificaciones de Shelly recibidas a través de este topic:

- **src** es el origen de la notificación (en este caso, un dispositivo).
- **dst** es el destino de la notificación (en este caso, un usuario).
- **method** es el método de la notificación. Los posibles métodos relativos a notificaciones son:
    - NotifyStatus: Notifica sobre un cambio en el estado del componente y lleva información sobre los cambios ocurridos.
    - NotifyFullStatus: Contiene la misma semántica que NotifyStatus, pero el payload contiene el estado completo de los componentes.
    - NotifyEvent: Este método notifica sobre un evento ocurrido que no está reflejado en el estado del dispositivo (por ejemplo, se ha pulsado un botón, se ha cambiado la configuración, ...).

Podemos observar un ejemplo de una notificación recibida por MQTT en un dispositivo Shelly en el listing 2. En esta, observamos que viene de un dispositivo con id *shellypro4pm-f008d1d8b8b8*, con tipo de notificación *NotifyStatus*, en el tiempo con timestamp UTC *1631186545.04*, y notifica el estado del componente *switch:0*. En concreto, se está notificando que el componente de tipo *button* (botón) tiene un parámetro *output* cuyo valor es *true* (verdadero) en el momento del envío de la notificación.

```
{
   "src": "shellypro4pm-f008d1d8b8b8",
   "method": "NotifyStatus",
   "params": {
      "ts": 1631186545.04,
      "switch:0": {
         "id": 0,
         "output": true,
         "source": "button"
      }
   }
}
```

Listing 2: Ejemplo de una notificación de Shelly recibida por MQTT en el tópico device_id/events/rpc.

2.- *shelly_id*/rpc

Tópico usado para ejecutar comandos en un dispositivo de Shelly con id *shelly_id* usando RPC a través de MQTT. Se usa el payload de Shelly de nuevo para esta tarea, con la novedad del uso del campo *id* para que el cliente sepa a qué mensaje se hace referencia al publicar la respuesta al comando. En este caso, en src se escribe un tópico al que el emisor del comando se debe haber suscrito previamente para recibir la respuesta al comando.

Un ejemplo de un mensaje de comando enviado a un dispositivo para configurarlo mediante este tópico sería el que podemos observar en el listing 3. En esta, observamos que el id del comando es *1*, con tipo de comando *Schedule.Create*. Es decir, se va a crear una tarea programada. Lo que va a hacer esta tarea y cuando lo va a hacer, lo determinan los valores que encontramos en *params*. El campo *timespec* indica cuando se va a realizar la tarea, y el campo *calls* indica el comando que realizará la tarea y sus parámetros necesarios en *params*. Este comando parece que encenderá el componente con id 0, invocando al método *Switch.Set* y poniendo el campo *on* a *true*.

```
{
  "id": 1,
```

```
  "src": "mynewtopic",
  "method": "Schedule.Create",
  "params": {
    "timespec": "0 0 22 * * MON",
    "calls": [{ "method": "Switch.Set", "params":{"id":0,"on":
   true} }]
  }
}
```

Listing 3: Ejemplo de una mensaje de comando (encender) enviado a un interruptor inteligente de Shelly por MQTT en el tópico device_id/events/rpc.

3.- device_id/online

Tópico que notifica el estado del dispositivo cuyo id es *device_id*. Solo se envía un mensaje cuyo contenido es *true* si el dispositivo está activo, o *false* (con Last Will Testament - LWT) si el dispositivo se ha forzado a desconectarse abruptamente.

3.4.3. Tópicos - Otros

Además de los tópicos principales descritos, los dispositivos de Shelly utilizan los siguientes otros tópicos.

Se suscribe a:

- **shellies/command:** se usa para mandar un comando 'broadcast' a todos los dispositivos.
- ***topic_prefix*/command:** otro tópico donde se reciven comandos, y donde *topic_prefix* es normalmente único por dispositivo.
- ***topic_prefix*/command/*component:id*:** donde componentes individuales de un dispositivo aceptan comandos específicos.

Publica en:

- **shellies/announce:** Información de un dispositivo Shelly en respuesta a un announce.

- ***topic_prefix*/status:** El estado completo de un dispositivo como respuesta a un status_update publicado en *.../command*.

- ***topic_prefix*/announce:** Información de un dispositivo Shelly en respuesta a un announce.

- ***topic_prefix*/status/*component:id*:** El estado completo actual del componente.

- ***topic_prefix*/error/*component:id*:** Mensaje de error informativo si un comando MQTT no ha podido ser procesado.

En los casos donde se use, si *topic_prefix* no está definido, se usará el identificador del dispositivo.

Por último, entre los comandos comúnmente aceptados por un dispositivo de Shelly encontramos:

- **status_update:** Para provocar que el componente publique su estado.

- **announce:** Para provocar que la información devuelta al invocar el comando *Shelly.GetDeviceInfo* se publique en los tópicos *announce*.

3.4.4. Conclusiones

Tras analizar el caso de uso de Shelly, podemos ver una de las vías que se pueden aplicar al usar MQTT. En este caso, el direccionamiento mediante los tópicos es más simple y sin una complejidad añadida, y los mensajes se envían mediante JSON recopilando toda la información necesaria. No se aplica el carácter ligero de MQTT, pero si que se conserva el carácter de su actuación en tiempo real y sus característicos desacoplamientos, añadiendo también la posibilidad de direccionar a nivel de categorías la información. No es quizás la aplicación más completa y aprovechadora del potencial del protocolo MQTT, pero es un buen acercamiento para dispositivos IoT de estas capacidades y descripciones.

Entre las ventajas de usar un sistema de estas especificaciones encontramos la libertad de agrupar la información por una categoría específica a la vez que existe la posibilidad de direccionar más finamente sobre un dispositivo concreto, que se pueda mantener la transmisión en

tiempo real de datos, y que los mensajes envíados en el payload se puedan extrapolar a otras tecnologías y unificar así los payloads de todas las posibles integraciones usadas en el sistema.

En cuanto a los inconvenientes, destacamos el alto tráfico de red generado por usar un payload más grande y la pérdida de la modularidad característica de MQTT (normalmente se direccionarían los tópicos de manera que en cada tópico se ubicara un solo valor, y no un payload completo con muchos valores).

3.5. Casos de uso reales - Tasmota

Como segundo caso de uso real de aplicación del protocolo MQTT, nos fijamos en Tasmota [28], firmware alternativo de código abierto desarrollado por Theo Arends (y otros contribuidores) para dispositivos basados en los microcontroladores ESP8266 y ESP32 que ofrece una fácil configuración, una interfaz web y control local completo mediante MQTT o HTTP, entre otras características.

3.5.1. Tópicos

Tasmota usa los tópicos de MQTT de una manera diferente e interesante. Utiliza tokens a modo de variables que se sustituyen dinámicamente en tiempo de ejecución para formar lo que llaman un *Full Topic* (Tópico Completo en español), que será el tópico final usado en MQTT. Los tokens disponibles para formar Full Topics son:

- **%prefix%** - Tasmota usa 3 posibles prefijos para formar Full Topics:
  - **cmnd:** Prefijo usado para ejecutar comandos y preguntar por estados.
  - **stat:** Prefijo usado para informar de estados.
  - **tele:** Prefijo usado para informar de telemetrías varias en intervalos especificados.
- **%topic%** - Todos los mensajes de estado se mandarán usando este token que debe ser único y designado por el usuario. Puede ser *bedroom* o *XP-TS_10* mientras que el usuario se acuerde de este y a qué está identificando. Por defecto, Tasmota usa como *%topic% tasmota_XXXXXX*, donde *XXXXXX* es único y viene de las últimas 6 cifras de la dirección MAC (eliminando los : separadores) del dispositivo al que se refiere. El uso de *%topic%*

permite el uso de las funcionalidades de *GroupTopic* y *FallbackTopic* que comentaremos a continuación.

- **%hostname %** - El nombre del host del dispositivo (mostrado en la interfaz web).
- **%id %** - Dirección MAC del dispositivo.

El orden de estos tokens no es importante, así que pueden combinarse de cualquier manera. Así, los siguientes serían ejemplos de Full Topics formados siguiendo las reglas descritas, que posteriormente serán resueltos en tópicos utilizables en MQTT:

- **%prefix %/ %topic %/** (el Full Topic por defecto del sistema en Tasmota)
- **tasmota/ %topic %/ %prefix %/**
- **tasmota/dormitorio/ %topic %/ %prefix %/**
- **casa/dormitorio1/baño2/ %topic %/ %prefix %/**
- **%prefix %/casa/azotea/ %topic %/**

En el sistema de tópicos de Tasmota, se definen las siguientes configuraciones especiales para la creación de tópicos:

- **Group Topic:** Se usa para agrupar tópicos que vayan dirigidos a un grupo de dispositivos. Si varios dispositivos hacen uso de un mismo *Group Topic* (Tópico Grupal), todos ellos recibirán los mensajes que se publiquen en este. Si se usa la opción por defecto, *tasmotas*, se publicará un mensaje que todos recibirán. Esto puede ser útil para órdenes más globales, como actualizar el firmware o, justamente para designar los grupos a los que irá cada dispositivo.
- **FallBack Topic:** Este tópico permite la comunicación MQTT con un dispositivo independientemente de los tópicos configurable. Es especialmente efectivo cuando se dan casos en el que varios dispositivos caigan en el mismo tópico configurable, y se quiera comunicar con ellos independientemente. Por defecto, el FallBack Topic de un dispositivo es *DVES_XXXXXX_fb*, donde *XXXXXX* de nuevo viene de las últimas 6 cifras de la dirección MAC del dispositivo al que se refiere.

- **LWT (Last Will Testament) Topic:** Tasmota hace uso de la característica de *Last Will Testament* (Testamento de Última Voluntad en español) a través de este tópico que se utiliza para generar una última notificación de un dispositivo que se ha desconectado o expirado. En concreto, el tópico tendrá la forma *tele/tasmota_XXXXXX/LWT* donde *XXXXXX* son las últimas 6 cifras de la dirección MAC del dispositivo al que se refiere.

3.5.2. Conclusiones

El sistema de tópicos del sistema de Tasmota es cuanto menos un sistema bastante interesante, que, al contrario que Shelly, vela por la completa caracterización de los tópicos. Este sistema nos permite formar tópicos de manera más libre y sin estructura fija, usando los Full Topics que son totalmente configurables, sin añadir más ambigüedad a la que ya podría existir.

Entre las ventajas que podemos encontrar de la forma de integración de MQTT que usa Tasmota destacamos la flexibilidad y libertad de diseño de los tópicos, el nivel de finura al que se puede alcanzar con los tópicos para generar un menor tráfico de red y el carácter de tiempo real característico de MQTT que se mantiene.

Por otra parte, las desventajas que podemos ver en este sistema son, entre otras, la complejidad de configuración inicial y de aprendizaje de la configuración y la posible sobrecarga de red dependiendo de la finura que se escoja en el diseño de los tópicos.

4

Caso de Uso

Para poder ver esta tecnología en acción, la siguiente parte del presente trabajo se enfocará en el desarrollo de un caso de uso en el que podremos observar la aplicabilidad real de MQTT con más facilidad. Este caso de uso se sitúa en un sistema de monitorización de diversos parámetros ambientales (la temperatura, la humedad y los niveles de CO2 y TVOC en el aire), donde, dependiendo del escenario concreto, la naturaleza de la comunicación entre dispositivos y sistemas puede ser crítica para el funcionamiento eficiente la aplicación. A lo largo de esta sección, examinaremos detenidamente el diseño, desarrollo e implementación de MQTT en este entorno, proporcionando una visión integral que abarca desde la toma de decisiones iniciales hasta la evaluación de resultados y las perspectivas futuras. Además, proporcionaremos una visión general de la estructura de nuestro caso de uso, incluyendo la arquitectura del sistema, los roles de los componentes clave y las consideraciones de diseño que han guiado la implementación.

4.1. Diseño de la Arquitectura Hardware

Para este diseño, se ha elegido un dispositivo Raspberry Pi 3B+, que actuará como broker MQTT encargado de gestionar los distintos tópicos y mensajes que se tramiten en el sistema.

El primer cliente físico, un ordenador particular, será el encargado suscribirse a los tópicos necesarios para recibir la información y mostrarla de una manera sencilla para el usuario. El segundo cliente físico y único dispositivo IoT del sistema, el microcontrolador ESP32, estará conectado a dos sensores: el CCS811, encargado de medir parámetros del aire (la cantidad de CO2, el tVOC, etc.); y el AM2302, encargado de medir la temperatura y la humedad en el ambiente.

En la Figura 2 podemos observar una representación realizada con un simulador de circuitos de la arquitectura hardware del dispositivo ESP32, y en la Figura 3 se muestra un esquemá-

tico de la arquitectura hardware del dispositivo ESP32. Ambas incluyen además los sensores y sus respectivas conexiones.

