UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería de Sonido e Imagen en Telecomunicaciones

Trabajo Fin de Grado

Implementación de un espacio de trabajo interconectado con dispositivos activos que se comunican utilizando el protocolo MQTT

Ignacio Diestro Gil

Noviembre, 2021

UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería de Sonido e Imagen en Telecomunicaciones

Trabajo Fin de Grado

Implementación de un espacio de trabajo interconectado con dispositivos activos que se comunican utilizando el protocolo MQTT

Autor: Ignacio Diestro Gil

Tutor: Antonio Gordillo Guerrero

RESUMEN

La rápida evolución que han sufrido las telecomunicaciones, nos ha permitido no solo conectarnos entre nosotros, sino también conectarnos con nuestro entorno a través de dispositivos “inteligentes”. Para este propósito, de conectarnos con el mundo físico que nos rodea, ha surgido el Internet de las Cosas (IoT).

Para poder dar soporte a esta comunicación, se necesita una infraestructura que soporte los sistemas, que se encargarán de transferir los datos, analizarlos y tomar decisiones en función de estos. Para este proyecto, se ha diseñado un sistema para el Internet de las Cosas. Desde los dispositivos que se instalarán en el mundo físico para interactuar con este, hasta los servidores que almacenarán toda la lógica del sistema. Todo esto, tomando como base el protocolo de comunicación MQTT.

Hemos logrado la creación de un sistema de comunicación fiable, escalable y estable. Utilizando para ello diversas tecnologías, como bases de datos temporales, puertas de enlace, sistemas de monitorización y lógicas programables, entre otras. El sistema se puede utilizar para dar soporte a una gran cantidad de soluciones IoT. Gracias a su diseño, la infraestructura se puede desplegar con facilidad y sin necesidad de invertir grandes cantidades de tiempo en su configuración.

ABSTRACT

The fast evolution of telecommunications has allowed us not only to connect with each other, but also to connect with our environment through "smart" devices. For this purpose, to connect us to the physical world around us, the Internet of Things (IoT) has emerged.

In order to allow this communication, an infrastructure is needed to support the systems that will be responsible for transferring data, analysing it and making decisions based on it. For this project, a system for the Internet of Things has been designed. From the devices that will be installed in the physical world to interact with it, to the servers that will store all the logic of the system. All this, based on the MQTT communication protocol.

We have achieved the creation of a reliable, scalable and stable communication system. Using various technologies, such as temporary databases, gateways, monitoring systems and programmable logics, among others. The system can be used to support a wide range of IoT solutions. Thanks to its design, the infrastructure can be deployed easily and without the need to invest large amounts of time in its configuration.

ÍNDICE GENERAL DE CONTENIDOS

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

En esta época, la sensorización de los espacios para convertirlos en inteligentes está a la orden del día. El mundo cada vez está más interconectado y hemos llegado a un punto de la evolución en que no solo lo hacemos entre nosotros, sino también con el entorno y entre los propios dispositivos sin interacción humana. Todo lo que puede ser un dispositivo, es un dispositivo. La sensorización de los espacios nos permite un amplio control del mundo físico que nos rodea y nuestra interacción con él. De esta corriente ha surgido el Internet de las Cosas (IoT), un concepto para la interconexión digital de objetos.

Uno de los muchos usos con los que cuenta el Internet de las Cosas son las ciudades y edificios inteligentes. Al convertir un edificio en inteligente, aportando "inteligencia" y comunicación a sus componentes, podemos contar con un control completo del entorno y adaptarlo mejor a nuestras necesidades, ayudando a la mejora de la calidad de vida de las personas.

Basándonos en todas las ventajas que nos ofrece IoT, se ha propuesto el desarrollo de un sistema completo para soportar esta lógica de intercomunicación. Utilizando tecnologías punteras y métodos empresariales para un control completo de la infraestructura. Esto abarca desde los sensores, los cuales son dispositivos con bajas capacidades de procesamiento, a los servidores, que almacenarán todo el software y lógica necesarios para su funcionamiento; puertas de enlace, bases de datos, monitorizadores, webhooks, etc…

1.1. Alcance del proyecto

Lo más importante a la hora de afrontar el proyecto, ha sido el enfoque profesional del mismo. Esto nos lleva a crear un sistema con unos pilares sólidos, bien documentado y que permite una escalabilidad continua. Con facilidad para configurar los diferentes parámetros que controlan la infraestructura, sin afectar a sus funciones, ni impedir su correcto funcionamiento.

Con estos pilares técnicos, el sistema es estable y puede trasladarse a diversos escenarios de uso. Solo es necesario ajustar los parámetros de configuración, no modificar su arquitectura.

1.2. Motivación

El proyecto, ha contado con un fuerte respaldo de motivación. Se ha afrontado con la mentalidad de aprender nuevas tecnologías y afianzar los conceptos con los que ya contaba.

Este proyecto cuenta con fuertes conceptos de informática, sobre todo infraestructura y microservicios, los cuales no se han estudiado en la carrera. Este es el caso del trabajo con servidores, bases de datos, APIs, etc.

Sin embargo, cuenta también con conceptos de programación y de comunicaciones inalámbricas que sí se han estudiado. Partiendo de una base de conocimientos, se han ampliado y perfeccionado. Como la programación en C, Python, comunicaciones inalámbricas, etc.

He visto en este proyecto, una oportunidad de adentrarme en un mundo en parte desconocido, que me ha permitido aprender tecnologías que seguramente me servirán el día de mañana.

2. OBJETIVOS

El proyecto llevado a cabo, tiene como finalidad crear un espacio de trabajo interconectado con dispositivos activos que se comunican utilizando el protocolo MQTT. Para ello, se ha hecho especial hincapié en la infraestructura que soporta los canales de comunicación, análisis y configuración del sistema.

Los objetivos propuestos son:

2.1. Objetivos funcionales

- Crear de un espacio inteligente interconectado.
- Utilizar MQTT para la conexión de los dispositivos.
- Generar una infraestructura sólida que mantenga todo el sistema de comunicación y análisis de datos.
- Alta escalabilidad del sistema.
- Facilitar el despliegue.
- Facilitar el implementar modificaciones.
- Enfocar de manera empresarial el proyecto.

2.2. Objetivos personales

- Conocer infraestructura de servidores, microservicios, backend y API.
- Ampliar conocimientos sobre IoT.
- Ampliar conocimientos en programación con Python.
- Ampliar conocimientos de hardware y dispositivos electrónicos.

3. ANTECEDENTES / ESTADO DEL ARTE

3.1. Internet de las Cosas (IoT)

Internet de las Cosas (**IoT**) por sus siglas en inglés '**Internet of Things**' (1) se basa en un concepto de interconexión de dispositivos que pueden interactuar entre sí y cooperar con otros objetos. De esta forma, conseguimos crear nuevas aplicaciones y servicios para alcanzar diferentes objetivos. Permitiendo que los objetos estén conectadas en todo momento, en cualquier lugar y con cualquier persona. Convirtiendo los objetos en "inteligentes", para que puedan interactuar con el mundo físico.

Este concepto fue extendido por **Kevin Ashton**, profesor del **MIT** (2). El cual, para el **RFID Journal** utilizó este término de Internet of Things por primera vez, para referirse a este sistema de conexión **M2M** (**Machine to Machine**). En este tipo de conexión, los objetos físicos utilizan tecnología incorporada para comunicarse e interactuar entre ellos. Este tipo de aplicaciones, pueden incluirse en los vehículos, casas o electrodomésticos inteligentes. Así, nos ofrecen notificaciones sobre su estado, permiten la interacción con su entorno y el ser humano, nos proporcionan seguridad y automatización, entre otras cosas.

Para el Internet de las Cosas, es importante mencionar la convergencia de redes utilizando IP. Pues se basa en el uso de una red IP multiservicio común, que admite una amplia gama de servicios y aplicaciones. De esta forma, no hay una arquitectura establecida, pudiendo utilizar un modelo distribuido e interconectado en lugar del tradicional centralizado, como podemos ver en la Figura 1.

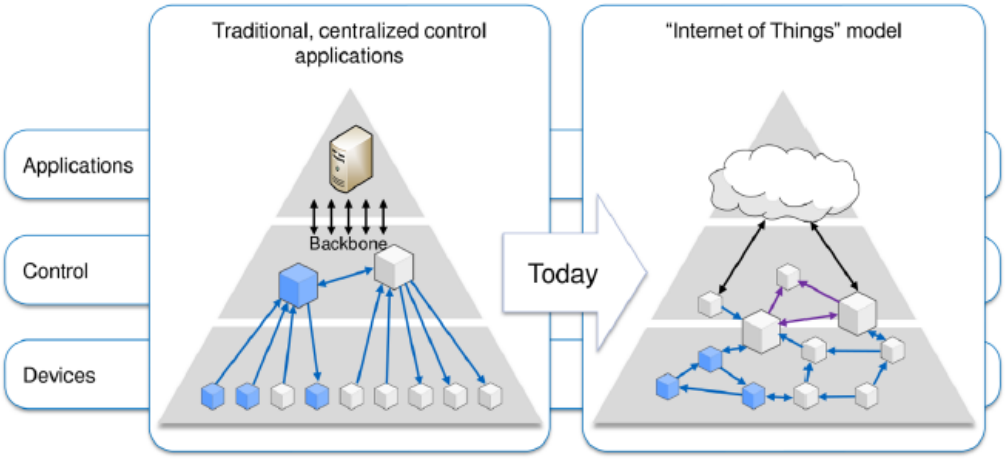

Figura 1: 'IoT' cooperativo interconectado (1)

Las características fundamentales que debe cumplir IoT son las siguientes:

- **Alta interconectividad:** todo se puede conectar con la infraestructura global de comunicación.
- **Servicios relacionados con cosas:** proporcionando servicios dentro de las limitaciones que encontramos, como la protección de privacidad o la falta de coherencia entre lo físico y lo virtual.
- **Heterogeneidad:** los servicios IoT se basan en diferentes plataformas de hardware o redes, para poder ser compatibles con otros dispositivos.
- **Cambios dinámicos:** tanto el estado como la cantidad de dispositivos se encuentra en constante cambio, por ejemplo, añadiendo sensores a la red o durmiendo otros que no necesitamos.
- **Alta escalabilidad:** los dispositivos que deben administrarse y comunicarse entre sí serán en un futuro de una gran magnitud en comparación con los que contamos hoy día.

Internet de las Cosas es un concepto global, que tiene en cuenta la amplia experiencia y las tecnologías necesarias; desde los sistemas de detección, comunicación, agregación de datos, procesado de estos y finalmente su prestación de servicios, para crear un solo sistema denominado Internet de las Cosas (IoT). Según

el **IERC** (European Research Cluster on the Internet of Things) junto con **ITU** (International Telecommunication Union), han dado esta definición para Internet of Things:

"Internet de las cosas (IoT): una infraestructura global para la sociedad de la información que permite servicios interconectando elementos (físicos y virtuales) basados en la evolución de las tecnologías de la información y las comunicaciones interoperables. NOTA 1 - A través de la explotación de identificación, captura de datos, procesamiento y capacidades de comunicación, IoT hace un uso completo de las cosas que ofrecen servicios a todo tipo de aplicaciones, garantizando al mismo tiempo que la seguridad y la privacidad cumplen los requisitos. NOTA 2 - Desde una perspectiva más amplia, IoT puede ser percibido como una visión con implicaciones tecnológicas y sociales"

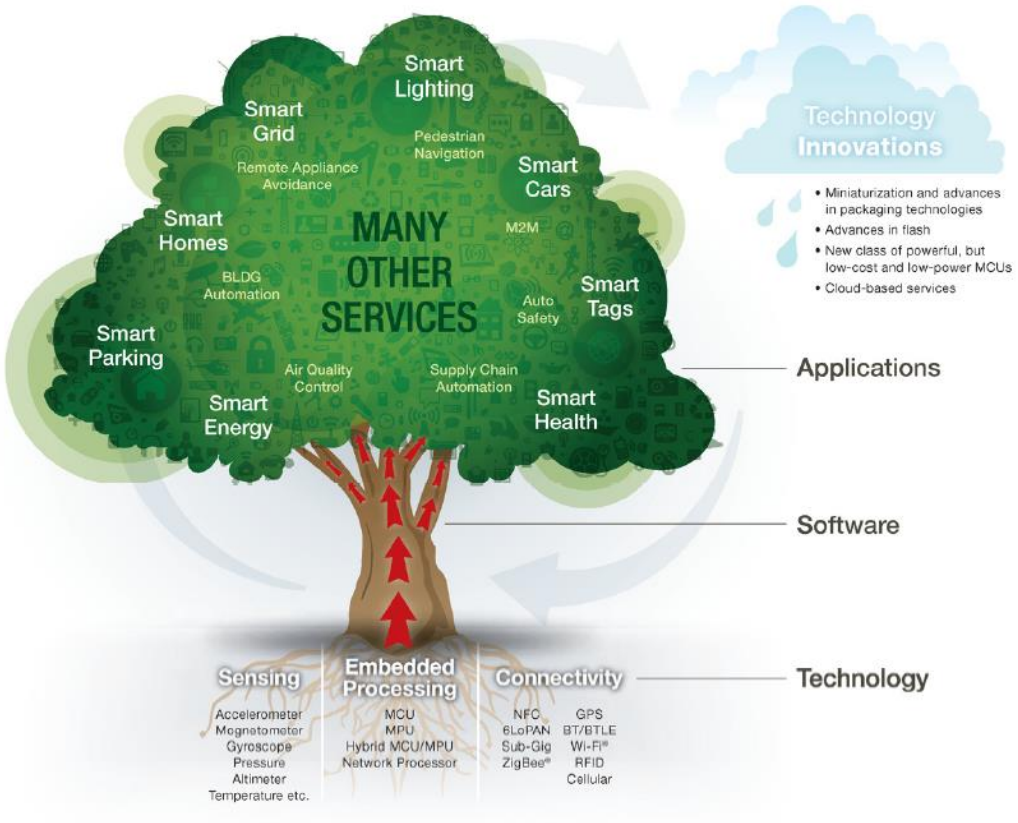

Figura 2: Diferentes servicios, tecnologías y significados de IoT (1)

3.1.1. Capas de la arquitectura IoT

Como infraestructura, el Internet de las Cosas adquiere una arquitectura por capas (3). Este enfoque modular de dividir la estructura en varios niveles y centrarse en cada uno de forma independiente, ayuda a gestionar la complejidad de los sistemas IoT. Aquí se incluyen diferentes aspectos (4), como los físicos (las cosas) y los virtuales (servicios y protocolos). Como mencionábamos anteriormente, no hay una arquitectura establecida, pero sí que existen diferentes propuestas (5), como son la arquitectura de 3 capas, arquitectura de 5 capas, arquitectura niebla o la arquitectura de Computación de Borde, entre otras.

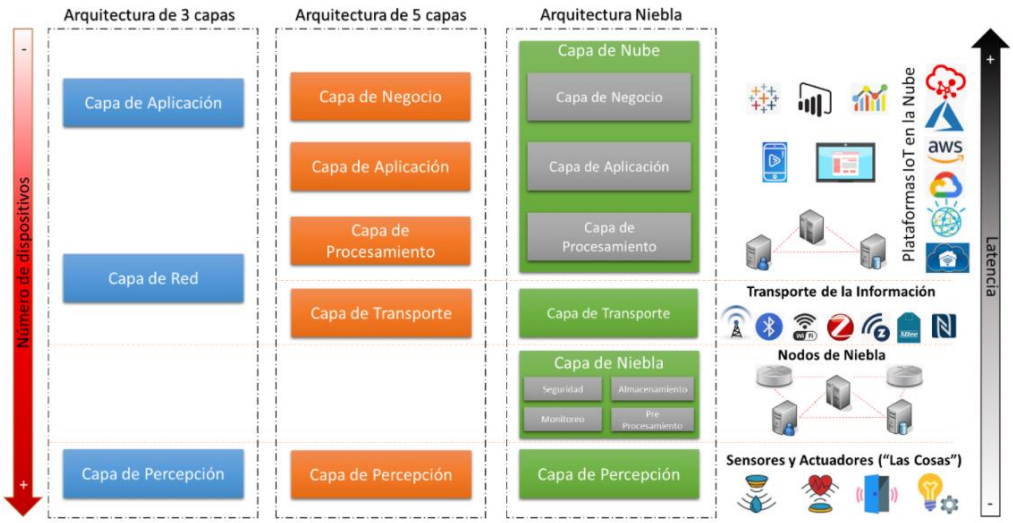

Figura 3: Arquitecturas IoT (5)

En nuestro caso, vamos a explicar la arquitectura más básica y general que es la de 3 capas. Ya que en IoT, las arquitecturas más detalladas las podemos incluir como subcapas verticales que se encuentran agrupadas dentro de las generales. Las tres principales partes de la **arquitectura de 3 capas** son: capa de percepción o de dispositivos, capa de red o de borde y capa de aplicación o de nubes.

3.1.1.1. Capa de percepción o de dispositivos

Es la capa inferior de la infraestructura. Está compuesta por los sensores y actuadores físicos, los cuales por sí solos no se consideran 'inteligentes', por lo que se deben conectar a los componentes de la arquitectura que cuentan con más capacidad de procesamiento, casi siempre a través de las puertas de enlace. Estos

dispositivos, normalmente transmiten la información utilizando protocolos inalámbricos como puede ser Bluetooth, Zigbee, Wifi, celular, RFID o con protocolos de cable como Ethernet.

3.1.1.2. Capa de red o de borde

Es la relacionada con los servicios de análisis y procesamiento situados en el borde de la red, sirviendo de integración para la capa de dispositivos. Proporciona control y enrutamiento para las capas superiores.

Este análisis se produce en tiempo real (o casi). Se realizan tareas como el filtrado y la agregación de datos, transfiriéndolos en sentido ascendente para su posterior procesado y análisis. Dado que suele ser la primera capa en contacto con la red pública de internet, una de las principales misiones de la capa es mantener el control y la seguridad de acceso.

3.1.1.3. Capa de aplicación o de nubes

Esta es la capa final de la infraestructura, cuando los datos ya han sido recogidos y preparados para su procesamiento, almacenamiento y uso en las aplicaciones de la nube. Estas aplicaciones a menudo se complementan con aplicaciones móviles y web, ayudando a presentar los datos a los usuarios finales de una forma más amistosa y organizada. También contamos con la opción de administrar y monitorizar todo el sistema a alto nivel e incluso utilizar APIs públicas para permitir la interacción con otros sistemas.

3.1.2. Componentes de la arquitectura IoT

Para la construcción del Sistema de IoT, la arquitectura cuenta con componentes que garantizan que los datos captados por los sensores se recopilan, almacenan y procesan (6) (4). A continuación, se muestra una enumeración de los componentes principales de la arquitectura IoT:

3.1.2.1. Cosas

Una ‘cosa’ es un objeto equipado con sensores que recopila datos para ser transferidos a través de la red y actuadores que permiten interactuar. Aquí, podemos encontrar diversos dispositivos como neveras, farolas, edificios, vehículos o incluso maquinaria industrial entre otros.

3.1.2.2. Gateways

Los gateways (puertas de enlace) proporcionan conectividad entre las cosas y la parte de la nube de la solución de IoT. Es decir, permiten el preprocesado y filtrado de los datos antes de pasarlos a la nube (ahorrando costes computacionales) y la transmisión de comandos de control que van desde la nube a las cosas, ya sea de forma directa o a través de actuadores.

3.1.2.3. Cloud Gateways

Las puertas de enlace a la nube, facilitan la comprensión y la transmisión segura de datos entre las Gateways de campo y los servidores IoT de la nube. También asegura la compatibilidad de los protocolos entre las diferentes puertas de enlace.

3.1.2.4. Procesadores de datos en Streaming

Se encargan del procesamiento de datos en tiempo real, para distribuir los procedentes de los sensores entre los componentes de las diferentes soluciones de IoT. Garantizando una transmisión efectiva de los datos de entrada a una base de datos y aplicaciones de control.

3.1.2.5. Data Lakes

Se utilizan para almacenar los datos generados por los sensores en su formato natural. Encontramos estos macrodatos en ‘lotes’ o ‘flujos’.

3.1.2.6. Big Data Warehouse

Aquí, encontramos los datos procesados y filtrados que son necesarios para obtener información valiosa. Estos se extraen de los Data Lakes. En un Big Data Warehouse encontramos solo datos limpios, estructurados y combinados. Además, encontramos información de contexto sobre cosas, sensores o comandos de control.

3.1.2.7. Machine Learning

Gracias al aprendizaje automático, contamos con la posibilidad de crear modelos precisos y eficientes para las aplicaciones de control. Utilizando datos actuales y pasados para establecer patrones de acción.

3.1.2.8. Aplicaciones de control

Estas aplicaciones se encargan de enviar los comandos y alertas automáticas a los actuadores. Para ello utilizan los datos recogidos por los sensores que posteriormente se han almacenado y analizado. Estas aplicaciones, pueden estar basadas en reglas prestablecidas o en aprendizaje automático.

3.1.2.9. Aplicaciones de usuario

Estas aplicaciones, permiten la interacción con el usuario de una forma más amistosa, son componentes software que permiten monitorear y controlar los sistemas. Esto puede ser a través de una aplicación móvil o web, enviando comandos de control o configurando comportamientos.

3.1.3. Principales aplicaciones para IoT

En los últimos años, se ha producido una evolución en los mercados y las aplicaciones, creando oportunidades significativas. El desarrollo de estas tendencias se ve facilitada por el uso del Internet de las Cosas. De esta forma, se crean sistemas que se pueden amortizar en periodos cortos de tiempo y que hacen posibles las infinitas ideas que queramos llevar a cabo. Las aplicaciones de IoT están limitadas casi por nuestra imaginación y las podemos ver en todos los aspectos de la vida. A continuación, se muestra una lista con algunos de los dominios de aplicación de IoT identificados por **IERC** (1), basados en aportaciones de expertos, encuestas e informes:

Figura 4: Internet Of Things: proliferación de dispositivos conectados en todos los sectores (1)

3.1.3.1. Smart Health

Gracias a la red de sensores, podemos monitorizar las constantes vitales de los pacientes para un mejor pronóstico y seguimiento de las enfermedades. Además, la interconexión de los dispositivos de gestión médicos nos permite un mayor control de los historiales médicos y las comunicaciones entre diferentes centros para el estudio y análisis de enfermedades.

3.1.3.2. Smart Buildings

¿Un edificio que se gestiona a si mismo? Sí, esta es una de las muchas posibilidades que nos aporta IoT. Sensorizando las distintas instalaciones, podemos contar con termostatos inteligentes, alarmas, controles de acceso, climatización y seguridad frente a agentes como inundación o incendio, entre otros.

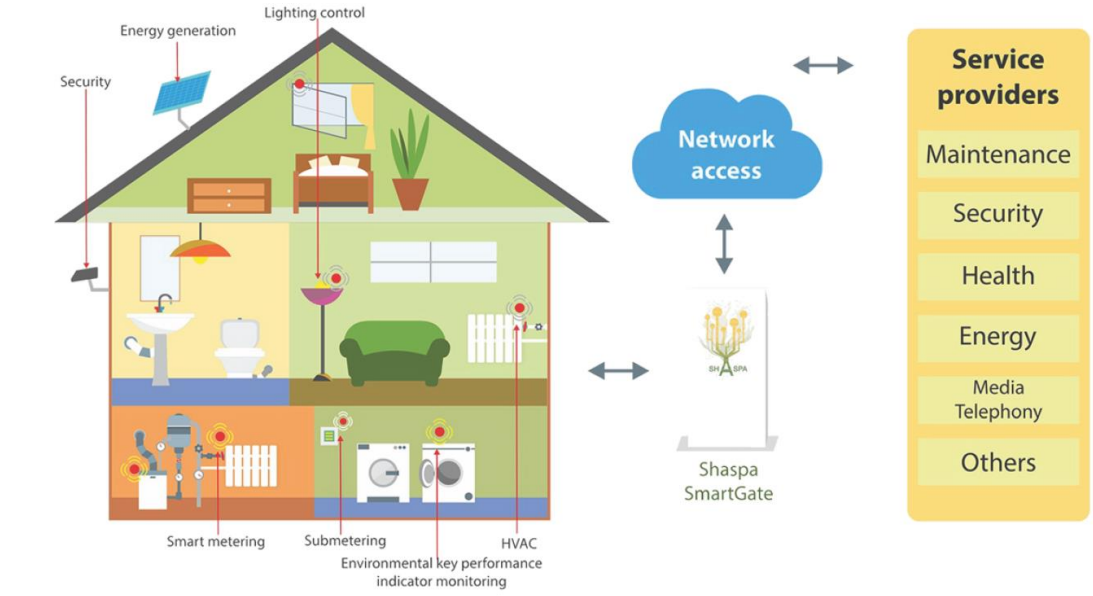

Figura 5: Equipos y aparatos integrados (1)

3.1.3.3. Smart Living

Muy extendido en nuestros días, el control de todo nuestro entorno por ejemplo a través del móvil. Nos permite medir los consumos de nuestra casa, controlar los aparatos de forma remota, y obtener análisis del estado de electrodomésticos y demás aparatos electrónicos que tengamos en nuestra red.