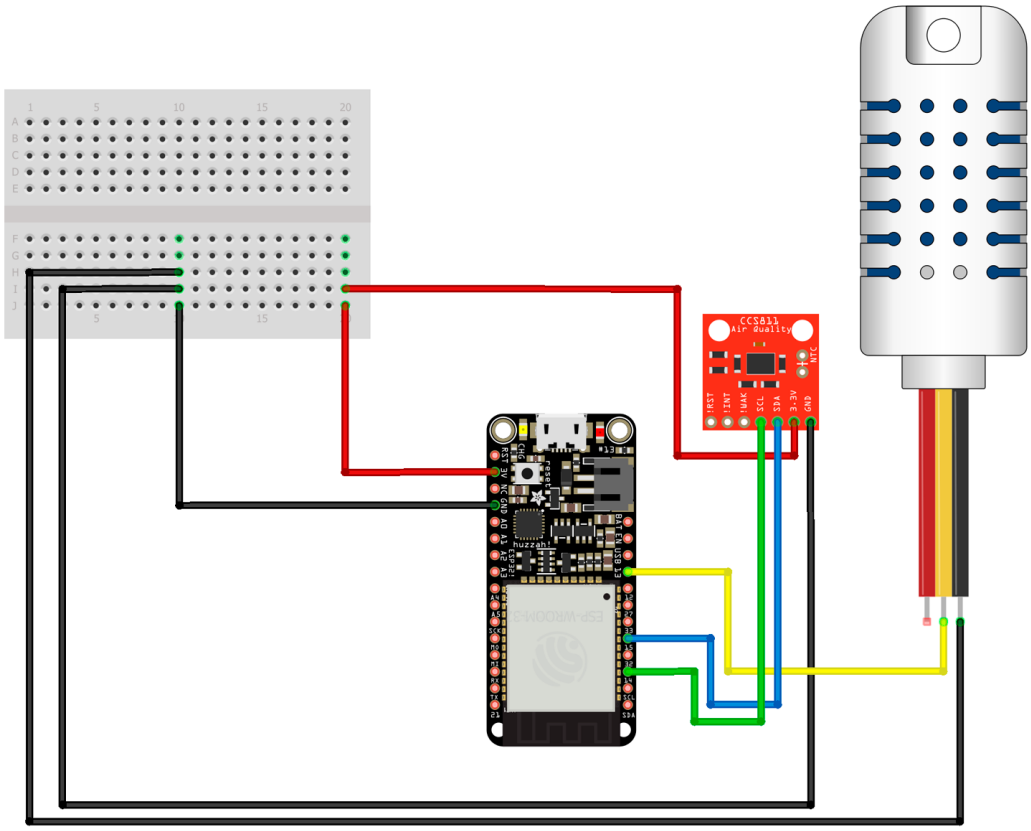

Figura 2: Representación de la arquitectura hardware del sistema IoT

4.2. Diseño de la Arquitectura Software

La arquitectura descrita se puede visualizar en el diagrama presentado en la Figura 4, y consta de las partes que se detallan a continuación.

El dispositivo Raspberry Pi 3B+ gestionará los distintos tópicos y mensajes a través de un contenedor virtual de Docker con la imagen del broker Mosquitto. Además, se pueden registrar usuarios en el broker, protegidos por la contraseña que elijan. Estos pares <usuario,contraseña>serán utilizados para autentificar la comunicación y se almacenan en el archi-

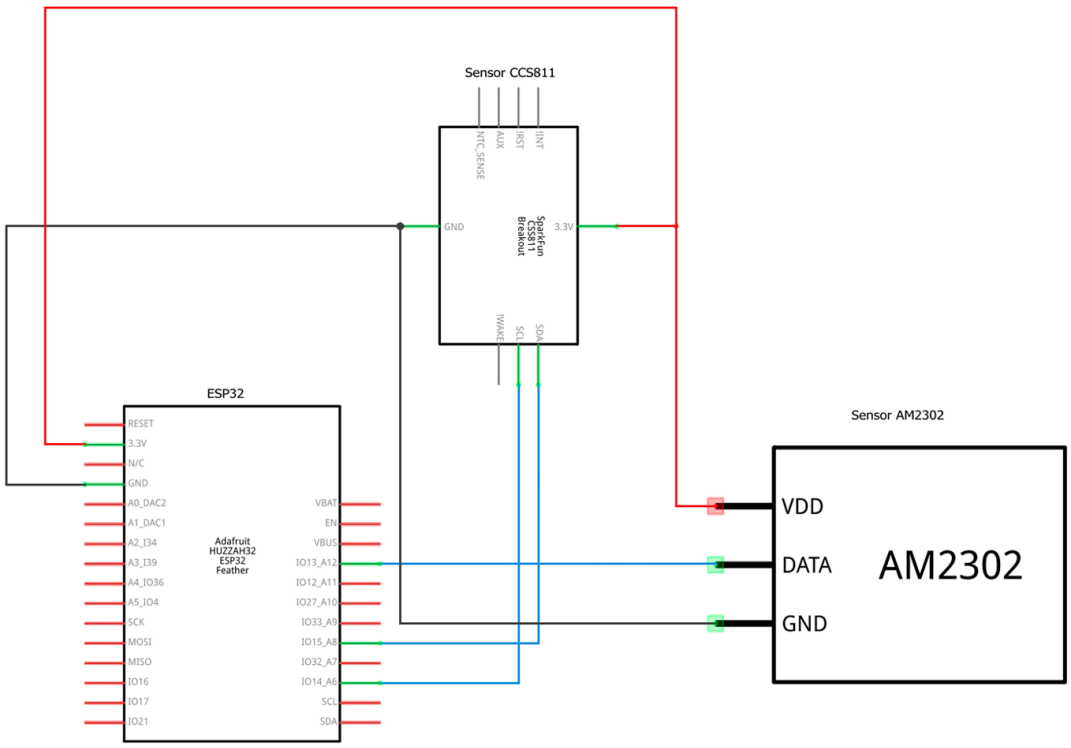

Figura 3: Esquemático de la arquitectura hardware del sistema IoT

vo *pwfile*. Cada dispositivo del sistema estará identificado con un identificador único, en este caso, su dirección MAC.

El dispositivo ESP32 ejecuta programas escritos en C/C++ o variantes como Arduino. En este caso, el lenguaje usado es el de Arduino por la facilidad de conexión al microcontrolador desde su software IDE dedicado y por la integración del sistema con librerías externas que pudieran ser necesarias para la programación y compilación del sistema. Los datos medidos por los sensores se pueden acceder gracias a dos librerías de Arduino. La primera de ellas es la librería de código libre *AM2302-Sensor* [7], creada y mantenida por el usuario de GitHub *ha-senradball*, que nos ayuda a obtener la información del sensor AM2320 de una manera legible. La segunda librería usada, *SparkFun_CCS811_Arduino_Library* [9], es otra librería de código abierto para poder leer la información del sensor CCS811 de Sparkfun, creada y mantenida por la propia empresa Sparkfun.

El otro cliente que añadiremos al sistema es un ordenador particular. En un grupo de contenedores de Docker, tendremos ejecutando los servicios Telegraf, InfluxDB 2 y Grafana en el

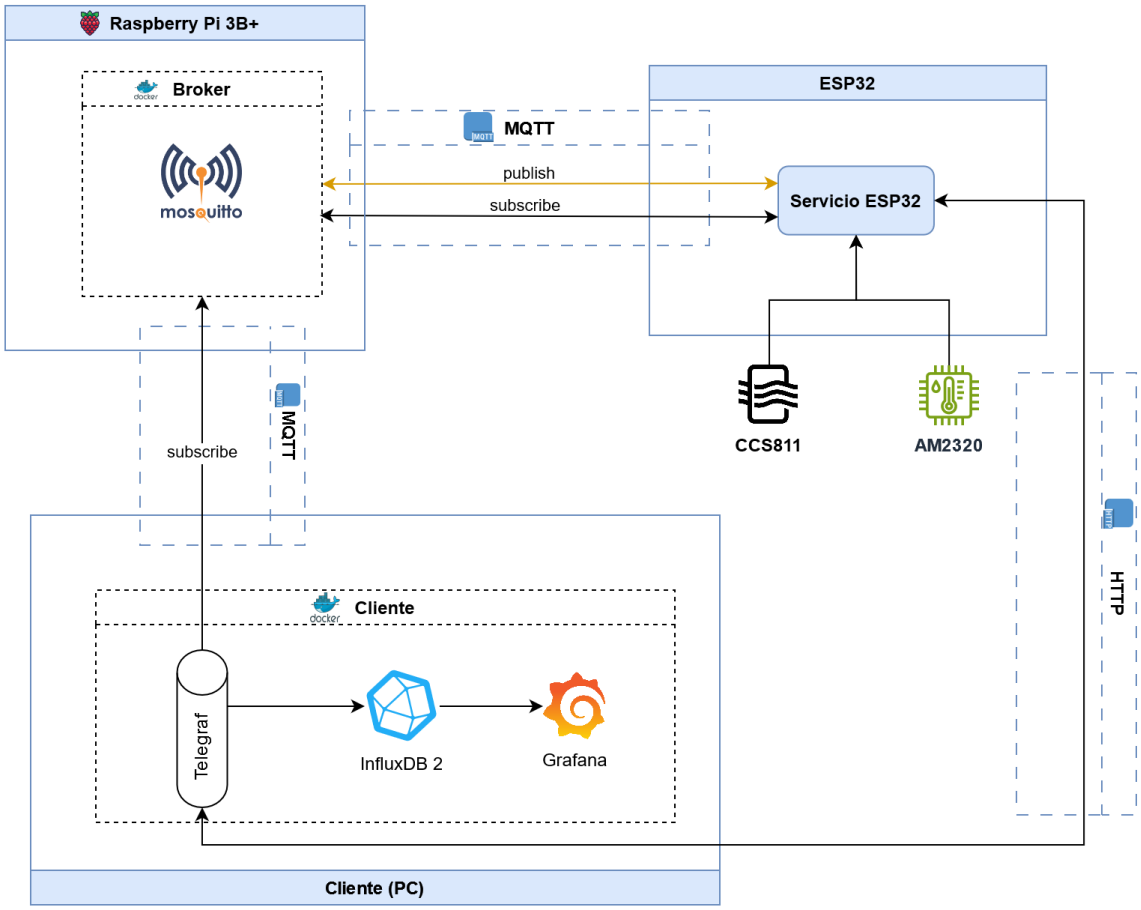

Figura 4: Diseño de la arquitectura software del sistema

cliente. Telegraf será el intermediario entre el emisor (el broker en caso de usar MQTT, o el propio dispositivo ESP32 en caso de usar HTTP) y el resto de servicios, publicando los datos en la base de datos de InfluxDB. Telegraf usa un servicio de plugins (complementos o módulos) para definir sus entradas y salidas. En este sistema se usarán las entradas MQTT y HTTP para obtener la información de los sensores a través del broker en la Raspberry y del servidor HTTP en el dispositivo ESP32, respectivamente; y los parámetros del procesador, memoria y red (del contenedor de Docker donde se encuentra Telegraf) mediante los plugins nativos de Telegraf disponibles para ello; y la salida InfluxDB mediante el plugin nativo de Telegraf dedicado para ello. Es decir, Telegraf agrupará todas las entradas que se le configuren y las distribuirá a todas las salidas especificadas.

A continuación tenemos InfluxDB, una base de datos para series temporales que se encar-

ga de persistir la información en el sistema y servir de repositorio de donde se leerá la información para mostrarla posteriormente. Por último, el servicio Grafana será el encargado de proporcionar una visualización de la información de una manera más estética y accesible mediante una serie de gráficas y visualizadores diversos. A través de la integración con InfluxDB que proporciona Grafana, se accederá a la información almacenada en InfluxDB por medio de *queries* (consultas) en lenguaje Flux, el nuevo lenguaje propio de consultas de InfluxDB, y la dispondrá en la página web haciendo uso de dos *dashboards* (paneles) de Grafana, uno para las métricas de la comunicación y del sistema, y otro para la información como tal de los sensores. Para los dashboards, se ha creado un dashboard personalizado para la información de los sensores, y se ha adaptado otro dashboard [29], creado por Jorge de la Cruz, para obtener las métricas del sistema (manteniendo algunas gráficas, descartando otras, y añadiendo algunas personalizadas como las relativas a la red).

4.3. Diseño de la Comunicación

Las comunicaciones del proyecto consistirán de un formato común para el contenido de los mensajes (el *Payload*), una serie de comandos aceptados por los dispositivos, y los canales (los *Topics*) designados para el intercambio de información. En general se mantendrá esta estructura tanto para MQTT como HTTP, siendo el canal de comunicación lo único que cambia y no el contenido en sí. El único caso donde serán diferentes es en el envío de comandos por HTTP, ya que se recibirán los comandos en forma de peticiones a unas URLs que especificaremos más adelante.

4.3.1. Payload

El primer punto a considerar en las comunicaciones de nuestro sistema será el formato de los mensajes que se van a enviar. Tras un estudio inicial de la información que se quiere transmitir, comprobando exhaustivamente todos los casos de intercambio de información y otros casos de uso reales en el mercado actual, el modelo elegido para el payload es el que se muestra en el Listing 4.

```
{
    "src": src,
```

```
    "method": method,
    "ts": x,
    "params": {
        "param1": a,
        "param2": b,
         .
         .
         .
         "paramN": N
    }
}
```

Listing 4: Diseño del Payload en formato JSON

Donde:

- **src** es el origen del mensaje. Si el origen es un dispositivo IoT, src será su id, mientras que si es el broker (el mensaje es, por tanto, una orden para un dispositivo), src será el nombre del topic donde se espera recibir la posible respuesta al comando.

- **method** es el método/motivo que se invoca con el mensaje.

    - Si es un mensaje de orden será **cmd**.
    - Si es una respuesta a una orden será **cmdinfo**.
    - Si es un evento que ha ocurrido en el sistema será **info**.

- **ts** es el *timestamp* de Unix en UTC (Tiempo Universal Coordinado, el estándar de tiempo por el cual se regulan los relojes del mundo).

- **params** son los parámetros necesarios del comando o de la respuesta. Si es un mensaje de orden (*method* es *cmd*), el primer parámetro será el comando principal a ejecutar, y el resto los parámetros adicionales del comando.

4.3.2. Comandos

También necesitaremos la posibilidad de enviar órdenes o solicitar información específica al dispositivo. Para eso, definiremos una lista de comandos aceptados por el dispositivo, que reaccionará ante ellos y actuará en consecuencia de la manera que definimos a continuación:

- **get:*[X, Y, ...]*** - Si el dispositivo ESP32 recibe un mensaje de comando de este tipo, este devolverá la información actual de los campos indicados en el comando exclusivamente. Por ejemplo, si se recibe el comando *get:[temperature, humidity]*, se devolverá el payload con los valores actuales de los parámetros *temperature* (temperatura en español) y *humidity* (humedad en español) solamente.

- **restart** - Si el dispositivo ESP32 recibe este comando, mandará un mensaje cuyo único parámetro será OK (acuse de recibo), y tras ello, este se reiniciará. Esta respuesta es una respuesta a modo de *Last Will Testament* (testamento de última voluntad en español), es decir, el último mensaje que manda un dispositivo antes de desconectarse (reiniciarse en este caso).

En el caso de HTTP, por cuestión de simplificar el funcionamiento, los comandos se mandan como peticiones GET diferentes a las rutas que detallaremos más a fondo posteriormente.

Debido a la simpleza del alcance del sistema, no se pueden tener muchos más comandos. Se podría extender el sistema tanto como se quisiera, y estos comandos nos servirían para ofrecernos una idea inicial de como podría ser en un sistema más grande.

4.3.3. Topics

Una vez diseñada la arquitectura general del sistema, otra cuestión esencial a acordar en un proyecto que usa MQTT es los tópicos donde se va a intercambiar la información entre los dispositivos. Se debe pensar cautelosamente cuál es la mejor implementación de los tópicos para mantener la eficiencia de acuerdo a las especificaciones y objetivos del sistema, y presentamos las dos estrategias principales y la conclusión a la que se ha llegado:

- La primera estrategia trataría de reducir el número de tópicos al máximo, reutilizándolos e identificando los mensajes y sus respectivas respuestas por medio de un identificador que almacenaría y gestionaría el broker.

- La segunda estrategia trataría de reducir la carga del broker y utilizaría tantos tópicos como fueran necesarios.

El pensamiento repentino podría ser que la primera estrategia es la más óptima. Pero, al analizarlo detenidamente, podemos deducir que la segunda estrategia es superior, no solo a corto plazo, sino que también de cara a que el sistema escale de la manera más eficiente posible. Es cierto que si en un sistema en el que se use MQTT y aplique esta estrategia se tienen una cantidad alta de dispositivos transmitiendo y recibiendo información, el broker tendría que estar suscrito a una cantidad considerable de tópicos. Pero si se realiza una definición robusta de los tópicos, en realidad no se tendría una cantidad tan grande de tópicos como para descartar esta estrategia, y a la vez se estaría manteniendo esa eficiencia y rapidez que caracteriza a MQTT, al contrario que en la primera estrategia.

Así pues, una vez elegida la estrategia apropiada, detallamos los tópicos que se utilizarán en el sistema:

1. **MAC** - Este tópico se utilizará para que el dispositivo ESP32 reciba los comandos previamente descritos, y actúe en consecuencia. *MAC* hace referencia a la dirección *MAC* del dispositivo.

2. **MAC/<*responsetopic*>** - Este es un tópico variable, ya que *responsetopic* se definirá al enviar el comando al ESP32, y será el campo *src* del mensaje de orden recibido, como hemos comentado previamente. En este se recibirán las correspondientes respuestas generadas tras enviar un comando al dispositivo. *MAC* hace referencia a la dirección *MAC* del dispositivo.

3. **events** - En este tópico se recibirán los eventos del sistema. Esto es, el reporte constante de los valores de los sensores.

En este caso de prueba solo tendremos un dispositivo IoT, el ESP32, y por eso obtenemos una cantidad bastante pequeña de tópicos. De nuevo, el sistema siempre se podría extender y designar nuevos tópicos según sea necesario.

4.3.4. Rutas del servidor HTTP

En el caso de usar HTTP como tecnología de comunicación en el sistema, las rutas del servidor HTTP serán las siguientes:

1. **GET /** *(raíz)* - En esta ruta se recibirán los eventos del sistema. Esto es, el reporte constante de los valores de los sensores. A diferencia de MQTT, no se recibirán los datos cada vez que se actualicen, sino que se recibirán cada vez que se soliciten (lo que se conoce como hacer *polling* al servidor).

2. **GET /field?params** - Este ruta se utilizará para que el dispositivo ESP32 reciba los comandos de solicitud de campos específicos, y envíe la respuesta al comando. En la misma ruta, tras enviar la petición, se recibirán las correspondientes respuestas. Los campos que se vayan a solicitar, serán especificados mediante parámetros URL, es decir, con el formato ?*parametro1=valor1&parametro2=valor2...*

3. **GET /restart** - Esta ruta se utilizará para enviar el comando *restart* al sistema. Al igual que en la ruta anterior, la respuesta se recibirá en la misma ruta tras enviar la petición.

4.4. Desarrollo del Caso de Uso

4.4.1. Broker MQTT en el dispositivo Raspberry

La primera parte con la que comenzaremos el desarrollo del proyecto consistirá en montar el broker en la Raspberry. Para ello, vamos a utilizar la imagen del contenedor virtual de Mosquitto para Docker. Se usará la imagen existente, creada por la fundación Eclipse, y por ello se definirá el contendor por medio de un archivo YAML *docker-compose* (en el archivo MQTT/SERVER/docker-compose.yml), que podemos observar en el Listing 5. En este definiremos el nombre (*mqtt5*), la imagen que se está usando, los puertos que se exponen en el contenedor (1883 y 9001, los puertos usuales en MQTT) y los archivos que persistirán fuera del contenedor en el cliente (las carpetas *config*, *data* y *log* del servicio de Mosquitto).

```
services:
  mqtt5:
    image: eclipse-mosquitto
```

```
    container_name: mqtt5
    ports:
      - "1883:1883"
      - "9001:9001"
    volumes:
      - ./config:/mosquitto/config:rw
      - ./data:/mosquitto/data:rw
      - ./log:/mosquitto/log:rw
    restart: unless-stopped

volumes:
  config:
  data:
  log:

networks:
  default:
    name: red-mqtt
```

Listing 5: Configuración del servicio Mosquitto en el archivo *docker-compose.yml*

Tras poner en marcha el contenedor a partir de esta configuración, ya podremos acceder al broker a través de la dirección IP de la Raspberry y alguno de los puertos definidos. Para poder probarlo, en cualquier otro cliente podemos instalar las herramientas de línea de comando (CLI) de Mosquitto. Entre las herramientas que se ofrecen, encontramos una herramienta (***mosquitto_pub***) para publicar mensajes a un tópico, y otra (***mosquitto_sub***) para suscribirte a un tópico. En ambas herramientas, se puede usar el flag **-h** para especificar el host (dispositivo) donde se encuentra el broker MQTT, en este caso la dirección IP de la Raspberry. Otros flags que usaremos son el flag **-t** para especificar el tópico al que publicar o suscribirse, **-m** para designar el mensaje a publicar (en el caso de mosquitto_pub), y **-u** y **-P** para especificar el usuario y contraseña de la comunicación MQTT con el broker. De esta manera, podemos ejecutar los siguientes comandos para publicar un mensaje y suscribirnos a un tópico, respec-

tivamente:

- **mosquitto_pub -h** *<direcciónIP>*-**t** *<direcciónMACRaspberry>* **-m** *<mensaje>* **-u** *<usuario>* **-P** *<contraseña>*
- **mosquitto_sub -h** *<direcciónIP>* **-t** *<direcciónMACRaspberry>* **-u** *<usuario>* **-P** *<contraseña>*

4.4.2. Servicio en el dispositivo ESP32

A continuación describiremos el desarrollo del servicio que se ejecutará en el microcontrolador ESP32. La implementación del servicio está realizada en Arduino, por lo que consiste de un programa principal, main.ino, y las librerías personales correspondiente. En el programa principal tenemos dos etiquetas (funciones) características de los programas de Arduino: *setup* y *loop*. En la función *setup* inicializamos todo el sistema, y en la función *loop* se ejecuta el programa en sí en bucle. Además, se declaran en el propio programa otras funciones auxiliares tanto para las comunicaciones en MQTT o HTTP, como para el funcionamiento general del programa. Dado que todo se va a ejecutar desde este programa, y no vamos a tener un programa separado para distinguir entre HTTP y MQTT, tendremos una variable booleana de control *HTTP_MQTT* cuyo valor será 0 si se está usando HTTP, o 1 si se usa MQTT. Para la conexión por MQTT usaremos la librería *PubSubClient*, desarrollada por Nick O'Leary; y para el servidor de HTTP usaremos la librería de Arduino, *WebServer*, ambas incluidas con Arduino IDE.

En las variables del programa declaramos las variables necesarias, tanto los parámetros para las comunicaciones/conexiones, como las variables necesarias para el correcto funcionamiento del programa completo. En la función *setup*, inicializamos el monitor serial (para poder mandar mensajes de debug al terminal por medio de la conexión serial), inicializamos la conexión Wi-Fi con el router al que lo queramos conectar, inicializamos el cliente MQTT con los detalles de la conexión al broker, inicializamos los parámetros para ejecutar el servidor web, se define la función *callback* (retrollamada) que se ejecutará cuando se reciba un mensaje en alguno de los tópicos a los que está suscrito el dispositivo ESP32, se especifican los *handlers* (manejadores) para cada ruta definida en el servidor HTTP, y se inicializan los objetos de los sensores usados por el microcontrolador. La implementación de la función *setup* la podemos

ver en el Listing 6.

```
void setup()
{
    // Inicialización de la comunicación serie
    Serial.begin(115200);

    // Inicialización de la comunicación I2C
    Wire.begin();

    // Inicialización de la comunicación WiFi
    setupWifi();
    configTime(0, 0, ntpServer);

    // Configuración del topic MQTT donde recibir comandos (dirección
    MAC del dispositivo)
    topicCmd = client_id.c_str();

    // Si se ha configurado el uso de MQTT
    if (HTTP_MQTT)
    {
        // Conexion con el broker MQTT
        client.setServer(mqtt_broker, mqtt_port);
        // Callback para recibir mensajes
        client.setCallback(callback);
    }
    else
    { // Si se ha configurado el uso de HTTP
        // Inicialización del servidor HTTP
        startServer();
    }

    // Inicialización de los sensores
    am2302.begin();
```

```
    auto status = am2302.read();
    debug("\n\nEstado del sensor AM2302: ");
    debugln(status);

    if (!ccs811.begin())
    {
        debugln("Error al cargar el sensor CCS811. Comprueba las
   conexiones.");
    }

    // Esperar a que el sensor esté listo
    while (!ccs811.dataAvailable())
        ;
}
```

Listing 6: Implementación del método *setup*

En la función loop, se comprueba el estado de la conexión MQTT, y en caso de que ocurra cualquier error en esta, se fuerza la reconexión con el broker. En el caso del servidor HTTP se llama a la función *handleClient* que maneja a las conexiones entrantes y las reubica en el manejador correspondiente a la ruta accedida. Cada 10 segundos se obtiene la información actualizada de los sensores, y se publica en el tópico de notificaciones, *events*. En el caso de HTTP, solo se actualizará la información y se dispondrá la última información guardada cada vez que se solicite. La implementación de la función se dispone en el Listing 7.

```
void loop()
{
    // Si se ha configurado el uso de MQTT
    if (HTTP_MQTT) {
        // Si no se ha establecido la conexión con el broker MQTT o
   se ha perdido
        if (!client.connected())
        {
            // Intentar reconectar
```

```
            reconnect();
        }

        // Mantener la conexión con el broker MQTT
        client.loop();
    } else { // Si se ha configurado el uso de HTTP
        // Mantener la conexión con el servidor HTTP
        server.handleClient();
    }

    // Si han pasado más de 10 segundos desde el último mensaje
    unsigned long now = millis();
    if (now - lastMsg > 10000) {
        lastMsg = now;

        // Si hay datos disponibles en el sensor CCS811
        // Se leen los resultados del sensor CCS811 solo,
        // porque el sensor AM2302 no dispondrá de nuevos
        // datos hasta que se lea de nuevo.
        if (ccs811.dataAvailable()) {

            // Leer los resultados del algoritmo del sensor CCS811
            ccs811.readAlgorithmResults();

            // Leer los datos de los sensores
            temp = roundf(am2302.get_Temperature() * 100) / 100;
            humidity = roundf(am2302.get_Humidity() * 100) / 100;
            co2 = ccs811.getCO2();
            tvoc = ccs811.getTVOC();

            // Mostrar los datos de los sensores en el monitor serie
            debugSensors();
```

```
            // Crear el mensaje con los datos de los sensores
            msg = createInfoPayload(client_id, temp, humidity, co2,
    tvoc);
            const char *msgchar = msg.c_str();

            // Si se ha configurado el uso de MQTT
            if (HTTP_MQTT) {
                debug("MQTT - Publish message: ");
                debugln(msgchar);

                // Publicar el mensaje en el topic MQTT
                client.publish(topic, msgchar);
            }
        }
    }
}
```

Listing 7: Implementación del método *loop*

Además del archivo principal *main.ino*, también se ha implementado una librería adicional *utils.h*, definida en el archivo *utils.cpp*. En esta librería se implementan funciones para leer y escribir la información en formato JSON, para su posterior tratamiento o publicación de una manera uniforme y estandarizada en todo el sistema, además de otras funciones auxiliares útiles para el correcto funcionamiento de la librería. En el Listing 8 podemos observar uno de los métodos implementados en esta librería, el método *genPayload*, encargado de generar el string del payload en formato JSON, recibiendo como parámetros el ID del cliente (su dirección MAC), el tipo de método, y los parámetros.

```
String genPayload(String client, String method, JSONVar params)
{
    // Se crea el objeto JSON (JSONVar)
    JSONVar obj;

    // Se rellenan los campos que tenemos
    obj["src"] = client;
```

```
    obj["method"] = method;
    obj["ts"] = getTime();

    // Si se ha recibido un objeto JSONVar con los parámetros no
   vacío,
    // se rellena también.
    if(params.keys().length() > 0){
            obj["params"] = params;
    }

    debugln(obj);

    // Se devuelve el objeto JSON como un String
    return JSON.stringify(obj);
}
```

Listing 8: Implementación del método *genPayload* de la librería *utils*

El programa se ejecuta directamente desde Arduino IDE. Para ello, previamente hemos tenido que instalar el soporte para el microcontrolador ESP32 Feather de Adafruit, el dispositivo usado. Al ejecutarlo, se compilará y flasheará el programa en el dispositivo vía conexión serial. A partir de este momento, el programa se ejecutará en bucle de manera indefinida y continuará leyendo la información y publicándola en el caso de usar MQTT.

4.4.3. Servicio en el cliente PC

En el cliente (que será un ordenador particular) hemos creado un servicio, de nuevo en un contenedor virtual de Docker, ejecutando los servicios Telegraf, InfluxDB y Grafana. Para ello, hemos usado también un archivo YAML *docker-compose*. En él especificamos las imágenes de Docker correspondientes, los puertos a exponer, los volúmenes persistidos en el sistema local y las variables de entorno necesarias para la configuración de cada servicio.