3.1.3.4. Smart Transports and Mobility

Este ámbito de aplicación abarca una amplia gama de secciones de transporte y logística. Gracias al Internet de las Cosas, contamos con una monitorización continua del estado y las condiciones de los envíos, así como el control de la disposición de almacenamiento y seguimiento de flotas. Los vehículos pueden contar con diagnósticos y comunicación automática, se puede hacer una gestión integral de la flota y del estado de carreteras y estaciones, ya que conoceremos sus características en todo momento gracias a la interconexión de los dispositivos.

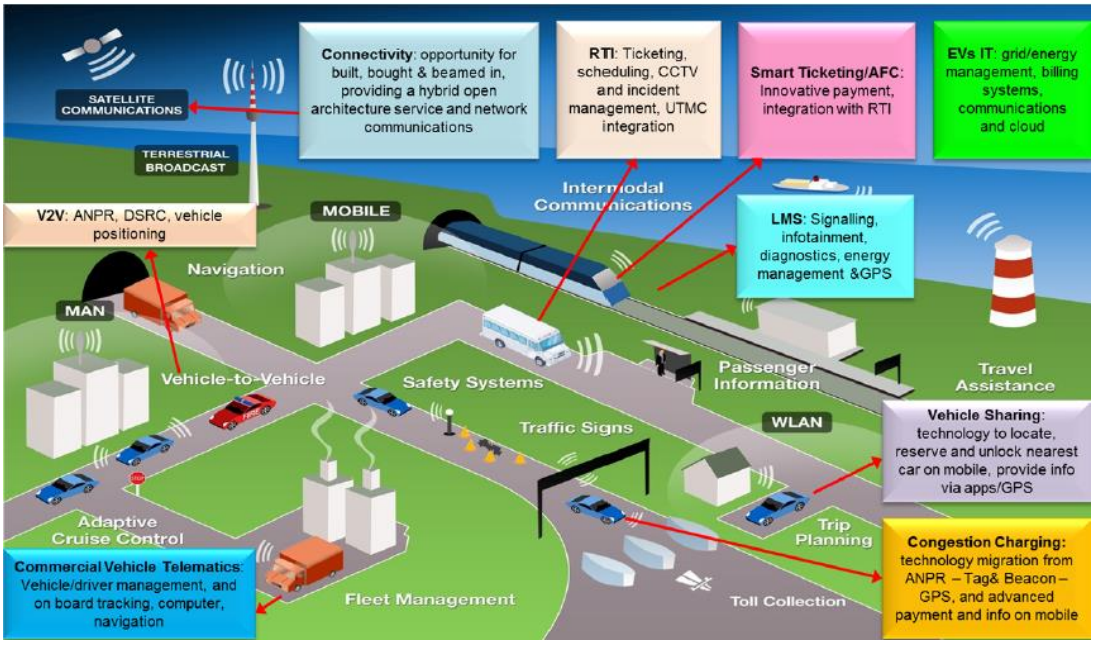

Figura 6: Ecosistema ITS (1)

3.1.3.5. Smart Industry

El papel del Internet de las Cosas es cada vez más dominante en las industrias, dando acceso a dispositivos y máquinas. Esto permite penetrar más en los sistemas de fabricación digitalizados. Manteniendo una monitorización de los procesos de fabricación para su constante análisis y toma de decisiones. Podríamos permitir la interacción de entidades externas con los sistemas de fabricación, como pueden ser proveedores de herramientas, encargados de logística y actores de mantenimiento y reacondicionamiento. El Internet de las Cosas está significando una gran revolución para los sistemas de fabricación modernos.

3.1.3.6. Smart City

Podemos definir una ciudad inteligente como una ciudad que cuenta con monitorización e integración de todas sus infraestructuras críticas, incluyendo carreteras, puentes, túneles, aeropuertos, comunicaciones, energía e incluso edificios, entre otros. Esto ayudará a optimizar mejor los recursos, planificar el mantenimiento preventivo y monitorear los aspectos de seguridad mientras se maximizan los servicios a los ciudadanos. Con este sistema de sensores integrados, se pueden recopilar, analizar y tomar decisiones que mejoren la gestión de la ciudad. Puede encontrarse un caso de éxito en la región en Villanueva de la Serena (7).

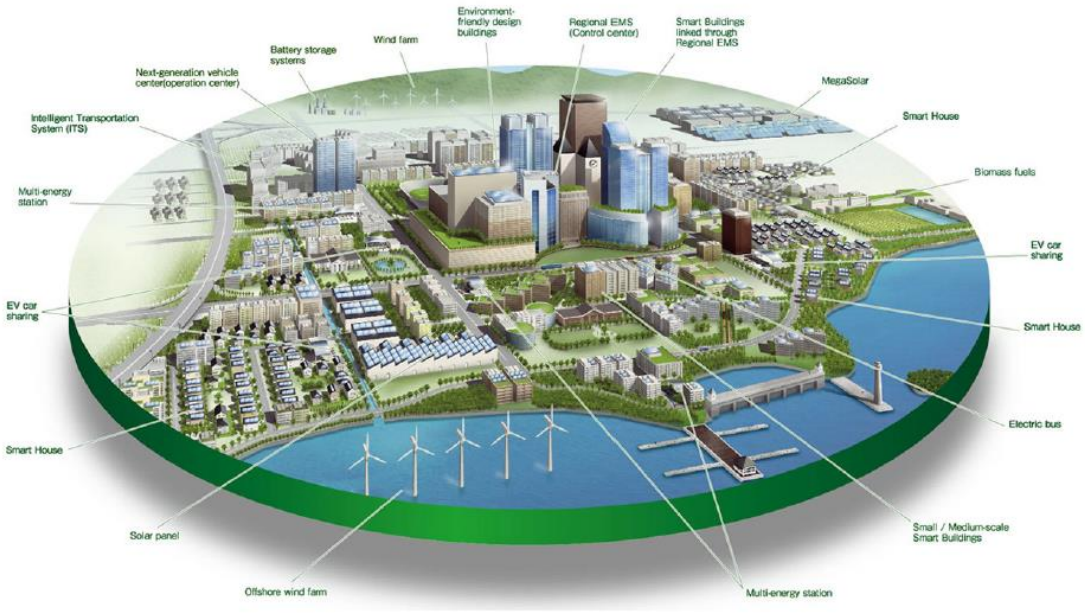

Figura 7: Concepto de Smart City (1)

3.1.4. Protocolos para IoT

En el Internet de las Cosas, encontramos una gran cantidad de dispositivos que necesitan conectarse entre sí. Además, cada uno de estos dispositivos realizan tareas diferentes, variando su velocidad de datos, tamaño, etc. Por esto, decimos que el sistema IoT es heterogéneo (8).

Para solucionar el problema de los dispositivos con características diferentes entre sí y poder conseguir que interactúen entre ellos, se han creado protocolos de comunicación específicos. Estos, son la piedra angular de cualquier sistema IoT. Crean un enlace heterogéneo entre los distintos dispositivos e implantan un sistema de comunicación aceptado por todos. Desde el punto de vista del cliente (9), todo el sistema puede ser abstraído y llamado como 'red de servicio', utilizando la tecnología que mejor se adapte al escenario seleccionado. Para el **mMTC** (Massive Machine Type Communications) (Comunicaciones de tipo máquina masiva), las tecnologías **LPWA** (Low-Power Wide-Area) (Baja-Potencia Área-Ancha) entran en juego en el mundo del Internet de las Cosas, pues en los dispositivos que se suelen utilizar se busca que necesiten el menor consumo de recursos posibles, tanto de potencia como de mensajería. Las tecnologías **LPWA** se caracterizan por una larga vida de las baterías, cobertura de comunicación ampliada y soporte para un elevado número de dispositivos. Según el **mMTC** este tipo de tecnologías representan más del **60%** del mercado de IoT.

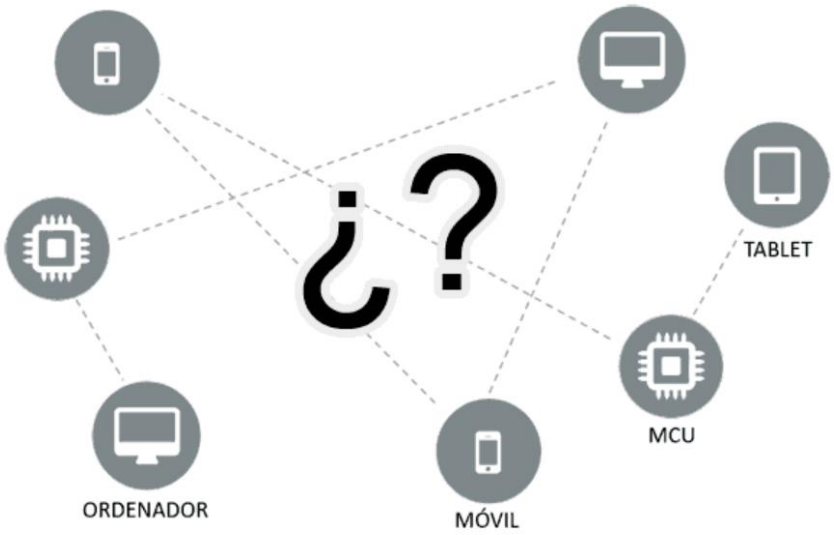

Figura 8: Interconexión de dispositivos (10)

Estos protocolos proporcionan la forma heterogénea de crear una comunicación **M2M** (Machine to Machine) (10). Gracias al impulso de las comunicaciones y el desarrollo de los sistemas para el Internet de las Cosas, muchas compañías han creado diferentes protocolos con características propias para poder utilizarlos según nuestras necesidades. Estos sistemas, deben cumplir las características básicas del Internet de las Cosas como son: alta escalabilidad, heterogeneidad, soportar cambios dinámicos, servicios relacionados con cosas y alta interconectividad. Además, contamos con el condicionante de la seguridad, pues los dispositivos utilizados estarán enlazados con internet, enviando información privada e incluso de control de sistemas físicos.

En resumen (1), los dispositivos interconectados en IoT necesitan comunicarse utilizando protocolos ligeros que no requieran de una gran cantidad de uso de recursos CPU. Hoy en día, existen dos arquitecturas dominantes para los protocolos de intercambios de datos: **basados en Bus (Bus-Based)** y **basados en Broker (Broker-Based)**.

3.1.4.1. Broker-Based

En este tipo de arquitectura (1), también llamada **Message Queue**, el Broker controla la distribución de la información. Entre otras cosas, almacena, reenvía, filtra

y prioriza las solicitudes de publicación y suscripción. Los dispositivos pueden cambiar entre publicador o suscriptor según sus objetivos. Algunos de los protocolos más famosos de este tipo son: **AMQP** (Advanced Message Queuring Protocol), **CoAP** (Constrained Applications Protocol), **MQTT** (Message Queue Telemetry Transport y **JMS** (Java Message Service).

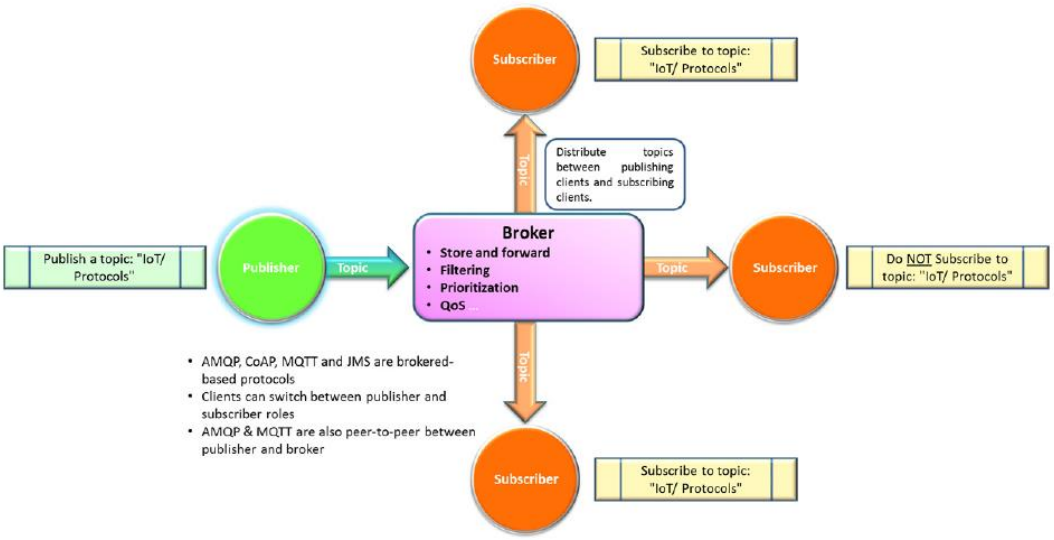

Figura 9: Protocolo basado en Broker para el intercambio de datos (1)

3.1.4.2. Bus-Based

En esta arquitectura (1), también llamada **Message Service**, los clientes publican mensajes para un tema específico y se entregan directamente a los suscriptores de ese tema. No hay broker centralizado. Algunos ejemplos de protocolos Bus-Based más famosos son: **DDS** (Data Distribution Service), **REST** (Representational State Transfer) y **XMPP** (Extensible Messaging and Presence Protocol).

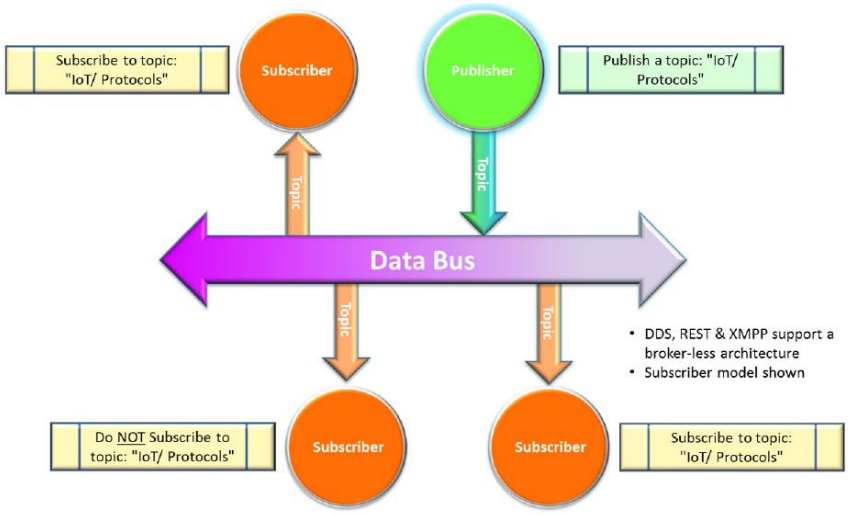

Figura 10: Protocolo basado en Bus para el intercambio de datos (1)

Una vez entendidas las diferencias entre las principales metodologías de comunicación que podemos encontrar en IoT, hablaremos en profundidad en los protocolos más usados actualmente:

3.1.4.3. CoAP

CoAP (Constrained Applications Protocol) (1) es un protocolo de transmisión de datos modelo cliente/servidor similar a **HTTP** ya que utiliza la tecnología **REST** (9). Un sensor es normalmente un 'servidor' de información y quien la consume es el 'cliente'. Admite protocolo one-to-one (uno-a-uno) para transmitir la información entre cliente y servidor.

CoAP utiliza el descubrimiento de contenido, esto permite a los dispositivos encontrarse unos a otros para el intercambio de datos. Fue diseñado para la interoperabilidad con la web, con protocolos como HTTP y RESTful como comentamos anteriormente, admitiendo comunicaciones asíncronas. Con este protocolo, encontramos facilidad para generar paquetes pequeños y se adapta a un modelo de transferencia de datos observando los cambios de estado a medida que ocurren, sin basarse simplemente en eventos.

Se utiliza el **UDP (User Datagram Protocol)** y admite la transmisión multidifusión. No siendo compatible con TCP, se realiza a través de datagramas sin conexión y se puede utilizar con otros protocolos de comunicación basados en

paquetes. Por lo general, UDP es más fácil de implementar que TCP, sin embargo, UDP no cuenta con las herramientas de seguridad necesarias para mantener una conexión fiable, como son **SSL o TLS**, pertenecientes a TCP.

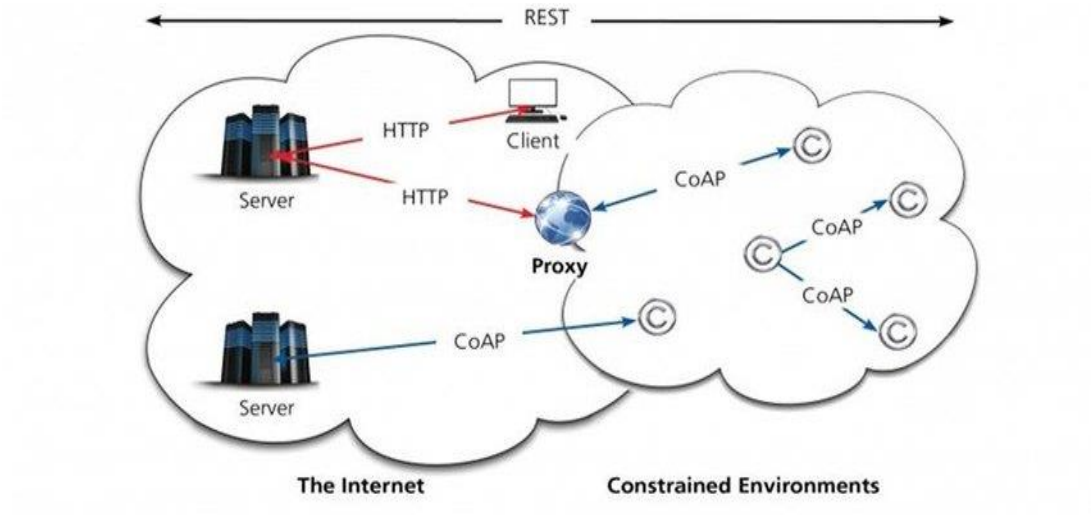

Figura 11: Protocolo CoAP, REST (11)

3.1.4.4. AMQP

Es un protocolo centrado en la capa de aplicación, surgido en el sector financiero con la finalidad de reemplazar los sistemas de mensajería no interoperables (1). Los puntos clave de **AMQP** son la orientación de mensajes, colas, enrutamiento (ya sea punto a punto o publicación/suscripción), fiabilidad y seguridad. Todo esto a través de un broker centralizado.

Proporciona comunicación orientada a mensajes y controla el flujo con garantías de entrega, que pueden ser de máximo una vez (donde cada mensaje se entrega una vez o ninguna), al menos una vez (donde cada mensaje se entrega con seguridad, pero puede hacerlo más de una vez) y exactamente una vez (donde el mensaje siempre llegará con seguridad una sola vez). Cuenta con autentificación y cifrado basado en **SASL y TLS**. Es un protocolo confiable gracias a la utilización de **TCP (Transmission Control Protocol)** que cuenta con **SSL** y **TLS**.

El protocolo establece el comportamiento del servidor que provee los mensajes y del cliente, de esta forma las implementaciones de diferentes proveedores

son realmente interoperables. A diferencia de JMS, que solo utiliza una API, AMQP es un protocolo a nivel de cable. Este protocolo se utiliza principalmente para fines corporativos, no resultando adecuado para aplicaciones de IoT con dispositivos de bajos recursos (10).

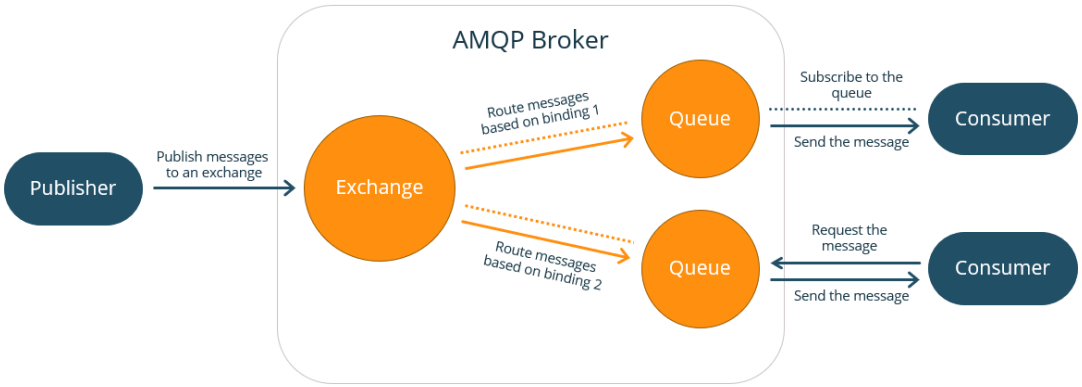

Figura 12: Protocolo AMQP (12)

3.1.4.5. MQTT

Es un protocolo de código abierto para el intercambio de mensajes entre múltiples clientes, a través de un intermediario central (1). Se diseñó para ser simple y fácil de implementar. Su arquitectura está basada en un Broker central y utiliza una conexión **TCP** de larga duración. MQTT, admite estructura jerárquica de sistema de archivos. Se puede utilizar para comunicaciones bidireccionales en redes poco fiables, donde el coste por bit transmitido es alto y con dispositivos de bajo consumo.

Es un protocolo ligero y muy adecuado para entornos con limitaciones. Suele utilizar la comunicación asíncrona y puede comunicar mensajes de desconexión automáticos. Existen diversas versiones de MQTT para abordar limitaciones específicas. Al utilizar TCP, contamos con seguridad **SSL/TLS**, aunque esto puede afectar negativamente a la eficiencia de la red cuando el número de nodos es superior a mil, provocando baja eficiencia. El protocolo no acepta descubrimiento automático de nuevos clientes.

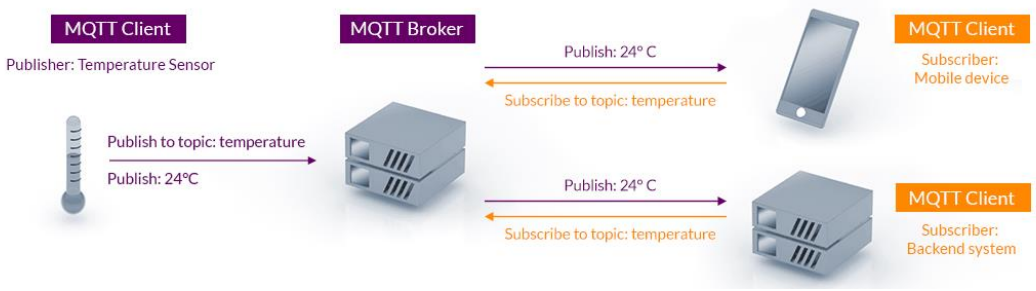

Figura 13: Protocolo MQTT

3.1.4.6. JMS

JMS, es una API middleware orientada a crear, leer, enviar y recibir mensajes entre dos o más clientes. Basado en **Java Enterprise**, está destinado a separar las funciones de la capa de transporte y aplicación, permitiendo la comunicación entre los diferentes componentes para una comunicación débilmente acoplada, fiable y asíncrona sobre **TCP/IP** (1).

Este protocolo admite sistemas de publicación/suscripción y punto a punto utilizando colas de mensajes. Las diferentes clases de Java se pueden utilizar para comunicarse con diferentes proveedores de JMS utilizando **Java Naming** and **Directory**.