En primer lugar, con respecto a la configuración de InfluxDB, se expondrá el puerto 8086. Este servicio genera una plataforma web en local vía dicho puerto desde el que podemos configurar el sistema, así como monitorear los datos en tiempo real y realizar *queries*, entre otras

funcionalidades. La información persistida de manera local en este servicio serán los archivos de configuración ubicados en la carpeta */etc/influxdb2*, y la base de datos en sí ubicada en la carpeta */var/lib/influxdb2*. Además, mediante variables de entorno, le especificaremos al servicio algunos datos como el usuario, contraseña y organización iniciales, y el nombre del *bucket* (cubo en español, es decir, la tabla de datos) sobre el que trabajaremos. La configuración descrita la podemos ver en el Listing 9.

```
influxdb:
  image: influxdb:latest
  ports:
    - 8086:8086
  volumes:
    - ./influxdb/influxdb-data:/var/lib/influxdb2
    - ./influxdb/influxdb-config:/etc/influxdb2
  environment:
    - DOCKER_INFLUXDB_INIT_MODE=setup
    - DOCKER_INFLUXDB_INIT_USERNAME=cliente
    - DOCKER_INFLUXDB_INIT_PASSWORD=testuma1
    - DOCKER_INFLUXDB_INIT_ORG=uma
    - DOCKER_INFLUXDB_INIT_BUCKET=esp32_db
```

Listing 9: Configuración del servicio InfluxDB en el archivo *docker-compose.yml*

En el caso de Telegraf, persistiremos localmente el archivo de configuración *telegraf.conf* ubicado en la carpeta */etc/telegraf*, en el que se especifica el comportamiento del servicio. Específicamente aquí se definen las entradas y salidas del servicio, que en nuestro caso serán MQTT y HTTP como entradas, e InfluxDB como salida. En el caso de la entrada de MQTT, se especifica que los tópicos donde se recibe la información son *events* y *30:AE:A4:28:3B:5/responseTopic*, además del broker MQTT, que será una URL con el formato *tcp://<direcciónIP-Raspberry>:1883*. En cuanto a la entrada de HTTP, tan solo tendremos que especificar las posibles rutas del servidor de donde se recopilará la información. En este caso usaremos la ruta raíz, donde se recibe toda la información (*http://<direcciónIP-ESP32>:80/*), y la ruta de field, usando como ejemplo el campo temperatura y el campo co2 con las siguientes URLS: *<direcciónIP-*

ESP32>:80/field?temperature=1 y *http://<direcciónIP-ESP32>/field?co2=1*. En la configuración de la salida por InfluxDB especificamos el *bucket*, la organización que hayamos especificado al crear el servicio de InfluxDB, el token de acceso generado en el servicio de InfluxDB, y la URL local del servicio dentro del grupo de contenedores de Docker. Por último, hay que especificar que este servicio depende del servicio de influxdb. Podemos ver la configuración en el Listing 10.

```
telegraf:
  image: telegraf:latest
  volumes:
    - ./telegraf/telegraf.conf:/etc/telegraf/telegraf.conf
  depends_on:
    - influxdb
```

Listing 10: Configuración del servicio Telegraf en el archivo *docker-compose.yml*

En última instancia, tenemos el servicio de Grafana. En este servicio expondremos el puerto 3000, y persistiremos la carpeta */etc/grafana/provisioning*, en la que encontramos los archivos de los *dashboards* y *datasources* (orígenes de datos en español, siendo estas las conexiones desde las cuales se recibe la información en Grafana), y la carpeta */var/lib/grafana*, en la que se encuentra almacenada la información del sistema general y los plugins instalados. También le especificaremos que siempre se reinicie el contenedor si ocurre algo (en el caso de detener el servicio manualmente, no se iniciará automáticamente, incluso al reiniciar Docker). Por último, indicaremos que las dependencias de este servicio son los dos anteriores, *influxdb* y *telegraf*. Observamos la configuración descrita en el Listing 11.

```
grafana:
  image: grafana/grafana-enterprise
  restart: unless-stopped
  ports:
    - '3000:3000'
  volumes:
    - ./grafana/grafana-provisioning:/etc/grafana/
provisioning
```

```
    - ./grafana/grafana-data:/var/lib/grafana
  depends_on:
    - influxdb
    - telegraf
```

Listing 11: Configuración del servicio Telegraf en el archivo *docker-compose.yml*

4.4.4. Prueba de flujo completo

Una vez definido el sistema y su correspondiente configuración, realizaremos una prueba de flujo completo. El orden de ejecución no es relevante, pues son servicios independientes, aunque en el servicio del cliente no habrán datos si no está el servicio de ESP32 en ejecución, y los mensajes publicados mediante MQTT por el servicio de ESP32 tampoco llegarán a ningún lugar si no se está ejecutando el servicio del broker en la Raspberry.

En primer lugar, iniciaremos el servicio del broker MQTT en la Raspberry. Para ello, primero necesitamos tener instalado un sistema operativo en el dispositivo (en nuestro caso, Raspberry Pi OS, basado en Linux). Luego, tendremos que haber instalado Docker en el sistema operativo, refiriéndonos para ello al manual de instalación oficial de Docker [30] (en este caso, seguimos el manual de instalación para Debian puesto que estamos usando Raspberry Pi OS de 64 bits). A continuación, copiamos la carpeta MQTT y su contenido en el dispositivo Raspberry (mediante GitHub o un pendrive USB), y desde la terminal nos situamos en la carpeta SERVER dentro de la carpeta MQTT (si usamos Linux y copiamos la carpeta en *HOME*, haremos ***cd MQTT*** y ***cd SERVER***).

Una vez situados en la carpeta desde el terminal, ejecutaremos el contenedor de docker mediante el comando ***docker compose up -d***, sirviendo el flag -d para ejecutar el contenedor en modo desacoplado (correrá en segundo plano). Si queremos ver el estado de el container, podemos ejecutar el comando ***docker ps***, lo que nos da una información detallada de todos los contenedores existentes en el sistema. Tras haber comprobado que el contenedor está ejecutándose (en la columna *Status* dirá *Running*), podemos comprobar el funcionamiento accediendo a la terminal del contenedor con ***docker exec -it mqtt5 sh*** (siendo *exec* el comando para ejecutar un comando en un contenedor, *mqtt5* el nombre del contenedor, *sh* el comando a ejecutar y *-it* los flags que indican que la terminal quedará activa en primer plano, y que

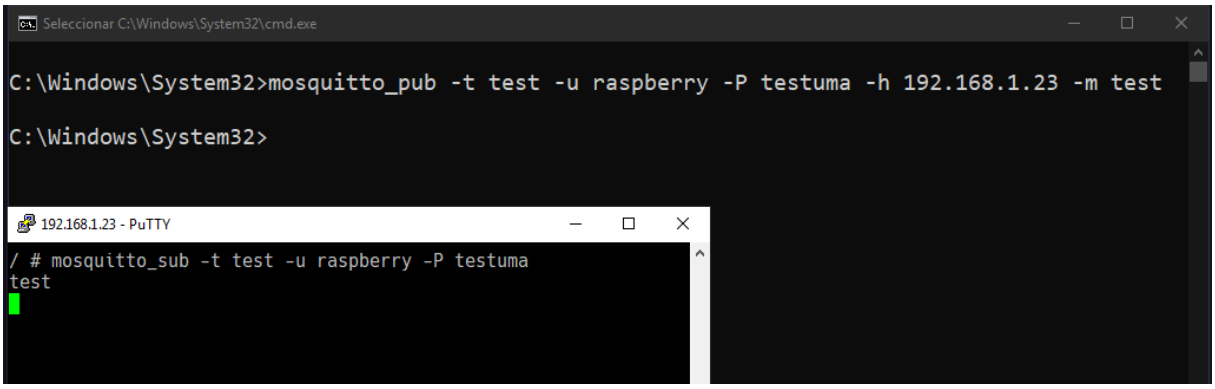

Figura 5: Prueba de publicación de mensaje de prueba desde el ordenador (arriba) hasta la Raspberry (abajo)

será un pseudoterminal) y suscribiéndo el dispositivo a cualquier tópico, por ejemplo el tópico *test*. Para suscribirnos al tópico, usaremos la herramienta CLI (de línea de comando) de Mosquitto, *mosquitto_sub*, con el comando ***mosquitto_sub -t test -u raspberry -P testuma***, donde el flag *-t test* especifica el tópico al cual suscribirse y los flags *-u raspberry -P testuma* especifican el usuario y contraseña, respectivamente. Desde otro dispositivo (por ejemplo el ordenador particular) instalaremos las herramientas CLI de Mosquitto. Una vez instaladas, haremos uso de la herramienta *mosquitto_pub* con el comando ***mosquitto_pub -t test -m test -h <direcciónIP-Raspberry>-u raspberry -P testuma***, donde los flags coinciden con los del comando ejecutado en la Raspberry, a excepción del flag *-m test* usado para indicar el mensaje a publicar en el tópico test y el flag *-h <direcciónIP-Raspberry>* para indicar el broker MQTT. Tras ejecutar este comando, podemos observar la terminal en el contenedor de MQTT en la Raspberry, y si todo ha ido correctamente, veremos que se ha recibido el mensaje *test*. Podemos ver el resultado de la prueba en la Figura 5.

El siguiente paso será ejecutar el servicio del cliente. De igual manera que en el paso anterior, nos ubicaremos en la carpeta cliente y ejecutaremos el comando ***docker compose up -d***. Antes de esto, configuraremos las entradas de Telegraf previamente descritas, con los datos que nos corresponda (las direcciones IPs correspondientes y demás configuraciones variables anteriormente mencionadas). Tras haber configurado y ejecutado los contenedores, deberíamos de tener los tres contenedores corriendo. Para comprobar que InfluxDB funciona correctamente, podemos abrir el navegador y abrir la URL *http://localhost:8086/*, donde nos aparecerá un menú de inicio de sesión del servicio web local de InfluxDB, como el que podemos observar en la Fi-

gura 6. Para comprobar el servicio de Grafana, podemos acceder a la URL *http://localhost:3000/*, la cual nos recibirá con la pantalla principal que podemos ver en la Figura 7.

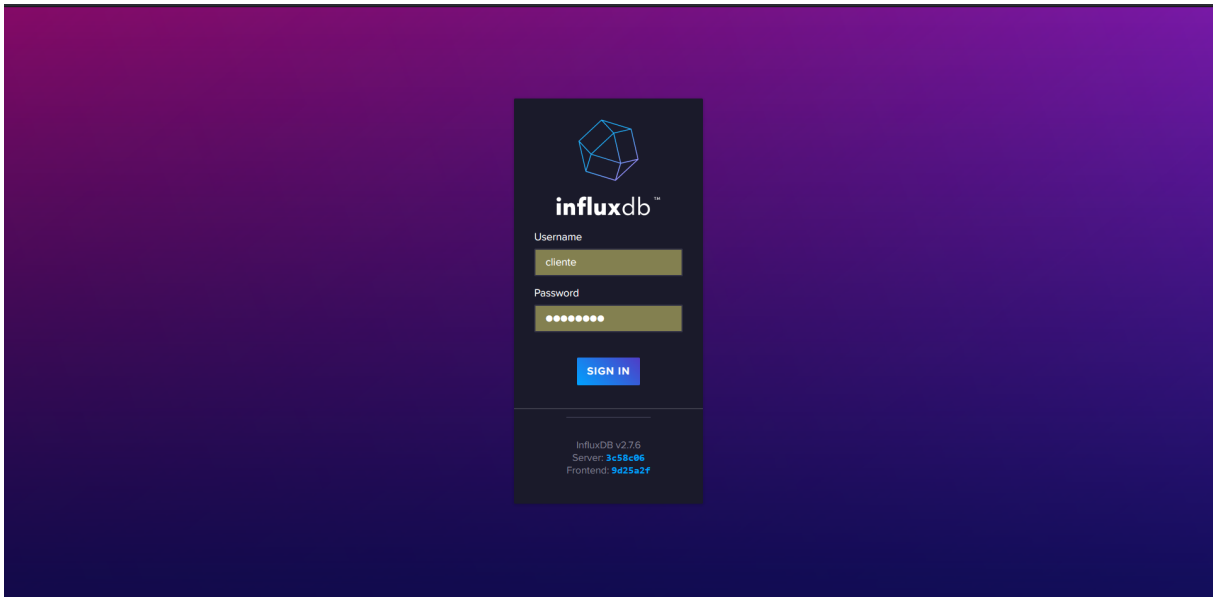

Figura 6: Menú de inicio de sesión en el servicio web local de InfluxDB

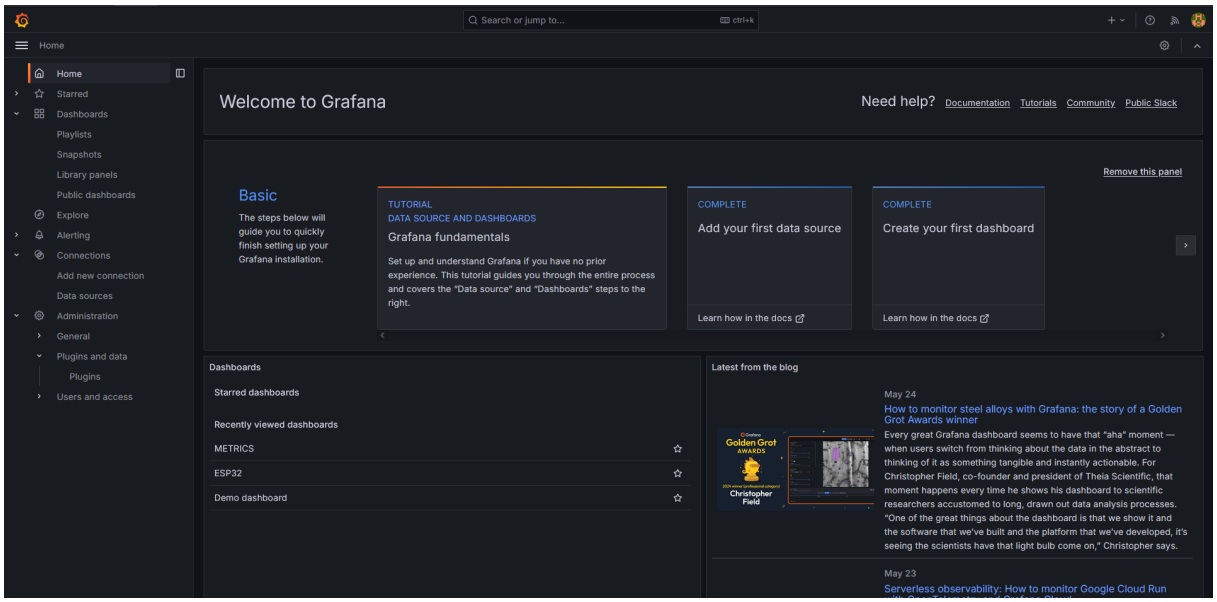

Figura 7: Página inicial del servicio web local de Grafana

Por último, ejecutaremos el servicio del ESP32. Necesitaremos el programa Arduino IDE, gratuito y descargable a través de la página oficial de Arduino [31]. Una vez instalado, tendremos que realizar una serie de configuraciones para poder compilar y subir el programa al dispo-

sitivo. Primero, tenemos que hacer que el software reconozca al dispositivo. Para ello, vamos al menú contextual superior del programa, hacemos click en File/Archivo ->Preferences/Preferencias. En la ventana que se abrirá, en el campo de texto que dice Additional Boards Manager URLs (URLs de Manejadores de Dispositivos Adicionales en español), introduciremos el siguiente enlace: https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json. Esto activará la posibilidad de instalar manejadores de dispositivos a los que Arduino no da soporte nativamente, como es el caso del nuestro. A continuación, para instalar el soporte de nuestro dispositivo, vamos de nuevo al menú contextual y hacemos click en Tools/Herramientas ->Board/Dispositivo ->Board Manager/Manejador de Dispositivo. Esto abrirá otro menú, esta vez con una barra de búsqueda. En la barra de búsqueda introducimos *esp32* y hacemos click en Install/Instalar en el único resultado que deberíamos obtener. Ahora, ya podremos seleccionar nuestro dispositivo en el IDE, yéndonos al menú superior y haciendo click en Tools/Herramientas ->Board/Dispositivo ->Adafruit ESP32 Feather. Tras haber seleccionado nuestro dispositivo, ya podemos abrir el proyecto en el programa, elegir el puerto y el dispositivo (Adafruit ESP32 Feather) en la parte superior, verificar el código, y cargarlo en el dispositivo. En el monitor del serial integrado en el programa podremos observar la salida del programa ejecutándose en el dispositivo.

Con los tres servicios ejecutándose, en los dos paneles de Grafana empezaremos a ver la información llegando y graficándose en su apartado correspondiente de manera similar a las imágenes de las figuras 8 y 9 en el caso de los datos de los sensores, y las figuras 10 y 11 en el caso de las métricas obtenidas durante el procesamiento y la comunicación de los datos. También podemos ver la información almacenada en tiempo real en el menú web de InfluxDB (localhost:8086), accediendo al apartado de *buckets*, y ejecutando una query Flux sobre el *bucket esp32_db*.

4.5. Métricas obtenidas y comparativa con HTTP

En este apartado observaremos las métricas que se obtienen en una comunicación normal en nuestro sistema. Cabe comentar que Telegraf hace una comunicación exhaustiva a partir de sus entradas definidas en el archivo de configuración. Es decir, en el caso de MQTT se suscribirá a todos los tópicos definidos, y en el de HTTP hará polling de todas las rutas definidas, y posteriormente unirá todos los datos en las salidas correspondientes. Para obtener las mé-

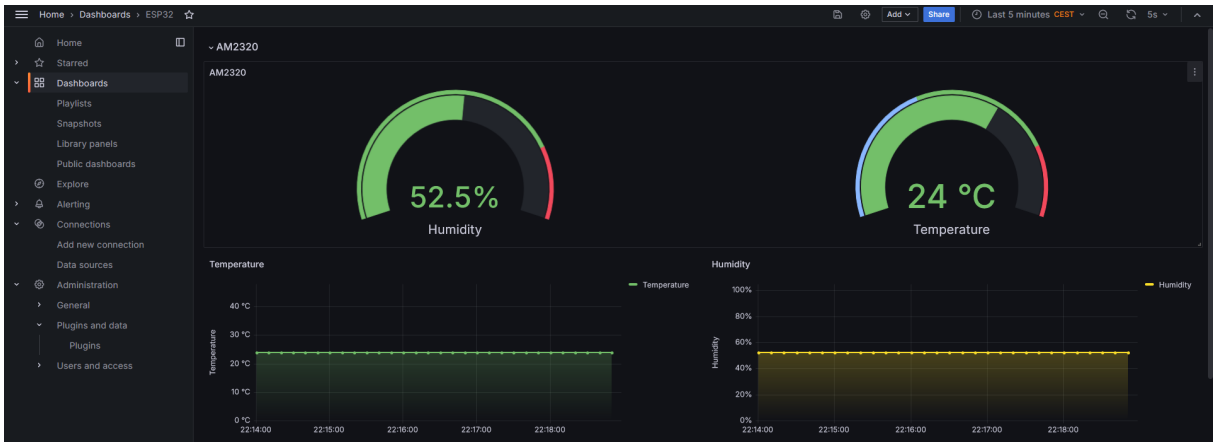

Figura 8: Vista de las gráficas a partir de los datos del sensor AM2320 en el panel de Grafana *ESP32*

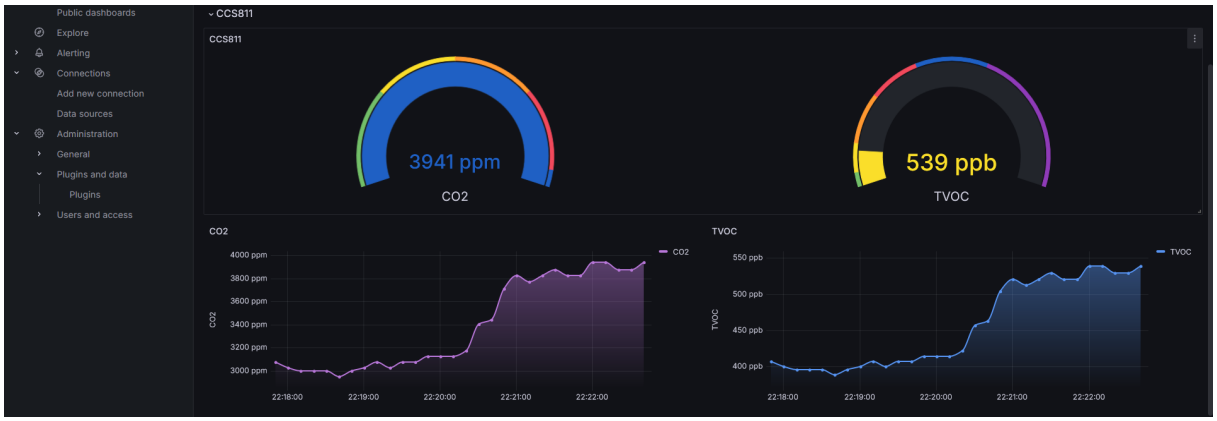

Figura 9: Vista de las gráficas a partir de los datos del sensor CCS811 en el panel de Grafana *ESP32*

tricas, haremos una prueba de flujo como la realizada en el apartado 4.4.4, y analizaremos los datos obtenidos en un período de tiempo de una hora. Tanto para los datos de los sensores, como para las métricas del sistema, los datos se obtienen de InfluxDB mediante el conector nativo de Grafana con InfluxDB (pre-configurado por la persistencia local a través de los archivos proporcionados) por queries Flux contra el bucket *esp32_db* como veremos un poco más adelante.

Las métricas que analizaremos serán las siguientes:

- **Métricas de red:** El RTT (Round Trip Time, o Tiempo de Ida y Vuelta en español), cuyo valor representa el tiempo que tarda una señal en enviarse de un punto a otro, más el tiempo que tarda en llegar de vuelta al emisor la confirmación de recibo de la señal

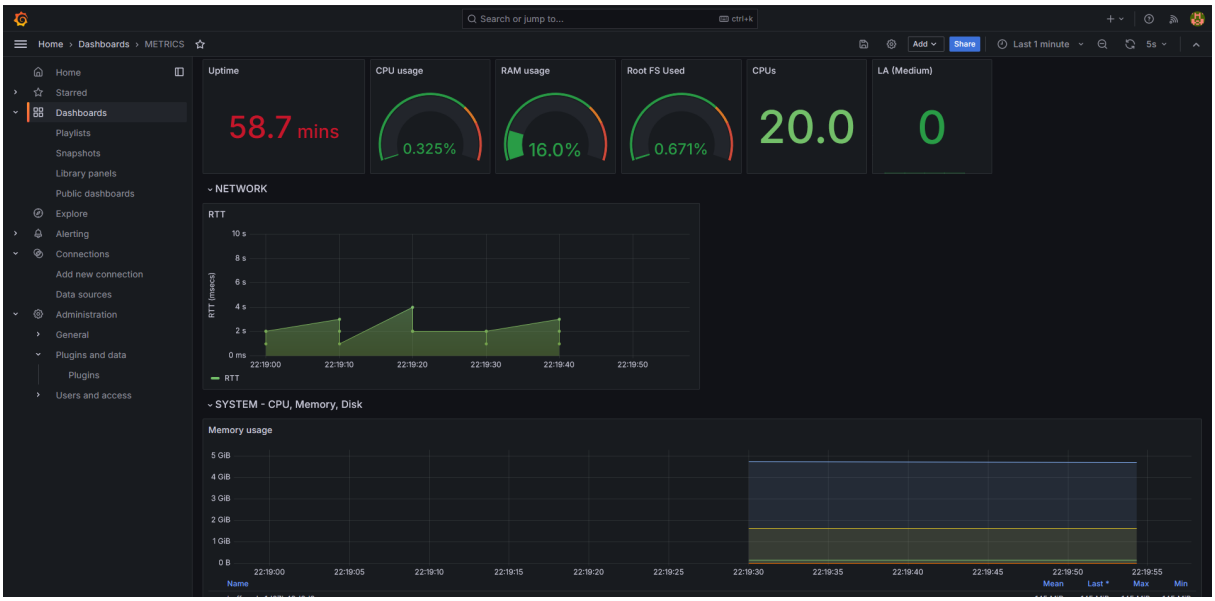

Figura 10: Vista 1 del panel de Grafana *METRICS*

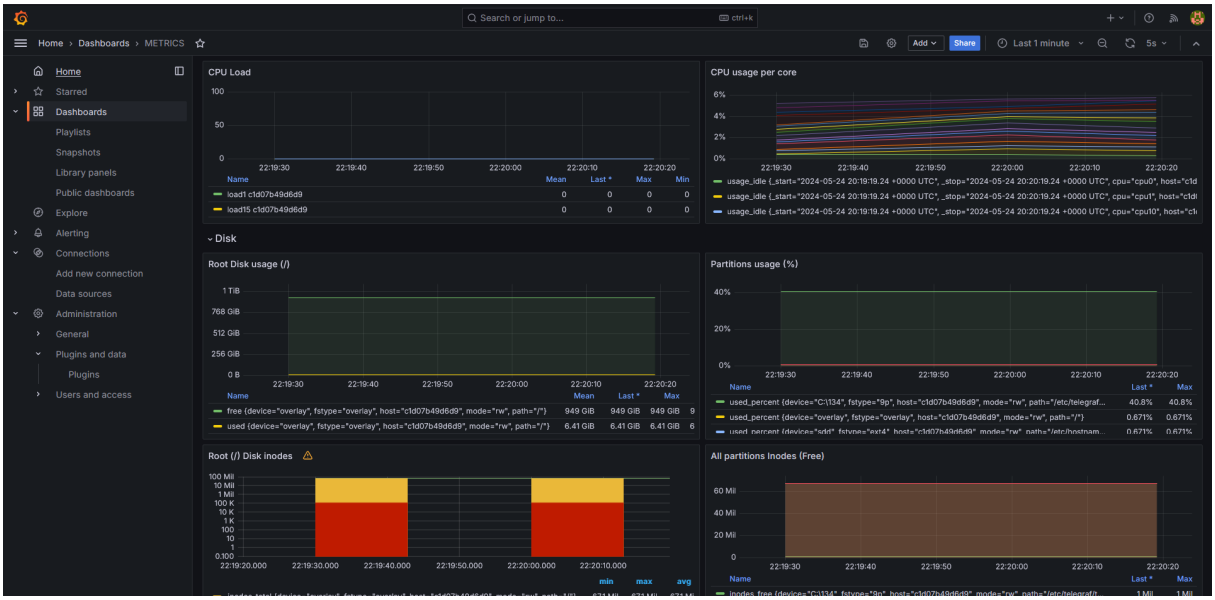

Figura 11: Vista 2 del panel de Grafana *METRICS*

enviada. En el Listing 12 podemos observar como se ha calculado el RTT en nuestro sistema mediante una query Flux. Este cálculo se realiza a partir del campo *ts* del mensaje, que es el tiempo UTC en el que el mensaje sale del ESP32, y el campo *_time* que InfluxDB genera para cada mensaje, y que simboliza el tiempo en el que el mensaje está correctamente insertado en el sistema (posterior a la confirmación de recibo de este inclusive). Con estos datos, solo tendremos que ajustar las unidades y restar el último

valor al primero para obtener el resultado del RTT del mensaje.