Al utilizar API de JMS, hay que tener en cuenta que no se puede garantizar la interoperabilidad entre clientes y servidores que utilicen diferentes implementaciones de JMS. Además, en sistemas de más de mil nodos, podemos encontrar problemas de rendimiento y complejidad.

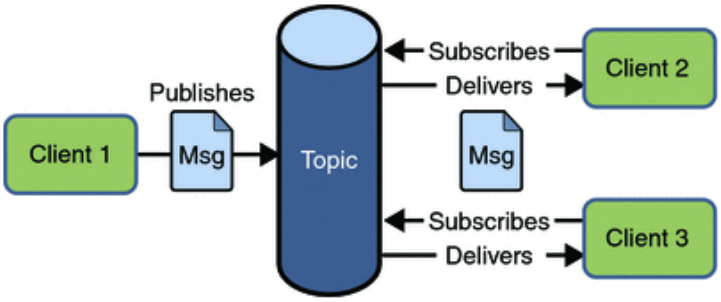

Figura 14: Protocolo JMS (Pub/Sub) (13)

3.1.4.7. DDS

Es un lenguaje de datos que se utiliza para permitir el intercambio de datos, escalable en tiempo real, de alto rendimiento e interoperable. Se diseñó para el comercio financiero, control de tráfico aéreo, gestión de redes y otras aplicaciones críticas de big data (1).

Es un protocolo descentralizado, sin intermediarios, con comunicaciones directas de igual a igual entre publicadores y suscriptores. Independiente del lenguaje y del sistema operativo. **DDS** envía y recibe información sobre **UDP**, aunque también puede utilizarse con **IP Multicast**, **TCP/IP**, **memoria compartida**, etc. Soporta conectividad de gran cantidad de dispositivos en tiempo real y con descubrimiento. Las aplicaciones con DDS están desacopladas y no requieren de aplicaciones de usuario, esto puede simplificar la compleja programación de la red. Cuenta con parámetros **QoS** para garantizar la entrega de los mensajes.

Aún están pendientes las especificaciones de seguridad en DDS, teniendo en cuenta que necesita **DSSI** (protocolo de cable) para asegurar la interoperabilidad.

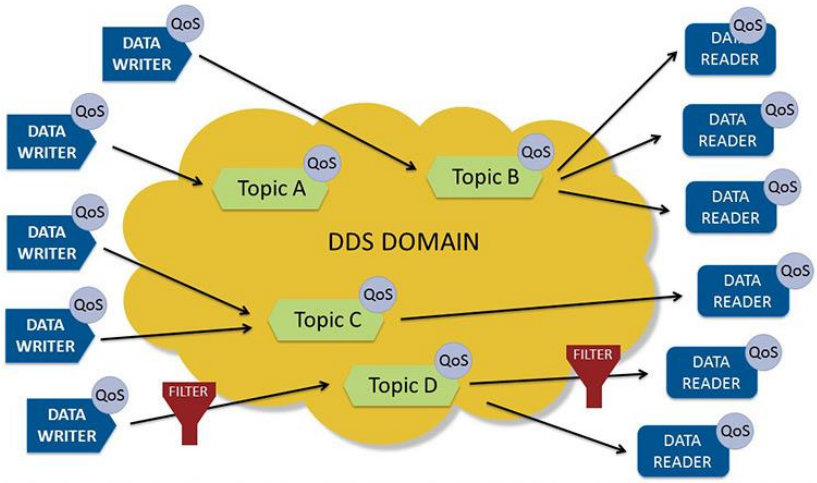

Figura 15: Protocolo DDS (14)

3.1.4.8. XMPP

XMPP es un protocolo middleware orientado a mensajes de texto **XML** (1). Cuenta con un modelo cliente-servidor descentralizado sin intermediarios, utilizado por aplicaciones de mensajería de texto.

Es casi en tiempo real y escalable masivamente a cientos de miles de nodos. Los datos binarios deben estar codificados en **base64** antes de ser transmitidos. Se utiliza para dispositivos con gran tráfico, potencialmente complicado y que requieren seguridad adicional. Puede aislar la seguridad sin necesidad de **TCP** o de la web.

Los usuarios mantienen el control a través de configuración de las preferencias, se están añadiendo nuevas extensiones, entre ellas, el transporte sobre **HTTP**.

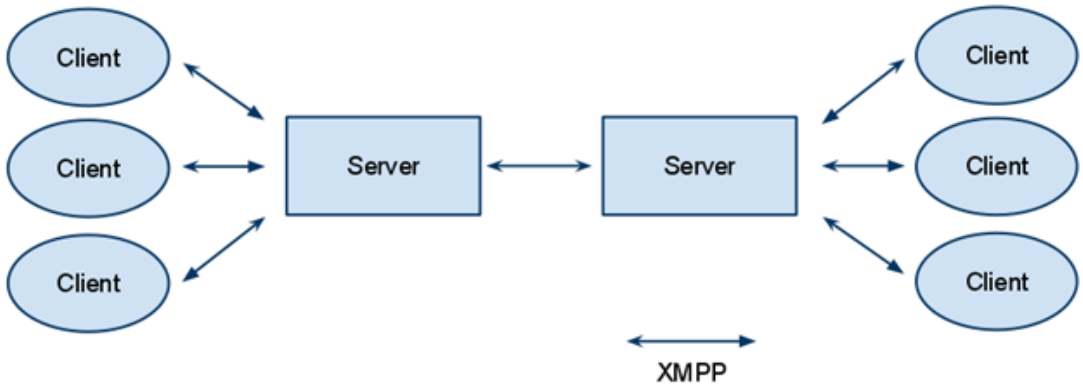

Figura 16: Protocolo XMPP (15)

3.2. MQTT

3.2.1. Introducción

MQTT (16), es un protocolo de transporte de mensajería del tipo publicación/suscripción entre cliente y servidor. Es abierto, simple, liviano debido a su mínima sobrecarga de paquetes y está diseñado para que sea fácil de implementar. Estas características lo hacen ideal para su uso en muchas situaciones, incluidos entornos restringidos, la comunicación **M2M** (Maquina a máquina) e Internet de las Cosas (**IoT**), donde se necesita una pequeña huella de código y el ancho de la red disponible es escaso. (17) Es un protocolo que sobresale cuando se transfieren datos por cable en comparación con protocolos como HTTP, siendo muy fácil de implementar por parte del cliente.

Este protocolo fue diseñado en 1999 por **Ando Stanford-Clark** (IBM) y **Arlen Nipper** (Arcom, actualmente Cirrus Link) (17). Se buscaba contar con un sistema con mínima pérdida de batería y ancho de banda, para conectarse con oleoductos vía satélite. Se establecieron cinco requisitos principales en el momento de su creación:

- Implementación simple.
- Calidad de servicio en la entrega.
- Ligero y eficiente en ancho de banda.
- Independiente de los datos.
- Conciencia continua de sesión.

Aunque estos conceptos sigan siendo los pilares del protocolo, el enfoque principal ha cambiado, de sistemas integrados propietarios a casos de uso abiertos de Internet de las Cosas. De este cambio, encontramos la confusión generalizada de lo que significa el nombre MQTT, ya que en un principio las siglas significaban ‘MQ Telemetry Transport’. Debido a esta confusión, muchas fuentes etiquetan a este protocolo del tipo cola de mensajes, que, aunque sea posible utilizar este sistema, no es el funcionamiento principal de MQTT.

Actualmente, el protocolo **MQTT (Message Queue Telemetry Transport),** está estandarizado bajo la organización **OASIS** (18), que tiene como propósito promover estándares. La estandarización se aprobó oficialmente el 29 de octubre de 2014. Hoy en día, la última versión disponible del protocolo es 'MQTT 5', introduciendo mejoras para la implementación con Internet de las Cosas utilizando sistemas en la nube, requiriendo mayor confiabilidad y manejo de errores.

3.2.2. Estructura de comunicación

El protocolo MQTT se basa en **TCP/IP** (19). Tanto el cliente como el intermediario deben contar con la pila **TCP/IP**. Si buscamos cuadrar el protocolo en el modelo OSI de siete capas (para una mejor comprensión), este se encontraría ocupando las capas de sesión, presentación y aplicación, contando con TCP en la capa de transporte e IP en la de red.

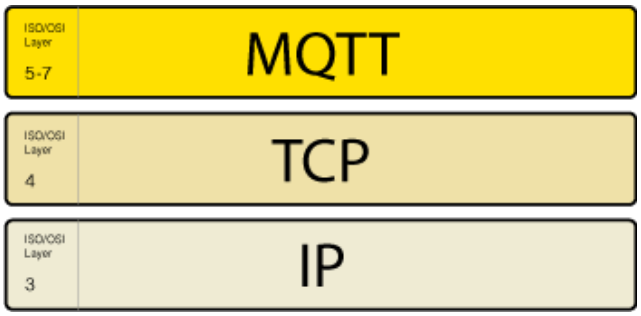

Figura 17: MQTT en modelo OSI (19)

Si profundizamos en el protocolo (16), dando un paso más y analizando la estructura de sus mensajes, nos encontramos con que MQTT opera intercambiando una serie de paquetes de control de manera definida. Estos paquetes constan de tres partes, siempre con el mismo orden: encabezado fijo (siempre presente), encabezado variable (presente en algunos paquetes) y carga útil (presente en algunos paquetes).

Fixed Header, present in all MQTT Control Packets
Variable Header, present in some MQTT Control Packets
Payload, present in some MQTT Control Packets

Figura 18: Estructura de un paquete de control MQTT (16)

Los campos de texto de los paquetes de control (carga útil), están codificados con cadenas **UTF-8**, una codificación eficiente de caracteres **Unicode**, que optimiza la codificación de caracteres **ASCII** para las comunicaciones basadas en texto. Cada una de estas cadenas cuentan con un campo de longitud entera de dos bytes como prefijo, de la siguiente forma:

Bit	7	6	5	4	3	2	1	0
byte 1	String length MSB							
byte 2	String length LSB							
byte 3 ….	UTF-8 encoded character data, if length > 0.							

Figura 19: Estructura de cadenas codificadas en UTF-8 (16)

La carga útil del mensaje (20), también denominada tema, es lo que utiliza el Broker para filtrar los mensajes. Cada tema consta de uno o más niveles, cada uno separado por una barra inclinada, como por ejemplo:

"MiCasa / PlantaBaja / Salon / Temperatura"

Estos temas, son muy ligeros en comparación con una cola de mensajes. El cliente no necesita crear el tema antes de publicarlo, el Broker acepta cualquier tema válido sin inicialización previa. Existen comodines para la suscripción a varios temas de forma simultánea, estos se explicarán más adelante en la sección de implementación.

Es importante tener en cuenta, que MQTT cuenta con temas reservados para estadísticas internas del servidor. Los clientes no pueden publicar sobre estos temas. Por lo general, se utiliza "$ SYS" para nombrarlos.

3.2.3. Patrón Publicación/Suscripción

(21) Los patrones de publicación/suscripción (Pub/Sub) proporcionan una alternativa a la arquitectura cliente-servidor, en la que el cliente se comunica directamente con el punto final. El modelo Pub/Sub, desacopla al cliente (editor) que envía el mensaje del que lo recibe (suscriptor). Gracias a esto, los clientes nunca se comunican directamente entre sí, es más, no son conscientes de la existencia el uno del otro.

La comunicación entre los distintos clientes se realiza a través de un Broker centralizado, el cual filtra los mensajes entrantes (publicaciones) y los distribuye entre los suscriptores. Esto es el aspecto más importante de pub/sub, ya que consigue un desacople total entre publicador y suscriptor. Esta conexión intermedia con el Broker, permite una mejor escalabilidad del sistema, gracias a que las operaciones en el intermediario se pueden paralelizar y los mensajes se procesan de forma controlada por eventos.

Los principales aspectos pub/sub que podemos encontrar en MQTT son:

- **Desacople espacial:** para publicar y recibir mensajes, los clientes solo necesitan conocer el nombre host/IP y el puerto del Broker
- **Desacople temporal:** aunque la mayoría de los mensajes se entregan casi en tiempo real, el Broker puede almacenar mensajes para clientes que no están en línea.
- **Funcionamiento asíncrono:** las tareas no se bloquean mientras esperan a que se reciba o transmita un mensaje. Esto es gracias a que la mayoría de las bibliotecas cliente funcionan de manera asíncrona, aunque también se pueda implementar un modelo de trabajo síncrono.

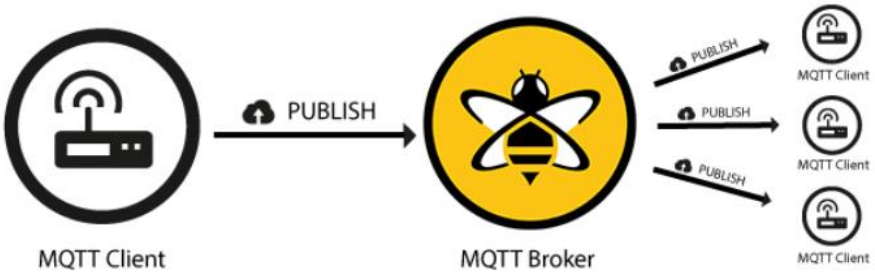

Figura 20: Estructura Pub/Sub MQTT

3.2.4. Broker

El broker (19), es el corazón de cualquier protocolo de publicación/suscripción y con MQTT no iba a ser diferente. Dependiendo de la implementación, el Broker puede controlar hasta millones de clientes simultáneamente.

El Broker, es el encargado de recibir todos los mensajes, filtrarlos, determinar quién está suscrito a ellos y enviar el mensaje a los clientes suscritos. También guarda los datos de sesión de todos los clientes con sesiones persistentes, suscripciones y mensajes perdidos. Se encarga de la autorización, autenticación e integración de los sistemas de backend. Esta integración es muy relevante, ya que el intermediario (broker) es el componente del sistema que se encuentra directamente conectado a internet, pues además de transmitir la información entre clientes, también se encarga de pasar los datos recopilados a los sistemas de procesado y análisis que se encuentran en el backend.

A la hora de implementar el broker, nos encontramos con gran variedad de programas que nos pueden ayudar a recepcionar y distribuir los mensajes, cada uno con sus características propias. La elección de un broker adecuado a nuestras necesidades (22), condiciona enormemente el funcionamiento de nuestro sistema. A continuación, se describen algunos de los principales brokers Open Source disponibles para MQTT:

3.2.4.1. Mosquitto

Eclipse Mosquitto es un intermediario de mensajes de código abierto con licencia **EPL/EDL** (23). Es liviano y adecuado para su uso en la mayor parte de dispositivos, desde computadoras de baja potencia hasta servidores completos. Los clientes MQTT se implementan a través de una biblioteca en **C**. Mosquitto es parte de la **Fundación Eclipse**.

3.2.4.2. Mosca

Es un broker **Open Source** para **Node.js**. Desarrollado por **Matteo Collina**. Se emplea como aplicación independiente o embebido para proyectos Node.js (22).

3.2.4.3. Aedes

También se utiliza, para proyectos **Node.js**. Diseñado por **Matteo Collina** se creó para reemplazar a Mosca

3.2.4.4. HBMQTT

Proporciona una API sencilla basada en corrutinas (24), facilitando una escritura de aplicaciones altamente concurrentes. Se diseñó sobre el marco E/S asíncrono estándar de **Python**. Acepta los mismos requisitos de MQTT, como pueden ser QoS, autenticación, temas restringidos, SSL, websocket, etc.

3.2.4.5. EMQTT

EMQ (Erlang MQTT Broker) es un broker MQTT de código abierto escrito en **Erlang/OTP** (25). **Erlang/OTP** es una plataforma de programación concurrente, tolerante a fallos, casi en tiempo real y distribuida. El proyecto EMQ tiene como objetivo implementar un agente MQTT escalable, confiable y de nivel empresarial para aplicaciones de mensajería móvil, hardware inteligente, M2M e IoT.

3.2.4.6. RabbitMQ

Es un corredor de mensajes de código abierto liviano y fácil de implementar en los dispositivos y en la nube. (26) Admite múltiples protocolos de mensajería. RabbitMQ se puede implementar en configuraciones distribuidas y federadas para cumplir con los requisitos de alta disponibilidad y gran escala. Ejecuta muchos sistemas operativos y muchos idiomas de programación populares como: **Java, .NET, Ruby, Python, PHP, Node,** etc.

3.2.4.7. HiveMQ

(27) Es un agente MQTT y plataforma de mensajería basada en el cliente, diseñada para movimiento rápido, eficiente y confiable de datos entre dispositivos IoT conectados. Su piedra angular es el protocolo MQTT para el envío instantáneo y bidireccional de datos entre los dispositivos y los sistemas.

Se diseñó principalmente para aplicaciones críticas de negocio confiables y escalables, entrega de datos rápida, uso eficiente de hardware, red y de recursos en la nube.

3.2.5. Cliente.

En MQTT, todos los dispositivos conectados, son considerados clientes. Para diferenciarlos, se utilizan las etiquetas de editor y suscriptor según el cliente esté recibiendo mensajes o publicándolos (19). Por lo tanto, un cliente MQTT es cualquier dispositivo (desde un microcontrolador a un servidor completo) que ejecuta una biblioteca MQTT y se conecta a un agente MQTT a través de la red.

Las bibliotecas de cliente MQTT, están disponibles para una gran variedad de lenguajes de programación como: Android, C, C++, Arduino, Go, iOS, Java, JavaScript y .NET.

3.2.6. Calidad del servicio.

La calidad de servicio (**QoS**) es un acuerdo entre el receptor y el remitente de un mensaje que define la garantía de entrega específica de este. Es una característica principal de MQTT, dándole al cliente la posibilidad de elegir el nivel de servicio en función de la confiabilidad de la red y la aplicación. Ya que el protocolo gestiona el intercambio y garantiza la entrega de los mensajes, **QoS** hace que la comunicación en redes no confiables sea más fácil de implementar (28).

Hay tres niveles de QoS disponibles en MQTT: como máximo una vez (0), al menos una vez (1) y exactamente una vez (2). Para una comprensión completa del sistema de calidad de entrega, debemos tener en cuenta los dos lados de entrega del mensaje. Entrega de mensajes desde el cliente de publicación al broker y la entrega del mensaje desde el broker al cliente suscriptor.

El cliente emisor del mensaje utiliza un nivel específico de QoS al enviarlo al broker. Este, retransmite el mensaje a los clientes suscritos basándose en el nivel de QoS de los clientes receptores. Cuando emisor y receptor especifican niveles diferentes de calidad de entrega, el broker utiliza siempre la calidad de servicio más baja.

3.2.6.1. QoS 0 – Como máximo una vez

Es el mínimo nivel QoS del que dispone el protocolo. Este nivel garantiza una comunicación con el mejor esfuerzo, sin haber garantía de entrega. Cuando el emisor

envía el mensaje, el receptor no reconoce la recepción de este. Tampoco se almacena ni retransmite de nuevo. Este nivel se denomina “disparar y olvidar” y solo se ajusta a la garantía de entrega con la que cuenta el protocolo subyacente, es decir, TCP (28).

Figura 21: QoS 0 (28)

3.2.6.2. QoS 1 – Al menos una vez

Este nivel garantiza que el mensaje ha sido recibido al menos una vez por el receptor. El emisor almacena el mensaje hasta que recibe el paquete de confinación (**PUBACK**) de recepción. De esta forma, se garantiza la entrega, pero es posible que el mensaje se envíe y entregue varias veces.

Para la comunicación de recepción entre emisor y receptor, se utiliza el identificador de paquete. Si el remitente no recibe una confirmación en un periodo de tiempo razonable, vuelve a enviar el paquete. Si esto sucede y se vuelve a enviar el mensaje, se establece el indicador de duplicado (**DUP**), que se utiliza para fines internos y no es procesado por broker o cliente (28).

Figura 22: QoS 1 (28)

3.2.6.3. QoS 2 – Exactamente una vez

QoS 2 es el nivel de servicio más alto de MQTT. Aquí, se garantiza que cada mensaje sea recibido solo una vez por los receptores. Es el nivel más alto, pero a la vez el más lento debido a la transmisión que se lleva a cabo. La garantía se proporciona por al menos dos flujos de solicitud/respuesta (Four-Part HandShake) entre remitente y receptor. Como en QoS 1, se utiliza el identificador de paquete del mensaje de publicación (PUBLISH).

Cuando el receptor recibe un mensaje con QoS 2, procesa el mensaje de publicación y responde al remitente con un **PUBREC** que reconoce el paquete. Si el emisor no recibe este paquete, se envía de nuevo el mensaje con el indicador de duplicado (**DUP**) hasta que acuse el recibo. Una vez se ha recibido el mensaje PUBREC del receptor, se puede descartar el paquete inicial. Almacenando la confirmación y respondiendo con un paquete **PUBREL**.

Cuando el receptor recibe el **PUBREL**, descarta todos los estados almacenados y responde con un **PUBCOMP**, de la misma forma el emisor al recibir este mensaje devolverá el mismo paquete **PUBCOMP**. Este paso es muy importante para garantizar que la entrega se realiza solo una vez (28).

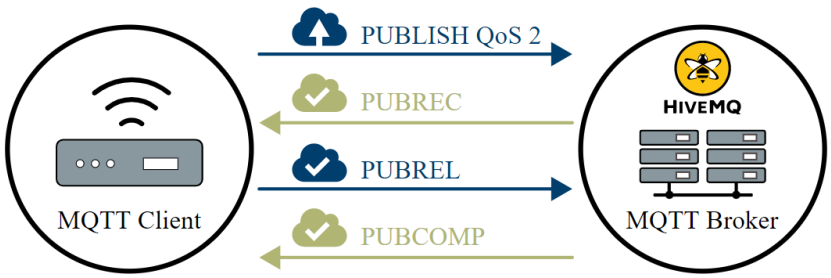

Figura 23: QoS 2 (28)

3.3. Microservicios

Los microservicios, son un enfoque de arquitectura en el que una sola aplicación se compone de muchos servicios más pequeños, poco acoplados y que pueden implementarse de forma independiente (29).

Normalmente, estos servicios tienen su propia pila de tecnologías y se comunican entre si a través de **API REST**. Esto permite a las organizaciones actualizar el código más fácilmente, añadiendo nuevas funcionalidades sin modificar el resto de la aplicación o escalar los componentes de manera independiente unos de otros.

La principal ventaja de la **arquitectura de microservicios**, es la de evitar pérdidas de tiempo y frustraciones a la hora de implementar nuevas partes de código o prestaciones para un sistema. Este modelo permite a una organización crear pequeños equipos multifuncionales entorno a un servicio o colección de servicios y hacer que operen de manera ágil. Al dividir una aplicación en servicios más pequeños que trabajan en conjunto, se consigue un grado de aislamiento frente a fallos, una mejor capacidad de recuperar aplicaciones y combinando esto con el pequeño tamaño de los servicios, unos límites claros y patrones de comunicación eficaz, facilita que nuevos equipos de trabajo contribuyan rápidamente al desarrollo de la aplicación.

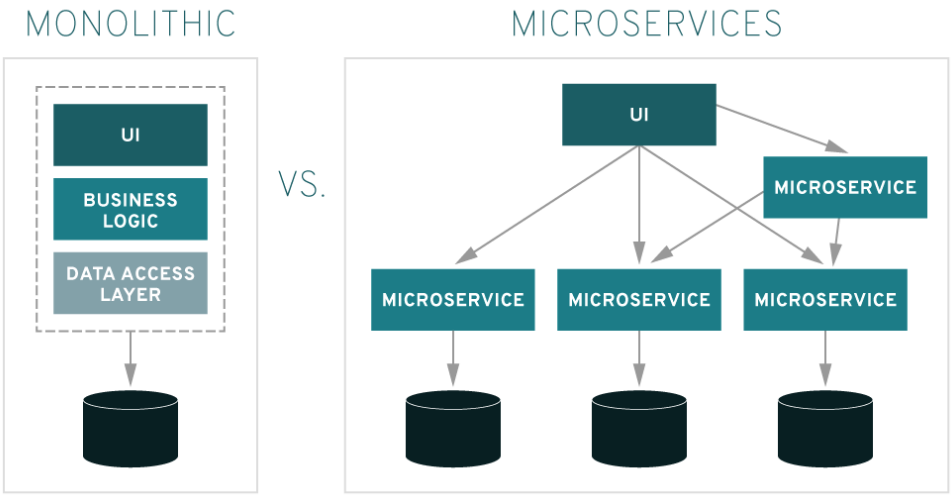

Figura 24: Arquitectura monolítica VS microservicios (30)

En la arquitectura tradicional **monolítica**, existen patrones de n niveles. Normalmente, la aplicación comparte una gran base de datos para todo el sistema y una pila común. Contando con un fuerte acoplamiento y una dificultad añadida a la hora de implementar nuevos cambios o desarrollos dentro del sistema.