- **Métricas de sistema y disco:** En esta sección disponemos de valores relacionados con la CPU, la memoria y el disco del contenedor virtual de Telegraf. Los valores representados en este apartado se recopilan de manera automática por la entrada nativa de Telegraf que se ha especificado previamente en el archivo de configuración del servicio. Concretamente, las métricas usadas referentes a esta sección son el uso de memoria, el uso (total y por núcleos) y la carga de CPU, el número de inodos y el uso del disco root(/), el porcentaje de uso de cada partición de disco y el número total de inodos libres en cada partición.

```
from(bucket: "esp32_db")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_field"] == "ts") // Filtramos por el
    campo "ts"
  |> map(fn: (r) => {
      timeInSeconds = float(v: uint(v: r._time)) / float(v: "1e
    +9")
      tsInSeconds = float(v: r._value)

      // Calculamos el RTT y transformarlo a milisegundos
      rttInSeconds = (timeInSeconds - tsInSeconds) * 1000.0

      // Devuelve el nuevo registro con las columnas
      // RTT, el _time original, y ts
      return {
          _time: r._time,   // Mantener _time original
          mqtttime: tsInSeconds,   // Mantener ts original
          systime: timeInSeconds, // _time original en segundos
          _value: rttInSeconds   // RTT en segundos
      }
  })
```

```
  |> yield(name: "result")
```

Listing 12: Query Flux para obtener el RTT de un mensaje en InfluxDB

En cuanto a los datos de los sensores, basta con hacer una query Flux contra la base de datos especificandole el rango de tiempo deseado, filtrando por el campo que queremos obtener y agregándolo de manera que se calcule la media de los valores en una ventana temporal de 1 hora. Podemos ver el ejemplo de una query para obtener la media de la temperatura en el Listing 13, siendo este modelo de queries el usado para todos los parámetros del sistema, teniendo que cambiar únicamente la línea de filtrado colocando el nombre del campo deseado.

```
from(bucket: "esp32_db")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_field"] == "params_temperature")
  |> aggregateWindow(every: 1h, fn: mean, createEmpty: false)
  |> yield(name: "mean")
```

Listing 13: Query Flux para obtener la media del parámetro *params_temperature* por hora

4.5.1. Métricas obtenidas con MQTT

Tras explicar el método de recolección de datos y métricas, pasamos a analizar las métricas obtenidas en el sistema a lo largo de un flujo de uso con MQTT de una hora. En primer lugar, podemos observar que al usar MQTT, el comportamiento del RTT (ilustrado en la Figura 12) parece seguir un patrón. Como vemos, el RTT es constante y estable, aunque haya experimentado una caída a mitad de la gráfica estudiada. Esta estabilidad prácticamente constante es característica de MQTT, debido a la naturaleza persistente del protocolo. En cuanto a la caída que sufre la gráfica, se puede deber a varios factores. El primer posible motivo de la caída es la optimización de la conexión, ya que es posible que se haya realizado una optimización en la configuración de MQTT o en la infraestructura de red alrededor de las 22:20, resultando en una mejora drástica del RTT, aunque esto quedaría fuera de nuestro estudio. Otro posible motivo es la persistencia de la conexión, ya que MQTT utiliza conexiones persistentes, lo que elimina la necesidad de reconectar para cada mensaje, reduciendo significativamente la latencia y variabilidad. Posibles motivos adicionales incluyen la disminución de la carga de datos enviados en el sistema, aunque esta opción queda descartada ya que no hay cambios en los

mensajes enviados para esta prueba. En general, todo apunta a que los culpables son la propia red y la optimización característica del protocolo MQTT. Factores como interferencias de red, fluctuaciones en el ancho de banda disponible o congestión en la red podrían contribuir a esta tendencia, más aún considerando que estamos operando por Wi-Fi.

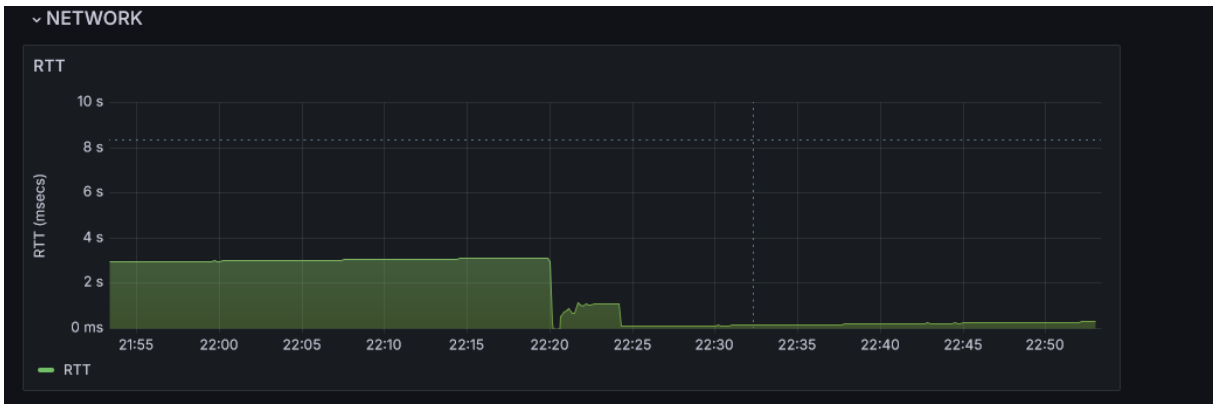

Figura 12: Valor promedio del RTT en una hora usando MQTT

En cuanto a las métricas relacionadas con la memoria del contenedor virtual, no podemos comentar nada muy relevante. La gráfica en la Figura 13 muestra una constancia en cuanto a los niveles de uso, que son bastantes bajos, con una media del 16.9 % de memoria siendo usada. La cantidad de memoria libre es bastante alta, alrededor de 4.63 GiB en promedio, y se mantiene muy estable a lo largo del tiempo. Esto indica que el sistema tiene una cantidad significativa de memoria no utilizada disponible, y unido al bajo porcentaje de uso, nos sugiere que no hay problemas de memoria o falta de recursos en el sistema. Por lo tanto, la conclusión es que el sistema tiene una gestión de memoria muy estable y eficiente (al menos durante el periodo observado).

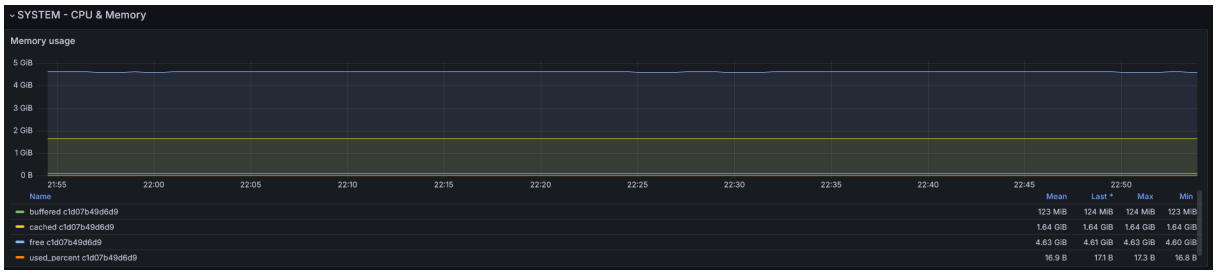

Figura 13: Valor promedio del uso de memoria del contenedor virtual en una hora usando MQTT

Moviéndonos a las métricas relacionadas con el uso de CPU del contenedor, nos fijaremos en las imágenes de las Figuras 14 y 15. Antes de las métricas mostradas en la Figura, en el panel tenemos otras métricas, usage_guest, usage_iowait, usage_irq, y usage_nice, y todas tienen un valor que es 0 o es prácticamente 0 ya que miden el uso de CPU por máquinas virtuales, operaciones de entrada/salida (I/O), manejadores de interrupciones hardware y procesos con prioridad modificada (nice) respectivamente. En nuestro caso, estas métricas no son importantes ya que no hacemos uso de ninguna de estas funciones, pero en general si lo suelen ser, y es por ello que las mantenemos en los paneles.

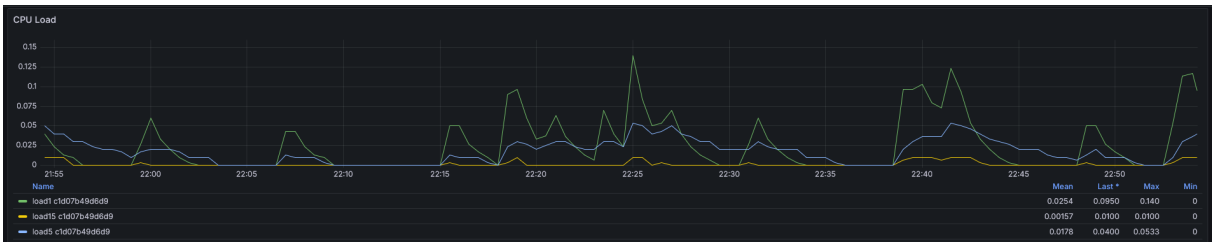

Figura 14: Valor promedio de la carga del CPU del contenedor virtual en una hora usando MQTT

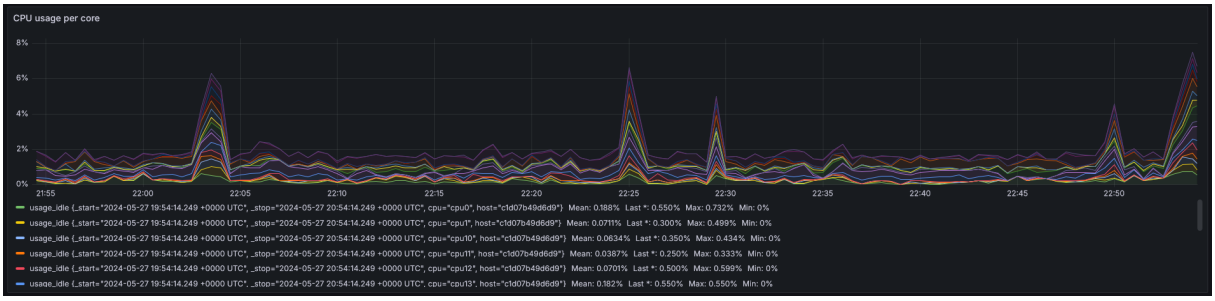

Figura 15: Valor promedio del uso (total y por núcleos) del CPU del contenedor virtual en una hora usando MQTT

Las que si son verdaderamente relevantes para nuestro caso, y que aparecen en las figuras, son la carga de CPU y el uso por núcleo. En el caso de la carga, tenemos tres parámetros siendo mostrados, *load1*, *load5* y *load15*, que representan la carga promedia de la CPU en el último minuto, los últimos 5 minutos, y los últimos 15 minutos, respectivamente. Los resultados de estos parámetros representan la cantidad media de procesos que estaban esperando para usar la CPU en un momento dado. Los valores de *load1* y *load5* son bajos, con un promedio de 0.0254 y 0.0178 y un pico máximo de 0.140 y 0.0533 respectivamente, lo que sugiere que la

carga de trabajo es constantemente liviana a corto plazo. En el caso de *load15*, los valores obtenidos son 0.00157 de media y 0.0100 de pico máximo. Estos valores nos indican que la carga es efectivamente ligera, e incluso lo es más a largo plazo. Como era de esperar, la carga en el último minuto es la que registra más actividad, ya que el dispositivo ESP32 publica los mensajes y Telegraf refresca sus entradas, ambos cada 10 segundos.

En cuanto a la métrica del uso de CPU por núcleo, lo que podemos observar es que los valores del parámetro *usage_idle*, que representa el tiempo que cada núcleo de la CPU está en uso, están alrededor del 0 % y el 8 %. Es decir, la CPU está inactiva en torno al 92 % - 100 % (100 menos el porcentaje de actividad) del tiempo, lo que sugiere que los núcleos están mayormente inactivos. Hay picos ocasionales en el uso de la CPU, que no superan el 8 % por núcleo, y que además están distribuidos uniformemente entre los núcleos, lo que indica que la carga se reparte bien entre ellos. Estos picos se dan alrededor de las 22:00, 22:05, 22:25, 22:30 y 22:50, y coinciden con los picos observados en la carga de la CPU, sugiriendo momentos de alta demanda de procesamiento. Los picos son manejados de manera efectiva sin un impacto prolongado en el rendimiento del sistema. Esta capacidad de manejar picos sugiere que el sistema está bien dimensionado para su carga actual y puede soportar aumentos temporales en la demanda de procesamiento. La baja carga promedia indica que hay suficiente capacidad disponible para escalar el sistema si se considerara necesario.

Por último, tenemos métricas que nos informan sobre el disco del contenedor virtual mientras está en ejecución. En concreto medimos los parámetros del valor promedio de uso y el número de inodos del disco root y sus particiones en las figuras 16 y 17. Los inodos son estructuras de datos que describen sistemas de archivos (archivos o directorios), almacenando la información relevante de estos. En cuanto al uso del disco, la gráfica nos muestra que tanto el tamaño promedio como el valor máximo de disco usado son de 6.21 GiB frente a 949 GiB de tamaño de disco libre. Con respecto a las particiones, parece que hay solamente una partición siendo usada mayormente, la cual se atribuye a la ruta */etc/telegraf* (en concreto al archivo telegraf.conf), como podemos ver en los atributos del parámetro. Esta partición tiene un porcentaje de uso promedio aproximado entre el 50 % y el 55 %, valores bastante más altos que el resto de particiones, que rondan el 1 % de uso.

En cuanto a los inodos del disco, podemos ver en la gráfica el número de inodos totales en amarillo, el número de inodos libres en verde, y el número de inodos totales usados en rojo. El

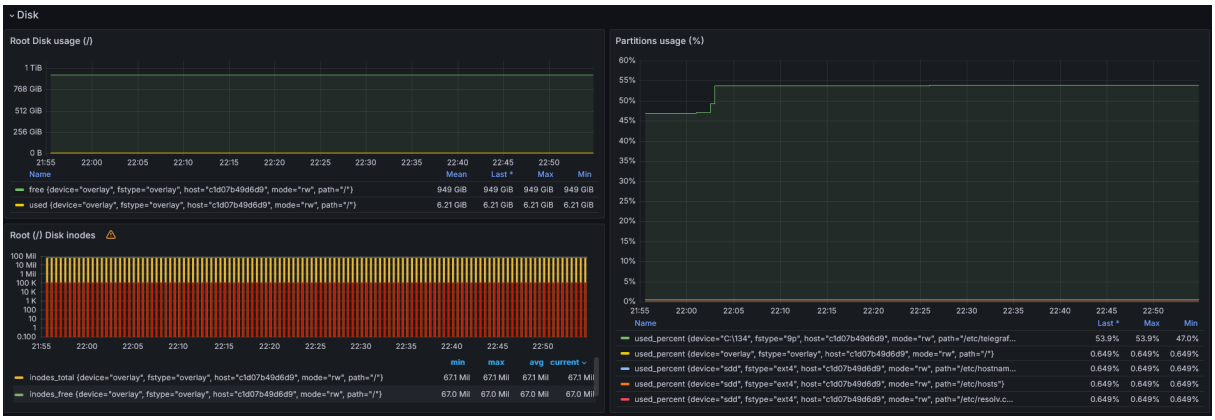

Figura 16: Valor promedio del uso del disco root y las particiones, y de los inodos del disco root del contenedor virtual en una hora usando MQTT

Figura 17: Valor promedio de los inodos de las particiones del contenedor virtual en una hora usando MQTT

valor de este último es la resta de los inodos totales menos los inodos libres. Siendo el valor promedio del numero de inodos totales 67.1 millones, y el del numero de inodos libres 67.0 millones, observamos que el número de inodos usados asciende a unos 123 mil. Con la última métrica, el número de inodos libres, de las particiones pasa algo parecido a lo que pasaba con el porcentaje de uso de las particiones, y es que el número de inodos libres referente a la ruta */etc/telegraf/telegraf.conf* es considerablemente menor al del resto de particiones del sistema. Podemos comprobar que el número de inodos libres de esta partición asciende a 1 millón de media, mientras que el resto se mantiene en 67 millones, lo cual nos indica que efectivamente aquí es donde se ubica la zona de mayor uso de disco del sistema mientras se está ejecutando. Aún así, en general las gráficas indican que el sistema tiene una gran cantidad de espacio de almacenamiento disponible y una abundancia de inodos libres, lo que sugiere

que no hay problemas de almacenamiento inminentes. La partición raíz (/) está utilizando solo una pequeña fracción de su capacidad total, y las otras particiones tienen un uso mínimo, lo que asegura una buena salud general del sistema de archivos y su capacidad para manejar futuras demandas de almacenamiento

4.5.2. Métricas obtenidas con HTTP

Veamos a continuación las mismas métricas, pero esta vez obtenidas usando HTTP para la transmisión de los mensajes. La primera diferencia notable es el valor del RTT, que podemos observar en la Figura 18. La gráfica con HTTP muestra más saltos debido a la variabilidad inherente en el establecimiento de conexiones y el mayor *overhead* del protocolo. El overhead es el hecho de que cada solicitud HTTP generalmente implica abrir una nueva conexión TCP, realizar el *handshake* (saludo inicial de una conexión HTTP) y luego enviar la solicitud, lo cual introduce una mayor latencia y variabilidad. En contraste, la misma gráfica pero haciendo uso de MQTT muestra una línea más recta y estable debido a su diseño optimizado para mantener conexiones persistentes y minimizar la latencia.

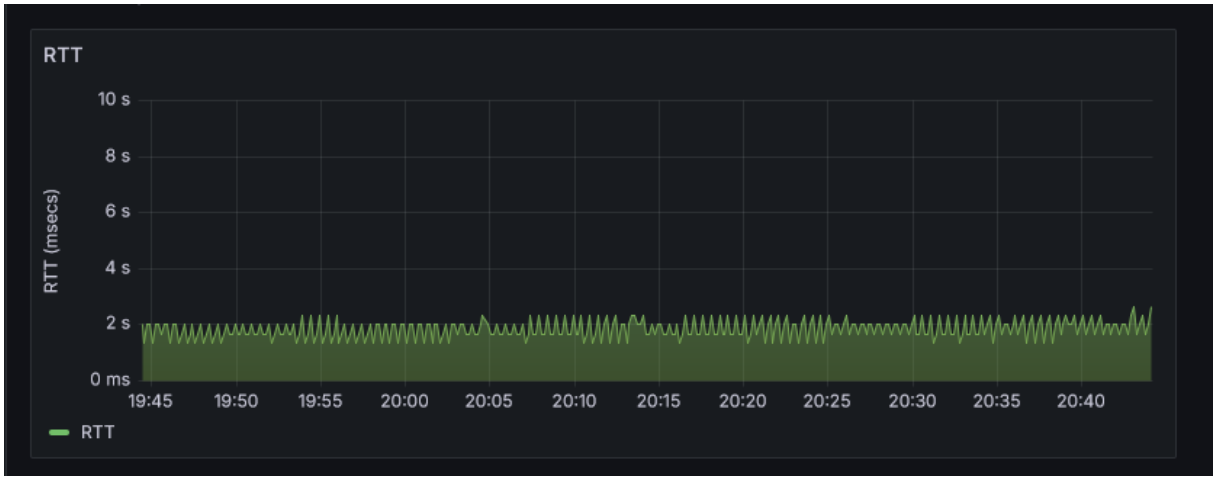

Figura 18: Valor promedio del RTT en una hora usando HTTP

Vamos a analizar ahora la carga y uso de CPU haciendo uso de HTTP, en comparación con los resultados obtenidos usando MQTT, fijándonos para ello en la imagen de las Figuras 19 y 20. En primer lugar, observamos que los valores relacionados a la carga de la CPU no son tan dispares. En el caso más cercano temporalmente, *load1*, vemos que HTTP registra valores ligeramente superiores como podíamos esperar, con una media de 0.0257 y un pico máximo

de 0.210. En las otras dos medidas de carga, *load5* y *load15*, observamos que de media si que obtienen un valor un poco inferior al obtenido con MQTT, registrando 0.0157 y 0.00105 respectivamente. Aún así, los valores máximos de estas dos medidas son mayores usando HTTP, con 0.0600 y 0.0167, respectivamente. Donde si que podemos observar un cambio más notable es en el uso de la CPU por núcleo. Usando HTTP, tanto la media como los valores máximos de este parámetro son prácticamente todos superiores a los obtenidos usando MQTT. Esta comparativa nos acerca más a la confirmación de nuestra hipótesis de que MQTT es más ligero, y por tanto mucho más apropiado que HTTP en un contexto de un sistema IoT.

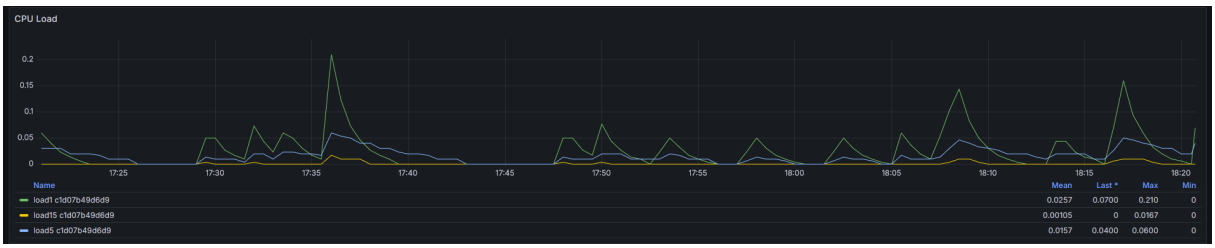

Figura 19: Valor promedio de la carga del CPU del contenedor virtual en una hora usando HTTP

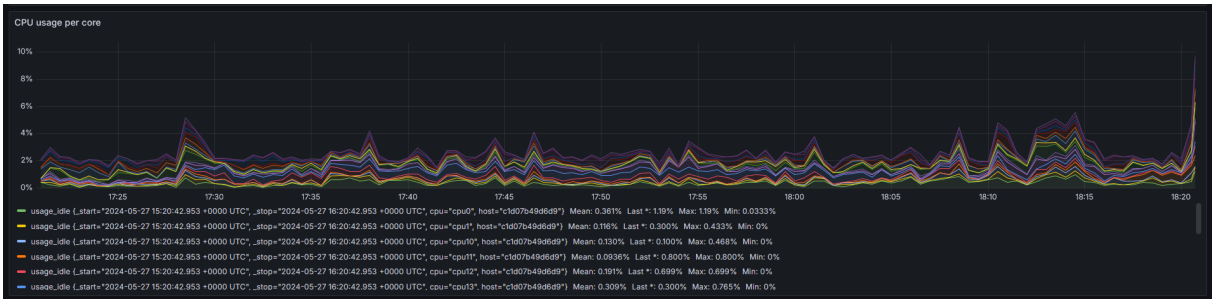

Figura 20: Valor promedio del uso (total y por núcleos) del contenedor virtual en una hora usando HTTP

En cuanto al uso de memoria del contenedor virtual, no vemos un cambio tan drástico con respecto a los valores obtenidos usando MQTT, como vemos en la Figura 21. Lo mismo ocurre con las métricas relacionadas con el disco, los valores son prácticamente iguales (figuras 22 y 23).

Por último, podemos ver una tabla comparativa de todos los valores obtenidos para cada parámetro que ha sido medido usando MQTT y HTTP en la Tabla 3.

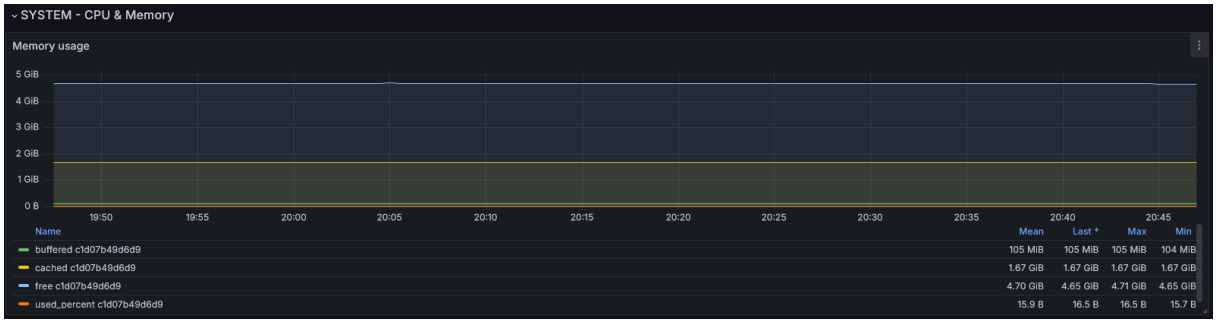

Figura 21: Valor promedio del uso de memoria del contenedor virtual en una hora usando HTTP

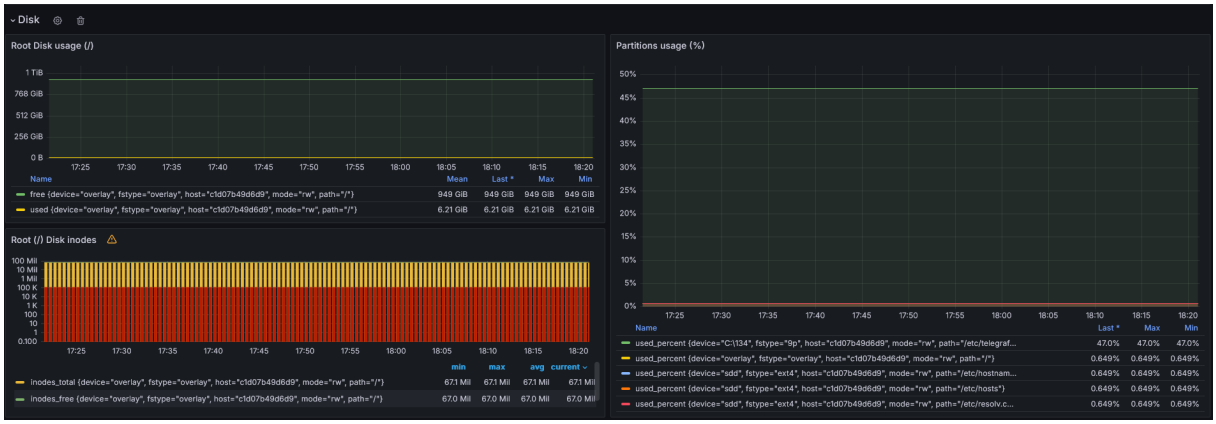

Figura 22: Valor promedio del uso del disco root y las particiones, y de los inodos del disco root del contenedor virtual en una hora usando HTTP

Figura 23: Valor promedio de los inodos de las particiones del contenedor virtual en una hora usando HTTP

Parámetro medido	Valor obtenido usando MQTT	Valor obtenido usando HTTP	Diferencia
RTT	Rango de 0.3 a 2.7 (1.5 de media)	Rango de 1.8 a 2.3 (2.05 de media)	+1.5 (mín), -0.4 (máx), +0.55 (media)
Uso promedio de memoria	16.9 %	15.9 %	-1 %
Carga de CPU - load1	0.0254 (media) - 0.140 (máx)	0.0257 (media) - 0.210 (máx)	+0.0003 (media) - +0.07 (máx)
Carga de CPU - load5	0.0178 (media) - 0.0533 (máx)	0.0157 (media) - 0.0600 (máx)	-0.0021 (media) - +0.0067 (máx)
Carga de CPU - load15	0.00157 (media) - 0.0100 (máx)	0.00105 (media) - 0.0167 (máx)	-0.00052 (media) - +0.0067 (máx)
Uso de CPU (por núcleos)	Pico en 7.8 %	Pico en 10 %	+2.2 %
Uso promedio de disco root	6.21 GiB	6.21 GiB	0
Promedio de inodos usados por el disco root	123K	123K	0
Uso promedio de partición telegraf	50.45 %	47.0 %	-3.45 %
Promedio de inodos libres en la partición telegraf	1M	1M	0

Tabla 3: Comparación de los valores de los parámetros medidos usando MQTT y HTTP

5

Conclusiones y Líneas Futuras

5.1. Conclusiones

Lo que nos dice la teoría es que el protocolo MQTT es más apropiado para un entorno IoT del calibre del ejemplo práctico que hemos presentado como caso de uso. En el estudio del protocolo hemos observado sus características principales y el funcionamiento que presenta, que respaldan esta teoría. Desde los inicios del protocolo, se empezó a desarrollar con el objetivo de servir en el contexto que comentamos de una mejor manera que las tecnologías presentes y dominantes en el sector en aquel momento, e incluso que lo siguen siendo hoy en día. Es por eso que MQTT se presenta como una de las tecnologías de comunicación predominantes en el mercado hoy en día. Y es por esto que mostramos interés y ponemos el foco en el protocolo, lo explicamos a fondo y lo ponemos a prueba.