Por el contrario, en la arquitectura de microservicios, los componentes son independientes manteniendo una comunicación **REST** entre ellos, permitiendo optimizar la pila de cada servicio individualmente. Los microservicios requieren menos infraestructura que las aplicaciones monolíticas ya que permiten un escalado preciso e individualizado. Este modelo de implementación también cuenta con sus desafíos, ya que requiere mayor complejidad de gestión, más servicios controlados por muchos equipos e incluso probablemente implementados en zonas geográficas distintas.

Este tipo de tecnologías, se describen como optimizadas para **DevOps** e integración/entrega continua (**CI/CD**). Ya que, dado el aumento masivo de la complejidad al implementar esta arquitectura, las partes móviles y dependencias que conlleva, hay que hacer inversiones significativas en implementación, monitoreo y automatización del ciclo de vida.

3.3.1. Docker

En 2013, Docker introdujo lo que se convertiría en el estándar de la industria para los contenedores. Los contenedores son una unidad estandarizada de software que permite a los desarrolladores aislar la aplicación del entorno, evitando el problema de incompatibilidad entre unas máquinas y otras (31).

Un contenedor es una unidad estándar de software que empaqueta el código y todas sus dependencias, para que la aplicación se ejecute de forma rápida y confiable en un entorno informático u otro. Una imagen de contenedores Docker es un paquete de software ligero que incluye todo lo necesario para ejecutar una aplicación: tiempo de ejecución, código, bibliotecas y herramientas del sistema (32).

Debido a que los contenedores no tienen la sobrecarga de su propio sistema operativo, son más pequeños y livianos que las máquinas virtuales tradicionales. Esto los convierte en servicios más pequeños y livianos que se integran perfectamente con las arquitecturas de microservicios. A continuación, se exponen algunas diferencias entre máquinas virtuales y contenedores:

Máquinas virtuales

Las máquinas virtuales (VM) son una abstracción del hardware físico que convierte un servidor en muchos servidores. Cada máquina virtual incluye una copia completa de un sistema operativo; la aplicación, los archivos binarios, las bibliotecas. Esto puede ocupar decenas de GB y pueden ser algo lentas para arrancarlas.

Contenedores

Los contenedores son una abstracción en la capa de la aplicación, que empaqueta el código y las dependencias juntos. Pueden ejecutarse varios contenedores en una misma máquina y compartir el kernel del sistema operativo, cada uno ejecutándose como procesos aislados. Ocupan menos espacio que las máquinas virtuales (alrededor de las decenas de MB).

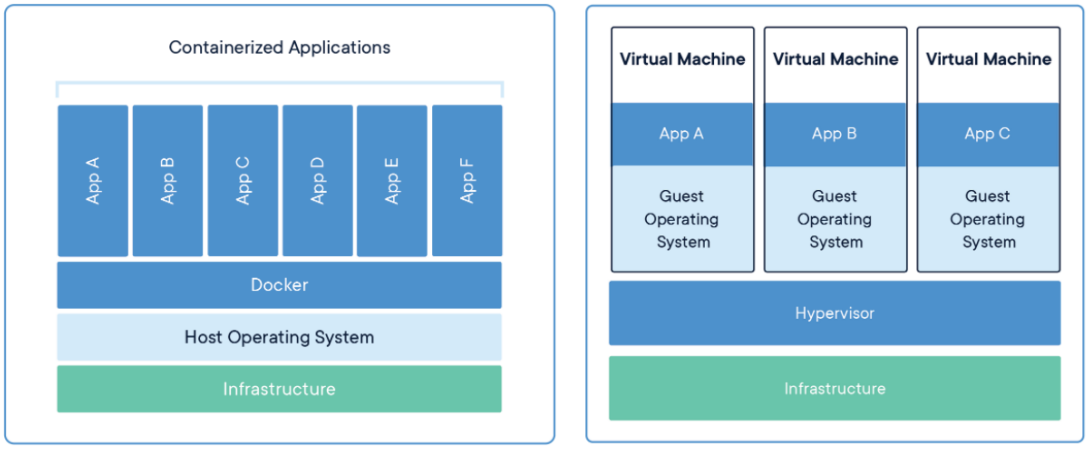

Figura 25: Contenedores VS máquinas virtuales (32)

3.4. Time series database (TSDB)

Como su propio nombre indica, una base de datos de series de tiempo (TSDB) es una base de datos optimizada para datos con marcas de tiempo o series de tiempo. Estos datos, son simplemente mediciones o eventos que se rastrean, monitorean, reducen y agregan a lo largo del tiempo. Pueden ser métricas de servidor, monitoreo de aplicaciones, datos de red, dato de sensores o muchos otros tipos de datos analíticos (33).

Los datos de series de tiempo, también denominados datos con marcas de tiempo, son una secuencia de puntos de datos indexados en orden de tiempo. Los datos con sello de tiempo se recopilan en diferentes momentos. Estos puntos de datos suelen

consistir en mediciones sucesivas, realizadas desde la misma fuente durante un intervalo de tiempo y se utilizan para realizar algún seguimiento de los cambios a lo largo del tiempo (34).

En sus inicios, las TSDB se utilizaban principalmente para analizar datos financieros, acciones y negociación. Esto ha cambiado con las numerosas aplicaciones que se han generado en la industria. Las condiciones de la informática han cambiado drásticamente en la última década, todo se ha dividido en funcionalidades, todo lo que puede ser un componente por sí solo, es un componente, ajustándose así al modelo de arquitectura de microservicios. Estamos siendo testigos de la instrumentación de todas las superficies, todo tiene o tendrá un sensor. Por lo que ahora todo fuera y dentro de la empresa está emitiendo un flujo de métricas y eventos o datos de series de tiempo.

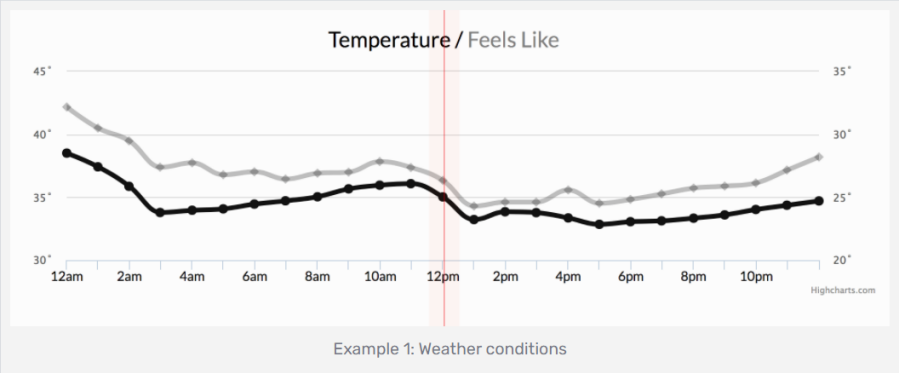

Figura 26: Ejemplo de datos de series de tiempo

Las bases de datos de series de tiempo, tienen propiedades clave de diseño que las hacen muy diferentes a otras bases de datos. Entre ellas están el almacenamiento y la compresión de datos con marcas de tiempo, la gestión del ciclo de vida de los datos, el resumen de los datos, la capacidad de manejar exploraciones dependientes de series de tiempo grandes con muchos registros y las consultas con reconocimiento de series de tiempo.

3.4.1. Prometheus

Prometheus es un conjunto de herramientas de monitoreo y alerta de sistemas de código abierto construido originalmente por SoundCloud. Desde que sus inicios

en 2012 muchas empresas y organizaciones han optado por utilizar Prometheus. El proyecto cuenta con una comunidad de usuarios y desarrolladores muy activa (35).

Actualmente, es un proyecto de código abierto independiente y se mantiene al margen de cualquier empresa. Prometheus se unión a la Cloud Native Computing Foundation en 2016 como el segundo proyecto aloja, después de Kubernetes.

Prometheus recopila y almacena sus métricas como datos de series de tiempo junto con pares clave-valor denominados etiquetas.

3.4.1.1. Arquitectura

Prometheus trabaja extrayendo métricas de trabajos organizados, ya sea directamente o mediante una puerta de enlace "push" intermedia para trabajos de corta duración. Almacena todas las muestras extraídas localmente y ejecuta reglas sobre estos datos para agregar y registrar nuevas series de tiempo a partir de datos existentes o generar alertas. Prometheus se puede complementar con muchos consumidores de API como Grafana para visualizar los datos recopilados.

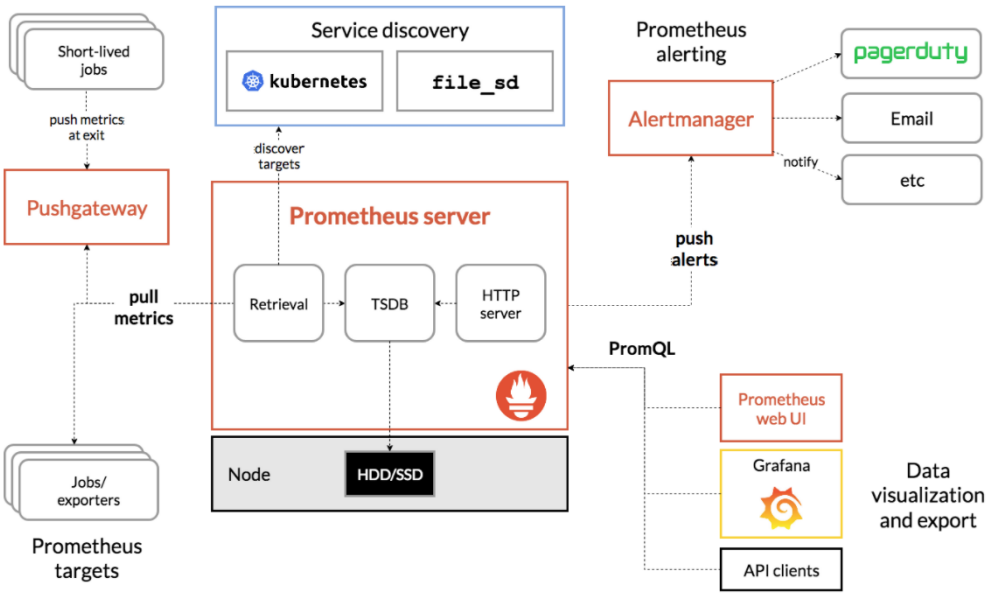

Figura 27: Ecosistema de Prometheus (35)

3.4.1.2. Almacenamiento

Prometheus incluye una base de datos de series de tiempo local en disco, aunque también se puede integrar con sistemas de almacenamiento remoto (36).

Almacenamiento local

Las muestras ingeridas se agrupan en bloques de dos horas. Cada uno de estos bloques consta de un directorio que contiene un subdirectorio de fragmentos con todas las muestras de series de tiempo para ese rango de tiempo, un archivo de metadatos y un archivo de índice. Las muestras en el directorio de fragmentos se agrupan en uno o más archivos de hasta 512 MB. Cuando las series se eliminan a través de la API, los registros de eliminación se almacenan en archivos de desecho separados (en lugar de eliminarlos de inmediato).

El bloque actual para las muestras entrantes se mantiene en memoria y no se conserva por completo. Está protegido frente a fallos mediante un registro de escritura anticipada (WAL). Estos archivos de escritura anticipada se almacenan en el directorio 'wal' en segmentos de 128 MB. Contienen datos sin procesar que aún no se han compactado, por lo que son más grandes que los archivos de bloques normales. Prometheus conserva un mínimo de tres archivos de registro de escritura anticipada para mantener al menos dos horas de datos sin procesar.

Almacenamiento remoto

Prometheus puede integrarse con sistemas de almacenamiento remoto de tres formas:

- Puede escribir muestras que ingiere en una URL remota en formato estandarizado.
- Puede recibir muestras de otros servidores Prometheus en formato estandarizado.
- Puede leer datos de muestra de una URL remota en formato estandarizado.

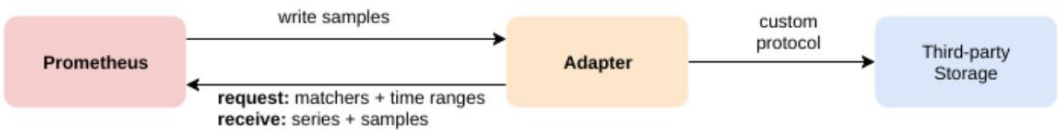

Figura 28: Almacenamiento remoto Prometheus (36)

3.4.1.3. Consultas

Prometheus proporciona un lenguaje de consulta funcional llamada PromQL (Prometheus Query Language) que permite al usuario seleccionar y agregar datos de

series de tiempo en tiempo real. El resultado puede mostrarse como gráficos, datos tubulares o consumirse por sistemas externos a través de la API HTTP.

Existen cuatro tipos de expresiones para las consultas:

- **Vector instantáneo:** conjunto de series de tiempo que contienen una sola muestra para cada serie. Todas comparten la misma marca de tiempo.
- **Vector de rango:** un conjunto de series de tiempo que contiene un rango de puntos a lo largo del tiempo para cada serie de tiempo.
- **Escalar:** un valor en punto flotante numérico simple.
- **Cadena:** un valor de cadena simplemente. Actualmente no se utiliza.

Los selectores vectoriales instantáneos, permiten la selección de un conjunto de series de tiempo y un valor de muestra único para cada una, en una marca de tiempo instantánea. Puede partir de una forma simple, como especificar solo el nombre de la métrica, dando como resultado un vector instantáneo que contiene elementos para todas las series de tiempo que tienen este nombre:

mqtt_temperature

Podemos filtrar aún más estas series de tiempo agregando una lista separada por comas de separadores de etiquetas entre llaves ({}):

mqtt_temperature{topic="casa_comedor_temperatura", job="MQTT-Exporter"}

3.5. Monitorización

Las herramientas de monitorización nos proporcionan la capacidad de contar con alertas y visualización, analizando las salidas de datos que otros sistemas proporcionan. Las alertas son básicamente la comprensión sintetizada de los recursos o procesos negativos del sistema y las visualizaciones son representaciones estructurada que facilitan la comprensión del usuario (37).

Para la visualización de los datos, necesitamos proporcionar representaciones simples, de salidas de sistemas complejos, para transmitir una gran cantidad de información.

3.5.1. Grafana

Grafana es un software libre basado en licencia Apache 2.0 que permite la visualización y el formato de datos métricos. Permite generar cuadros de mando, gráficos a partir de múltiples fuentes, entre ellas bases de datos de series de tiempo como Graphite, InfluxDb o Prometheus. Está escrito en lenguaje Go y cuenta con un HTTP API completo (38).

Se trata de un software multiplataforma y extensible mediante plugins en el que los usuarios pueden construir de forma personalizada su panel de visualización de datos y gestión de alertas (39).

Las principales características de Grafana son:

- Visualización de múltiples tipos de gráficas (histogramas, mapas geográficos…) los cuales se pueden enriquecer.
- Creación de paneles de control dinámicos y reutilizables.
- Utilización de diversas fuentes de datos de consulta, de las que pueden obtener métricas personalizadas, así como filtrar datos y realizar anotaciones en tiempo real.
- Definición de alerta y notificaciones.

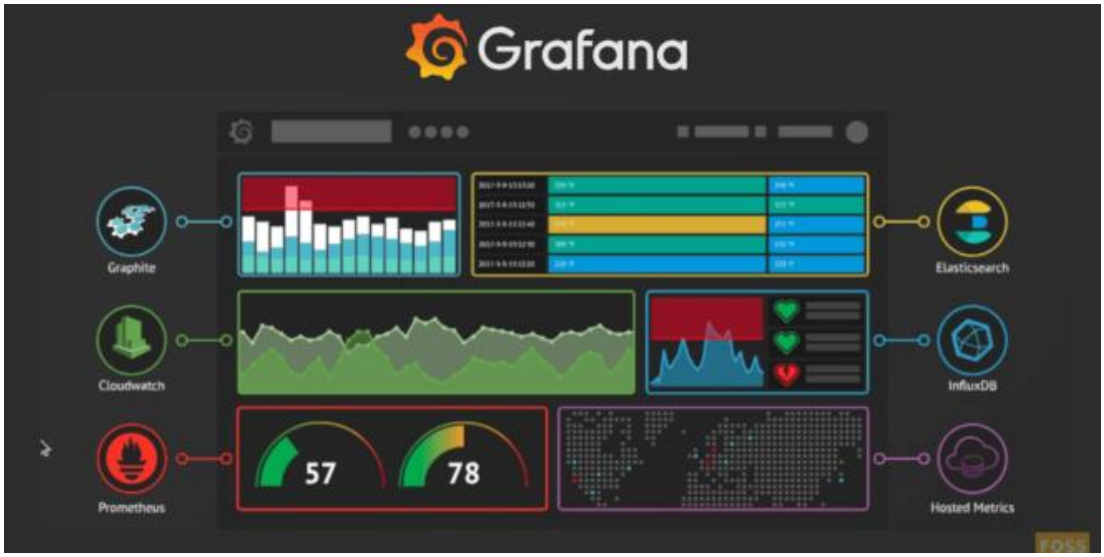

Figura 29: Grafana (40)

3.6. Hardware y dispositivos

3.6.1. Raspberry Pi 4 Model B

3.6.1.1. Introducción

Raspberry Pi 4 Model B es el último producto de la popular gama de ordenadores Raspberry Pi (41). Para el usuario final, la **Raspberry Pi Model B** ofrece un rendimiento de escritorio comparable al de los sistemas "**PC x86**" de nivel básico. Las principales características de este producto incluyen; un procesador de cuatro núcleos de **64 bits**, soporte de doble pantalla con resoluciones de hasta 4K, a través de un par de puertos micro **HDMI**, hasta **8 GB** de RAM, LAN inalámbrica de doble banda de 2,4/5,0 GHz.

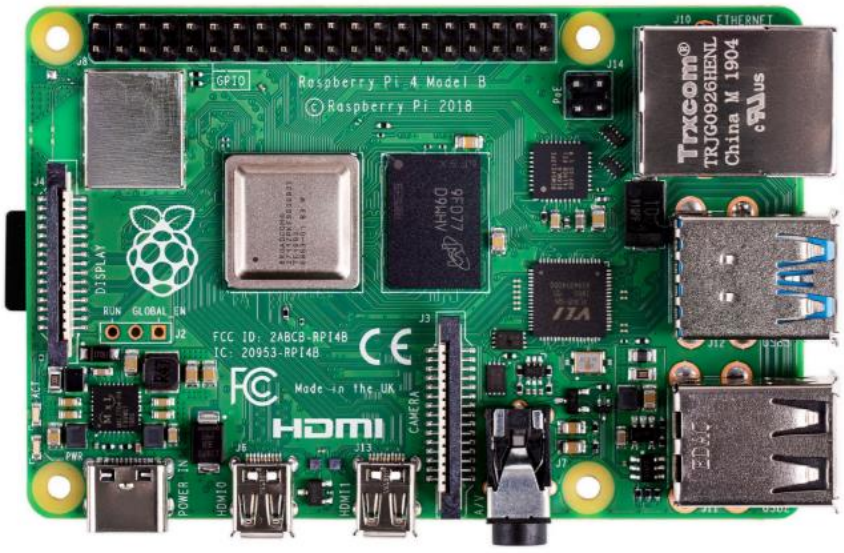

Figura 30: Raspberry Pi Model B (41)

3.6.1.2. Especificaciones

En la siguiente tabla se muestran las especificaciones técnicas más importantes de la Raspberry Pi 4 Model B:

Tabla 1: Especificaciones Raspberry Pi 4 Model B (41)

Procesador	Broadcom BCM2711, SoC de cuatro núcleos Cortex-A72 (ARM v8) de 64 bits a 1,5 GHz

Memoria	1GB, 2GB, 4GB o 8GB LPDDR4 (dependiendo del modelo) con ECC integrado
Conectividad:	LAN inalámbrica IEEE 802.11b/g/n/ac a 2,4 GHz y 5,0 GHz
	LAN, Bluetooth 5.0, BLE
	Ethernet Gigabit
	2 × puertos USB 3.0
	2 × puertos USB 2.0
GPIO:	Cabezal GPIO estándar de 40 pines (totalmente compatible con placas anteriores)
Video y sonido	2 × puertos micro HDMI (compatibles con hasta 4Kp60)
	Puerto de pantalla MIPI DSI de 2 carriles
	Puerto de cámara MIPI CSI de 2 carriles
	Puerto de audio estéreo de 4 polos y de vídeo compuesto
Multimedia:	H.265 (decodificación 4Kp60);
	H.264 (decodificación 1080p60, codificación 1080p30);
	Gráficos OpenGL ES, 3.0
SD card support:	Ranura para tarjetas Micro SD para cargar el sistema operativo y almacenar datos
Alimentación	5V DC a través del conector USB-C (mínimo 3A)
	5V DC vía pin GPIO (mínimo 3A)
	Con alimentación a través de Ethernet (PoE)
	(requiere un HAT PoE separado)
Entorno	Temperatura de funcionamiento 0-50ºC
Compilación	Para obtener una lista completa de las aprobaciones locales y regionales del producto, visite https://www.raspberrypi.org/documentation/ hardware/raspberrypi/conformity.md
Vida del producto	La Raspberry Pi 4 Modelo B seguirá produciéndose al menos hasta enero de 2026.

3.6.2. ESP32

3.6.2.1. Introducción

ESP32-WROOM-32 es un módulo de MCU Wifi, Bluetooth y BLE genérico y potente destinado a una amplia variedad de aplicaciones, que van desde redes de sensores de baja potencia hasta tareas más exigentes; como codificación de voz, transmisión de música y decodificación de MP3.

En el núcleo del módulo encontramos un chip **ESP32D0WDQ6**. Este chip está diseñado para ser escalable y adaptado. Hay dos núcleos CPU que se pueden controlar individualmente y cuenta con una frecuencia de reloj ajustable desde 80 MHz a 240 MHz. Tiene un coprocesador de baja potencia que se puede utilizar en lugar de la CPU para ahorrar energía en tareas que no requieren alta potencia informática. El ESP32, cuenta con integración con un amplio conjunto de periféricos que van desde sensores táctiles capacitivos, sensores Hall, tarjetas SD, Ethernet, SPI de alta velocidad, UART, I^2S e I^2C (42).

Figura 31: ESP32-WROOM-32 (43)

La integración de la tecnología Bluetooth y Wifi aseguran una amplia gama de aplicaciones. El Wifi le permite un gran alcance físico y una conexión directa a internet, mientras que el uso de Bluetooth permite al usuario conectarse cómodamente al teléfono o transmitir con baja energía.

La corriente en reposo del ESP32 es de 5 µA, por lo que se hace adecuado para aplicaciones de electrónica portátil. El módulo admite una velocidad de hasta 150 Mbps y una potencia de salida de 20 dBm en la antena para garantizar un rango físico más amplio.

3.6.2.2. Especificaciones

Para exponer de forma más ordenada las características técnicas generales del módulo ESP32 se adjunta la siguiente tabla:

Tabla 2: Especificaciones ESP32 (42)

Categorías	**Elementos**	**Especificaciones**
Certificaciones	Certificación RF	FCC/CE-RED/IC/TELEC/KCC/SRRC/NCC
	Certificación Wi-Fi	Wi-Fi Alliance
	Certificación Bluetooth	BQB
	Certificación ecológica	RoHS/REACH
Test	Fiabilidad	HTOL/HTSL/uHAST/TCT/ESD
Wi-Fi	Protocolos	802.11 b/g/n (802.11n up to 150 Mbps)
		Agregación de A-MPDU y A-MSDU y soporte del intervalo de guarda de 0,4 µs
	Rango de frecuencias	2.4 GHz ~ 2.5 GHz
Bluetooth	Protocolos	Especificación Bluetooth v4.2 BR/EDR y BLE
	Radio	Receptor NZIF con sensibilidad de -97 dBm
		Transmisor de clase 1, clase 2 y clase 3
		AFH
	Audio	CVSD y SBC
Hardware	Interfaces del módulo	Tarjeta SD, UART, SPI, SDIO, I2C, LED PWM, Motor PWM, I 2S, IR, contador de pulsos, GPIO, sensor táctil capacitivo, ADC, DAC, Two-Wire Automotive Interface (TWAI®), compatible con ISO11898-1 (especificación CAN 2.0)
	Sensor en el chip	Sensor Hall
	Cristal integrado	Cristal de 40 MHz
	Flash SPI integrado	4 MB
	Tensión de funcionamiento/alimentación	3.0 V ~ 3.6 V
	Corriente de funcionamiento	Media: 80 mA
	Corriente mínima suministrada por la fuente de alimentación	500 mA

	Rango de temperatura de funcionamiento recomendado	–40 °C ~ +85 °C
	Tamaño del paquete	(18.00±0.10) mm × (25.50±0.10) mm × (3.10±0.10) mm
	Nivel de sensibilidad a la humedad (MSL)	Nivel 3

3.6.2.3. Pinout

El módulo ESP32-WROOM-32 cuenta con una amplia variedad de pines, tanto de entrada como de salida para poder interactuar con otros sensores. En total cuenta con 38 sensores, los cuales tienen funciones diversas, pueden observarse los más importantes en la figura 32. Las más importantes son (44):

- 18 canales de conversión analógico-digital (ADC)
- 10 GPIO de detección capacitiva
- 3 interfaces UART
- 3 interfaces SPI
- 2 interfaces I^2C
- 16 canales de salida PWM
- 2 convertidores de digital a analógico (DAC)
- 2 interfaces I^2S

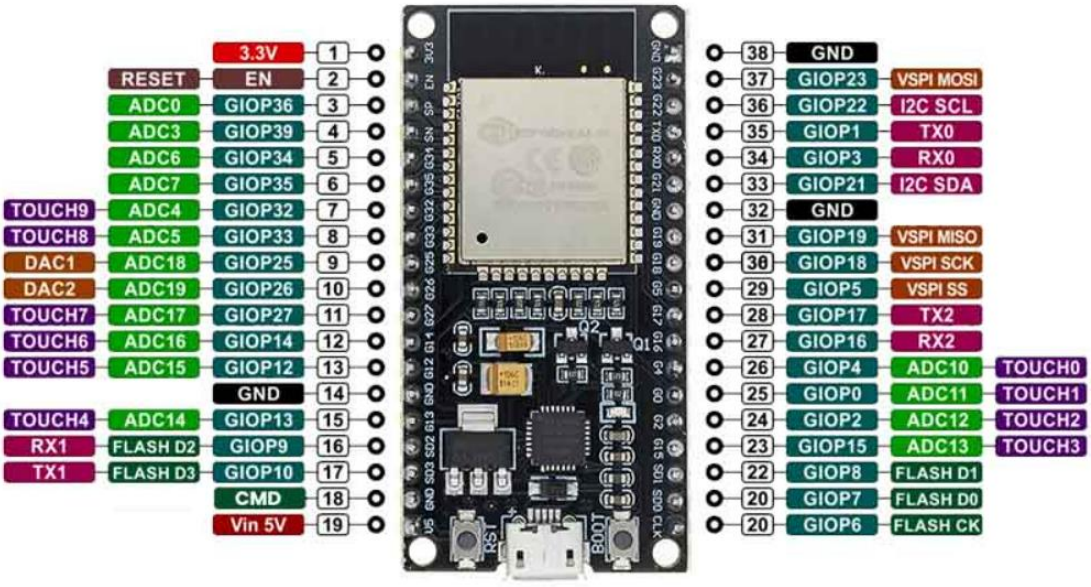

Figura 32: Pinout ESP32-WROOM-32 (45)

3.6.2.4. CPU y memoria interna

El módulo ESP32-D0WDQ6 contiene dos procesadores Xtensa® 32-bit LX6 de baja potencia. La memoria interna incluye:

- 448 KB de ROM para arranque y funciones básicas.
- 520 KB de SRAM en el chip para datos e instrucciones.
- 8 KB de SRAM en el RTC, que se denomina memoria RTC FAST y puede utilizarse para el almacenamiento de datos; se accede a ella mediante la CPU principal durante el arranque del RTC desde el modo Deep-sleep.
- 8 KB de SRAM en el RTC, que se denomina memoria RTC SLOW y a la que puede acceder el coprocesador durante el modo Deep-sleep.
- 1 Kbit de eFuse: 256 bits se utilizan para el sistema (dirección MAC y configuración del chip) y los restantes 768 bits se reservan para las aplicaciones del cliente, incluyendo la encriptación flash y el chip-ID.

3.6.2.5. Flash y SRAM externos

El ESP32 soporta múltiples flashes QSPI y chips SRAM. También admite el cifrado/descifrado por hardware basado en AES para proteger los datos y los programas de los desarrolladores en la memoria flash (42).

Puede acceder a la memoria flash externa QSPI y a la SRAM a través de cachés de alta velocidad. La flash externa puede asignarse al espacio de memoria de instrucciones de la CPU y al espacio de memoria de solo lectura.

- Si la flash externa se asigna al espacio de memoria de instrucciones de la CPU, se pueden asignar hasta 11MB + 248 KB a la vez. Aunque esto reducirá el rendimiento.
- Si la flash externa se asigna al espacio de memoria de datos de solo lectura, se pueden asignar hasta 4 MB a la vez, admitiendo lecturas de 8, 16 y 32 bits.

El módulo ESP32-WROOM-32 integra una flash SPI de 4 MB, que se conecta a GPIO6, GPIO7, GPIO8, GPIO9, GPIO10 y GPIO11. Haciendo que estos pines no puedan ser utilizados como pines normales.

4. MÉTODOLOGÍA

Para afrontar este proyecto, se ha utilizado una metodología de desarrollo iterativo e incremental. Esta metodología planifica el proyecto en distintos bloques temporales llamador iteraciones (46).

Las iteraciones, pueden entenderse como miniproyectos que proporcionan un resultado completo sobre el producto final. Así, el cliente puede obtener los beneficios del proyecto de forma incremental.

Por cada iteración se establecen una serie de requisitos, que deben completarse en el tiempo especificado. Esto permite evolucionar el producto (entrega incremental) y mantener un contacto continuo con el cliente, para la creación de nuevos requisitos, modificar los existentes y analizar el producto en busca de nuevas mejoras u orientaciones del mismo.

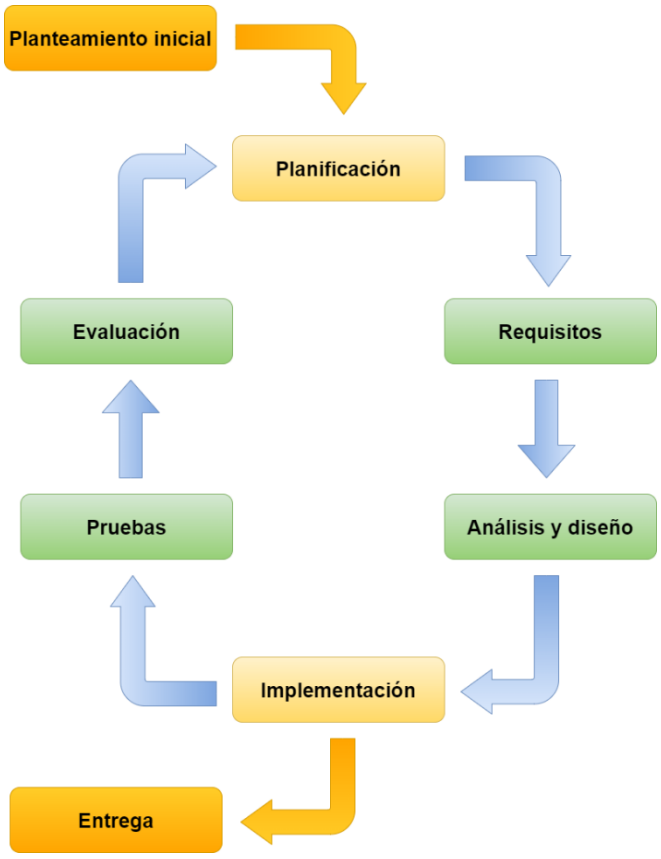

Figura 33: Desarrollo iterativo

4.1. Fases del proyecto

Para el desarrollo de este proyecto, se han llevado a cabo un total de seis iteraciones. Se explican de forma detallada a continuación.

4.1.1. Iteración 1

En esta primera iteración, se han definido los objetivos principales del proyecto. Las necesidades con las que contábamos y las diferentes opciones para solventarlas.

Objetivos

- Definición del proyecto.
- Análisis de las funciones principales.

Resultados

- Definición de una función principal, en este caso un espacio inteligente sensorizado.
- Diseño de un sistema altamente escalable, con facilidad de configuración y monitorización por parte del usuario.

4.1.2. Iteración 2

En esta segunda iteración, se han estudiado las tecnologías disponibles para llevar a cabo el proyecto y se han fijado las primeras bases para la infraestructura.

Objetivos

- Estudio y análisis de las tecnologías utilizadas para IoT.
- Planteamiento de una infraestructura base.

Resultados

- Tras análisis y comparativas de distintas tecnologías, se han elegido las que se utilizarán en este proyecto.
- Utilización de microservicios, dispositivos fácilmente programables, que requieran de poca potencia de computación y con bajo coste.

4.1.3. Iteración 3

En la tercera iteración, se ha implementado un pequeño sistema de interconexión, para familiarizarnos con el software y hardware a utilizar para una comunicación básica.

Objetivos

- Despliegue de infraestructura básica.
- Análisis de funcionamiento.
- Documentación

Resultados

- Se ha creado la infraestructura en el host con los microservicios básicos: host, Mosquitto, MQTT-Exporter, Prometheus y Grafana.
- Programación de scripts de Python para lanzar comunicación a través de la terminal y comprobar que los datos llegan al servicio de monitorización.
- Se ha comenzado con la documentación del proyecto.

4.1.4. Iteración 4

Una vez la infraestructura estaba desplegada, el paso de esta iteración ha sido la de conectar los dispositivos externos que se comunicarán con el host. Además, se ha planificado la lógica de respuesta para las órdenes de estos.

Objetivos

- Conexión de dispositivos receptores/actuadores (ESP32).
- Planificación de lógica de respuesta.

Resultados

- Se han programado los dispositivos para mantener una conexión estable con el servidor. Tanto de envío como de recepción de información.

- Se ha diseñado una lógica de funcionamiento altamente escalable y modificable. Utilizando scripts y alertas y evitando la reprogramación de los dispositivos de manera física.

4.1.5. Iteración 5

En esta iteración, ya se cuenta con todo el sistema desplegado y en funcionamiento. Se han configurado los parámetros de monitorización y configuración de alertas. Además, se han desplegado los scripts de control.

Objetivos

- Monitorización y alertas
- Despliegue de scripts de control

Resultados

- Se ha configurado el servicio de monitorización para evaluar todos los datos recibidos y configurar alertas para valores que se consideran peligrosos.
- Se han desplegado los scripts de control centrados en la monitorización y las alertas.

4.1.6. Iteración 6

Con el sistema ya completamente desplegado y automatizado, se han llevado a cabo pruebas de control de la calidad.

Objetivos

- Pruebas de control de calidad y funcionamiento.
- Entrega del proyecto

Resultados

- Se ha probado todo el sistema con diferentes casos de uso, haciendo efectivas pruebas funcionales en escenarios extremos, para comprobar un correcto funcionamiento del sistema.
- Se planifica la entrega del proyecto finalizado.

4.2. Planificación del proyecto

Todas las fases del desarrollo del sistema, se reflejan en un diagrama de Gantt. Esto nos aporta facilidad visual a la hora de afrontar los tiempos y las necesidades de un producto.

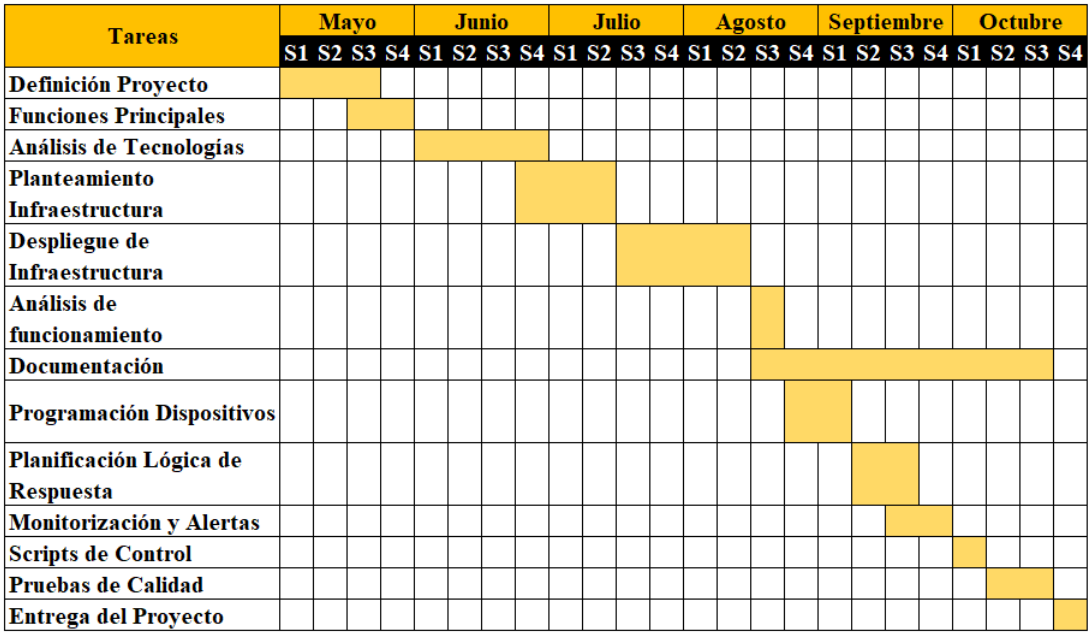

Figura 34: Diagrama de Gantt de la planificación del proyecto

5. IMPLEMENTACIÓN Y DESARROLLO

A lo largo de esta sección, se explicarán de forma más práctica la implementación y el desarrollo del proyecto. Centrándonos en el diseño, configuración y funcionalidad del mismo.

5.1. Selección de protocolo IoT

La elección del protocolo de comunicación correcto, puede marcar una enorme diferencia entre contar con una implementación eficiente del sistema o no. Todos cuentan con buenos métodos de comunicación e interconexión, pero eso no nos garantiza una conexión eficiente. Cada uno cuenta con características propias, que lo hacen destacar respecto al resto de protocolos para un ámbito u otro. De esta forma, no es lo mismo contar con grandes dispositivos con una capacidad de procesamiento elevada a contar con dispositivos pequeños, ya que la carga de trabajo no podrá ser la misma. También debemos tener en cuenta el tipo de red que vamos a utilizar, si es fiable o no, si contamos con un gran ancho de banda o en cambio contamos con un espectro estrecho en el que colocar nuestras comunicaciones. Por último, pero no menos importante, la capacidad de implementación, mejora, y programación. La facilidad a la hora de crear un sistema y mantenerlo también es un gran punto a tener en cuenta.

Para comenzar, tendremos en cuenta el sistema que vamos a implementar. Si necesitamos que los dispositivos se reconozcan entre sí y tengan interacción directa nos centraremos en un protocolo **Bus-Based**. En cambio, si nos interesa que los clientes no tengan conexión directa utilizaremos un protocolo **Broker-Based** de publicación/suscripción. Así, cada dispositivo será independiente uno de otro y podrán trabajar de forma individual y asíncrona. En este caso, resulta interesante la opción de una independencia total entre los dispositivos, de esta forma, cada uno tendrá conexión directa con el broker y nos facilitará enormemente la implementación de nuevos dispositivos o la sustitución de otros que ya no necesitemos o requieran mantenimiento.

Partiendo de esta idea básica del sistema (alta escalabilidad y cambio constante) el siguiente punto es fijarnos en los tipos de dispositivos que pensamos utilizar. En

nuestro caso, contaremos con sensores pequeños, que requieran de poca energía y que no cuentan con una gran capacidad de procesamiento. Así, nos centraremos en un protocolo en el que la carga de información enviada en cada mensaje sea ligera y fácil de sintetizar.

Todo esto, nos lleva a decantar nuestra implementación por el protocolo de comunicación **MQTT**, ya que cumple con todas las características que necesitamos, es potente y cuenta con gran cantidad de documentación para conocerlo en profundidad.

5.1 Despliegue de la infraestructura

Como comentábamos anteriormente en el apartado 3.1.2, un sistema de Internet de las Cosas está formado por una serie de componentes que definen el sistema. Sin esta arquitectura, nuestro sistema no se sustentaría y quedaría incompleto. Haciendo dos grandes clasificaciones, podemos diferenciar el host de los dispositivos periféricos.

El host, almacenará todos los programas que mantendrán la ejecución constante para monitorización, almacenamiento y lógica del sistema.

Los periféricos son nuestra puerta de enlace entre el mundo físico sensorizado y nuestro host.

5.1.1 Host

Para un correcto diseño del sistema, hemos implementado todas las funcionalidades en el mismo Host. Para poder implementar microservicios y dividir el sistema en partes más pequeñas hemos utilizado contenedores.

Nuestro host está formado por una Raspberry, la cual cuenta con un kernel Ubuntu. Nos hemos decantado por este dispositivo ya que es pequeño, barato y cuenta con la suficiente potencia como para mantener el sistema implementado sin problema. Para poder dividir el Host en servicios más pequeños hemos utilizado

Docker, pues una vez instalado, nos permite crear, modificar incluso eliminar contenedores de manera sencilla.

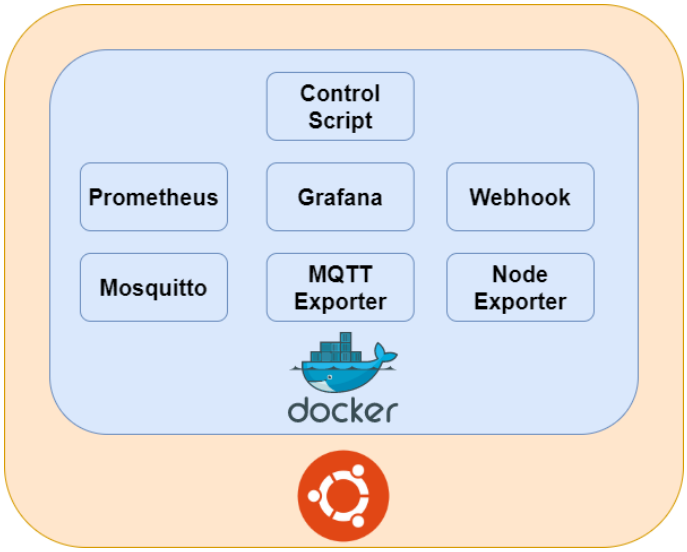

Figura 35: Organización de Host y contenedores

Como podemos ver en la imagen anterior, cada funcionalidad está incluida en un contenedor, esto nos permite configurar cada una de forma individual sin gran esfuerzo y evitar futuros fallos.

Para una mejor visualización del estado de los contenedores y su seguimiento, nos hemos ayudado de 'Portainer'. Portainer es un gestor de contenedores Docker con una interfaz más amigable al usuario y que permite configurar los parámetros sin necesidad de código en la terminal.

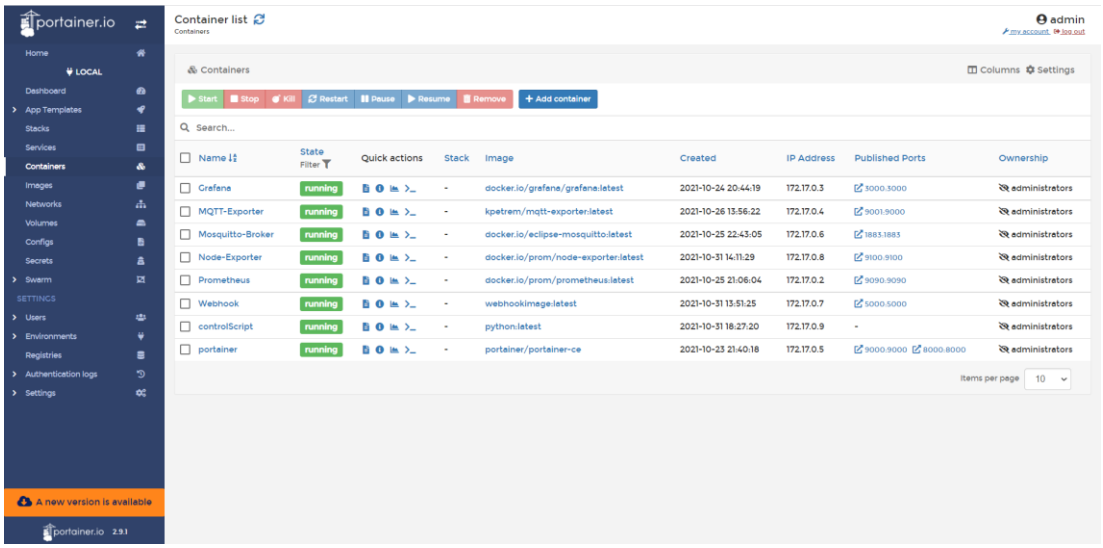

Figura 36: Contenedores en Portainer

Para el despliegue de la infraestructura en el host, podríamos haber instalado directamente todos los programas en nuestra Raspberry. Pero al utilizar contenedores y Docker, nos permite implementar de forma sencilla nuestro sistema en cualquier otro servidor. Esto es importante a la hora de implementar nuevas configuraciones, o nuevos hosts de alojamiento para la aplicación.

Los contenedores implementados, cuentan con una carpeta de configuración almacenada directamente en el Host. Al tener esta carpeta genérica para todo Docker, podemos crear copias de seguridad del sistema sin esfuerzos, pues solo tendremos que hacer copias de seguridad de la ruta '*/docker*'.

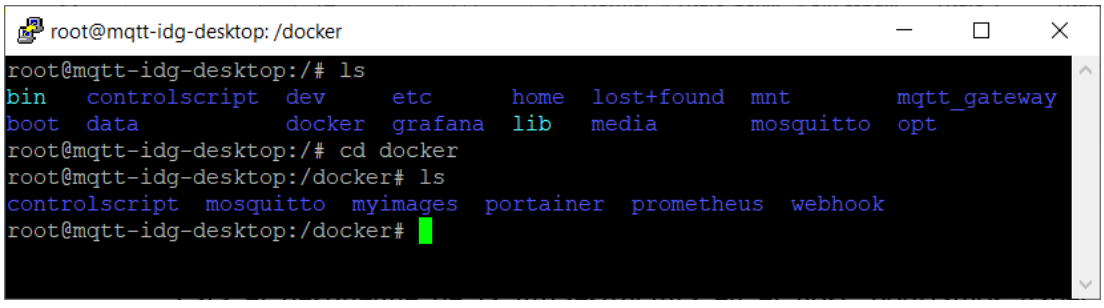

Figura 37: Rutas de almacenamiento docker en Host

Para la conexión continua con el Host, se ha utilizado PuTTY como cliente SSH para conseguir un control remoto y seguro del servidor.

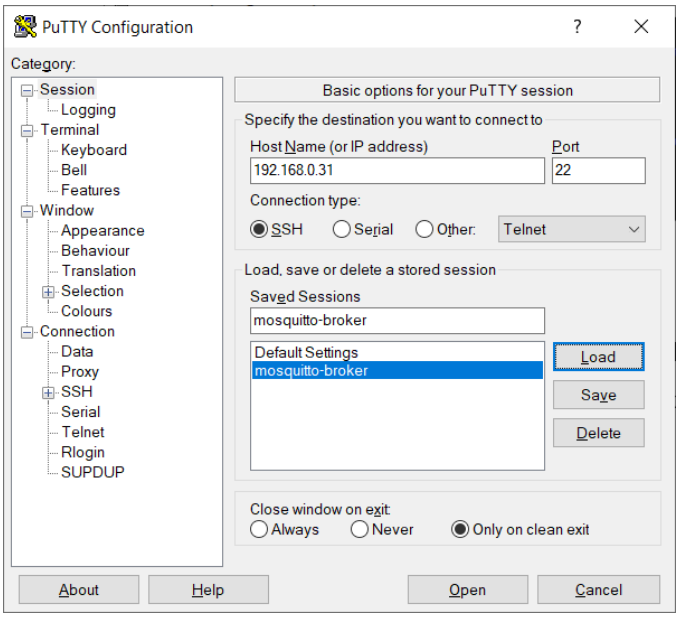

Figura 38: Interfaz PuTTY

A la hora de implementar los contenedores, hemos utilizado imágenes, tanto prediseñadas como propias. Al contar con imágenes genéricas como las de Prometheus, Grafana, Mosquitto, MQTT exporter y Node Exporter únicamente hemos tenido que ajustar sus archivos de configuración para adecuarlos a nuestro sistema.

En otras situaciones, para tareas más específicas o que se consideraba mejor un diseño propio, se han creado imágenes propias como las del Webhook o los Controlscripts.

A continuación, se explica el despliegue de cada software/contenedor y su configuración.

5.1.1.1 Mosquitto

Para nuestro broker MQTT hemos utilizado Eclipse Mosquitto. No es de los más potentes del mercado, sin embargo, nos encontramos con una distribución de código abierto, que nos permite añadir seguridad de identificación de usuario en incluso SSH/TLS. Es ideal para dispositivos de bajos recursos y su configuración es muy fácil de establecer, incluso pudiendo usarse sin ninguna configuración previa.

```
1635721988: mosquitto version 2.0.12 starting
1635721988: Config loaded from /mosquitto/config/mosquitto.conf.
1635721988: Opening ipv4 listen socket on port 1883.
1635721988: Opening ipv6 listen socket on port 1883.
1635721988: mosquitto version 2.0.12 running
```

Figura 39: Terminal de Mosquitto al iniciarse

Para poder configurar el broker, se han creado tres directorios: el de configuración, el de datos y el de logs. Además, contamos con un archivo txt en el que hemos guardado los datos de identificación de usuarios.

```
/mosquitto # ls
config           data             log              passwordfile.txt
/mosquitto # ls config
mosquitto.conf       mosquitto.conf.save
/mosquitto # ls data
mosquitto.db
/mosquitto # ls log
mosquitto.log
/mosquitto #
```

Figura 40: Terminal Mosquitto y directorios de configuración

Config: en este directorio se guarda el archivo 'mosquitto.conf' el cual utilizará Mosquitto al iniciarse y cargar la configuración. Aquí encontramos los datos básicos como persistencia, puertos, permitir conexiones anónimas…

```
root@mqtt-idg-desktop: /docker/mosquitto/config
  GNU nano 5.4                    mosquitto.conf
#Permitir conexiones anonimas falso
allow_anonymous false

#Archivo de configuracion de user y password
password_file /mosquitto/passwordfile.txt

#Puerto de escucha
listener 1883

#Persistencia
persistence true
persistence_file mosquitto.db
persistence_location /mosquitto/data

#Archivo de logs
#log_dest file /mosquitto/log/mosquitto.log

#Archivo ACL para restriccion de topics
acl_file /mosquitto/config/aclfile.acl


^G Help        ^O Write Out   ^W Where Is    ^K Cut         ^T Execute
^X Exit        ^R Read File   ^\ Replace     ^U Paste       ^J Justify
```

Figura 41: Archivo mosquitto.conf

En este caso, se han denegado las conexiones de forma anónima. Así, cada cliente que se conecte debe contar con el usuario y la contraseña establecidos. Se ha establecido el puerto 1883 como puerto de escucha y se ha configurado la persistencia de datos.

Data: en este directorio encontramos una base de datos propia de Mosquitto en la que almacena mensajes y datos para contar con persistencia de mensajes y clientes.

Log: en este directorio encontramos un archivo de texto con los registros de los logs que se han realizado en el broker. En este caso está inhabilitado debido a que se trabaja mejor mostrando los logs por la terminal en lugar de solo almacenarlos.

5.1.1.2 MQTT-exporter

Este software se encarga de exportar los datos que recibe el broker MQTT y transformarlos en un lenguaje aceptado por Prometheus. Escucha todos los topics y los envía a Prometheus para su almacenamiento y análisis.

Por ejemplo, si el broker recibe un mensaje con las siguientes características:

topic 'casa/comedor', payload `'{"temperature": 26.24, "humidity": 43.5}'`

MQTT-exporter lo recibe y lo transforma en formato PromQL de la siguiente forma:

```
mqtt_temperature {topic='casa_comedor'} 26.24
mqtt_humidity {topic='casa_comedor'} 43.5
```

```
INFO:mqtt-exporter:listening to "#"
INFO:mqtt-exporter:listening to "#"
INFO:mqtt-exporter:listening to "#"
INFO:mqtt-exporter:creating prometheus metric: mqtt_temperature
INFO:mqtt-exporter:creating prometheus metric: mqtt_humidity
INFO:mqtt-exporter:listening to "#"
```

Figura 42: Logs en contenedor MQTT-exporter

MQTT-Exporter se encuentra en el puerto 9001, el cual utiliza Prometheus para extraer la información.

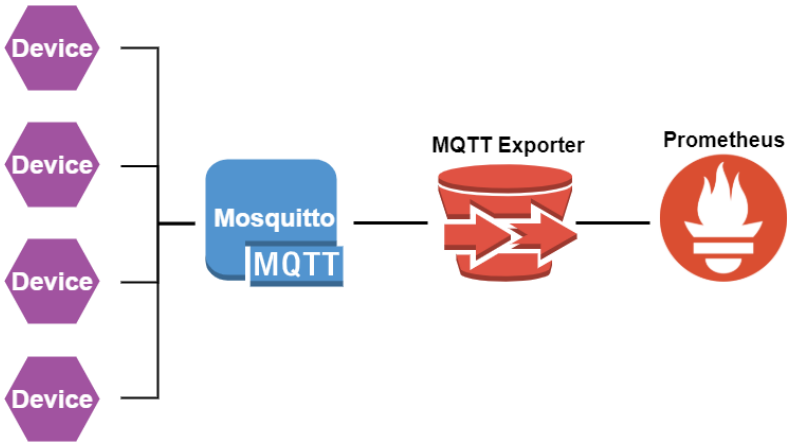

Figura 43: Esquema funcionamiento MQTT-Exporter

5.1.1.3 Node-Exporter

Al igual que en el apartado anterior, aquí también contamos con un exportador de datos para Prometheus. Node-Exporter es un software destinado al análisis del sistema Host. No requiere de configuración, pues viene ya diseñado para trabajar con sistemas de distribución Linux.

Con Node-Exporter, podremos tener conocimiento en todo momento de las funciones que realiza nuestro Host. Entre otras muchas funciones para monitorizar encontramos: temperatura, tráfico de red, uso de la CPU, uso de RAM, espacio en disco, características del hardware, etc.

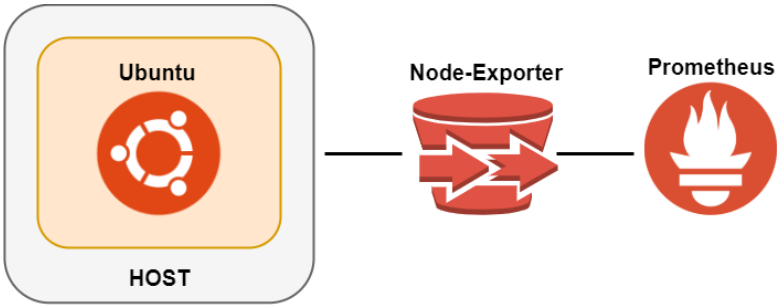

Figura 44: Esquema funcionamiento Node-Exporter

5.1.1.4 Prometheus

Como se mencionaba anteriormente, en el apartado 3.4.1, Prometheus es una base de datos de series de tiempo (TSDB). Es muy utilizado por las empresas y

servicios DevOps para monitorización continua de los diferentes entornos y servidores.

No es una de las más usadas para MQTT, sin embargo, cuenta con todo lo que necesitamos, pues nos permite almacenar tanto los datos de los sensores, como los del Host para poder mantener una monitorización constante del sistema completo.

Para un funcionamiento correcto de Prometheus, tenemos que ajustar el archivo de configuración que utiliza por defecto: prometheus.yml

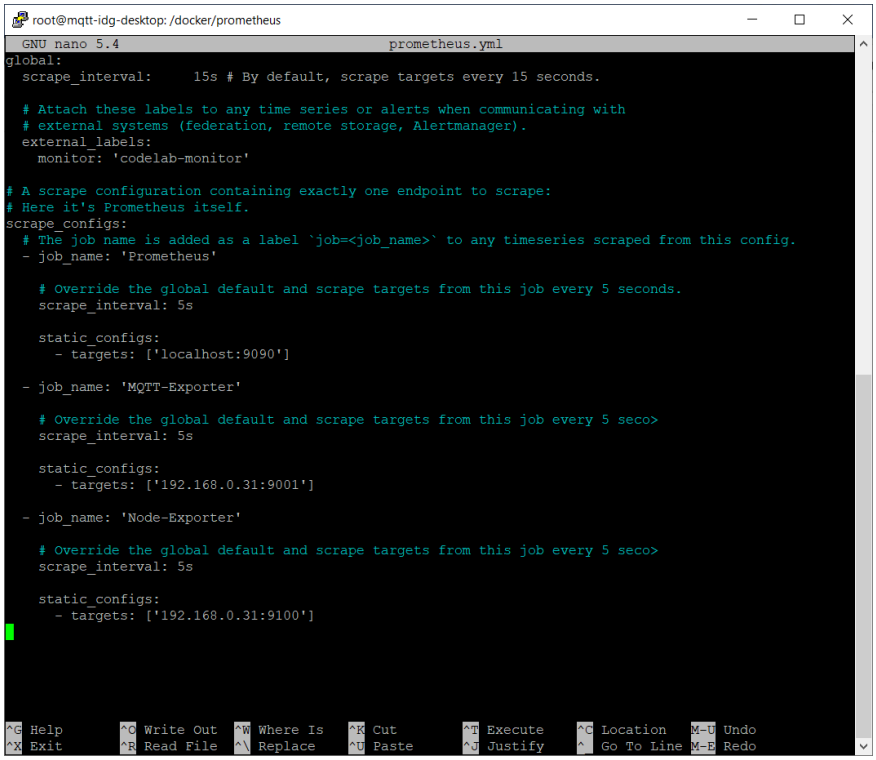

Figura 45: Archivo de configuración prometheus.yml

Prometheus admite muchos tipos de configuraciones, en este caso, hemos optado por añadir únicamente los tiempos de análisis y las fuentes de datos, que en este caso son: MQTT-exporter, Prometheus (análisis propio de sus funciones) y Node-Exporter.

Una vez contamos con el archivo de configuración preparado, podemos comprobar el estado de las fuentes de datos ingresando en la interfaz de Prometheus, en nuestro caso: *http://192.168.0.31:9090/targets*.

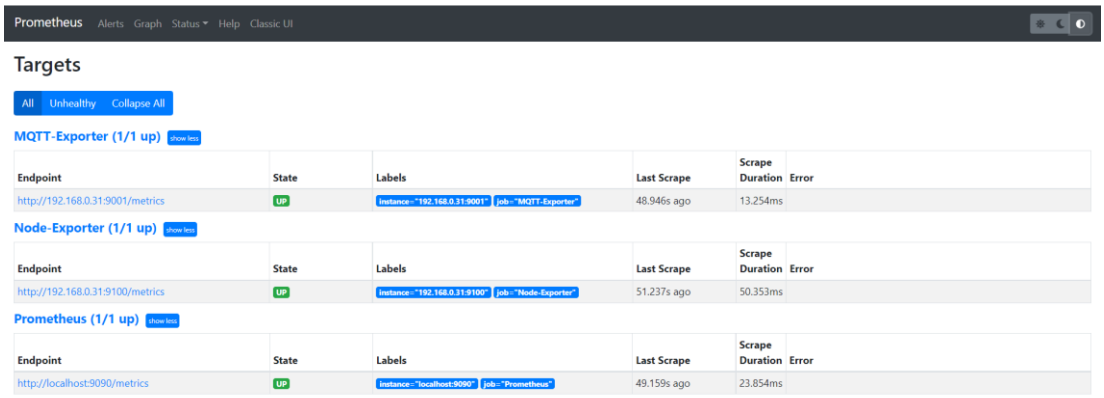

Figura 46: Targets en Prometheus

Prometheus cuenta con sus propias bases para consultas e incluso alertas (con ayuda de Alertmanager). Sin embargo, se ha utilizado Grafana como servicio de monitoreo, pues nos ofrece mejores características a la hora de configurar alertas, normas y gráficos.

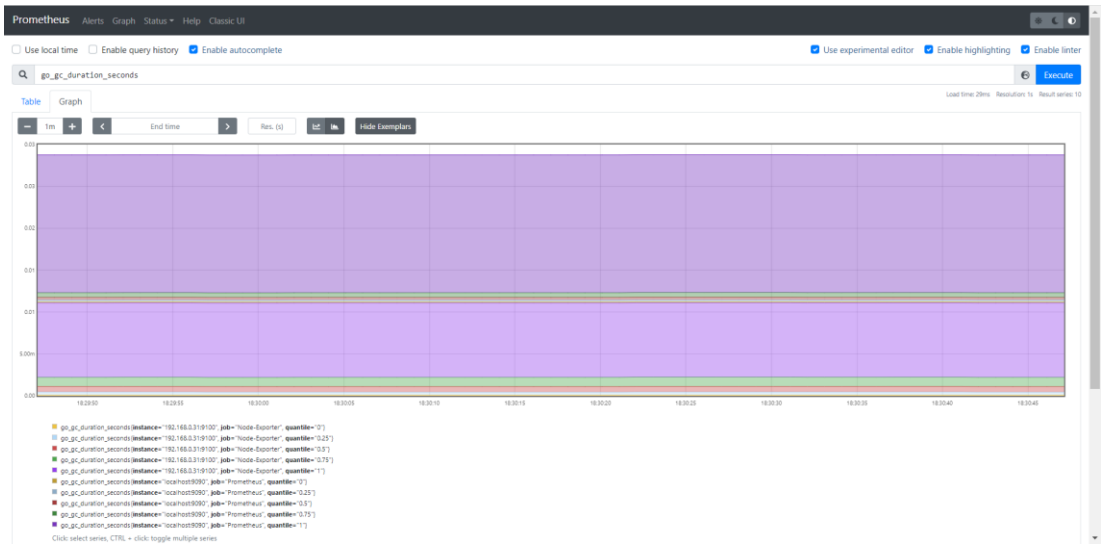

Figura 47: Consultas PromQL en Prometheus graph

5.1.1.5 Grafana

Para la monitorización de los datos, nos hemos ayudado del software Grafana. Es enormemente conocido por la facilidad y potencia que nos ofrece para la visualización y el control de datos. Nos permite con facilidad crear tableros individuales para cada función y dentro de estos, distintas gráficas que nos muestran análisis detallados de los datos.

Grafana no necesita configurarse a través de un archivo de texto. Se puede hacer directamente a través de la interfaz de usuario accediendo mediante la url: *http://192.168.0.31:3000/.*

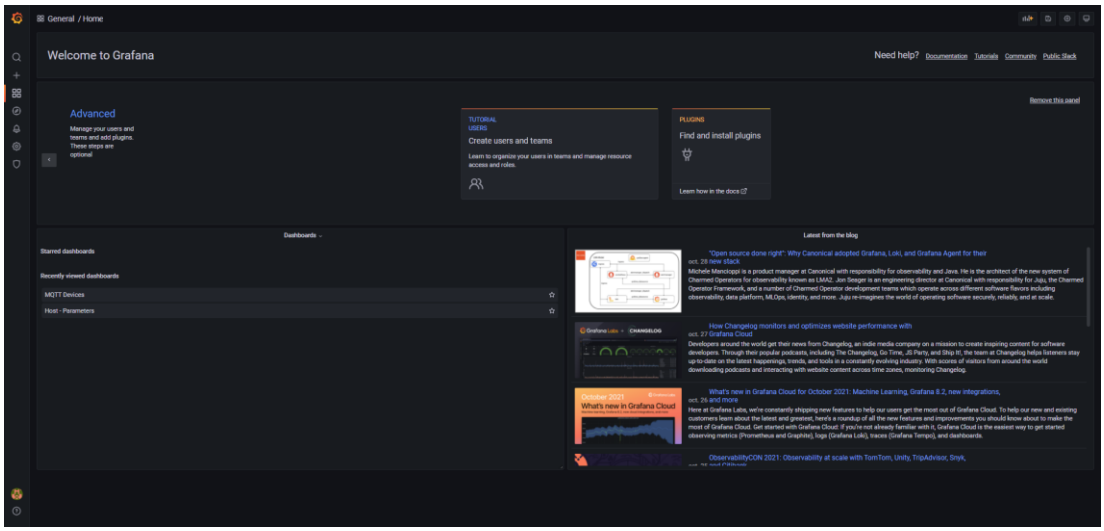

Figura 48: Interfaz inicial de Grafana

5.1.1.5.1 Tableros

Se han implementado dos tableros, uno para el control del Host y otro para el control de los dispositivos MQTT. Se considera relevante la creación de tableros diferentes ya que ambas funciones no están relacionadas y no se necesita compaginar los datos entre ellas.

Host – Parameters Dashboard

Figura 49: Host Dashboard en Grafana

En este tablero, se han configurado las gráficas para visualizar la mayor parte de los parámetros del Host. Algunos de los más importantes son: CPU, memoria, disco, procesos del sistema, temperatura, tráfico de red.

MQTT Devices Dashboard

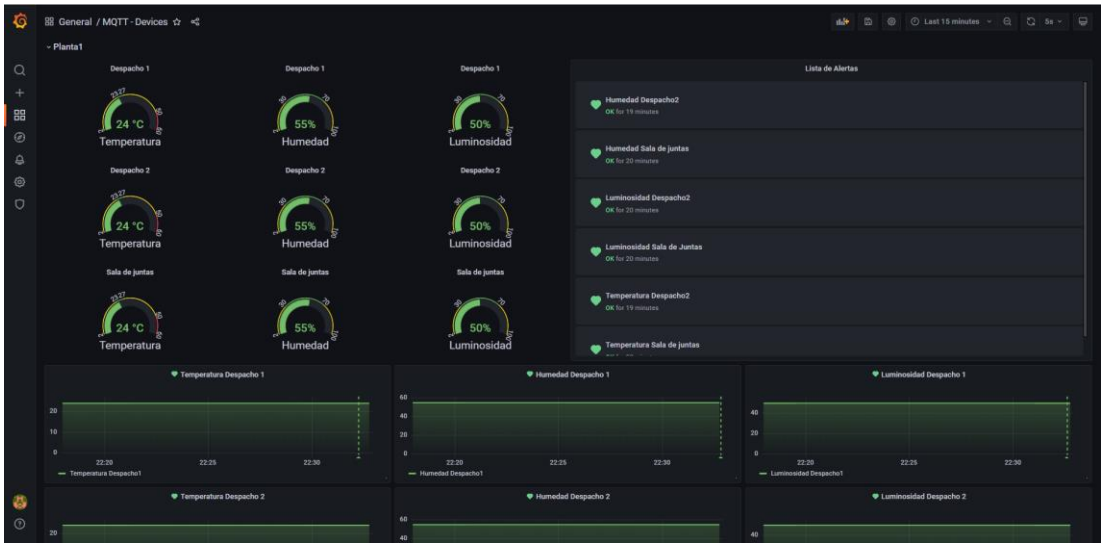

Figura 50: Tablero de dispositivos MQTT en grafana

Para nuestro caso, hemos simulado un edificio de varias plantas, en cada una tiene dos despachos y una sala de juntas. Todas cuentan con monitoreo de temperatura, humedad y luminosidad.

5.1.1.5.2 Alertas

Uno de los puntos fuertes de Grafana es su facilidad para configurar y enviar alertas. En este caso hemos utilizado dos canales de alerta:

Webhook: se encarga de recibir las alertas a través de HTTP REST y orquestarlas a los scripts de control a través de MQTT, se explicará su funcionamiento más adelante.

Telegram: envía los datos configurados de las alertas a un canal de telegram a través de un bot creado específicamente para esta función.

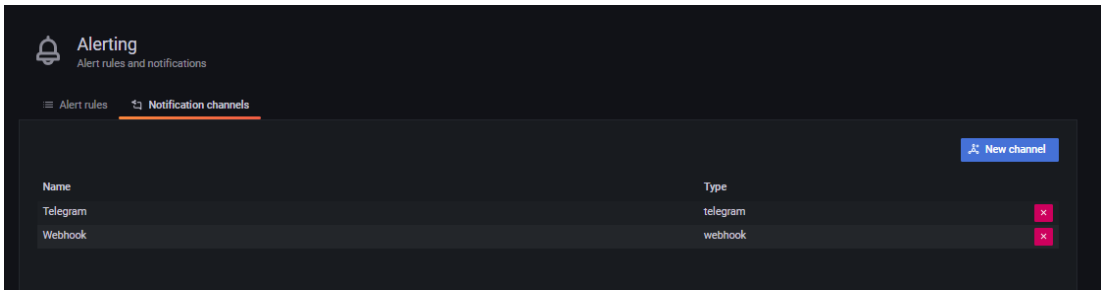

Figura 51: Canales de alerta Grafana

5.1.1.6 Webhook

Para el control de nuestros sistemas, se ha creado un Webhook el cual recibe la solicitud HTTP POST a través de Grafana cuando se dispara alguna de las alertas definidas.

El webhook se encarga de analizar el mensaje de la alerta, filtrando el topic del que proviene y orquesta los scripts de control que analizarán estos datos para mandar señales de control a los periféricos.

```
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://172.17.0.7:5000/ (Press CTRL+C to quit)
172.17.0.1 - - [31/Oct/2021 23:53:39] "POST /webhook HTTP/1.1" 200 -
172.17.0.1 - - [31/Oct/2021 23:53:40] "POST /webhook HTTP/1.1" 200 -
172.17.0.1 - - [31/Oct/2021 23:53:42] "POST /webhook HTTP/1.1" 200 -
```

Figura 52: Terminal del webhook

Este sistema ha sido implementado con Python para la lógica de funcionamiento.

En este punto del sistema, el webhook se comunica con los scripts de control mediante el broker y MQTT.

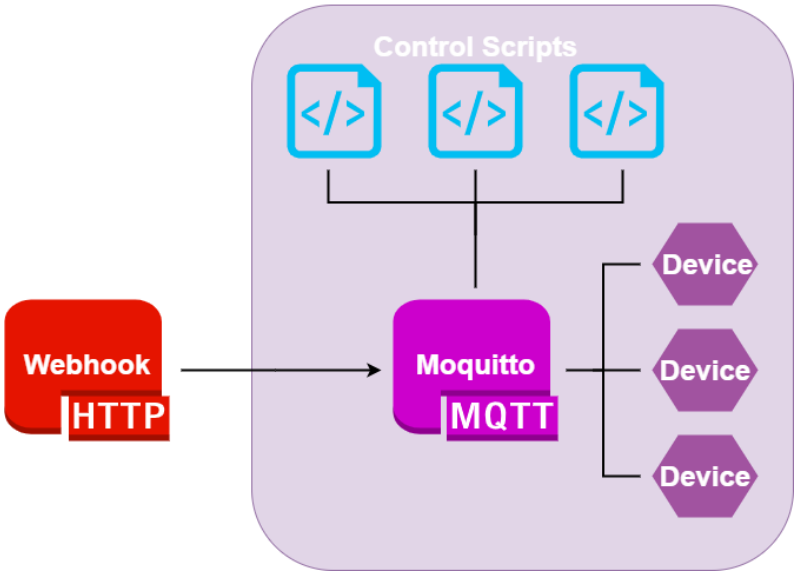

Figura 53: Conexión webhook con MQTT

Puede verse el código completo del Webhook en el **Anexo 5**.

5.1.1.7 Controlscripts

Los Controlscripts, son el cerebro de nuestro sistema. Se encargan de analizar los datos provenientes de las alarmas, para decidir qué acción llevan a cabo los dispositivos físicos.

Este sistema, se basa en diferentes scripts programados en Python que están conectados al sistema a través de MQTT, como un cliente más. Contamos con uno para cada función, en este caso luz, temperatura y humedad. Las lógicas de actuación son independientes según el tipo de dato que tratan.

Al contar con la toma de decisiones en un script separado de los dispositivos físicos, podemos realizar cambios sin necesidad de desmontar la infraestructura desplegada, de forma sencilla y rápida.

El flujo del sistema comienza cuando se recibe el mensaje completo de alerta por parte del Webhook en el topic nombrado con el tipo de dato a tratar, en el caso de temperatura: '*temperatura/'*. Aquí, el script analiza los datos y en función de los valores recibidos decide activar una de las tres salidas preparadas previamente en las placas *'pin1, pin2, pin3'*.

root@mqtt-idg-desktop: /docker/controlscript

```
  GNU nano 5.4
#Leer Json del webhook
def readJson(data):
        global state, valor, topic_respuesta, tag
        state = data["state"]
        title = data["title"]
        print(title)
        topic_respuesta = data["message"]
        if ('alerting' in state):
                for p in data["evalMatches"]:
                        valor = p["value"]
                        valor = int(valor)

        else:
                valor = None
        tag = data["tags"]["tipo"]

        return 'OK'

#Crear Json con valores de salida
def createpayload(value1, value2, value3):
        mensaje = {}
        mensaje["titulo"] = tag
        mensaje["pin1"] = value1
        mensaje["pin2"] = value2
        mensaje["pin3"] = value3
        return mensaje

#Control de decisiones
def controlScript(valor):
        if (valor == None):
                pin1=0
                pin2=0
                pin3=0
        else:
                if (valor >= 50):
                        pin1 = 1
                        pin2 = 0
                        pin3 = 0
                if ((valor>=27) and (valor<50)):
                        pin1 = 0
                        pin2 = 1
                        pin3 = 0
                if (valor<=23):
                        pin1 = 0
                        pin2 = 0
                        pin3 = 1
```

Figura 54: Lógica de análisis y actuación Controlscript Temperatura

Los Controlscripts actúan enviando las órdenes a través del topic '*control/'*. De esta forma, todos los dispositivos físicos están suscritos a este topic, cada uno en

sus correspondientes datos y localización. Este flujo se explicará en detalle más adelante.

Los códigos completos de los Controlscripts se muestran en: **Anexo 2**, **Anexo 3** y **Anexo 4**.

5.1.2 Dispositivos

En este punto, nos centramos en los dispositivos que interactúan de forma directa con el mundo físico. Como se comentó anteriormente, la programación de los ESP32 no lleva incluida lógica de decisión. Únicamente se dedican a tomar datos, enviarlos y recibir órdenes para activar salidas concretas de la placa. Estas salidas estarán conectadas a los dispositivos de actuación del espacio inteligente, como pueden ser sistemas de refrigeración, alarmas de incendio, etc.

Como contamos con pines de entrada y salida suficientes en los dispositivos, además de potencia de procesamiento, estos se encargan simultáneamente de recolectar los datos y de actuar. Teniendo pines de entrada para los sensores y pines de salida para los dispositivos de actuación.

Ver **Anexo1** para código de programación del ESP32.

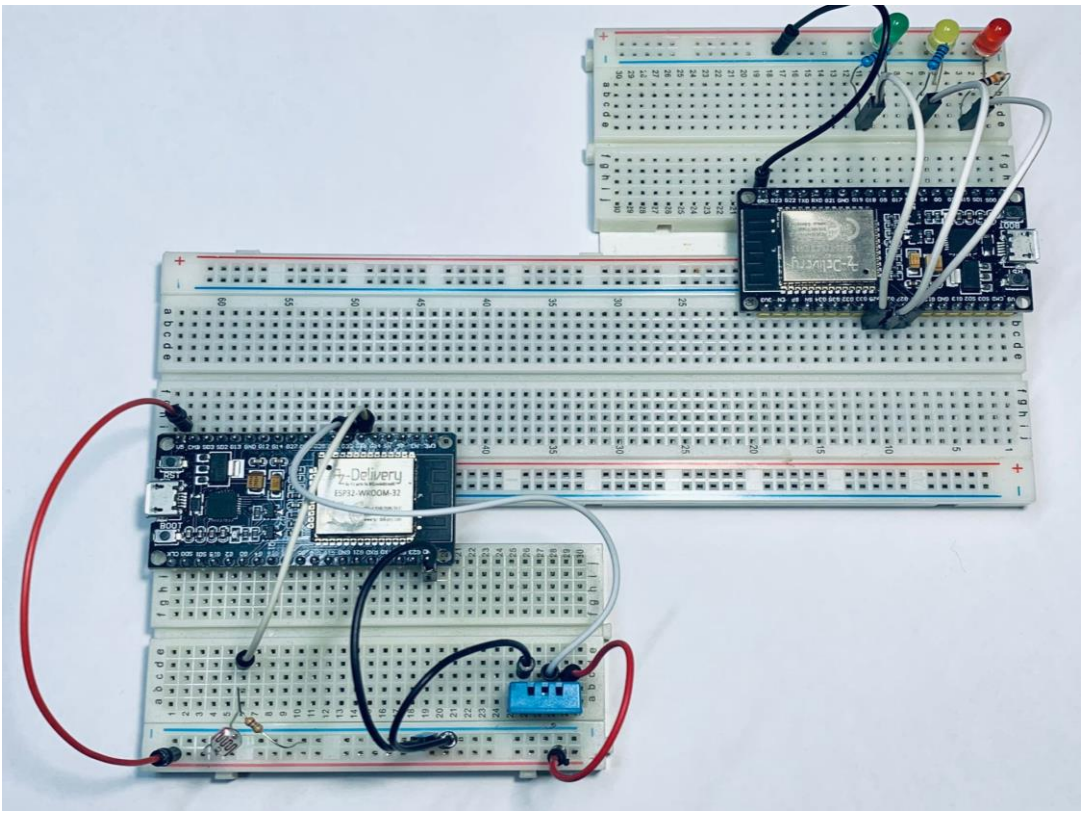

Figura 55: Dispositivos utilizados para el prototipado

5.2 Lógica del sistema

El diseño del sistema se ha orientado hacia la automatización de procesos y la facilidad de implementar cambios. La lógica de operaciones de toda la infraestructura está incluida en los contenedores, de forma que es fácil acceder a ella y modificarla a conveniencia. Esto evita los problemas derivados de tener que desmontar todos los dispositivos físicos implementados (ESP32) para cambiar la configuración.

Así, los ESP32 únicamente se encargan de mandar datos hacia Prometheus y de recibir las configuraciones de control por parte de los scripts.

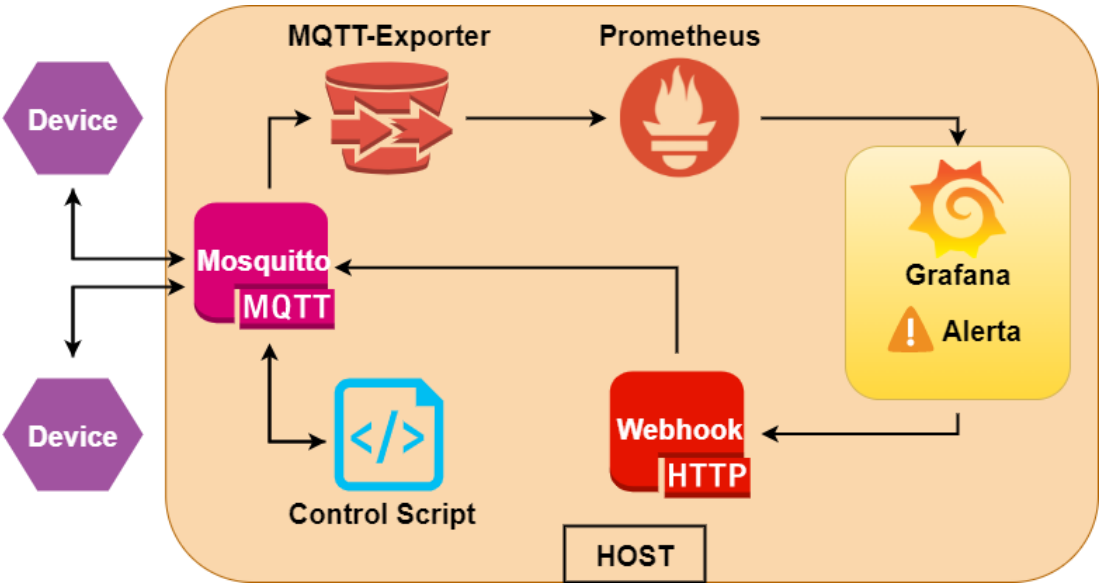

Figura 56: Lógica y funcionamiento del sistema

5.2.1 Flujo principal

A continuación, se explica paso a paso el recorrido seguido por un mensaje creado a raíz de un sensor y la actuación que se desencadena (puede apoyarse en la Figura 56 para una mejor comprensión):

- ESP32(receptor), situado en el despacho número 1, de la primera planta, capta un nivel de temperatura '*temperature=55'*.
- Este mensaje, se envía al broker con el siguiente formato como carga útil:

  `{"temperature":55}`

  En el topic:

  'planta1/despacho1'

- El broker recibe el mensaje.
- MQTT-Exporter envía el mensaje a Prometheus con el siguiente formato:

  `mqtt_temperature{topic='planta1_despacho1'} 55.`

- Prometheus analiza el dato y lo almacena en su TSDB.
- Grafana, que está configurada para tener como fuente de datos a Prometheus, analiza el dato. Previamente, se ha configurado una alerta

para que envíe un mensaje de control cuando la temperatura no se encuentre en los niveles de confort (24-27 ºC).

- Cuando Grafana detecta este valor, dispara la alerta. Enviando información del error a Telegram y a nuestro Webhook. El formato del mensaje enviado al Webhook es:


```
{
   "title":"[Alerting] Temperatura Planta1/Despacho1",
   "ruleId":2,
   "ruleName":"Temperatura Despacho1",
   "state":"alerting",
   "evalMatches":[
      {
         "value":55,
         "metric":"Temperature - ESP32",
         "tags":{
            "__name__":"mqtt_temperature",
            "instance":"192.168.0.31:9001",
            "job":"MQTT-Exporter",
            "topic":"planta1/despacho1"
         }
      }
   ],
   "orgId":1,
   "dashboardId":1,
   "panelId":36,
   "tags":{
      "tipo":"temperatura",
   },
   "ruleUrl":"http://localhost:3000/d/hbBPP9d7z/mqtt-
devices?tab=alert&viewPanel=36&orgId=1",
   "message":"temperatura/planta1/despacho1"
}
```


- El Webhook, recibe la alerta y envía los datos al broker, filtrando el topic sobre el que queremos enviar la información, recibido en la clave '*message'* del mensaje de Grafana:

'*temperatura/planta1/despacho1'*

Además de una carta útil, que contiene exactamente el mismo mensaje que se ha recibido de Grafana.

- El script de control que se encarga de la lógica para la temperatura recibe los datos, los procesa y envía las acciones a los dispositivos suscritos al topic:

  ‘*control/temperatura/planta1/despacho1’*

  Cada script de control está suscrito al topic que le caracteriza, en este caso, al ser el control de la temperatura, el topic es:

  ‘*temperatura/#*

  Los mensajes de acción que envía el script, cuentan con el siguiente formato:

  ```
  {"titulo":"temperatura","pin1":1,"pin2":0,"pin3":0}
  ```

- Los dispositivos, suscritos al topic:

  ‘control/#’

  recibirán la información para activar los pines del dispositivo en función de la acción requerida. En este caso, contamos con una alerta de temperatura, por lo que activará la salida designada para ese valor de temperatura.

Es importante destacar que, durante todo el proceso de los datos, se utiliza el formato JSON para enviar la información entre las distintas partes de la infraestructura.

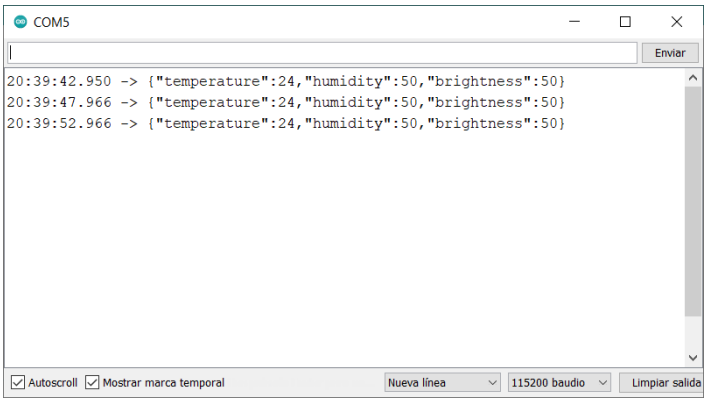

Figura 57: Envío de datos en formato JSON desde ESP32

5.2.2 Designación de topics en MQTT y restricciones

Para la comunicación entre los distintos componentes del sistema que utilizan MQTT, se han establecido un conjunto de topics, para un flujo correcto de la información. Aquí, podemos encontrar dos tipos principales:

Topics Generales:

Aquí encontramos los topics en los que se publicarán los datos de los sensores, el topic está compuesto basado la localización del dispositivo. Utilizando el esquema:

'localización1/sublocalización2'

Ej: *'planta1/despacho1'*

Cualquier dispositivo puede enviar información en estos topics.

Topics de Control:

Estos topics, son los que se utilizan para la transmisión de señales de control del sistema. Cuentan con restricciones de publicación y QoS. Aquí encontramos dos categorías:

- **Tipo de dato:** son los topics que se utilizan para la comunicación entre el Webhook y los Controlscripts. Utilizan el formato:

‘tipodato/localización1/sublocalización2’

Ej: *‘temperatura/planta1/despacho1’*

Estos topics solo son accesibles para el Webhook, ningún otro componente del sistema puede enviar información en ellos.

- **Actuación:** son los topics que se utilizan para la comunicación entre los Controlscripts y los dispositivos físicos (ESP32). Utilizan el formato:

  ‘control/tipodato/localización1/sublocalización2’

  Ej: *‘control/temperatura/planta1/despacho1’*

  Estos topics solo son accesibles para los Controlscripts, ningún otro componente del sistema puede enviar información en ellos.

Los mensajes publicados en los **topics de control**, cuentan con un **QoS1**, es decir, nos aseguramos que los mensajes llegan al menos una vez. No nos importa que llegue más de un mensaje con el mismo valor, pues no alterará la reacción del sistema.

Al contar con una jerarquía de topics, con restricciones de publicación y niveles de QoS específicos, contamos con un seguro frente a fallos que puedan generarse a raíz de comunicaciones no esperadas o erróneas entre dispositivos.

5.1 Despliegue en distintos servidores

Para garantizarnos un despliegue sencillo y rápido del Host y los Dispositivos, se han creado repositorios en **GitHub** en el que se encuentran los ficheros más importantes del sistema:

- Host:

  https://github.com/idiestro/TFG-IDG-dockercompose

  https://github.com/idiestro/TFG-IDG-Host

  https://github.com/idiestro/TFG-IDG-Webhook

- Dispositivos:

  https://github.com/idiestro/TFG-IDG-esp32

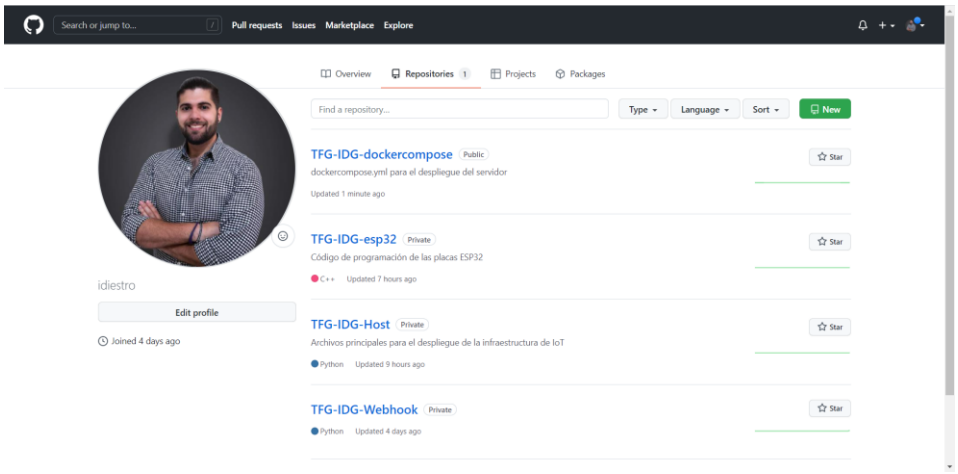

Figura 58: Tablero de repositorios del sistema GitHub

En los repositorios ***'TFG-IDG-Host'***, ***'TFG-IDG-Webhook'*** y ***'TFG-IDG-esp32'***, encontramos una copia de todos los archivos de configuración individuales de cada componente del sistema, además del código implementado en la memoria del ESP32.

Uno de los repositorios, ***'TFG-IDG-dockercompose'***, cuenta con un archivo de configuración **'.yml'** para Docker. Este archivo, nos permite crear todo el sistema de una sola vez de forma rápida, con todos sus volúmenes, dependencias y conexiones internas previamente configuradas.

6. RESULTADOS Y DISCUSIÓN

Una vez con el proyecto terminado, evaluando los objetivos planteados al principio, podemos asegurar que se han podido cumplir todos:

- Se ha creado un espacio inteligente interconectado, contamos con sensores instalados en distintos puntos del recinto para monitorizar datos del mundo físico y para llevar a cabo acciones en función de las ordenes provenientes de los sistemas de control.
- Se ha conseguido utilizar de forma eficiente y con éxito el protocolo de comunicación MQTT, para dispositivos de bajos recursos y que necesitan mensajería poco pesada.
- Se ha generado todo un sistema backend, que mantiene la infraestructura de comunicación de forma estable. Contando con bases de datos, puertas de enlace, broker para los dispositivos y servicios de monitorización y alertas, entre otros.
- Al contar con una lógica del sistema basada en el backend y no en los dispositivos físicos, es fácil añadir más sensores y actuadores sin tener que modificar el sistema, solo se requiere una configuración inicial de los dispositivos.

  También contamos con la utilización de microservicios y contenedores para cada aplicación, el host es altamente escalable, permitiendo añadir funcionalidades nuevas, o modificar las existentes.
- Todos los softwares y sus configuraciones se encuentran almacenadas en GitHub, por lo que una vez que contamos con un servidor, solo hay que utilizar los archivos del repositorio, para levantar el sistema de forma rápida y segura.
- Para llevar a cabo todo el proyecto, se ha seguido un enfoque empresarial. Esto se ha basado en utilizar sistemas que están instaurados a día de hoy en gran cantidad de empresas; como Prometheus, Docker o Grafana. Además, contamos con una lógica fácilmente escalable y modificable.

7. CONCLUSIONES

Para la creación de este proyecto, se ha pasado por diversas etapas de diseño e implementación. Cada una de ellas suponía un reto a la hora de implementarla, pues con cada paso, se presentaban distintas dificultades, nuevas posibilidades y diversas ramas de desarrollo.

En el proyecto se puede apreciar el uso de tecnologías punteras y conocimientos que no se imparten en la titulación de telecomunicaciones. Las competencias y habilidades que me ha proporcionado el grado, han permitido afrontar el reto de forma eficaz. Considero este proyecto ambicioso y a la vez útil. Realmente, se puede implementar el sistema y utilizarlo en un entorno real. Cuenta con pilares sólidos y configuraciones fácilmente modificables.

Aunque pueda parecer una infraestructura compleja, el sistema está diseñado como un puzle. Está compuesto de pequeñas piezas (softwares) con funcionalidades concretas y no demasiado complejas, cada parte tiene su función. Todas estas piezas cuentan con una interconexión ligera, que al juntarlas conforman todo el sistema, y proporcionan toda la capacidad y potencia de la infraestructura.

Personalmente, el proyecto me ha aportado una gran cantidad de conocimientos, en campos en los que no tenía mucha experiencia. Sin embargo, son campos que están a la orden del día a nivel empresarial y laboral, por lo que todo lo aquí aprendido me ayudará a desarrollarme como profesional en el mundo de las telecomunicaciones.

Dejando a un lado las conclusiones técnicas. Es una sensación muy placentera el poder ver como el proyecto, que partió de una pequeña idea, ha cogido forma y está en funcionamiento. Cómo, definiendo objetivos, planificando y sobre todo, poniendo ganas e interés, pueden conseguirse grandes cosas.

8. TRABAJOS FUTUROS

El proyecto llevado a cabo, se centra en un sistema de comunicación sólido. Fácilmente configurable y escalable. Esto nos abre las puertas a gran cantidad de nuevas implementaciones para adecuar el sistema a distintos escenarios o mejorar el ya existente.

Algunos de los trabajos futuros sobre el proyecto podrían ser:

- Implementación del sistema en un ambiente industrial o empresarial. Es decir, la lógica y el funcionamiento básicos están puestos en marcha. Sin embargo, podría ampliarse el proyecto haciendo un estudio de los dispositivos hardware necesarios para conseguir que los ESP32 interactúen con hardware que trabaja con líneas eléctricas de 220V o de nivel industrial.

  Esto sería por ejemplo el estudio de relés y maquinaria para la interconexión del sistema con los dispositivos finales de actuación (maquinaria industrial, sistemas de ventilación, sistemas de alumbrado, etc).

- Añadir al sistema un almacenamiento de datos persistente a largo plazo. Bases de datos que nos permitan tener un control de los datos antiguos, para su posterior utilización.

- Añadir nuevas métricas de sensorización al sistema, o incluso mejorar las ya creadas. Una nueva sensorización que podría ser importante es la medición de CO2 y el control de la ventilación en función de los valores obtenidos.

- Implementar seguridad SSH entre el Broker y los dispositivos. En este caso no ha sido necesaria debido a que se contaba con una red confiable. Sin embargo, es una buena opción de seguridad en redes poco fiables.

- Implementar el sistema repartido en diferentes Host, utilizando tecnología clustering. Esto nos permitiría utilizar la llamada ‘informática sin servidor’, escalando horizontal y verticalmente el sistema automáticamente en función de las necesidades del mismo y ahorrarnos posibles fallos.

REFERENCIAS

1. **VERMESAN, Ovidiu and FRIESS, Peter.** Internet of Things - From Research and Innovation to Market Deployment. *River Publishers.* [Online] 2014. [Cited: agosto 16, 2021.] https://www.riverpublishers.com/pdf/ebook/RP_E9788793102958.pdf.

2. **CENDÓN, Bruno.** El origen de IoT. *Bruno Cendón.* [Online] 2019. [Cited: agosto 23, 2021.] https://www.bcendon.com/el-origen-del-iot/.

3. **GERBER, Anna and KANSAL, Satwik.** Simplify the development of your IoT solutions with IoT architectures. *IBM Developer.* [Online] 2017. [Cited: agosto 24, 2021.] https://developer.ibm.com/articles/iot-lp201-iot-architectures/.

4. **CRESPO, Enrique.** Arquitecturas IoT. *Aprendiendo Arduino.* [Online] 2018. [Cited: agosto 24, 2021.] https://aprendiendoarduino.wordpress.com/2018/11/11/arquitecturas-iot/.

5. **GARCÍA, Luis.** ¿Cuáles son las arquitecturas y componentes de una red IoT? *Rede Móviles.* [Online] 2020. [Cited: agosto 25, 2021.] https://redesmoviles.com/iot/arquitecturas-iot/.

6. **GRIZHNEVICH, Alex.** IoT Architecture: Building Blocks and How They Work. *Science Soft.* [Online] 2018. [Cited: agosto 25, 2021.] https://www.scnsoft.com/blog/iot-architecture-in-a-nutshell-and-how-it-works.

7. **AYUNTAMIENTO DE VILLANUEVA DE LA SERENA.** SMART CITY. *Ayuntamiento de Villanueva de la Serena.* [Online] 2017. [Cited: septiembre 16, 2021.] https://villanuevadelaserena.es/concejalias-ayv/smartcity/.

8. **ÇORAK, Burak, et al.** Comparative Analysis of IoT Communication. *IEEE 978-1-5386-3779-1.* [Online] 2018. [Cited: agosto 25, 2021.]

9. **STUSEK, Martin, et al.** IoT Protocols for Low-power Massive IoT: A Communication Perspective. *IEEE 978-1-7281-5764-1/19/.* [Online] 2019. [Cited: agosto 25, 2021.]

10. **LLAMAS, Luis.** Protocolos de Comunicación para IoT. *Luis Llamas.* [Online] 2019. [Cited: agosto 27, 2021.] https://www.luisllamas.es/protocolos-de-comunicacion-para-iot/.

11. **M HASAN, Hamid.** ResearchGate. *IoT Protocol for Health Care Systems: A Comparative Study.* [Online] 2018. [Cited: septiemnre 15, 2021.] https://www.researchgate.net/figure/Constrained-Application-Protocol-CoAP_fig1_328967304.

12. **SMARTBEAR.** Smartbear. *AMQP Testing.* [Online] 2021. [Cited: septiembre 15, 2021.] https://support.smartbear.com/readyapi/docs/testing/amqp.html.

13. **JAVAPOINT.** JAVAPOINT. *JMS Tutorial.* [Online] 2014. [Cited: septiembre 15, 2021.] https://www.javatpoint.com/jms-tutorial.

14. **DDS Foundation.** DDS Foundation. *What is DDS?* [Online] 2015. [Cited: septiembre 15, 2021.] https://www.dds-foundation.org/what-is-dds-3/.

15. **DEQING SUN.** Adafruit. *What is XMPP and why using it.* [Online] 2013. [Cited: septiembre 15, 2021.]

16. **OASIS.** MQTT Version 5.0. *OASIS Open.* [Online] 2014. [Cited: septiembre 22, 2021.] https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

17. **HIVEMQ.** Introducing the MQTT Protocol - MQTT Essentials: Part 1. *HIVEMQ.* [Online] 2015. [Cited: septiembre 21, 2021.] https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/.

18. **OASIS.** OASIS Open. [Online] [Cited: septiembre 21, 2021.] https://www.oasis-open.org/.

19. **HIVEMQ.** MQTT Client and Broker and MQTT Server and Connection Establishment Explained - MQTT Essentials: Part 3. *HIVEMQ.* [Online] 2019. [Cited: septiembre 21, 2021.] https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment/.

20. **HIVEMQ**. MQTT Topics & Best Practices - MQTT Essentials: Part 5. *HIVEMQ.* [Online] 2019. [Cited: septiembre 21, 2021.] https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/.

21. **HIVEMQ**. Publish & Subscribe - MQTT Essentials: Part 2. *HIVEMQ.* [Online] 2015. [Cited: septiembre 21, 2021.]

22. **LLAMAS, Luis.** PRINCIPALES BROKER MQTT OPEN SOURCE PARA PROYECTOS IOT. *Luis Llamas.* [Online] 2019. [Cited: septiembre 22, 2021.] https://www.luisllamas.es/principales-broker-mqtt-open-source-para-proyectos-iot/.

23. **ECLIPSE FOUNDATION.** Eclipse Mosquitto™ An open source MQTT broker. *mosquitto.* [Online] 2020. [Cited: septiembre 22, 2021.] https://mosquitto.org/.

24. **HBMQTT.** HBMQTT. *HBMQTT.* [Online] 2015. [Cited: septiembre 22, 2021.] https://hbmqtt.readthedocs.io/en/latest/index.html.

25. **EMQ.** EMQ - Erlang MQTT Broker. *EMQ.* [Online] 2018. [Cited: septiembre 22, 2021.] https://emqtt.io/docs/v2/index.html.

26. **RABBITMQ.** Clients Libraries and Developer Tools. *RabbitMQ.* [Online] 2019. [Cited: septiembre 23, 2021.] https://www.rabbitmq.com/devtools.html.

27. **HIVEMQ.** HiveMQ MQTT Broker. *HiveMQ.* [Online] 2019. [Cited: septiembre 23, 2021.]

28. **HIVEMQ**. Quality of Service 0,1 & 2 - MQTT Essentials: Part 6. *HIVEMQ.* [Online] 2015. [Cited: septiembre 27, 2021.] https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/.

29. **IBM Cloud Education.** Microservices. *IBM.* [Online] 2021. [Cited: Septiembre 29, 2021.] https://www.ibm.com/cloud/learn/microservices.

30. **DIGITIZE01.** MicroServices Architecture (MSA) & Containerization. *Shlule.* [Online] 2019. [Cited: septiembre 29, 2021.]

31. **DOCKER.** Developers bring their ideas to life with Docker. *Docker.* [Online] 2020. [Cited: septiembre 29, 2021.] https://www.docker.com/why-docker.

32. **DOCKER**. Use containers to Build, Share and Run your applications. *Docker.* [Online] 2020. [Cited: septiembre 29, 2021.] https://www.docker.com/resources/what-container.

33. **INFLUXDATA.** Time series database (TSDB) explained. *influxdata.* [Online] 2017. [Cited: septiembre 30, 2021.] https://www.influxdata.com/time-series-database/.

34. **INFLUXDATA**. What is time series data? *influxdata.* [Online] 2020. [Cited: septiembre 30, 2021.] https://www.influxdata.com/what-is-time-series-data/.

35. **PROMETHEUS AUTHORS.** OVERVIEW. *Prometheus.* [Online] 2020. [Cited: septiembre 30, 2021.] https://prometheus.io/docs/introduction/overview/.

36. **PROMETHEUS AUTHORS**. STORAGE. *Prometheus.* [Online] https://prometheus.io/docs/prometheus/latest/storage/.

37. **BARKER, DAN.** 5 alerting and visualization tools for sysadmins. *opensource.com.* [Online] 2018. [Cited: septiembre 30, 2021.] https://web.archive.org/web/20181010141616/https://opensource.com/article/18/10/alerting-and-visualization-tools-sysadmins.

38. **WIKIPEDIA.** Grafana. *Wikipedia.* [Online] 2021. [Cited: septiembre 30, 2021.] https://es.wikipedia.org/wiki/Grafana#cite_note-2.

39. **MANCOMUN.** Grafana. *mancomun.* [Online] 2021. [Cited: septiembre 30, 2021.] https://www.mancomun.gal/es/solucion-tic/grafana/#:~:text=Grafana%20es%20un%20sistema%20de,una%20serie%20de%20servicios%20a%C3%B1adidos..

40. **LEMUS, ISAAC.** Grafana: Software libre para el análisis y la supervisión. *conocimiento libre.* [Online] 2021. [Cited: septiembre 30, 2021.] https://conocimientolibre.mx/grafana/.

41. **RASPBERRY PI TRADING LTD.** Raspberry Pi 4 Computer Model B. *Raspberry Pi.* [Online] 2021. [Cited: septiembre 30, 2021.] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

42. **ESPRESSIF SYSTEMS.** ESP32-WROOM-32 Datasheet. *Espressif.* [Online] 2021. [Cited: septiembre 30, 2021.] https://www.espressif.com/en/products/modules/esp32.

43. **BRICOGEEK.** ESP32 Wroom WIFI + Bluetooth. *BricoGeek.* [Online] 2021. https://tienda.bricogeek.com/arduino-compatibles/1274-esp32-wroom-wifi-bluetooth.html.

44. **DESCUBRE ARDUINO.** ESP32, módulo ESP32-WROOM GPIO Pinout. *Descubre Arduino.* [Online] 2020. [Cited: septiembre 30, 2021.]

45. **UNIT Electronic.** ESP32 38 Pines ESP WROOM 32. *UNIT Electronic.* [Online] 2021. [Cited: septiembre 30, 2021.] https://uelectronics.com/producto/esp32-38-pines-esp-wroom-32/.

46. **PROYECTOS AGILES.** Desarrollo iterativo e incremental. *proyectos agiles.* [Online] 2008. [Cited: octubre 2021, 2.] https://proyectosagiles.org/desarrollo-iterativo-incremental/.

ANEXOS

Anexo 1

```
//Librerias generales
  #include <WiFi.h>
  #include <PubSubClient.h>
  #include <ArduinoJson.hpp>
  #include <ArduinoJson.h>
  #include <Wire.h>

//Librerias especificas
  #include <DHT.h>

//Declaramos ID
const char* clientID;

//Definimos pines de entrada
  #define DHTPIN 32
  #define DHTTYPE DHT11
  DHT dht (DHTPIN, DHTTYPE);
  #define pinLUZ 35

//Definimos pines de salida
  const char pinSalidaT[] = {25, 26, 27}; //pinSalida0=25,
pinSalida1=26, pinSalida2=27
  const char pinSalidaH[] = {21, 22, 23}; //pinSalida0=33,
pinSalida1=32, pinSalida2=35
  const char pinSalidaL[] = {33, 18, 19}; //pinSalida0=17,
pinSalida1=18, pinSalida2=19

//Ingresamos datos del Wifi
  const char* ssid = "Pakitos_Wifi";
  const char* password = "IrNacFranCar55";

//Ingresamos direccion del broker
  const char* mqtt_server = "192.168.0.31";
  int mqtt_port = 1883;
  const char* user_mqtt = "nacho";
  const char* password_mqtt = "1234";

//Declaramos topic de recepcion
  const char* topic_recepcion = "control/#";

//Declaramos topics de publicacion
  const char* topic_despacho1 = "planta1/despacho1";
  const char* topic_despacho2 = "planta1/despacho2";
  const char* topic_salajuntas = "planta1/salajuntas";
  const char* topic_comedor = "planta1/comedor";

//Declaramos Json de entrada
  StaticJsonDocument<100> doc_recibido;

//Declaramos array para datos recibidos
  int valores_callback[] = {0,0,0};
```

```
  const char* titulo = "None";


//Configuramos cliente ESP
  WiFiClient espClient;
  PubSubClient client(espClient);

//Tiempo para mensajes salientes
  long lastMsg = 0;
  char msg[50];
  int value = 0;

//Declaramos variables
  float temperature = 0;
  float humidity = 0;
  int brightness = 0;

//Modulo de inicio del sistema
void setup() {
  Serial.begin(115200);
  dht.begin();
  //Inicializamos  pines de salida
  int a;
  for (a=0;a<3;a++){
    pinMode(pinSalidaT[a], OUTPUT);
    pinMode(pinSalidaH[a], OUTPUT);
    pinMode(pinSalidaL[a], OUTPUT);
  }
  //Conectar a Wifi
  setup_wifi();
  //Conectar a Broker
  client.setServer(mqtt_server, mqtt_port);
  //Recibimos mensajes
  client.setCallback(callback);
  delay(5000);
}

//Conexion Wifi - modulo
void setup_wifi() {
  delay(10);
  // Conexion a una red wifi
  Serial.println("Conectando a: ");
  Serial.println(ssid);
  //Conexion a red wifi
  WiFi.begin(ssid, password);
  //Espera hasta que se conecte
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  //Mostramos configuracion de conexiones
  Serial.println("Conectado a Wifi");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
}

//Callback por si recibimos mensaje - Modulo
void callback(char* topic, byte* message, unsigned int length) {
```

```
  Serial.print("Mensaje recibido en topic: ");
  Serial.println(topic);
  Serial.print("-Mensaje: ");
  String mensaje_recibido;
  //Mostramos mensaje recibido
  for (int i = 0; i < length; i++) {
    Serial.print((char)message[i]);
    mensaje_recibido += (char)message[i];
  }
  Serial.println();
  leerJson(mensaje_recibido);
  actuador();
}

//Conexion con Broker-MQTT
void reconnect() {
  //Leemos ID del dispositivo
  String chipID = WiFi.macAddress();
  clientID = chipID.c_str();
  //Loop hasta conectar
  while (!client.connected()) {
    Serial.print("Esperando conexion a Broker...");
    //Attempt to connect
    if (client.connect(clientID, user_mqtt,password_mqtt)) {
      Serial.println("Connectado");
      Serial.print("ESP32 ID: ");
      Serial.println(clientID);
      // Subscribe
      client.subscribe(topic_recepcion);
      client.setKeepAlive(30);
      }
      else {
      Serial.print("Connection failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

//Modulo para convertir a json
String mensajeJson(float temp, float hum, float brigh){
    String json;
    StaticJsonDocument<300> doc;
    doc["temperature"] = temp;
    doc["humidity"] = hum;
    doc["brightness"] = brigh;
    serializeJson(doc,json);
    return json;
}

//Modulo para leer json
int leerJson(String mensaje_recibido){
  int b;
  DeserializationError error = deserializeJson(doc_recibido,
mensaje_recibido);
```

```
  if (error) {
    Serial.print(F("deserializeJson() failed: "));
    Serial.println(error.f_str());
  return 0;
  }
  titulo = doc_recibido["titulo"];
  int valores[] =
{doc_recibido["pin1"],doc_recibido["pin2"],doc_recibido["pin3"]};
  for (b=0; b<3; b++){
    valores_callback[b] = valores[b];
    }
  return 0;
 }

void actuador(){
  int i;
  if (strcmp(titulo, "temperatura") == 0){
    for ( i=0; i<3; i++){
      if(valores_callback[i] == 1){
      digitalWrite(pinSalidaT[i],HIGH);
      }
      else{
      digitalWrite(pinSalidaT[i],LOW);
      }
    }
  }
  if (strcmp(titulo, "humedad") == 0){
    for ( i=0; i<3; i++){
      if(valores_callback[i] == 1){
      digitalWrite(pinSalidaH[i],HIGH);
      }
      else{
      digitalWrite(pinSalidaH[i],LOW);
      }
    }
  }
  if (strcmp(titulo, "luminosidad") == 0){
    for ( i=0; i<3; i++){
      if(valores_callback[i] == 1){
      digitalWrite(pinSalidaL[i],HIGH);
      }
      else{
      digitalWrite(pinSalidaL[i],LOW);
      }
    }
  }
}

//Modulo para publicar datos
void publicarDatos(String json){
    //Convertimos String a Char
    char mensaje[80];
    json.toCharArray(mensaje,80);
    //Mostramos por pantalla
    Serial.println(mensaje);
    //Publicamos en broker
    client.publish(topic_despacho1, mensaje);
    client.publish(topic_despacho2, mensaje);
```

```
    client.publish(topic_salajuntas, mensaje);
}

//Módulo de lectura de sensores
int leerSensores(){
  humidity =dht.readHumidity();
  temperature =dht.readTemperature();
  float analog_value = analogRead(pinLUZ);
  brightness = map(analog_value,0,4095, 0,50); //Escala la lectura
analógica map(valor recibido, minimo,maximo, %minimo,%maximo)
  //Confirmamos que no hay error de lectura
  if (isnan(humidity) || isnan(temperature)) {
    Serial.println("Error al leer del sensor");
    humidity = 0; temperature = 0; brightness = 0;
    return humidity,temperature, brightness;
    }

  return temperature,humidity, brightness;
  }

void loop() {

  if (!client.connected()) {
    reconnect();
  }
  client.loop();

  long now = millis();
  if (now - lastMsg > 10000) {
    lastMsg = now;
    //Tomamos las medidas de los sensores
    temperature, humidity, brightness = leerSensores();
    //Creamos Json con los datos
    String json = mensajeJson(temperature,humidity,brightness);
    //Publicamos tema en el topic
    publicarDatos(json);
  }


}
```

Anexo 2

```
#Script de control de temperatura

import paho.mqtt.client as mqtt
import time
import json

#Variables de conexion
user_mqtt = "controltemperatura"
pass_mqtt = "1234"
host_mqtt = "192.168.0.31"
port_mqtt = 1883
clientID = "ControlScriptTemperatura"
main_topic = "control/"

#Leer Json del webhook
def readJson(data):
        global state, valor, topic_respuesta, tag
        state = data["state"]
        title = data["title"]
        print(title)
        topic_respuesta = data["message"]
        if ('alerting' in state):
                for p in data["evalMatches"]:
                        valor = p["value"]
                        valor = int(valor)

        else:
                valor = None
        tag = data["tags"]["tipo"]

        return 'OK'

#Crear Json con valores de salida
def createpayload(value1, value2, value3):
        mensaje = {}
        mensaje["titulo"] = tag
        mensaje["pin1"] = value1
        mensaje["pin2"] = value2
        mensaje["pin3"] = value3
        return mensaje

#Control de decisiones
def controlScript(valor):
        if (valor == None):
                pin1=0
                pin2=0
                pin3=0
        else:
                if (valor >= 50):
                        pin1 = 1
                        pin2 = 0
                        pin3 = 0
                if ((valor>=27) and (valor<50)):
                        pin1 = 0
                        pin2 = 1
                        pin3 = 0
```

```
                if (valor<=23):
                        pin1 = 0
                        pin2 = 0
                        pin3 = 1
                if ((valor > 23) and (valor < 27)):
                        pin1 = 0
                        pin2 = 0
                        pin3 = 0
        datos = createpayload(pin1,pin2,pin3)
        return datos

# Callback para respuesta de conexion del broker
def ConectadoMQTT(client, userdata, flags, rc):
    print("Conectado con código "+str(rc))

    #suscripción a topic de temperatura
    client.subscribe("temperatura/#", qos=1)

# Callback para cuando recibimos un mensaje
def MensajeMQTT(client, userdata, msg):
    texto = json.loads(msg.payload)
    readJson(texto)
    texto_logica = controlScript(valor)
    payload = json.dumps(texto_logica)
    topic_envio = main_topic+topic_respuesta
    MiMQTT.publish(topic_envio,payload, qos=1,retain=True )
    print(payload)
    print(topic_envio)

#Conexiones y enlaces con broker
MiMQTT = mqtt.Client(clientID)
MiMQTT.on_connect = ConectadoMQTT
MiMQTT.on_message = MensajeMQTT

MiMQTT.username_pw_set(user_mqtt, pass_mqtt)
MiMQTT.connect(host_mqtt, port_mqtt)
#Loop de interacción continua con el servidor
MiMQTT.loop_forever()
```

Anexo 3

```
#Script de control de Humedad

import paho.mqtt.client as mqtt
import time
import json

#Variables de conexion
user_mqtt = "controlhumedad"
pass_mqtt = "1234"
host_mqtt = "192.168.0.31"
port_mqtt = 1883
clientID = "ControlScriptHumedad"
main_topic = "control/"
#Leer Json del webhook
def readJson(data):
        global state, valor, topic_respuesta, tag
        state = data["state"]
        title = data["title"]
        print(title)
        topic_respuesta = data["message"]
        if ('alerting' in state):
                for p in data["evalMatches"]:
                        valor = p["value"]
                        valor = int(valor)
        else:
                valor = None
        #tag = data["tags"]["tipo"]
        return 'OK'

#Crear Json con valores de salida
def createpayload(value1, value2, value3):
    mensaje = {}
    #mensaje["titulo"] = tag
    mensaje["pin1"] = value1
    mensaje["pin2"] = value2
    mensaje["pin3"] = value3
    return mensaje

#Control de decisiones
def controlScript(valor):
        if (valor == None):
                pin1=0
                pin2=0
                pin3=0
        else:
                if (valor >= 70):
                        pin1 = 1
                        pin2 = 0
                        pin3 = 0
                if ((valor>=40) and (valor<70)):
                        pin1 = 0
                        pin2 = 0
                        pin3 = 0
                if (valor<40):
                        pin1 = 0
                        pin2 = 1
```

```
                    pin3 = 0
        datos = createpayload(pin1,pin2,pin3)
        return datos

# Callback para respuesta de conexion del broker
def ConectadoMQTT(client, userdata, flags, rc):
    print("Conectado con código "+str(rc))

    #suscripción a topic de temperatura
    client.subscribe("humedad/#",qos=1)

# Callback para cuando recibimos un mensaje
def MensajeMQTT(client, userdata, msg):
    texto = json.loads(msg.payload)
    readJson(texto)
    texto_logica = controlScript(valor)
    payload = json.dumps(texto_logica)
    topic_envio = main_topic+topic_respuesta
    MiMQTT.publish(topic_envio,payload, qos=1, retain=True)
    print(payload)
    print(topic_envio)

#Conexiones y enlaces con broker
MiMQTT = mqtt.Client(clientID)
MiMQTT.on_connect = ConectadoMQTT
MiMQTT.on_message = MensajeMQTT

MiMQTT.username_pw_set(user_mqtt, pass_mqtt)
MiMQTT.connect(host_mqtt, port_mqtt)

#Loop de interacción continua con el servidor
MiMQTT.loop_forever()
```

Anexo 4

```
#Script de control de Luminosidad

import paho.mqtt.client as mqtt
import time
import json

#Variables de conexion
user_mqtt = "controlluminosidad"
pass_mqtt = "1234"
host_mqtt = "192.168.0.31"
port_mqtt = 1883
clientID = "ControlScriptLuminosidad"
main_topic = "control/"

#Leer Json del webhook
def readJson(data):
        global state, valor, topic_respuesta, tag
        state = data["state"]
        title = data["title"]
        print(title)
        topic_respuesta = data["message"]
        if ('alerting' in state):
                for p in data["evalMatches"]:
                        valor = p["value"]
                        valor = int(valor)
        else:
                valor = None
        tag = data["tags"]["tipo"]
        return 'OK'

#Crear Json con valores de salida
def createpayload(value1, value2, value3):
    mensaje = {}
    mensaje["titulo"] = tag
    mensaje["pin1"] = value1
    mensaje["pin2"] = value2
    mensaje["pin3"] = value3
    return mensaje

#Control de decisiones
def controlScript(valor):
        if (valor == None):
                pin1=0
                pin2=0
                pin3=0
        else:
                if (valor >= 70):
                        pin1 = 1
                        pin2 = 0
                        pin3 = 0
                if ((valor>=40) and (valor<70)):
                        pin1 = 0
                        pin2 = 0
                        pin3 = 0
                if (valor<40):
                        pin1 = 0
```

```
                        pin2 = 0
                        pin3 = 1
        datos = createpayload(pin1,pin2,pin3)
        return datos

# Callback para respuesta de conexion del broker
def ConectadoMQTT(client, userdata, flags, rc):
    print("Conectado con código "+str(rc))

    #suscripción a topic de temperatura
    client.subscribe("luminosidad/#",qos=1)

# Callback para cuando recibimos un mensaje
def MensajeMQTT(client, userdata, msg):
    texto = json.loads(msg.payload)
    readJson(texto)
    texto_logica = controlScript(valor)
    payload = json.dumps(texto_logica)
    topic_envio = main_topic+topic_respuesta
    MiMQTT.publish(topic_envio,payload, qos=1, retain=True)
    print(payload)
    print(topic_envio)

#Conexiones y enlaces con broker
MiMQTT = mqtt.Client()
MiMQTT.on_connect = ConectadoMQTT
MiMQTT.on_message = MensajeMQTT

MiMQTT.username_pw_set(user_mqtt, pass_mqtt)
MiMQTT.connect(host_mqtt, port_mqtt)

#Loop de interacción continua con el servidor
MiMQTT.loop_forever()
```

Anexo 5

```
from flask import Flask, request, abort
import paho.mqtt.client as mqtt
import json
app = Flask(__name__)


@app.route('/webhook', methods=['POST'])

def ppal():
        data = readJson(request.json)
        mqtt_conection(data,request.json)
        return 'OK'
def readJson(json):
        #data = {}
        #data["topic"] = json["message"]
        data = json["message"]
        return data

def mqtt_conection(data,message):
        topic = data
        payload = json.dumps(message)
        sendMQTT = mqtt.Client(client_id='Webhook')
        sendMQTT.username_pw_set(username="webhook",
password="1234")
        sendMQTT.connect('192.168.0.31',1883)
        sendMQTT.publish(topic, payload, qos=1, retain=True)
        return 'OK'


if __name__ == '__main__':
        app.run(host='0.0.0.0')
```