Los resultados obtenidos en esta comparativa nos han indicado en general que se mantiene en pie la hipótesis inicial de que MQTT es un protocolo más apropiado que HTTP en el contexto de un sistema IoT donde los dispositivos necesiten comunicarse regularmente. Es bastante probable que escalando el mismo sistema a un sistema más grande, se obtengan las diferencias de manera más exagerada y acentuada. Aún así, la comparativa realizada ha revelado diferencias significativas entre las dos tecnologías en varios aspectos, incluyendo la red, el uso/carga de CPU y memoria, y el uso de disco. A continuación, se presenta un análisis detallado de estas diferencias, además de otras no obtenidas directamente en el caso de uso, pero que son interesantes de resaltar:

1. Red

    a) RTT

En HTTP hemos observado un RTT mayor y más variable debido al *overhead* que comentábamos previamente, que está presente en el sistema por la necesidad de establecer y cerrar conexiones TCP para cada solicitud, inherente al funcionamiento protocolo. Además, el uso de HTTP hace al sistema menos eficiente en términos de velocidad de transmisión para mensajes pequeños y frecuentes, comunes en aplicaciones IoT.

En cambio, usando MQTT, en general obtenemos una latencia inferior, debido a la persistencia de la conexión (una vez establecida, se reutiliza para múltiples mensajes), y hemos podido comprobar que el protocolo está optimizado para la transmisión rápida de mensajes, especialmente pequeños, lo que resulta en un RTT más constante e inferior.

b) Consumo de Ancho de Banda

Esto no lo hemos podido comprobar directamente, pero podemos estimar que haciendo uso de HTTP se tenga un mayor consumo de ancho de banda debido al tamaño superior de las cabeceras y a la re-negociación de conexiones, lo cual es menos eficiente para aplicaciones IoT que requieren la transmisión frecuente de datos.

Por otro lado, MQTT hace uso de cabeceras más ligeras y una conexión persistente, lo que reduce el consumo de ancho de banda, y es mejor para dispositivos con limitaciones de ancho de banda y aplicaciones que requieren comunicación frecuente.

2. Uso/Carga de CPU y Memoria

a) CPU

Hemos observado que mientras se usaba el protocolo HTTP para la comunicación del sistema, hemos obtenido una mayor carga de CPU debido de nuevo a la necesidad de manejar conexiones repetitivas y el *overhead* de HTTP. Esto se hace especialmente evidente en el uso del CPU por núcleos, como podemos ver reflejado en los picos de la gráfica correspondiente en comparación a la análoga en la parte de MQTT.

En cuanto a MQTT, obtenemos una menor y más constante carga de CPU debido

a la eficiencia del protocolo ajustada para el sistema que hemos desarrollado, y de nuevo por la persistencia de la conexión, que provoca que la CPU no trabaje de más en un intercambio regular de información. Encontramos que la carga de CPU es más predecible y estable, lo cual es ideal para dispositivos con recursos limitados.

b) Memoria

Refiriéndonos al uso de memoria del sistema, no podemos comentar una diferencia muy notable en el caso de uso desarrollado. Quizás en un sistema de características más grandes y exigentes si que pueda llegar a presentar unas diferencias evidentes el uso de un protocolo u otro como tecnología de comunicaciones del sistema. Lo que podríamos esperar es que el uso de HTTP requiriera de *buffers* más grandes para manejar las cabeceras de los mensajes y los datos temporales de cada conexión, con respecto a un menor uso de memoria y un mejor manejo de recursos para dispositivos con limitaciones de memoria con el uso de MQTT, ya que la conexión persistente requiere menos *buffers* y manejo de estado.

3. Uso de Disco

En este aspecto tampoco hemos encontrado una diferencia muy grande, al igual que pasa con la memoria del sistema. Podemos hacer la misma suposición de que en un sistema más grande podríamos encontrarnos con unas diferencias más notables. Entre ellas, podríamos observar que en HTTP se necesite almacenamiento adicional, ya que se genera una cantidad mayor de logs debido a la cantidad de conexiones y solicitudes que se generan en el sistema, además de tener que manejar posibles retransmisiones y estados temporales de cada conexión, como comentábamos previamente.

En cambio, con MQTT no generaríamos tantos logs gracias a la persistencia y el *overhead* inferior que presenta el protocolo.

4. Aspectos Adicionales

a) Seguridad

En cuanto a la seguridad se refiere, ambos protocolos presentan mecanismos para asegurar el máximo nivel de seguridad en las comunicaciones. HTTP hace uso de HTTPS para esto, lo cual añade *overhead* pero es ampliamente soportado.

MQTT puede usar TLS/SSL para asegurar las conexiones, aunque esto puede aumentar la carga de CPU. Además, también proporciona mecanismos de autenticación y autorización (que hemos usado en el caso de uso) adecuados para entornos IoT.

b) Escalabilidad

La naturaleza del protocolo HTTP lo hace brillar en aplicaciones donde las solicitudes son esporádicas y no tan frecuentes. Si lo que necesitamos en cambio es un sistema donde las comunicaciones ocurran de manera regular, HTTP presenta una escalabilidad limitada en dichos entornos, debido al *overhead* y la gestión de múltiples conexiones que describíamos previamente, que se van a ver acentuados al añadir incluso más conexiones todavía.

MQTT, en cambio, presenta un alto nivel de escabilidad en este tipo de entornos debido a su eficiencia y su menor uso de recursos, lo que lo hace ideal para este tipo de aplicaciones que requieren una transmisión de datos en tiempo real y en alta frecuencia. En el supuesto de añadir más dispositivos y/o más sensores al sistema, lo único que esto supondría sería que el broker se suscribiera a más tópicos, pero al publicarse la información en tópicos a los que se pueden suscribir muchos dispositivos, no hay que tener una conexión específica con cada uno de los dispositivos que necesite acceder a la información como pasa con HTTP.

c) Facilidad de Implementación

En este aspecto, es debatible pero razonable pensar que HTTP es superior a MQTT. HTTP es un protocolo bien conocido y establecido en el panorama actual, con un amplio soporte mediante bibliotecas y herramientas al alcance de cualquier persona que lo quiera aprender a usar.

Por otro lado, MQTT está diseñado específicamente para sistemas IoT, que aunque presente un creciente soporte y muchas bibliotecas disponibles para los lenguajes de programación principales, sigue siendo un nicho en el mercado actual. Además, MQTT no es solo un protocolo distinto, si no que hace uso de un paradigma de comunicación diferente (el modelo Publicación-Suscripción) al que está ya implantado en la sociedad tecnológica actual (el modelo Petición-Respuesta). Es por

esto que aún, a día de hoy, su curva de aprendizaje pueda presentarse mayor para aquellos nuevos en el protocolo y en los entornos IoT en general.

En resumen, MQTT es claramente más adecuado para aplicaciones IoT debido a su baja latencia, menor uso de ancho de banda, y eficiencia en el uso de CPU y memoria. Otras tecnologías como HTTP, aunque puedan ser universalmente soportadas y fáciles de implementar, presentan desventajas significativas en términos de *overhead* y eficiencia, especialmente en aplicaciones donde la transmisión frecuente y en tiempo real de datos es crítica. Para dispositivos IoT con limitaciones de recursos, MQTT proporciona un rendimiento superior y es más apropiado para escenarios de comunicación en tiempo real o con una comunicación constante para los que se requiera obtener una baja latencia.

5.2. Líneas Futuras

Como líneas futuras, se plantea escalar el sistema de manera que haya más sensores distintos y más dispositivos en general, para que se pueda así estudiar de manera más amplia el comportamiento del protocolo y se puedan observar de forma más precisa sus ventajas en un entorno más exigente. En el presente trabajo hemos usado un microcontrolador con recursos limitados, pero también se probaría el proyecto y se adaptaría a dispositivos con restricciones energéticas, por ejemplo. Además, se podría extrapolar esta investigación a un análisis del protocolo desde el punto de vista de la seguridad, debido a la relevancia que presenta la ciberseguridad en la tecnología actual. Por último, se puede estudiar la aplicación de la tecnología a otros ámbitos más específicos de las tecnologías IoT, como lo serían el ámbito IoT industrial (como en la automatización y monitoreo de equipos de fábricas, la gestión de cadenas de suministro, etc.) o incluso las populares *Smart Cities*, o ciudades inteligentes en español (como el monitoreo del tráfico, la gestión de servicios públicos, y la seguridad pública).

Referencias

[1] *Página web de MQTT*. Disponible online: https://mqtt.org/. (accedida el 6 de marzo de 2024).

[2] *Página web de Docker*. Disponible online: https://www.docker.com/. (accedida el 6 de marzo de 2024).

[3] *Página web de Eclipse Paho*. Disponible online: https://eclipse.dev/paho/. (accedida el 6 de marzo de 2024).

[4] *Página web de ESP32*. Disponible online: http://esp32.net/. (accedida el 6 de marzo de 2024).

[5] *Página web con el resumen del dispositivo Adafruit HUZZAH32 - ESP32 Feather*. Disponible online: https://learn.adafruit.com/adafruit-huzzah32-esp32-feather/. (accedida el 20 de mayo de 2024).

[6] *Página de compra del sensor AM2302*. Disponible online: https://www.adafruit.com/product/393. (accedida el 6 de marzo de 2024).

[7] Hasenradball. *Repositorio en GitHub de la librería del sensor AM2302 en Arduino*. Disponible online: https://github.com/hasenradball/AM2302-Sensor. (accedida el 18 de abril de 2024).

[8] *Página de compra del sensor CCS811*. Disponible online: https://www.sparkfun.com/products/retired/14193. (accedida el 6 de marzo de 2024).

[9] *Repositorio en GitHub de la librería del sensor CCS811 de Sparkfun en Arduino*. Disponible online: https://github.com/sparkfun/SparkFun_CCS811_Arduino_Library. (accedida el 18 de abril de 2024).

[10] *Página web de Git*. Disponible online: https://git-scm.com/. (accedida el 6 de marzo de 2024).

[11] *Página web de GitHub*. Disponible online: https://github.com/. (accedida el 6 de marzo de 2024).

[12] *Página web de Grafana*. Disponible online: https://grafana.com/. (accedida el 6 de marzo de 2024).

[13] *Página web de InfluxDB*. Disponible online: https://www.influxdata.com/. (accedida el 6 de marzo de 2024).

[14] *Página web de Mosquitto*. Disponible online: https://mosquitto.org/. (accedida el 6 de marzo de 2024).

[15] *Página web de Raspberry*. Disponible online: https://www.raspberrypi.com/. (accedida el 6 de marzo de 2024).

[16] *Página web de Telegraf*. Disponible online: https://www.influxdata.com/time-series-platform/telegraf/. (accedida el 6 de marzo de 2024).

[17] *Página web de Visual Studio Code*. Disponible online: https://code.visualstudio.com/. (accedida el 6 de marzo de 2024).

[18] Andy Stanford-Clark. *MQ Integrator Pervasive Device Protocol - MQ/SCADA Protocol*. Version 2.3. Disponible online: https://stanford-clark.com/MQIpdp/. (accedida el 6 de marzo de 2024). Oct. de 1999.

[19] Patrick Th. Eugster et al. "The Many Faces of Publish/Subscribe". En: *ACM Computing Surveys* 35.2 (2003), págs. 114-131. ISSN: 03600300. DOI: 10.1145/857076.857078. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0345565888&doi=10.1145%2f857076.857078&partnerID=40&md5=2f73e727073b55714a15f02b5e61bedf.

[20] Andy Stanford-Clark. *MQTT Version 3*. Version 3.0. Disponible online: https://stanford-clark.com/MQTT/. (accedida el 6 de marzo de 2024). Oct. de 1999.

[21] IBM y Eurotech. *MQTT Version 3.1*. Version 3.1. Disponible online: https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html. (accedida el 6 de marzo de 2024). 2013.

[22] Andrew Banks y Rahul Gupta. *MQTT Version 3.1.1*. Version 3.1.1. Disponible online: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. (accedida el 6 de marzo de 2024). Oct. de 2014.

[23] *Página web de OASIS Open*. Disponible online: https://www.oasis-open.org/. (accedida el 7 de abril de 2024).

[24] Jens Deters. ""Secret" unveiled: Why MQTT v4.0 will never be released". En: (feb. de 2017). (accedido el 8 de abril de 2024). URL: https://www.jensd.de/wordpress/?p=2667.

[25] Andrew Banks et al. *MQTT Version 5.0*. Version 5.0. Disponible online: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. (accedida el 6 de marzo de 2024). Mar. de 2019.

[26] Hong Linh Truong Andy Stanford-Clark. *MQTT-SN V1.2 Protocol Specification*. Version 5.0. Disponible online: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. (accedida el 6 de marzo de 2024). Mar. de 2019.

[27] *Página web de Shelly*. Disponible online: https://www.shelly.com/. (accedida el 13 de abril de 2024).

[28] Theo Arends. *Repositorio en GitHub de Tasmota*. Disponible online: https://github.com/arendst/Tasmota. (accedida el 18 de abril de 2024).

[29] *Página web del dashboard de Grafana VMware vSphere - Overview*. Disponible online: https://grafana.com/grafana/dashboards/8159-vmware-vsphere-overview/. (accedida el 18 de abril de 2024).

[30] *Página web del manual de instalación de Docker en Debian*. Disponible online: https://docs.docker.com/engine/install/debian/. (accedida el 10 de mayo de 2024).

[31] *Página web para la descarga del software Arduino IDE*. Disponible online: https://www.arduino.cc/en/software. (accedida el 10 de mayo de 2024).

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA