UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

"IMPLEMENTACIÓN DE UN SISTEMA DE MONITORIZACIÓN REMOTA DE SENSORES MEDIANTE MQTT Y KUBERNETES"

TRABAJO FIN DE GRADO

Septiembre - 2022

AUTOR: Jose Lucerga Marco

DIRECTORES: Katja Gilly De La Sierra-Llamazares

Eduardo Yubero Funes

AGRADECIMIENTOS

Me gustaría agradecer a mi familia toda la ayuda y apoyo que me han transmitido durante estos años de carrera. Gracias por acudir a las celebraciones en los buenos momentos y gracias por el apoyo en los no tan buenos.

Quisiera agradecer también los consejos, paciencia, tiempo e implicación de mis dos directores de proyecto, Katja Gilly De La Sierra-Llamazares y Eduardo Yubero Funes, ya que sin ellos este proyecto no hubiera sido posible.

También quiero agradecer a todos los profesores y compañeros con los que he tenido la suerte de compartir lecciones y momentos que me han servido para desarrollarme como mejor ingeniero y persona.

Finalmente agradecer a todas las personas cercanas que se han interesado por este camino que emprendí hace años y que está llegando a su fin.

Índice general

Índice de ilustraciones

Índice de tablas

1. INTRODUCCIÓN

El presente Trabajo Fin de Grado consiste en el estudio, diseño e implementación de un servicio de medición y monitorización en tiempo real, vía web, de las variables meteorológicas de temperatura y humedad. Para ello implementamos una solución de Internet of Things (IoT) sobre un servidor de Kubernetes, también conocido como K8s, una solución de computación virtualizada basada en contenedores, muy popular en implementaciones recientes por su optimización de recursos y escalabilidad

1.1. Motivación

Dada la situación sanitaria actual, el control de calidad del aire en espacios cerrados se ha convertido en un punto clave a la hora de cuidar nuestra salud, sobre todo, en el trabajo, escuelas y centros sanitarios. La monitorización de las variables que caracterizan la calidad del aire es el primer paso a la hora de mantener un control. La posibilidad de obtener estos datos en tiempo real permite agilizar la toma de decisiones, mientras que el estudio de los históricos permite aumentar nuestro conocimiento en este campo. La obtención de las medidas de estas variables supone el uso de una enorme cantidad de dispositivos de medición, ya que al menos debemos emplear uno por estancia.

"*Internet of things"* (IoT) es una tecnología ideal para implementar este servicio, ya que desarrolla eficientemente el envío de información desde una gran cantidad de dispositivos hacia un único servidor. Una vez obtenemos la información, esta debe ser almacenada, tratada y enviada en una forma visualmente estructurada, que permita su estudio y facilite la toma de decisiones.

Para llevar a cabo este envío y procesado de la información, es necesario el uso de diferentes aplicaciones o microservicios que, trabajando en serie o cascada, estructuran y envían la información a la siguiente aplicación, dando lugar a un servicio completo.

Desde el punto de vista de la infraestructura, existen diferentes modelos de asociación a la hora de estructurar la disposición de las diferentes aplicaciones que forman nuestro servicio global.

En primer lugar, se presenta la estructura tradicional, en el que varias aplicaciones trabajan en un mismo servidor físico. Esto fuerza a las distintas aplicaciones a competir con el resto para conseguir recursos. Este modelo de servidor no presenta opciones viables de escalabilidad ni herramientas de aislamiento y seguridad entre aplicaciones.

Mas tarde surgió la virtualización, que consiste en crear sobre un Sistema Operativo (S.O), entornos independientes completos dentro de un mismo servidor físico, que reciben el nombre de máquinas virtuales. Esto da lugar a una mejora en el uso medio de recursos, seguridad e independencia de aplicaciones, pero debido a que cada entorno virtualizado está corriendo un S.O, puede suponer un uso ineficiente de los recursos en casos concretos, especialmente cuando se trate de entornos IoT.

Finalmente tenemos la estructura basada de contenedores, paquetes de software que incluyen todo lo necesario para ejecutar un programa. En este modelo, disponemos de un servidor físico, configurado con cualquier S.O, sobre el que instalamos un *container-runtime*, un entorno de ejecución ligero sobre el que desplegamos los contenedores. El *container-runtime* también se encarga de actuar como puente para la comunicación entre el S.O y los contenedores. Al consumir pocos recursos permite que la carga computacional esté concentrada en las aplicaciones. Por otro lado, al ejecutarse de manera independiente al S.O, permite que las aplicaciones estén aisladas, y por lo tanto protegidas.

De esta manera, el uso de contenedores se ha convertido en la manera más eficaz a la hora de desplegar un gran número de proyectos, principalmente relacionados con

microservicios. Sin embargo, presenta una serie de debilidades que limitan su enorme potencial:

- Difícil administración de servidores formados por cientos de contenedores.
- Malgasto de recursos, cuando el número de peticiones por parte de los usuarios tienen un descenso, ya que seguimos manteniendo activos contenedores en desuso.
- Limitación en la potencia, cuando aumentan las peticiones y ya hemos alcanzado el máximo en el uso de recursos.

A lo largo de este proyecto vamos a explicar y demostrar como el uso de Kubernetes elimina estas debilidades y otorga un gran número de mejoras actuando como un administrador de contenedores, más conocido como el orquestador de contenedores.

1.2. Estado del arte

A lo largo de la historia de Internet se han empleado diferentes modelos de infraestructura o despliegue de servicios, en los que las aplicaciones que brindan de servicios a los usuarios se han instalado en las máquinas de computación siguiendo los siguientes modelos.

1.2.1. Despliegue tradicional

Al comienzo de la era de Internet, las organizaciones ejecutaban aplicaciones en servidores físicos. No había ninguna manera de definir una frontera de recursos para las aplicaciones que residían en un mismo servidor y esto causaba problemas de asignación de recursos.

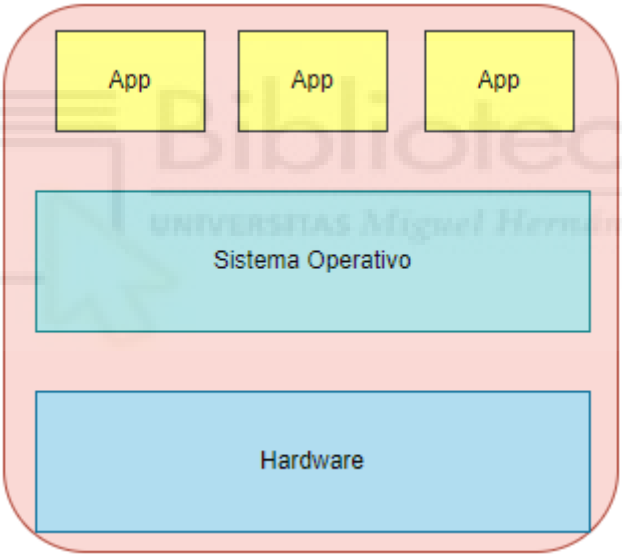

Figura 1.1: Arquitectura servidor tradicional.

Por ejemplo, si varias aplicaciones se están ejecutando en un mismo servidor físico, puede haber ocasiones en las que una aplicación monopolice la mayoría de los recursos, por lo tanto, la otra aplicación que comparte servidor sufrirá un descenso en su rendimiento.

Una solución para este tipo de arquitectura es ejecutar las aplicaciones en distintos servidores físicos, de manera que cada aplicación disponga de recursos individuales. En

este caso es imprescindible conocer la cantidad de recursos necesarios para cada aplicación y diseñar servidores a medida. Sin embargo, esta solución supone un alto coste de puesta en marcha y mantenimiento, y no es viable en cuanto a la escalabilidad del servicio.

1.2.2. Despliegue Virtualizado

Como solución, surgió la virtualización. Permite ejecutar varias Máquinas Virtuales (VMs) en un único servidor. Cada VM actúa como un elemento computacional independiente y aislado. Para ello, los recursos del servidor son virtualizados, es decir, asociados a cada VMs según las necesidades de la aplicación que va a ejecutar. La virtualización permite a las aplicaciones que corren en un mismo servidor estar aisladas del resto y aporta un mayor nivel de seguridad, ya que las aplicaciones no pueden acceder a la información que se encuentre en otra VMs.

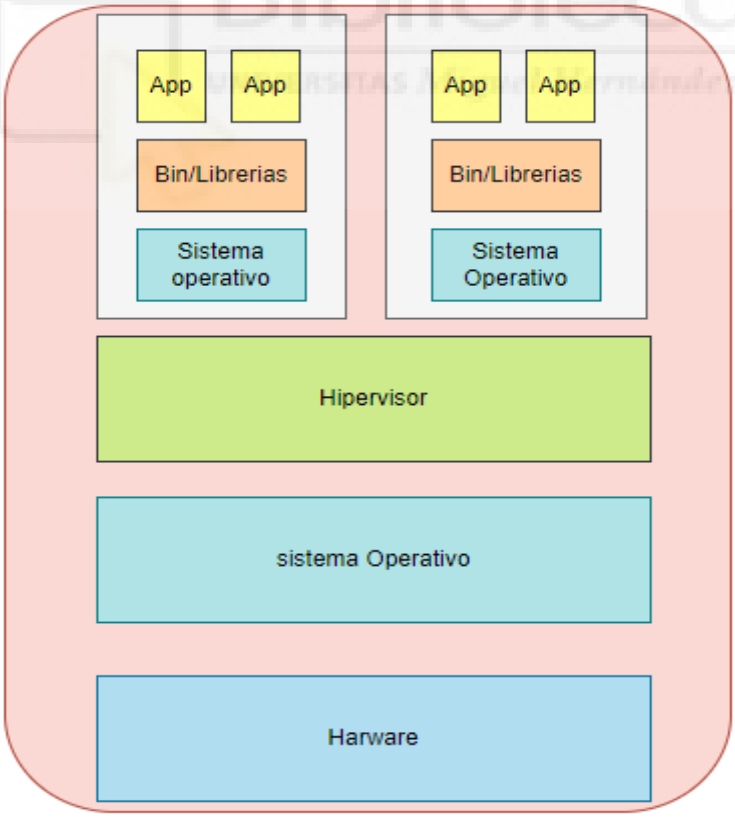

Figura 1.2: Arquitectura servidor virtualizado.

La virtualización permite una mejor utilización de los recursos del servidor y permite una escalabilidad superior ya que las aplicaciones se pueden actualizar de manera más sencilla y segura, sin afectar al resto de VMs, reduciendo costes y tiempo.

Cada VM es una instalación completa que ejecuta todos los componentes, incluido su propio sistema operativo, gracias a su hardware virtualizado. La debilidad de este modelo radica en que cada VM debe ejecutar un S.O completo, para lo que consume una cantidad notable de recursos.

1.2.3. Despliegue Container

Los contenedores son parecidos a las VMs, tienen su propio sistema de archivos, usan un porcentaje asignados del CPU, RAM y disco duro.

Sin embargo, en vez de ejecutarse sobre un Sistema Operativo distinto por cada VMs, los contenedores se ejecutan sobre un *container-runtime*. Este realiza la función de un S.O mucho más ligero que permite realizar todas las funciones necesarias y tener una asignación virtual de recursos. De esta manera no se gastan recursos en mantener la ejecución de un S.O clásico y se aprovechan para la ejecución de la aplicación.

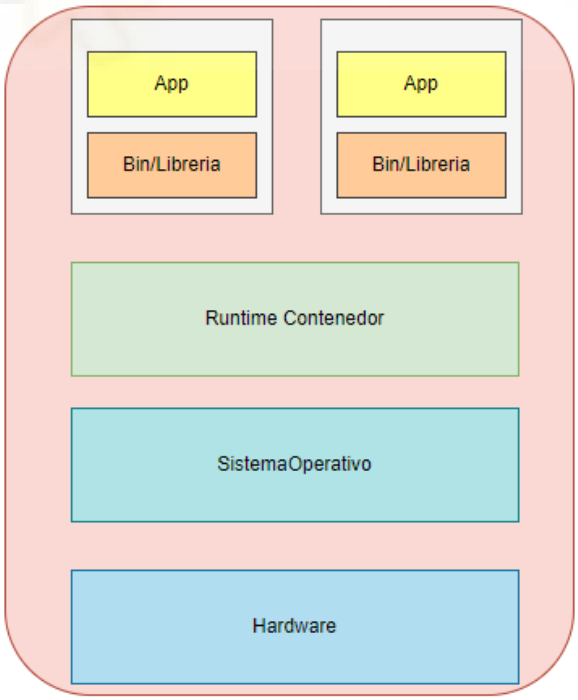

Figura 1.3: Arquitectura servidor contenerizado.

Gracias al conjunto contenedor – *container-runtime*, los servicios que se llevan a cabo son ligeros e independientes del SO en el que se albergan, siendo portátiles a través de distintos S.O y la nube.

1.2.4. Despliegue K8s

Este modelo agrega un nivel a la arquitectura anterior, permitiendo administrar un conjunto de contenedores. Orquesta la infraestructura de cómputo, redes y almacenamiento para que los usuarios no tengan que hacerlo.

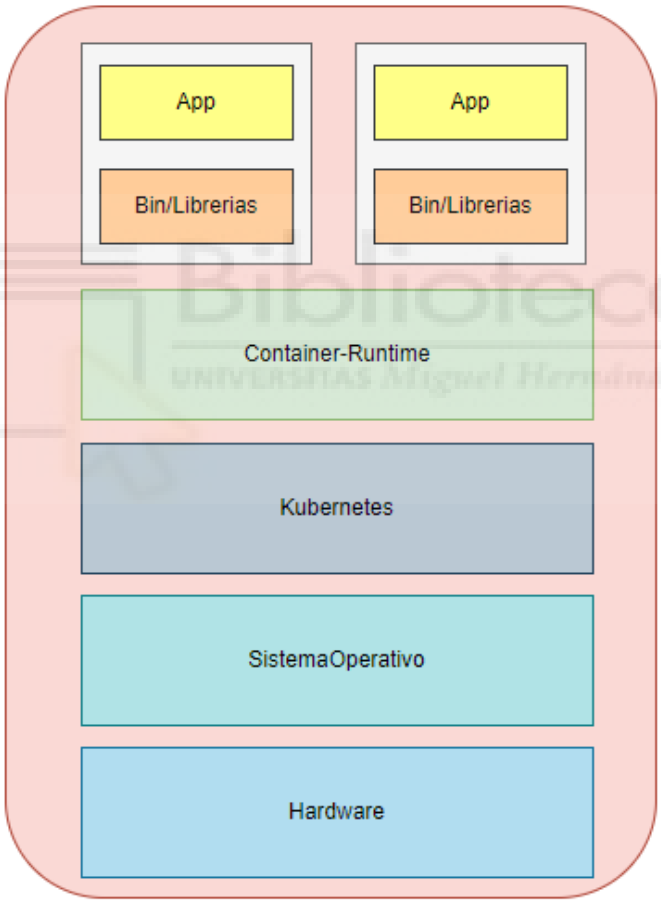

Figura 1.4: Arquitectura servidor de Kubernetes

Este nivel añadido automatiza el despliegue, configuración y mantenimiento de proyectos basado en microservicios.

1.3. Objetivos

El objetivo principal de este proyecto es el almacenamiento y monitorización remota en tiempo real de medidas meteorológicas captadas por sensores.

A nivel de utilidad, los objetivos son:

-Posibilidad de añadir sensores de distintas variables.

-Uso de alertas.

-Mantenimiento sencillo del servidor y de los dispositivos de medición.

-Coste reducido de los dispositivos y servidor.

Por otro lado, mediante el uso de las distintas funcionalidades y características de IoT y K8s pretendemos alcanzar los siguientes objetivos:

- **Escalabilidad:** Conseguir que nuestra aplicación sea capaz de mandar ordenes de escalado cuando se requiera un aumento o descenso de los recursos y que se obtengan de forma automática.
- **Despliegue y compatibilidad:** Hacer posible que la aplicación se pueda desarrollar y poner en producción de manera sencilla e independientemente del S.O que haya por debajo.
- **Bajo consumo de recursos**. Lograr maximizar la eficiencia de nuestro servidor manteniendo una correcta funcionalidad.
- **Disponibilidad**. Conseguir que la aplicación se encuentre disponible para los usuarios el máximo tiempo posible.
- **Baja inversión**: Conseguir que en una misma máquina puedan ejecutarse independientemente distintos microservicios que realicen funciones sencillas, que de manera conjunta den lugar a un proyecto complejo.

1.4. Estructura de la memoria

En el presente capítulo 1 se ha planteado el motivo por el que se ha decidido realizar este proyecto y se han establecido los principales objetivos que se deben alcanzar. En el capítulo 2 se hará una breve descripción del hardware y de las herramientas software y los lenguajes utilizados para el desarrollo de la plataforma web. Después, en el capítulo 3 se incluye parte del código fuente resultante de la programación, así como la descripción del flujo de nuestras medidas y las pruebas llevadas a cabo. Por último, se presentan las conclusiones y posibles ampliaciones futuras.

2. MATERIAL Y MÉTODOS

Nuestro proyecto consta de 2 componentes principales. En primer lugar disponemos de la estación, donde mediante el uso de distintos componentes electrónicos se toman los datos de temperatura y humedad, y se envían hacia nuestro segundo componente. Este es un servidor que recibe los datos tomados por la estación, los almacena y muestra gráficamente.

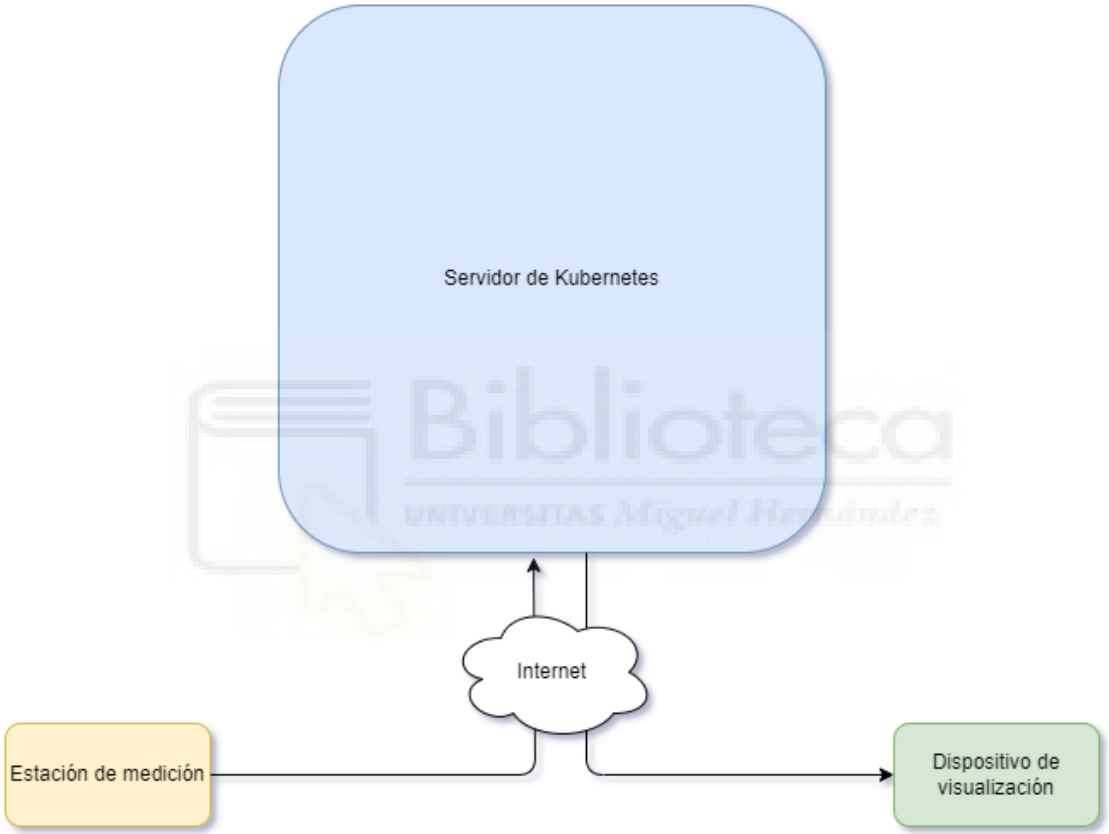

Figura 2.1: Visión general del proyecto

A lo largo de este apartado vamos a exponer las características principales de nuestros 2 componentes, dando a conocer sus principales elementos y funcionamiento básico, de una manera teórica. A continuación, vamos a explicar brevemente el protocolo de comunicación que empleamos entre nuestra estación de medición y nuestro servidor, el MQTT (protocolo de mensajería estándar para *Internet of Things*). Para concluir vamos a presentar los contendores que hemos seleccionado para llevar a cabo los distintos microservicios.

Cabe mencionar que en nuestro primer componente se ha desarrollado un sistema básico de registro de datos basados en un microcontrolador Arduino con acceso a web. El desarrollo de estos dispositivos ha sido muy importante durante la última década y existen muchos sistemas desarrollados tanto en la bibliografía como en la web. Es por esto que se ha centrado el desarrollo del proyecto en la segunda parte, ya que se basa en una tecnología más moderna y menos utilizada, que está siendo adoptada por las grandes empresas tecnológicas.

2.1. Estación de captura de datos

Nuestra estación está formada por una placa programable con Wi-Fi integrado que ejecuta Arduino, y dos sensores, uno de temperatura y otro de humedad. Uno de los objetivos a conseguir con este proyecto es que sea escalable, por ello hemos escogido una placa que acepta más conexiones, por lo que en un futuro podemos agregar nuevos sensores, como pueden ser de oxígeno (O2) y dióxido de carbono (CO2) o de luz.

A continuación, vamos a describir brevemente las características principales de estos componentes.

Placa NodeMCU V3 Wi-Fi - ESP8266: Es una placa programable, altamente empleada en el mundo IOT, basada en el chip ESP826 que permite la transferencia de datos a través de Wi-Fi mediante la pila completa TCP/IP. Por otro lado, la comunicación con los sensores se realiza a través del protocolo I2C. Esta placa es compatible con Arduino, facilitando la programación, y tiene un coste muy reducido.

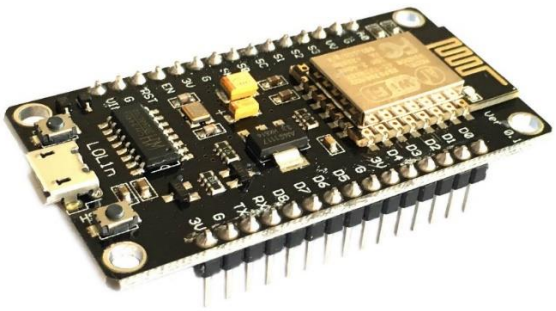

Figura 2.2: Placa ESP8266

Arduino: Es una plataforma basada en *open-source* implementada en un hardware y software muy sencillo de utilizar. Utiliza un software con una estructura básica muy sencilla, que consta de 2 partes fundamentales: En primer lugar configuramos el *setup(),* primera función que se ejecuta, donde se declaran las variables. En segundo lugar tenemos el *loop(),* que incluye el código que se ejecuta continuamente en bucle.

Sensores DHT 11 & 22: Se trata de 2 sensores muy sencillos formados por un sensor capacitivo de humedad y un termistor. Ambos incluyen un chip básico que realiza una conversión analógico-digital y emite una señal con la temperatura y la humedad. El tiempo de respuesta del DHTT11 es de 6 segundos en condiciones óptimas. Hemos añadido un margen de error de 2s a la toma de datos, configurando una captura cada 8 segundos y adaptando el resto del proyecto a este valor.

Figura 2.3: Sensores DHT11 y DHT22

2.2. Kubernetes

Podemos definir Kubernetes (K8s, K-ubernete-s, siendo el 8 las letras de la palabra ubernete) como un orquestador de contenedores, siendo el encargado de administrar todos los contenedores y las interacciones que permiten que se comuniquen y puedan trabajar de manera conjunta. Está basado en un sistema *open-source* para la automatización, despliegue, escalado y administración de aplicaciones contenerizadas. Como explicaremos más adelante, es un modelo de servidor que alcanza su máximo potencial cuando lo desplegamos en la nube.

Por lo tanto, para entender cuáles son las ventajas de K8s, primero debemos comprender la tecnología de los contenedores y conocer tanto sus características como su arquitectura y funcionamiento

2.2.1. Contenedores

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 de un entorno informático a otro. Se crean a partir de imágenes de contenedores que se despliegan sobre un entorno llamado *container-runtime*, que trabaja de manera conjunta con el sistema operativo (S.O) para ejecutar y mantener activos los procesos del contenedor.

Una imagen de contenedor es un paquete de software ligero, independiente y ejecutable que incluye todo lo necesario para ejecutar una aplicación: código, tiempo de ejecución, herramientas del sistema, bibliotecas del sistema y configuraciones.
Las imágenes de contenedor se convierten en contenedores en tiempo de ejecución. Está disponible para aplicaciones basadas en Linux y Windows. El software del contenedor siempre se ejecutará de la misma manera, independientemente de la infraestructura. Los contenedores aíslan el software de su entorno y garantizan que funcione de manera uniforme a pesar de las diferencias, por ejemplo, entre el desarrollo

y la puesta en escena. En un servidor de Kubernetes, los contenedores se ejecutan dentro de la unidad mínima, los *pods*.

2.2.2. Estructura de Kubernetes

Un *pod* puede estar formado por varios contenedores estrechamente relacionados o por un solo contenedor, como es el caso de este proyecto. Los *pods* se ejecutan dentro de nodos, que son las unidades básicas de computación en Kubernetes. Un nodo es una máquina física independiente. A su vez, un proyecto puede estar formado por diferentes nodos, dando lugar a un *cluster*. Todos estos nodos, llamados nodos trabajadores (*worker-nodes)*, son los encargados de realizar el cálculo computacional para ejecutar las aplicaciones y durante todo momento trabajan bajo el control y la supervisión del nodo maestro.

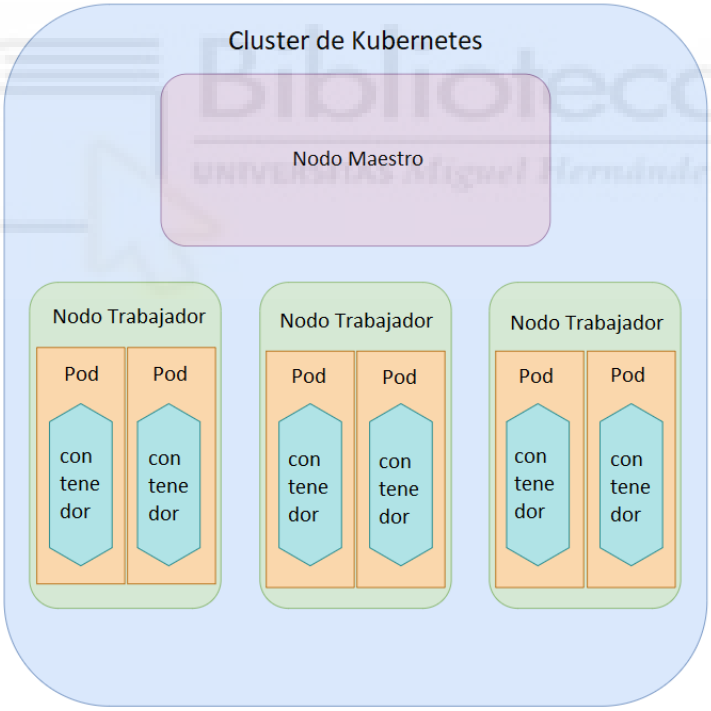

Figura 2.4: Estructura cluster K8s

En conclusión, un servidor de kubernetes está formado por un nodo maestro, que controla a uno o varios nodos trabajadores, donde residen los *pods* que despliegan los contenedores que ejecutan el código final de nuestro proyecto. A continuación, se explica las características principales de estos 2 nodos.

2.2.2.1. Nodo Maestro

El nodo maestro se encarga de gestionar el *cluster* de kubernetes, guarda información sobre los distintos nodos trabajadores, y monitoriza los contenedores y los *pods* donde se están ejecutando. Todas estas funciones las realiza a través de los siguientes componentes:

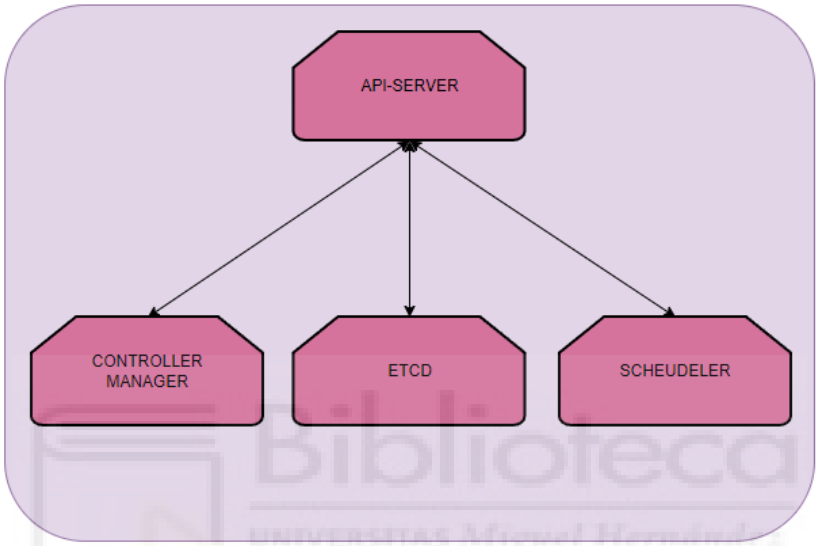

Figura 2.5: Componentes nodo maestro

- ***Kube API-Server***: Es el componente principal de administración. Es el encargado de orquestar todas las operaciones que se llevan a cabo dentro del *cluster* o servidor. Habilita el API-kubernetes, que permite al usuario realizar funciones de administración. Controla al resto de componentes y la comunicación entre los nodos trabajadores y maestro.
- ***Controller Manager***: Es el responsable de llevar a cabo diferentes funciones de control sobre el sistema. Se encarga de conectar y desconectar nodos, administrar situaciones en las que los nodos no están disponibles. También se encarga de que el número deseado de contenedores esté durante todo el tiempo trabajando dentro de un grupo de réplicas
- ***ETCD***: Es la base de datos que almacena la información en un formato de par clave-valor, en el que cada variable o clave, tiene un valor concreto.

- ***Scheduler***: Identifica el nodo correcto en el que hay que inicializar un contenedor basándose en los requerimientos de recursos, la capacidad disponible de los nodos trabajadores o cualquier otra política o reglas definidas.

2.2.2.2. Nodo Trabajador

El nodo trabajador se encarga de ejecutar las aplicaciones que corren dentro de los contenedores según las indicaciones del nodo maestro. Está formado por los siguientes componentes esenciales:

- ***Kubelet***: Es un agente que se encuentra en cada uno de los nodos. Es el responsable de supervisar todas las acciones que se llevan a cabo en el nodo. Se encarga de interactuar con el nodo maestro, escuchando órdenes del *kube api-server*. Contacta con este para unirse al *cluster*, recibir información sobre los contenedores que deben ser inicializados en su nodo y enviar reportes sobre el estado de su nodo e información sobre cada contenedor que está ejecutando. Se encarga de desplegar o destruir contenedores.

- ***Kube-proxy***: Es el encargado de facilitar una serie de reglas para la identificación de los diferentes contenedores, para que estos puedan intercambiar información con el resto de los contenedores y que se puedan comunicar con otros nodos.

- ***Container-runtime***: Es el software que ejecuta los contenedores y administra las imágenes de contenedores en un nodo. En nuestro proyecto vamos a emplear Containerd, el *container-runtime* con el que K8s trabaja por defecto.

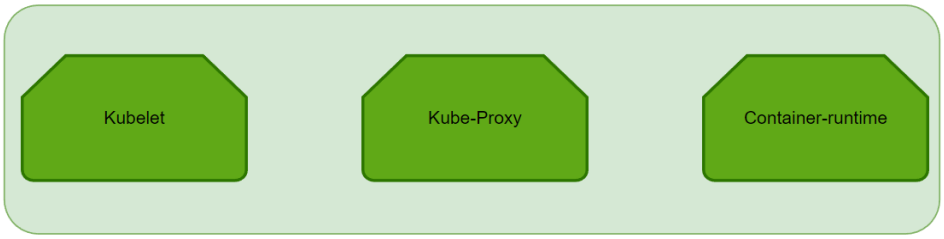

Figura 2.6: Componentes nodo trabajador

2.2.3. Componentes de Kubernetes

Una vez comprendidos los dos componentes que forman un *cluster* de K8s (nodos maestro y trabajador), a continuación, vamos a dar a conocer los distintos objetos y funcionalidades que se desarrollan en los nodos trabajadores.

2.2.3.1. *Pod*

Representan la unidad mínima dentro de nuestro sistema, en ellos residen los contenedores que llevan a cabo los procesos principales de nuestro proyecto.

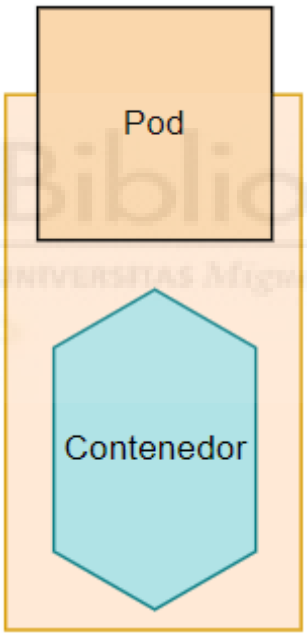

Figura 2.7: Pod

Gracias a los servicios, cada *pod* dispone de una IP única accesible desde cualquier otro *pod* que se encuentra dentro de nuestro *cluster*. Se recomienda un único contenedor por cada *pod*, aun así, se pueden disponer de varios cuando trabajan de manera conjunta, en estos casos siempre debe haber un contenedor principal. Los contenedores se comunican entre ellos dentro de un *pod* vía localhost y puerto. Cada *pod* tiene un identificador llamado *label*, que sirve para asociarlo a un *recplicaset*.

2.2.3.2. *Replicaset*

Es un controlador que se encarga de mantener activo en todo momento el número deseado de *pods* que forman parte de su grupo. Esta función la lleva a cabo gracias a dos herramientas que permiten identificar los *pods* asociados a un *replicaset* concreto y conocer el estado de los *pods*.

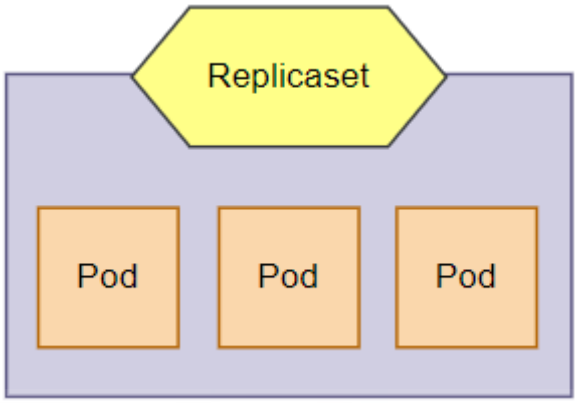

Figura 2.8: Replicaset

En primer lugar tenemos la etiqueta *matchLabels*, dentro del campo *selector*, donde especificaremos el valor del *label* configurado en los *pods* que deseamos que formen parte del *replicaset*. De esta manera, cuando el campo *label* de un *pod* coincida con el *matchlabels* de un *replicaset*, automáticamente el *pod* pasará a formar parte del *replicaset* y será administrado por el mismo.

En segundo lugar, disponemos del par "*Desired State* y *Current State*". *Desired State* es el número de *pods* que deseamos mantener, por lo que el *replicaset*, dependiendo del *Current State o* número de *pods* activos en cada momento, creará o eliminará *pods* asociados.

2.2.3.3. *Deployment*

Es un controlador que se encarga de crear, administrar y actualizar *replicasets*. Existen dos tipos de estrategias a la hora de disponer un *deployment*, que usaremos dependiendo del motivo por el cual deseamos modificar el *replicaset*.

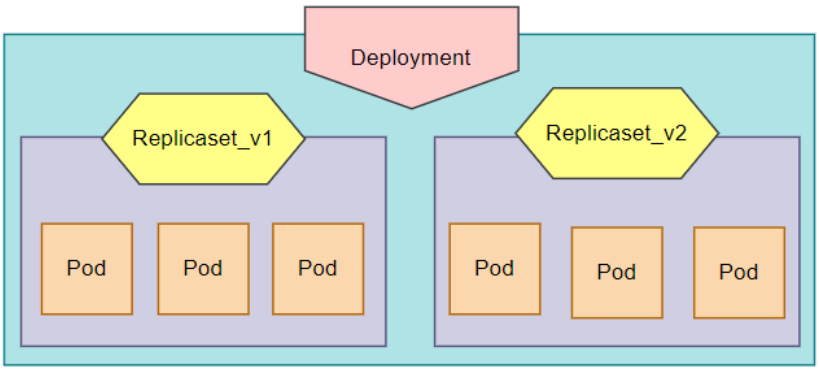

Figura 2.9: Deployment

Por un lado, tenemos el *RollingUpdate*, cuya característica principal es mantener el servicio al mismo tiempo que los *pods* involucrados se actualizan. Es la estrategia por defecto si no especificamos ninguna a la hora de declararlo en nuestro archivo de configuración YAML.

Tal y como observamos en la figura 2.10, al inicio de esta estrategia se crea un *replicaset* vacío. A continuación, se crea el primer *pod* con la versión actualizada del contenedor, y una vez verificado su correcto funcionamiento, se elimina un *pod* del *replicaset* anterior. De esta manera, a medida que se crean los *pods* en el nuevo *replicaset*, se eliminan del anterior. Finalmente disponemos del nuevo *replicaset* junto con sus *pod*s actualizados, de manera que se elimina el *replicaset* antiguo que había quedado vacío.

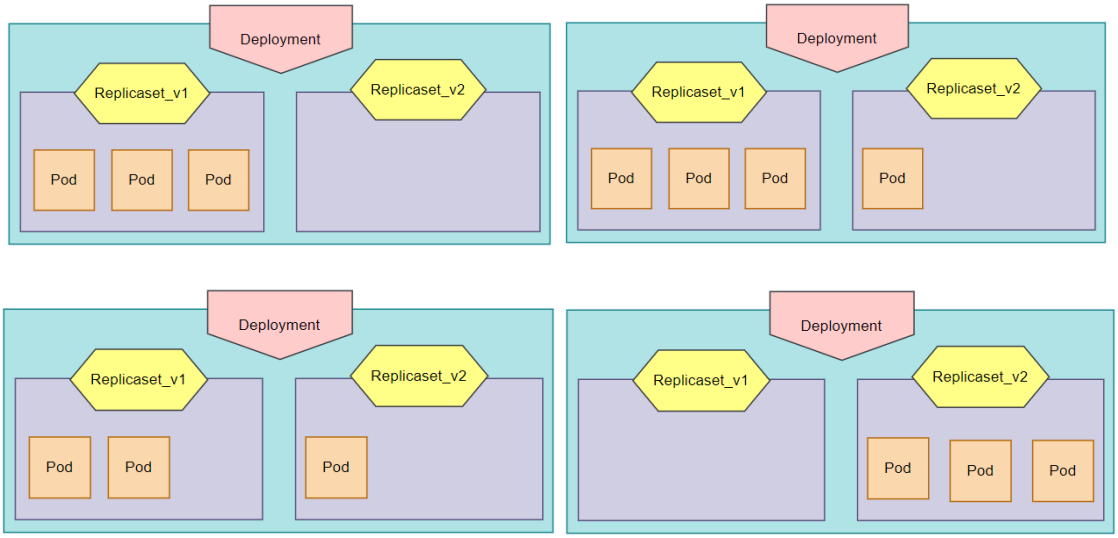

Figura 2.10: Estrategia de despliegue RollingUpdate

Por otro lado, disponemos de la estrategia *Recreate*, cuya característica es eliminar de inmediato el *replicaset*, y por lo tanto los *pods* asociados a esta. Se suele emplear en casos en los que se detectan vulnerabilidades de seguridad en alguno de los *pods*, por lo que la prioridad es detener el funcionamiento lo antes posible, pese a que el servicio se vea interrumpido.

Observando la figura 2.11, en primer lugar, se crea un nuevo *replicaset* sin *pods*. A continuación, se eliminan los *pods* de la versión anterior y finalmente se crean de golpe todos los *pods* en el nuevo *replicaset*.

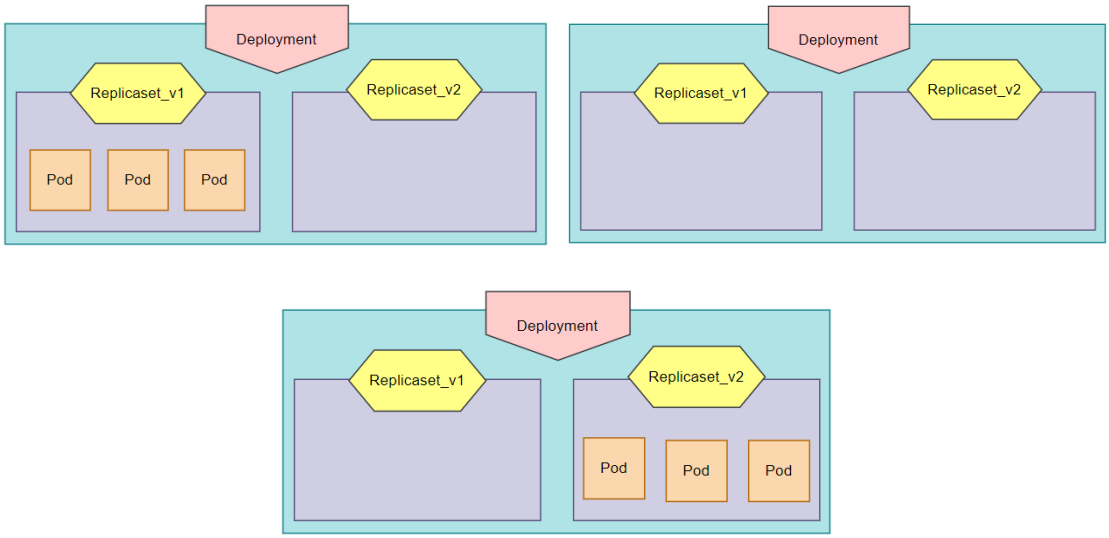

Figura 2.11: Estrategia de despliegue Recreate

2.2.3.4. Servicios

A través de los servicios dotamos de una IP y puerto a nuestros *pods*. De esta manera son capaces de identificarse y transferir información a través del protocolo de transporte TCP. En nuestro servidor se pueden establecer 2 tipos de comunicaciones, cada uno con un puerto distinto en cada servicio.

Para la comunicación entre dos *pod*s de nuestro nodo, se emplea el puerto descrito como "*port*", este será el puerto objetivo para las comunicaciones internas.

Para la conexión entre un agente externo a nuestro servidor y cualquiera de nuestros pods, utilizamos el "*nodeport*".

Por ejemplo, un *pod* que contiene una base de datos recibirá consultas de otros *pod*s a través del port, mientras que una consulta realizada por el administrador de manera remota debe ser lanzada hacia el nodeport.

2.2.3.5. *Job y Cronjob*

Estos objetos son *pod*s temporales. Los controladores de Jobs se encargan de supervisar *pod*s para que lleven a cabo tareas específicas. Existen 2 tipos de *jobs:*

-*Jos:* Este tipo de *pod* será eliminado una vez la tarea haya sido completada. Podemos utilizar esta funcionalidad para eliminar la información almacenada en una base de datos cada vez que el sistema se inicie.

-*CronJob:* Es utilizado para realizar una tarea específica cada cierto periodo de tiempo. Un ejemplo de uso es la eliminación de la información almacenada en una base de datos cada semana o cada hora.

2.2.3.6. *Configmap*

Un *ConfigMap* es un objeto API que se utiliza para almacenar datos no confidenciales en pares clave-valor. Los *pod*s pueden consumir *ConfigMaps* como variables de entorno, argumentos de línea de comandos o como archivos de configuración en un volumen.

En nuestro caso vamos a emplear los *ConfigMaps* para configurar valores dentro de nuestros contenedores. Por ejemplo, al desplegar nuestra base de datos prometheus, vamos a configurar variables como:

-Scrape Interval (s): Cada cuanto tiempo se realiza una consulta para añadir información a la base de datos.

-Target: Destino hacia el que se realiza la consulta de información.

2.2.4. YAML

A la hora de crear un proyecto de kubernetes, utilizamos archivos de configuración YAML. Es un formato de serialización de datos, su principal ventaja es la legibilidad y la capacidad de escritura. Presenta una sintaxis estricta respecto al sangrado.

A través de los archivos YAML vamos a configurar el despliegue de los *pod*s que van a albergar nuestros contenedores.

En primer lugar, vamos a desplegar los *pod*s a través de *deployments*, ya que optimizan la administración de *pod*s mediante la administración de *replicasets*.

En segundo lugar, vamos a configurar nuestros contenedores por medio de *configmaps*.

Finalmente, vamos a dotar a nuestros *pod*s de una IP y puertos, para que puedan comunicarse a través de los servicios.

Otra de las ventajas de K8s es que a la hora de desplegar nuestros objetos, mediante documentos YAML y archivos de configuración, disponemos de una herramienta que permite realizar un despliegue simultáneo. Esta herramienta es HELM.

2.2.5. Helm y Helmchart

Es un administrador de paquetes de configuración para Kubernetes. Su función es enviar un conjunto de documentos YAML y *values* a nuestro servidor K8s, que aplicará la configuración expuesta. Este conjunto de archivos se denomina Helm-chart. Es una manera conveniente de agrupar colecciones de archivos, de manera que mediante el uso de los siguientes comandos podemos instalar, realizar cambios en tiempo de ejecución y volver a versiones anteriores:

`Helm install` – Instalación de un proyecto de K8s.

`Helm upgrade` – Aplica cambios de configuración en nuestro servidor en tiempo real, sin que el servicio se vea afectado. Por ejemplo, si modificamos la versión de un contenedor, el *pod* antiguo donde reside el contenedor a cambiar no será eliminado hasta que la versión actualizada del contenedor se esté ejecutando en un nuevo *pod*.

`Helm rollback` – Establece una configuración anterior en tiempo real sin que el servicio se vea afectado.

Como hemos comentado anteriormente, un archivo esencial a la hora de configurar nuestros *pods* es el documento *values*. Es un documento *YAML* en el que vamos a especificar los recursos computacionales, en cuanto a memoria y CPU, asociados a cada *pod*. Debemos configurar 2 valores:

- Request: Son los valores garantizados para el *pod*.
- Limit: Valores máximos que el *pod* no puede superar.

La asociación se realiza gracias al identificador *label*. En caso de superar estos valores, nuestro nodo maestro se encargará de ejecutar un escalado de manera automática, previamente configurado.

2.2.6. Escalados

Kubernetes permite configurar diferentes modos de escalado automático, de manera que el sistema ajusta la cantidad de recursos empleados según la carga de trabajo en cada momento. De esta manera asegura una gran eficiencia y una rápida respuesta en el control de recursos frente a cambios en la actividad y demanda de servicio de nuestra plataforma, aumentando o disminuyendo los requerimientos de utilización de los recursos del sistema virtualizado.

Podemos definir los escalados según el tipo de objeto involucrado y según la estrategia de escalado.

Por un lado, distinguimos entre escalado de nodos y escalado de *pod*s.

Por otro lado, distinguimos entre escalado horizontal y vertical:

- Horizontal: Consiste en aumentar el número de *pod*s o nodos.
- Vertical: Consiste en aumentar la cantidad de recursos de CPU o memoria asociados a un *pod* o nodo.

	Horizontal	Vertical
Nodo	añadir o eliminar nodos	modificar CPU o memoria de un nodo
Pod	añadir o eliminar *pod*s	modificar CPU o memoria de un *pod*

Tabla 2.1: Tipos de escalado

El escalado horizontal de nodos es el principal motivo por el que se recomienda desplegar servidores basados K8s en la nube, ya que, en un momento de escasez de recursos computacionales, vamos a poder agregar una cantidad de nodos prácticamente ilimitada.

Dependiendo del objetivo de nuestro escalado, aplicaremos uno vertical u horizontal:

- **Vertical**: Se usa con tal de optimizar el uso de los recursos asociados al servidor. Un ejemplo sería revisar constantemente la cantidad de recursos empleados por un *pod* y en caso de salir del margen establecido en el documento values, ajustar estos valores.
- Horizontal: Se utiliza para asegurar un correcto funcionamiento del servidor ante un aumento en la carga de la aplicación. Un ejemplo es configurar un pod de nginx, encargado de recibir las peticiones http externas.

Por lo tanto, el escalado horizontal se encarga del rendimiento de nuestra aplicación, mientras que el escalado vertical se centra en el rendimiento del servidor.

Por otro lado, existe una arquitectura de K8s en la que tanto el nodo maestro como el trabajador se encuentran en una misma máquina física, MicroK8s.

2.2.7. MicroK8s

Es una versión de Kubernetes potente, sencilla y de cómodo despliegue. Es ideal para usuarios que se están iniciando en el mundo de K8s y cuyos proyectos de pequeña envergadura y testeo no requieren una gran cantidad de recursos. Esta es la solución que se ha empleado en este proyecto.

2.3. MQTT

Uno de los puntos clave del proyecto, es la comunicación entre nuestra estación de captura de datos y nuestro servidor. Como ya hemos comentado, esta transferencia de información se va a realizar a través de Internet, mediante el protocolo más usado en el mundo de IoT, el MQTT.

MQTT corresponde con las siglas de *Message Queuing Telemetry Transport,* es un protocolo ligero de comunicación entre dispositivos M2M, que trabaja bajo el protocolo TCP, basado en un sistema de publicaciones y suscripciones. Es simple y de un ancho de banda reducido. Puede realizar dos funciones: enviar un comando de control hacia un dispositivo, o leer y publicar información. Está basado en tópicos, que actúan como una palabra clave.

La comunicación está compuesta por 3 elementos: el cliente emisor, el servidor o *broker* y el cliente receptor.

Por un lado, el cliente emisor publica información en un tópico específico del *broker*. Por otro lado, el cliente receptor contacta con el *broker* y se suscribe a los tópicos sobre los que desea recibir información. Finalmente disponemos del *broker*, quien se encarga de recibir y enviar la información entre los clientes.

De esta manera, la información publicada en un tópico del *broker* por parte de un cliente emisor, será transmitida desde el *broker* hacia los clientes suscritos a este tópico.

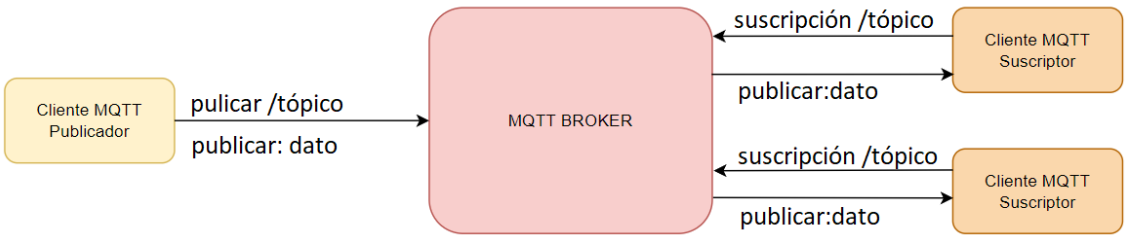

Figura 2.12: Funcionamiento protocolo MQTT

2.4. Contenedores utilizados

Una vez comprendido el funcionamiento básico de k8s y conocidas las herramientas que emplea, vamos a presentar el conjunto de contenedores que hemos escogido para llevar a cabo los microservicios que forman nuestro proyecto.

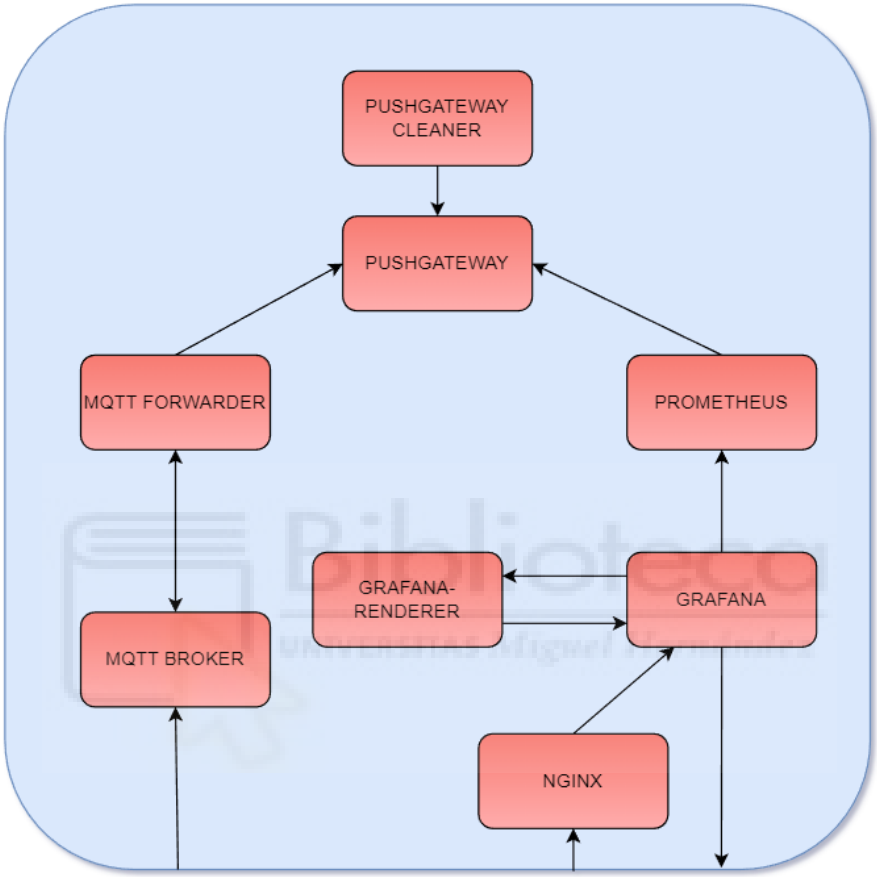

Figura 2.13: Flujo de datos entre contenedores

MQTT BROKER

En primer lugar, nuestro servidor debe ser capaz de recibir la información procedente de nuestro dispositivo de medición, que emplea MQTT como protocolo de comunicación, por lo que hemos seleccionado a Mosquitto como el *broker* que recibirá la información asociada a nuestro tópico. Hemos empleado el *broker* de Eclipse Mosquitto ya que es muy ligero y es uno de los más empleados en el mundo de IoT.

PROMETHEUS

A continuación, necesitamos una base de datos que almacene todas las medidas de temperatura y humedad para, más adelante, consultarlos a la hora de crear las gráficas presentadas al usuario final. Para ello emplearemos la base de datos Prometheus. Hemos seleccionado esta base de datos debido a que es ampliamente utilizada en el mundo de IoT, junto con el programa de visualización de datos Grafana. Presenta gran sencillez a la hora de configurar y realizar consultas. Emplea el lenguaje funcional de consultas Prometheus Query Language (promQL), que permite al usuario seleccionar y agregar datos de series temporales en tiempo real.

MQTT FORWARDER

En este punto nos encontramos con el primer obstáculo del proyecto a la hora de diseñar el flujo que seguirá nuestro dato, ya que nuestra base de datos no puede suscribirse al tópico de un *broker* MQTT, por lo que vamos a necesitar un contenedor que reciba esta información. Para ello empleamos el MQTT Forwarder, que actúa como cliente de MQTT suscrito a nuestro tópico en el *broker* de Mosquitto. Por otro lado, se encarga de modificar la estructura de la información recibida a través de MQTT, de manera que esté lista para almacenarla en la base de datos, y de enviarla al siguiente contenedor.

PUSHGATEWAY

Recibe los datos modificados por el MQTT Forwarder y los almacena a la espera de recibir la consulta por parte de nuestra base de datos.

PUSHGATEWAY CLEANER

Vamos a utilizar este *cronjob* para evitar que el Pushgateway almacene la misma información durante mucho tiempo. El proceso se ejecutará cada cierto periodo de tiempo, con tal de eliminar la información de aquellos dispositivos que lleven cierto tiempo sin enviar datos. Este contenedor recibe el nombre de Pushgateway Cleaner.

GRAFANA

Finalmente, disponemos de la información en nuestra base de datos Prometheus, por lo que únicamente nos falta darle forma y presentarla gráficamente al usuario final. Para ellos vamos a utilizar Grafana, que permite la presentación de los datos en tiempo real, así como la consulta de históricos. Otro de los motivos por los que hemos seleccionado esta plataforma es debido a que incorpora alertas, una funcionalidad muy interesante a la hora de aplicar en nuestro proyecto.

GRAFANA RENDERER

Corresponde con el *backend* que se encarga de renderizar las gráficas y paneles de Grafana.

NGINX

Es el servidor web que hemos configurado con un proxy inverso. Se encarga de recibir las peticiones por parte del usuario final y dirigirlas hacia el *pod* deseado. En nuestro caso lo hemos configurado para que reciba las solicitudes http de gráficas en el puerto 80 y las reenvíe hacia el puerto correspondiente del pod de Grafana.

3. RESULTADOS Y DISCUSIÓN

En los capítulos anteriores hemos explicado el funcionamiento básico de las tecnologías empleadas en este proyecto. A continuación, vamos a mostrar las configuraciones que hemos aplicado sobre estas herramientas y el resultado final obtenido.

En primer lugar, vamos a mostrar el flujo básico que recorre la información, desde la toma del valor en el sensor, hasta el muestreo vía web en un dispositivo, pasando por nuestro servidor de Kubernetes. A continuación, presentaremos la configuración realizada sobre nuestra estación de medición. Después vamos a explicar la configuración de los distintos *pod*s del servidor de K8s y el flujo de los datos a través de estos.

3.1. Flujo básico

Si estudiamos los dispositivos que forman nuestro proyecto como si fueran cajas negras, observamos que el flujo de la información es muy sencillo y podemos diferenciar los tres objetos principales que aportan utilidad a nuestra aplicación.

En primer lugar, disponemos de la estación de medición, que toma el dato y lo envía a través de MQTT hacia el servidor de K8s. Éste recibe el dato y tras procesarlo lo almacena, Finalmente, el usuario final mediante una petición web, accede a la información en forma de gráficas.

En la figura 3.1 podemos observar el gráfico del flujo de la información entre los distintos componentes.

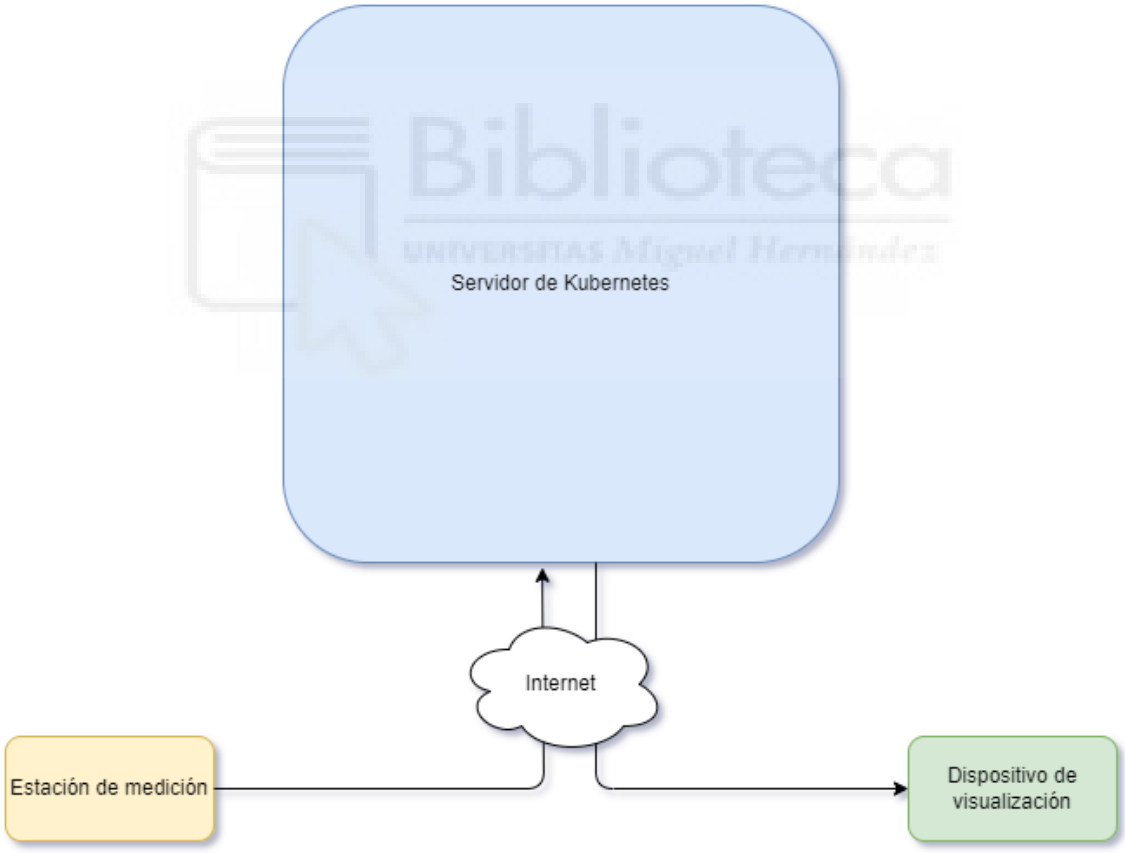

Figura 3.1: Visión general del proyecto

3.2. Estación de medición

Se encarga de realizar las siguientes funciones:

1-Toma del dato por parte del sensor.

2-Envío de la medida a placa ESP8266.

3-Procesamiento de los datos por parte de Arduino.

4- Envío de la información a través de conexión Wi-Fi, por parte del cliente MQTT hacia nuestro *broker*, en el tópico "*measurement*". Cada estación envía los datos identificados con el id del dispositivo, que corresponde con la dirección MAC de la placa ESP8266.

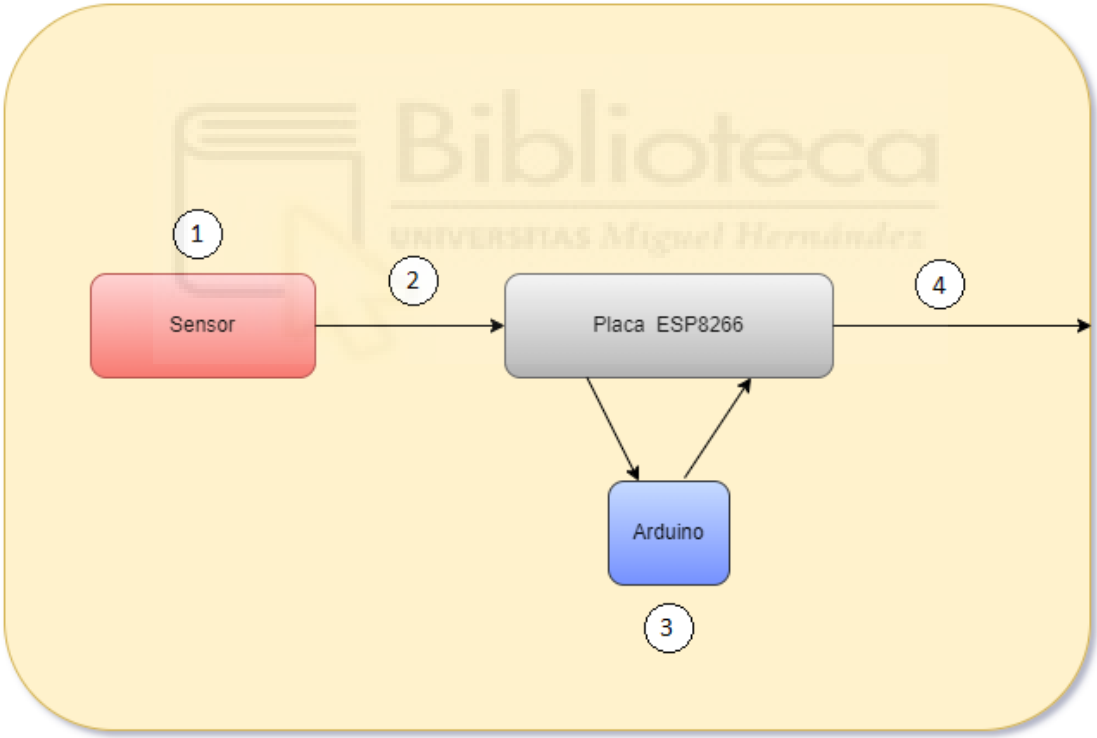

Figura 3.2: Flujo de datos en la estación de medición

La configuración de nuestra estación de medición la podemos dividir en 2 partes, una correspondiente al hardware, el montaje, y otra referida al software, la programación.

3.2.1. Montaje

Debido a que únicamente empleamos la placa ESP8266 y un sensor para medir la humedad y temperatura, el montaje es muy sencillo. Son necesarias 3 conexiones entre los dos componentes:

1. Tierra
2. Voltaje
3. Datos

En la figura 3.3 observamos las conexiones que hemos configurado para nuestro proyecto.

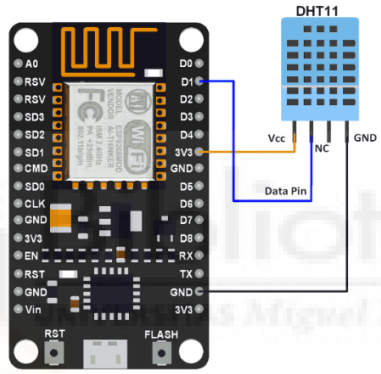

Figura 3.3: Montaje Placa ESP8266 – sensor DHT11

Finalmente debemos alimentar la placa ESP8266, para lo que disponemos de dos opciones.

La primera opción, que hemos usado a la hora de desarrollar el proyecto, es conectar la placa a un ordenador a través de un cable USB-microUSB. También podemos alimentarla a través de un transformador (cargador) de 5V.

En segundo lugar, ya que este dispositivo se va a emplear para la captura de datos meteorológicos, es posible que en el lugar en el que deseemos realizar la medición, no dispongamos de ningún dispositivo electrónico o enchufes al que podamos conectar la placa por micro-USB. En estos casos, podemos emplear una batería portátil, ya que el consumo de la placa es muy reducido.

3.2.2. Arduino

La parte de la programación de la placa ESP8266 la realizamos mediante Arduino.

A continuación, se muestra el código, cuyas funciones principales son:

- Conexión a la red Wi-Fi
- Conexión al *broker* MQTT
- Tomar el dato del sensor
- Conectarse al *broker* MQTT
- Crear y enviar el mensaje con las medidas a través de MQTT en el tópico *measurement*
- Reconectarse a la red Wi-Fi y al *broker* de MQTT en caso de pérdida de conexión.

En el segundo anexo se encuentra el código de Arduino empleado.

3.3. Kubernetes

En este apartado vamos a estudiar el flujo de la información dentro del servidor de Kubernetes.

Como hemos comentado en el apartado 2.3, vamos a emplear 8 *pod*s con sus correspondientes contenedores, que se van a encargar de recibir, almacenar y mostrar la información de las medidas obtenidas.

En la figura 3.4 podemos observar un gráfico del flujo de la información dentro de nuestro servidor y de las funciones que se emplean para lograrlo.

Disponemos de dos *pod*s accesibles desde el exterior:

-El MQTT *Broker* recibe a través de MQTT los datos captados por los sensores.

- Nginx, que tras una petición web, envía los datos de consulta a Grafana.

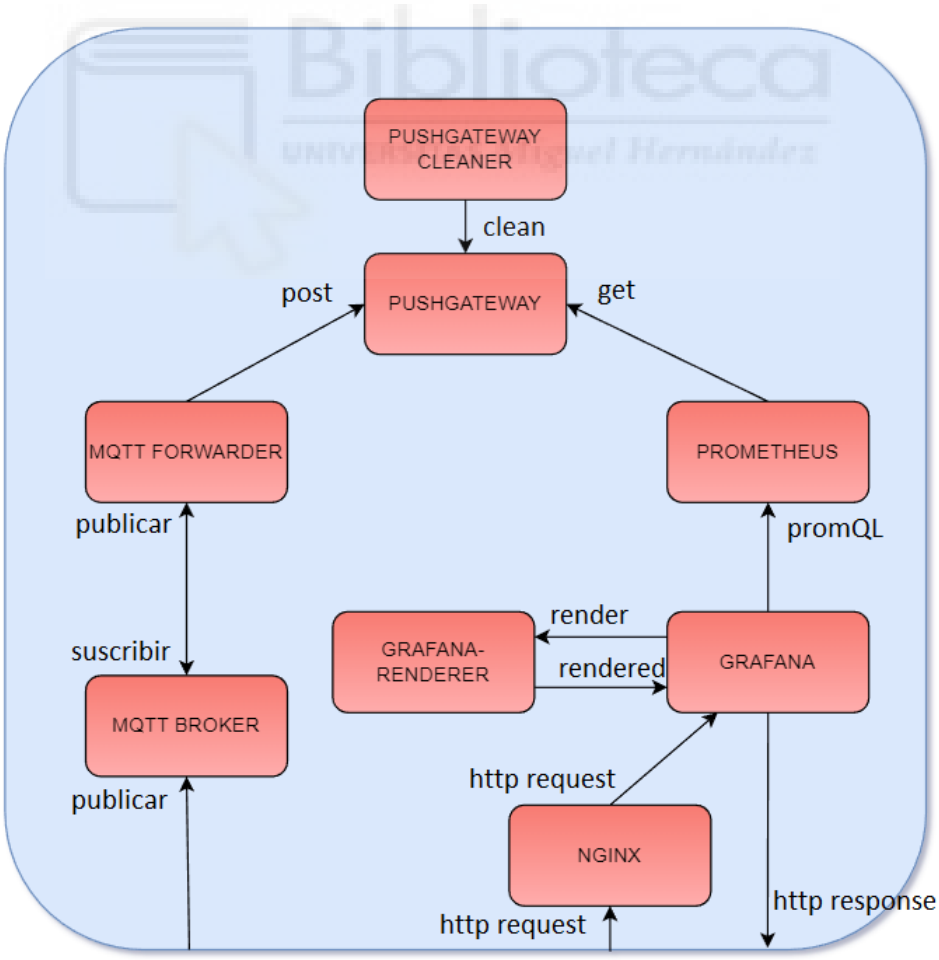

Figura 3.4: Flujo de datos y peticiones empleadas en el servidor K8s

3.3.1. MQTT Broker

Este *pod* alberga el contenedor del *broker* para el protocolo MQTT. Se encarga de recibir las publicaciones enviadas desde el cliente de nuestra estación de medición. Estas publicaciones son enviadas hacia el tópico *measurement*. A continuación, el *broker* se encarga de reenviar estas publicaciones a los clientes que se encuentran suscritos a este mismo tópico, es decir, las envía hacia nuestro siguiente *pod*, el MQTT Forwarder.

En la figura 3.5 podemos observar las 3 interacciones que lleva a cabo el *pod*:

- Nuestro cliente emisor alojado en la placa ESP8266 publica las medidas en el tópico *measurement*.
- Nuestro cliente receptor alojado en el *pod* MQTT Forwarder se suscribe al tópico *measurement*.
- El MQTT Broker envía los datos del tópico *measurement* a los clientes suscritos.

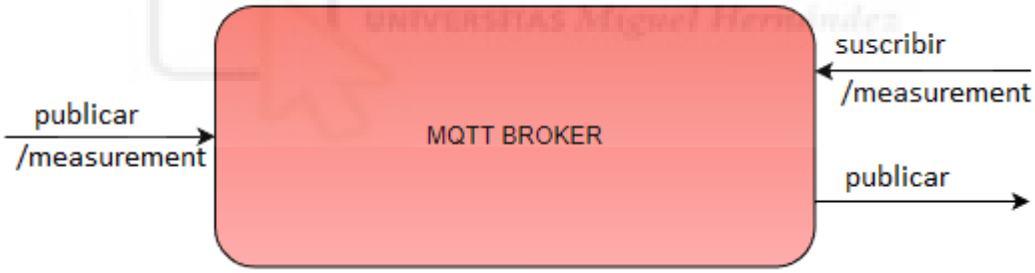

Figura 3.5: Pod MQTT Broker

3.3.2. MQTT Forwarder

Este *pod* alberga un contenedor que se encarga de 3 funciones principales:

En primer lugar, es un cliente de MQTT, que vamos a conectar a nuestro servidor de Mosquitto en nuestro *pod* MQTT Broker. Configuramos la suscripción al tópico *measurement*, por lo que va a recibir los datos de temperatura y humedad publicados en este tópico.

En segundo lugar, elimina la información sobre el tópico y mantiene la asociación entre el id del dispositivo de captura y las medidas de temperatura y humedad.

Finalmente, se encarga de enviar esta información a través de HTTP, hacia nuestro siguiente *pod*, PUSHGATEWAY.

En la figura 3.6 podemos observar las 3 interacciones que lleva a cabo el *pod*:

1. Suscripción al tópico *measurement* en el MQTT Broker.
2. Recepción de las publicaciones en el tópico *measurement*.
3. Envío de datos al pushgateway.

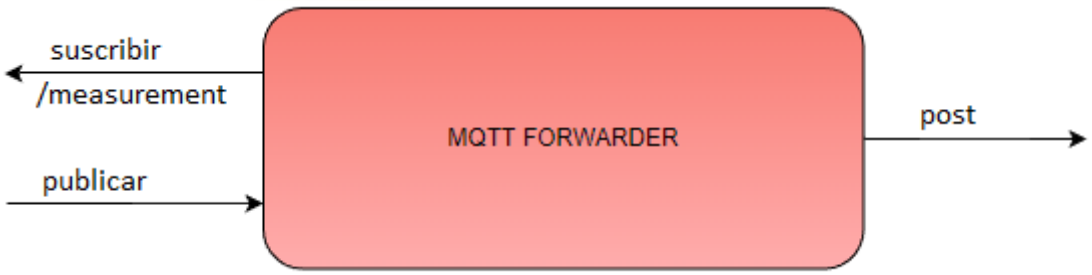

Figura 3.6: Pod MQTT Forwarder

3.3.3. Pushgateway

Se encarga de almacenar y presentar la información de manera que pueda recibir consultas desde nuestra base de datos que se encuentra en nuestro siguiente *pod*, Prometheus.

En la figura 3.7 podemos observar las 3 interacciones que lleva a cabo el *pod*:

1. Recepción de la información enviada por el MQTT Forwarder.
2. Recepción de consulta por parte de la base de datos del *pod* Prometheus.
3. Borrado de datos antiguos por parte del Pushgateway-Cleaner.

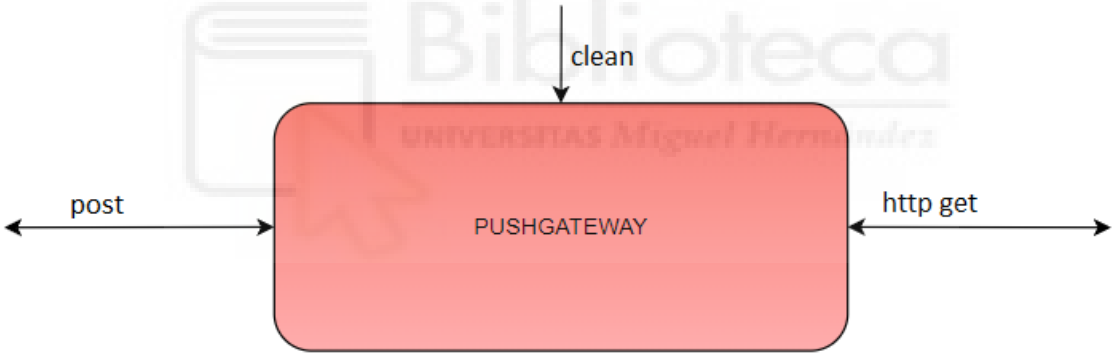

Figura 3.7: Pod Pusgateway

3.3.4. Pushgateway-Cleaner

Este *pod* corresponde con la funcionalidad de un cronjob. Como ya hemos comentado anteriormente, es un *pod* que se ejecuta cada cierto tiempo y se elimina tras realizar su función. En este caso, lo hemos configurado para que se encargue de eliminar del registro del PUSHGATEWAY las medidas de dispositivos que hace más de 5 minutos que no publican nuevos datos.

En la figura 3.8 podemos observar las 3 interacciones que lleva a cabo el *pod*:

- Cada 5 minutos accede al Pushgateway-Cleaner y se encarga de eliminar los datos asociados a los dispositivos que llevan más de 5 minutos sin publicar medidas nuevas.

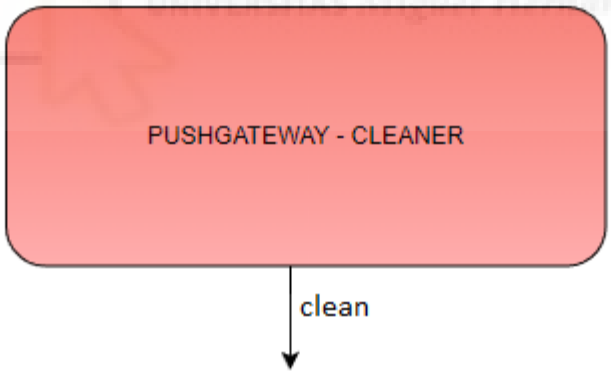

Figura 3.8: Pod Pushgateway-Cleaner

3.3.5. Prometheus

Este *pod* alberga el contenedor que almacena nuestro histórico de medidas, es nuestra base de datos. A través de una solicitud GET que se envía cada 8 segundos a nuestro *pod* Pushgateway, obtiene los datos de las medidas de temperatura y humedad, asociados al id del dispositivo de medición. Como se mencionó en el apartado 2.1, nuestra estación envía datos cada 8s, por lo que hemos configurado la misma frecuencia a esta solicitud GET. Este valor se ha configurado en el configmap y se define como “scrape interval”.

En la figura 3.9 podemos observar las 2 interacciones que lleva a cabo el *pod*:

1. Realiza una consulta de las medidas, a través de una petición GET, en el Pushgateway.
2. Recibe una consulta de las medidas, por parte de Grafana, a través de una petición promQL.

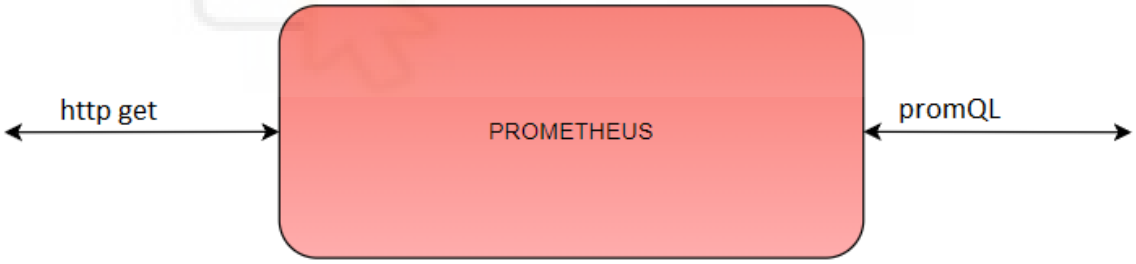

Figura 3.9: Pod Prometheus

3.3.6. Grafana

Es una plataforma de análisis gráfico de métricas. A través de una consulta promQL, obtiene las medidas almacenadas en Prometheus. Esta consulta la configuramos cada 8s, periodicidad que hemos configurado en el resto de componentes. Una vez realizada la consulta, tenemos una gran cantidad de maneras en las que podemos presentar el dato, este es uno de los motivos por lo que hemos seleccionado esta plataforma.

En la figura 3.10 podemos observar las 2 interacciones que lleva a cabo el *pod*:

1. Realiza una consulta contra la base de datos, alojada en el *pod* Prometheus, a través de una petición promQL.
2. Recibe una petición GET, a través de http, de un usuario que desea acceder a un gráfico.
3. Envía los datos a renderizar al pod Grafana-image-renderer y los recibe en forma de tabla.

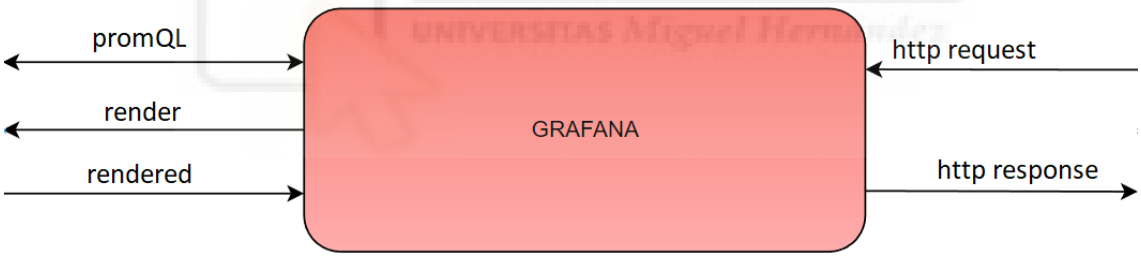

Figura 3.10: Pod Grafana

3.3.7. Grafana-Image-Renderer

Este contenedor es un plug-in de Grafana que actúa como motor de renderizado de gráficas y paneles de monitorización.

- Recibe la solicitud de renderizado de Grafana, la ejecuta y le devuelve la gráfica en formato PNG

Figura 3.11: Pod Grafana-Image-Renderer

3.3.8. Nginx

Es el contenedor que recibe las distintas peticiones https de los clientes, solicitando una gráfica específica. Se encarga de enrutar cada petición hacia la gráfica adecuada en Grafana. También permite modificar la URL entrante asociada a una gráfica.

1. Recibe la petición de una gráfica a través de una solicitud http request en el puerto 80 y la reenvía a la IP del *pod* de Grafana en el puerto 30300.

Figura 3.12: Pod Nginx

3.3.9. Diagrama de Flujo

A continuación, mostramos el flujo interno de mensajes en nuestro servidor K8s que se lleva a cabo para mostrar una medida a un usuario final, desde que es enviada por la estación de medición hasta que se muestra la gráfica al usuario final.

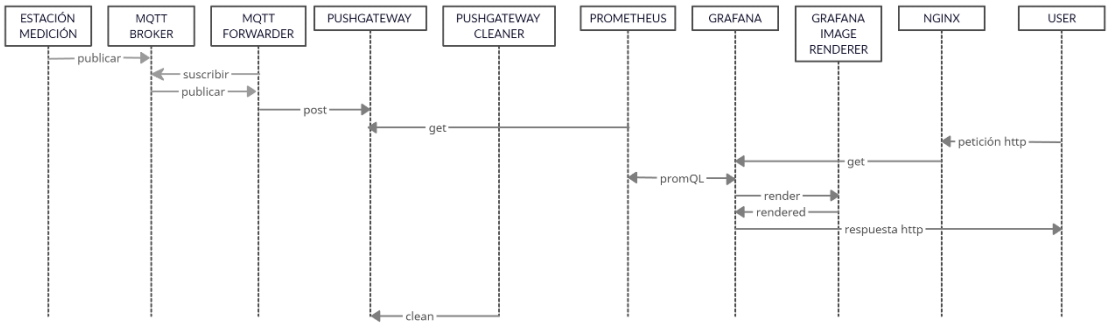

Figura 3.13: Diagrama de flujo de medida en el servidor de K8s

3.4. Puesta en marcha y uso

A continuación, vamos a mostrar la instalación, puesta en marcha del servidor y de nuestra estación de monitorización, así como las funcionalidades que hemos empleado a la hora de configurar este proyecto.

En primer lugar, debemos instalar microk8s, activar Helm e instalar nuestro proyecto previamente descargado.

```
sudo snap install microk8s –classic

sudo microk8s.enable dns

sudo microk8s.enable helm3

sudo microk8s.helm3 install proyectotfg proyecto/stack/tfg
```

Una vez instalado mediante el siguiente comando, arrancamos nuestro servidor:

```
microk8s start
```

En primer lugar, tras realizar numerosos arranques de prueba podemos observar como el tiempo medio de puesta en marcha de nuestro servidor es de 1m 5s

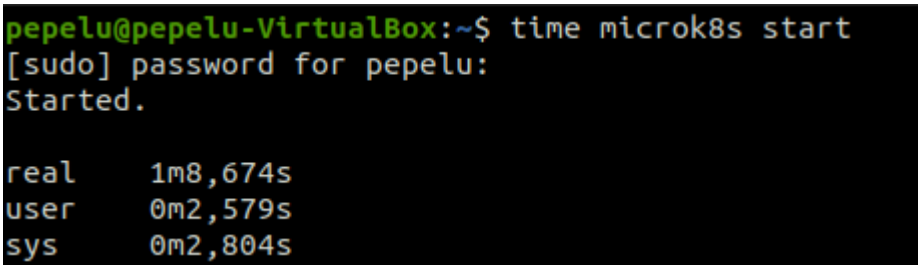

Figura 3.14: Tiempo de arranque del servidor K8s

En segundo lugar, tras realización de pruebas comprobamos que el servidor emplea una media de 1m 8s en detener el funcionamiento.

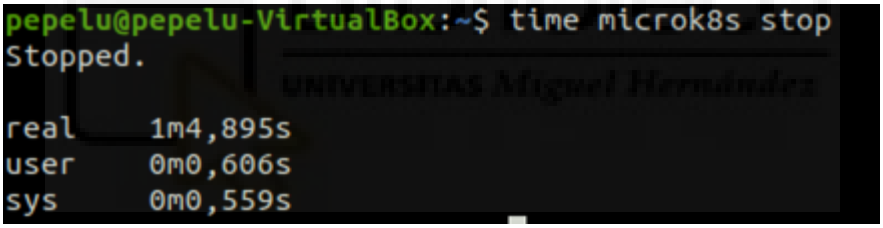

Figura 3.15: Tiempo de apagado del servidor K8s

Una vez arrancado, mediante el uso del siguiente comando, podemos observar el estado de los componentes internos de nuestro servidor:

Pods

```
pepelu@pepelu-VirtualBox:~$ microk8s kubectl get pods
NAME                                  READY   STATUS      RESTARTS   AGE
pushgateway-cleaner-1617123000-sf6d2  0/1     Completed   0          515d
pushgateway-cleaner-1617123300-vhgjc  0/1     Completed   0          515d
pushgateway-cleaner-1617123600-w6rg7  0/1     Completed   0          515d
pushgateway-67fd44fb5-k8dx9           1/1     Running     26         479d
prometheus-65844ff97c-z9s5t           1/1     Running     40         515d
mqttbroker-6f68c9645c-btncp           1/1     Running     44         515d
mqttforward-bd654bd7f-6cnrl           1/1     Running     42         515d
grafana-image-renderer-7fc78b9794-ztfxj   1/1     Running     4          31d
nginx-6ccc574c75-bq8jm                1/1     Running     4          32d
apiserver-7b85d5b545-44z7c            1/1     Running     4          32d
grafana-57bccbfdbf-bgqxt              1/1     Running     10         32d
```

Figura 3.16: Estado de los pods

Replicasets

```
pepelu@pepelu-VirtualBox:~$ microk8s kubectl get replicasets
NAME                                DESIRED   CURRENT   READY   AGE
pushgateway-67fd44fb5               1         1         1       515d
prometheus-65844ff97c               1         1         1       515d
mqttbroker-6f68c9645c               1         1         1       515d
mqttforward-bd654bd7f               1         1         1       515d
grafana-image-renderer-7fc78b9794   1         1         1       31d
nginx-6ccc574c75                    1         1         1       32d
apiserver-7b85d5b545                1         1         1       32d
grafana-57bccbfdbf                  1         1         1       515d
```

Figura 3.17: Estado de los replicasets

Deployments

```
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
pushgateway           1/1     1            1           515d
prometheus            1/1     1            1           515d
mqttbroker            1/1     1            1           515d
mqttforward           1/1     1            1           515d
grafana-image-renderer   1/1     1            1           515d
nginx                 1/1     1            1           32d
apiserver             1/1     1            1           32d
grafana               1/1     1            1           515d
```

Figura 3.18: Estado de los deployments

Servicios

```
pepelu@pepelu-VirtualBox:~$ microk8s kubectl get service
NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
kubernetes                  ClusterIP   10.152.183.1     <none>        443/TCP          515d
pushgateway-np              NodePort    10.152.183.207   <none>        9091:30991/TCP   515d
grafana-image-renderer-cip  NodePort    10.152.183.164   <none>        8081:32590/TCP   515d
grafana-np                  NodePort    10.152.183.186   <none>        3000:30300/TCP   515d
prometheus-cip              ClusterIP   10.152.183.31    <none>        9090/TCP         515d
mqtt-np                     NodePort    10.152.183.141   <none>        1883:30183/TCP   515d
```

Figura 3.19: Estado de los servicios

Jobs

```
pepelu@pepelu-VirtualBox:~$ microk8s kubectl get jobs
NAME                              COMPLETIONS   DURATION   AGE
pushgateway-cleaner-1617123000   1/1           66s        515d
pushgateway-cleaner-1617123300   1/1           55s        515d
pushgateway-cleaner-1617123600   1/1           56s        515d
```

Figura 3.20: Estado de los jobs

Cronjob

```
pepelu@pepelu-VirtualBox:~$ microk8s kubectl get cronjob
NAME                SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
pushgateway-cleaner   */5 * * * *   False     0        515d            515d
```

Figura 3.21: Estado del cronjob

Kubernetes también ofrece la posibilidad de activar un panel de control, donde podemos monitorizar, crear y modificar los distintos elementos de la arquitectura de K8s: *pods*, *replicasets*, *deployments*, servicios…

También podemos monitorizar en vivo el consumo de recursos de cada componente, así como la información que procesa cada *pod*.

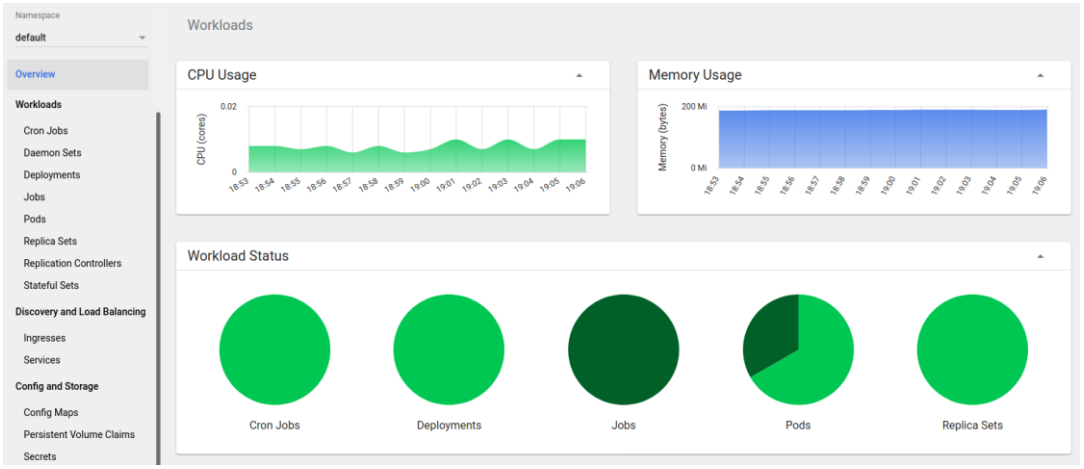

Figura 3.22: Dashboard K8s

Una vez el sistema está funcionando, el siguiente paso es inicializar nuestra estación de captura de medidas. Del fichero de Arduino, únicamente debemos modificar la IP destino, correspondiente con la IP local/pública en la que se encuentre nuestro servidor. En nuestro caso estamos empleando una IP local.

En la siguiente figura podemos observar los logs que obtenemos al iniciar nuestra placa ESP8266.

1.Conexión Wi-Fi.

2.Conexión al *broker* de MQTT.

3.Envío de medidas.

```
21:04:50.934 -> Iniciado proceso de activación de estación de toma de medidas.
21:04:53.032 -> Intentando conectar  con red Wi-Fi MOVISTAR_7FC0...
21:04:54.030 -> 1 2 3
21:04:56.036 ->
21:04:56.036 -> Conexión Wi-Fi establecida
21:04:56.036 -> IP address:     192.168.1.51
21:04:56.036 ->
21:04:56.036 ->
21:04:56.036 -> XxX Dispositivo Activado XxX
21:04:56.036 -> Intentando conectar al broker de MQTT 192.168.1.53:30183...
21:04:56.036 -> Conexión establecida con el broker MQTT!!
21:05:04.045 -> DHT11 Humedad: 59.00 %
21:05:04.078 -> DHT11 Temperatura: 31.40°C
21:05:04.078 -> Enviando mensaje MQTT  hacia el tópico: measurement
21:05:04.078 -> {id: d21584, control: 1, humedad: 59.000000, temperatura: 31.400000}
21:05:12.058 -> DHT11 Humedad: 59.00 %
21:05:12.058 -> DHT11 Temperatura: 31.30°C
21:05:12.058 -> Enviando mensaje MQTT  hacia el tópico: measurement
21:05:12.058 -> {id: d21584, control: 1, humedad: 59.000000, temperatura: 31.299999}
```

Figura 3.23: Debug Arduino al iniciar estación de toma de medidas

Una vez nuestro sensor se ha conectado correctamente al *broker* y comienza a publicar las medidas, estos datos comienzan a almacenarse en nuestra base de datos siguiendo el diagrama de fujo de la figura 3.13

El siguiente paso es acceder a la IP proporcionada por el servicio vinculado a Grafana y configurar los distintos paneles que deseamos visualizar. El usuario y contraseña por defecto son *admin* y admin.

Para ello debemos acceder al panel izquierdo y seleccionar: *Create* -> *Dashboard*.

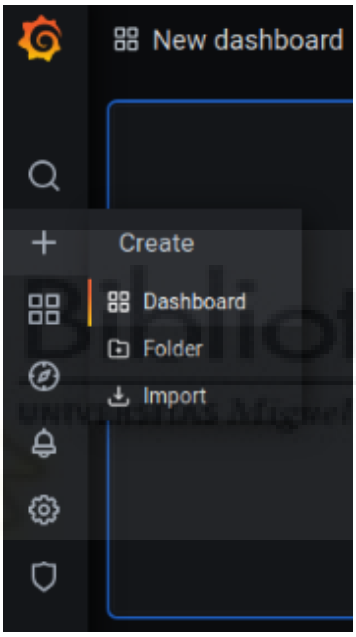

Figura 3.24: Creación de Dashboard

A continuación, presionamos sobre "*Add new panel*" y accedemos a la configuración del panel, donde los campos más importantes a tener en cuenta son:

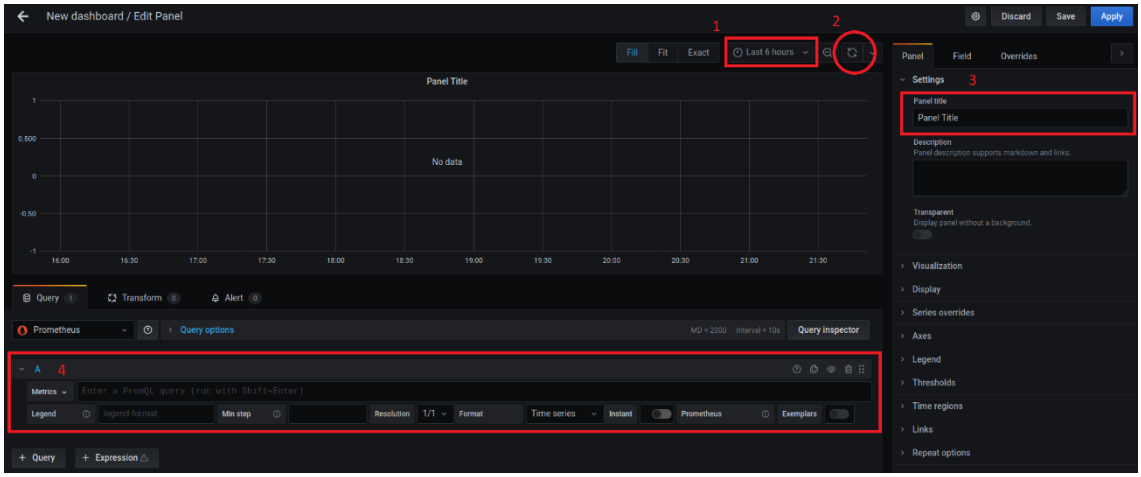

Figura 3.25: Configuración de panel.

1. El rango de tiempo que muestra la gráfica.

2. La tasa de actualización de los datos del panel.

3. Título del panel

4. La solicitud de datos que se va a enviar a Prometheus.

Las consultas a Prometheus las realizamos indicando el id del dispositivo y la variable deseada:

```
Temperature{exported_job="d21584"}
```

```
Humidity{exported_job="d21584"}
```

En las siguientes imágenes observamos los paneles de Grafana que hemos empleado a la hora de representar la temperatura y humedad

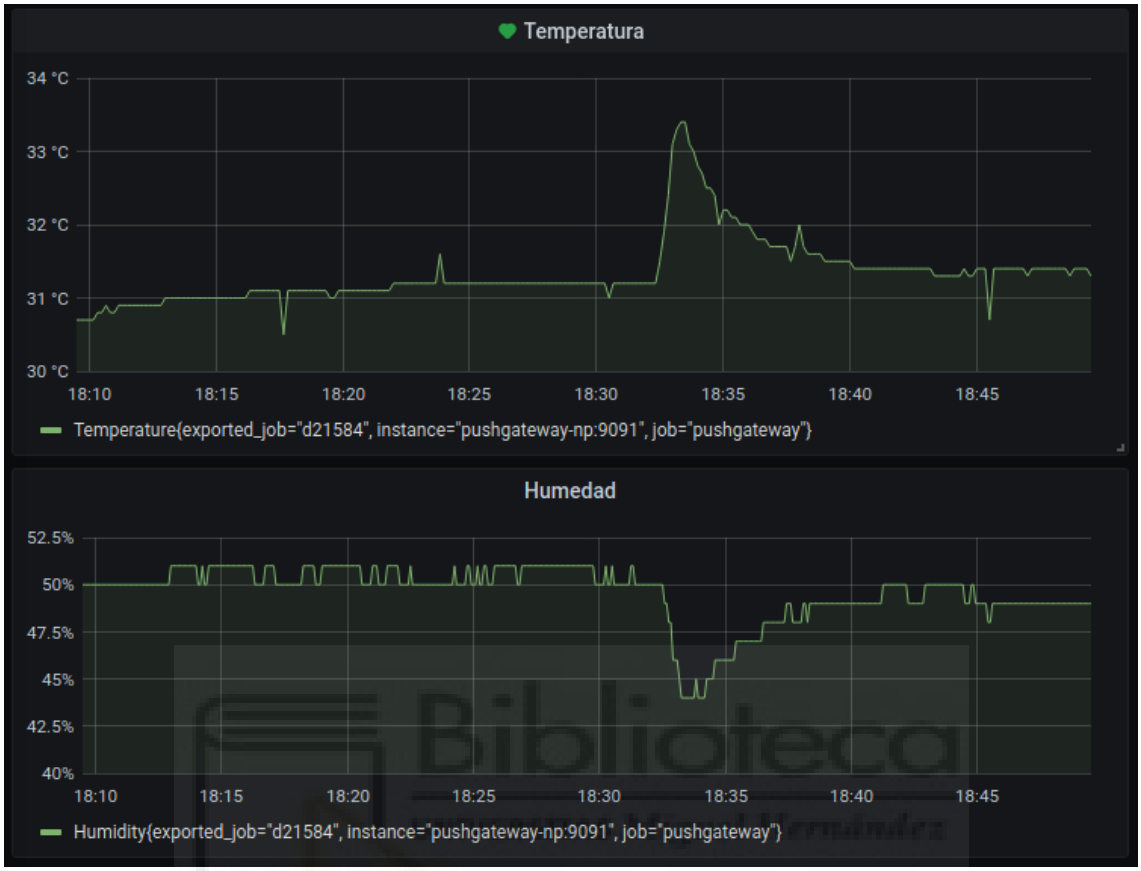

Figura 3.26: Gráfica Grafana 1

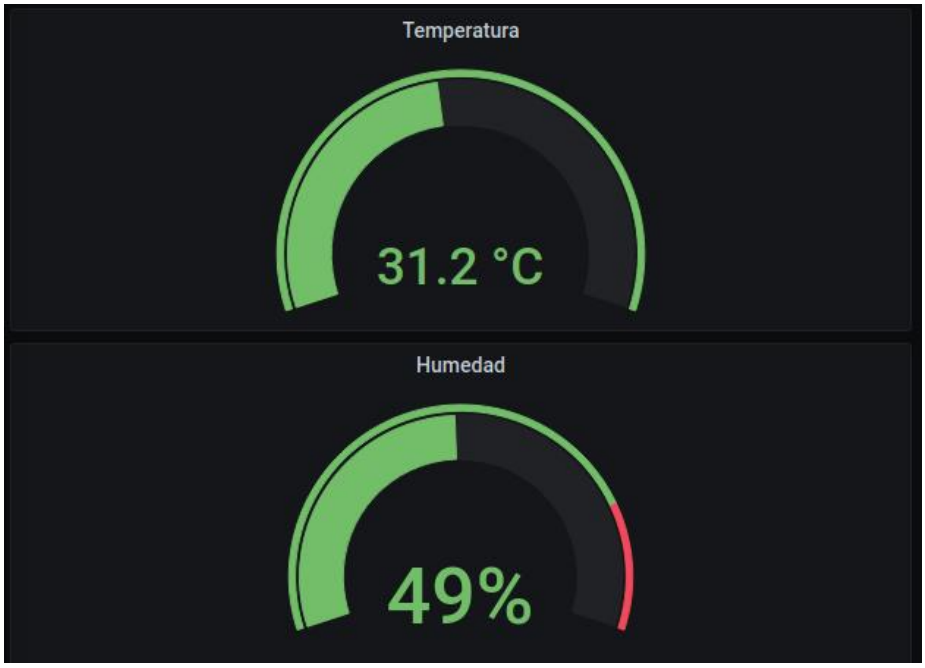

Figura 3.27: Gráfica Grafana 2

3.5. Pruebas realizadas

Finalmente vamos a mostrar el resultado de las pruebas realizadas.

Por un lado, nos hemos asegurado de que el escalado automático de *pod*s horizontal funciona correctamente, de manera que nuestro servidor de K8s siempre dispone de los recursos necesarios para responder ante un aumento de peticiones.

Por otro lado, comprobamos que nuestra estación de toma de medidas, en caso de sufrir una desconexión, es capaz de reconectarse tanto al *broker* de MQTT como a la red Wi-Fi.

3.5.1. Pruebas de escalado

A la hora de configurar el escalado automático de nuestro proyecto, tal y como observamos en la tabla 2.1, disponemos de 4 opciones.

Debido a que únicamente empleamos un nodo al que hemos asociado todos nuestros recursos computacionales y no disponemos de otras máquinas, descartamos el escalado de nodos.

Por otro lado, en cuanto al escalado de *pod*s, hemos aplicado el escalado horizontal automático o HPA, ya que nuestro objetivo es asegurar un correcto funcionamiento del servicio ante un aumento en las peticiones solicitadas por parte del usuario final.

Hemos aplicado el escalado al *pod* grafana-image-renderer, ya que podemos aumentar la carga de manera sencilla aumentando el número de peticiones de distintas gráficas.

A la hora de configurar el escalado, mediante el siguiente comando, indicamos el porcentaje de carga del CPU asociado a esta *pod* y los valores mínimos y máximos de replicasets:

```
microk8s kubectl autoscale deployment/grafana-image-renderer --cpu-percent=80 --min=1 --max=5
```

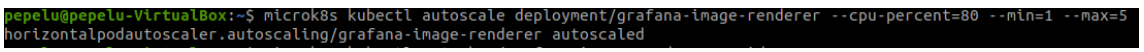

Figura 3.28: Configuración de escalado horizontal

A continuación, mediante el siguiente comando podemos monitorizar el estado del escalado:

```
microk8s.kubectl get  hpa/grafana-image-renderer -owide
```

```
2 pelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   200%/80%   1         5         3          5m21s
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   213%/80%   1         5         3          5m25s
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   146%/80%   1         5         5          7m20s
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   70%/80%   1         5         5          7m24s
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   70%/80%   1         5         5          7m25s
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   70%/80%   1         5         5          7m26s
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   20%/80%   1         5         5          8m30s
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   20%/80%   1         5         5          8m33s
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   20%/80%   1         5         3          8m35s
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   20%/80%   1         5         2          8m37s
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   20%/80%   1         5         2          8m39s
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   20%/80%   1         5         2          10m
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   20%/80%   1         5         1          10m
pepelu@pepelu-VirtualBox:~$ microk8s.kubectl get  hpa/grafana-image-renderer -owide
NAME                     REFERENCE                           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
grafana-image-renderer   Deployment/grafana-image-renderer   20%/80%   1         5         1          10m
```

Figura 3.29: Estado del escalado horizontal

En la figura 3.29, observamos como aumenta el número de réplicas una vez el umbral del uso de CPU ha sido sobrepasado.

Cuando la carga disminuye, el HPA espera una cierta cantidad de tiempo antes de reducir el escalado de la aplicación. Esto se conoce como *cooldown-delay* y ayuda a evitar que el número de réplicas se amplíe y reduzca con demasiada frecuencia. El resultado de esto es que, durante un tiempo determinado, la aplicación se ejecuta con el recuento alto de réplicas, aunque el valor de la métrica está muy por debajo del objetivo. Puede parecer que el HPA no responde a la disminución de la carga, pero eventualmente lo hará.

3.5.2. Pruebas Arduino

Una vez iniciada nuestra estación de toma de medidas, es posible que debido a distintos errores, se produzca una desconexión con el *broker* MQTT o una pérdida de conexión con la señal Wi-fi. Por ello, hemos configurado funciones de reconexión para ambos casos.

En las imágenes que se muestran a continuación, podemos observar los distintos logs obtenidos al poner a prueba ambas funciones:

```
21:04:50.934 -> Iniciado proceso de activación de estación de toma de medidas.
21:04:53.032 -> Intentando conectar  con red Wi-Fi MOVISTAR_7FC0...
21:04:54.030 -> 1 2 3
21:04:56.036 ->
21:04:56.036 -> Conexión Wi-Fi establecida
21:04:56.036 -> IP address:     192.168.1.51
21:04:56.036 ->
21:04:56.036 ->
21:04:56.036 -> XxX Dispositivo Activado XxX
21:04:56.036 -> Intentando conectar al broker de MQTT 192.168.1.53:30183...
21:04:56.036 -> Conexión establecida con el broker MQTT!!
21:05:04.045 -> DHT11 Humedad: 59.00 %
21:05:04.078 -> DHT11 Temperatura: 31.40°C
21:05:04.078 -> Enviando mensaje MQTT  hacia el tópico: measurement
21:05:04.078 -> {id: d21584, control: 1, humedad: 59.000000, temperatura: 31.400000}
21:05:12.058 -> DHT11 Humedad: 59.00 %
21:05:12.058 -> DHT11 Temperatura: 31.30°C
21:05:12.058 -> Enviando mensaje MQTT  hacia el tópico: measurement
21:05:12.058 -> {id: d21584, control: 1, humedad: 59.000000, temperatura: 31.299999}
```

Figura 3.30: Iniciación de estación de toma de medidas

```
21:06:32.074 -> DHT11 Humedad: 59.00 %
21:06:32.074 -> DHT11 Temperatura: 31.40°C
21:06:32.074 -> Enviando mensaje MQTT  hacia el tópico: measurement
21:06:32.074 -> {id: d21584, control: 1, humedad: 59.000000, temperatura: 31.400000}
21:06:32.074 -> Intentando conectar al broker de MQTT 192.168.1.53:30183...
21:06:32.074 -> Error en la conexión con el broker MQTT, intentando de nuevo en 10 segundos...
21:06:42.085 -> Intentando conectar al broker de MQTT 192.168.1.53:30183...
21:06:42.085 -> Error en la conexión con el broker MQTT, intentando de nuevo en 10 segundos...
21:06:52.080 -> Intentando conectar al broker de MQTT 192.168.1.53:30183...
21:06:52.113 -> Error en la conexión con el broker MQTT, intentando de nuevo en 10 segundos...
21:07:02.124 -> Intentando conectar al broker de MQTT 192.168.1.53:30183...
21:07:02.124 -> Error en la conexión con el broker MQTT, intentando de nuevo en 10 segundos...
21:07:12.150 -> Intentando conectar al broker de MQTT 192.168.1.53:30183...
21:07:12.150 -> Error en la conexión con el broker MQTT, intentando de nuevo en 10 segundos...
```

Figura 3.31: Desconexión del broker

```
21:09:34.071 -> Intentando conectar al broker de MQTT 192.168.1.53:30183...
21:09:34.105 -> Error en la conexión con el broker MQTT, intentando de nuevo en 10 segundos...
21:09:44.104 -> Intentando conectar al broker de MQTT 192.168.1.53:30183...
21:09:44.104 -> Conexión establecida con el broker MQTT!!
21:09:44.138 -> DHT11 Humedad: 59.00 %
21:09:44.138 -> DHT11 Temperatura: 31.40°C
21:09:44.138 -> Enviando mensaje MQTT  hacia el tópico: measurement
21:09:44.138 -> {id: d21584, control: 1, humedad: 59.000000, temperatura: 31.400000}
21:09:52.121 -> DHT11 Humedad: 59.00 %
21:09:52.121 -> DHT11 Temperatura: 31.30°C
21:09:52.121 -> Enviando mensaje MQTT  hacia el tópico: measurement
21:09:52.121 -> {id: d21584, control: 1, humedad: 59.000000, temperatura: 31.299999}
21:10:00.141 -> DHT11 Humedad: 59.00 %
21:10:00.141 -> DHT11 Temperatura: 31.40°C
21:10:00.141 -> Enviando mensaje MQTT  hacia el tópico: measurement
21:10:00.141 -> {id: d21584, control: 1, humedad: 59.000000, temperatura: 31.400000}
```

Figura 3.32: Reconexión del broker

```
21:11:04.127 -> DHT11 Humedad: 59.00 %
21:11:04.127 -> DHT11 Temperatura: 31.40°C
21:11:04.127 -> Enviando mensaje MQTT  hacia el tópico: measurement
21:11:04.127 -> {id: d21584, control: 1, humedad: 59.000000, temperatura: 31.400000}
21:11:08.954 -> Intentando conectar al broker de MQTT 192.168.1.53:30183...
21:11:08.954 -> Error en la conexión con el broker MQTT, intentando de nuevo en 10 segundos...
21:11:18.936 -> Se ha desconectado de la red WiFi.
21:11:20.020 -> Intentando reconectar con la red Wi-Fi MOVISTAR_7FC0...
21:11:25.028 -> .
21:11:30.039 -> .
21:11:35.041 -> .
21:11:40.040 -> .
21:11:45.047 -> .
21:11:50.041 -> .
21:11:55.047 -> .
21:12:00.040 -> .
21:12:05.038 -> .
21:12:10.035 -> .
21:12:15.046 -> .
21:12:20.034 -> .
21:12:25.020 -> .
21:12:30.052 -> .
21:12:35.048 -> .
21:12:40.047 -> .
21:12:45.021 -> .
21:12:50.026 -> .
21:12:55.032 -> .
21:13:00.029 -> .
21:13:05.025 -> .
21:13:10.050 -> .
21:13:10.050 ->
21:13:10.050 ->
21:13:10.050 -> Conexión Wi-FI establecida
21:13:10.050 -> IP address:     192.168.0.102
```

Figura 3.33: Desconexión y reconexión a la red Wi-Fi

4. CONCLUSIONES

Mediante este proyecto hemos conseguido que un usuario que se encuentre en cualquier parte del mundo sea capaz de monitorizar, en tiempo real a través de Internet, la temperatura y humedad de un lugar concreto.

Mediante el uso de una estación de medición y servidor propios, hemos conseguido los siguientes objetivos:

- Posibilidad de añadir sensores de distintas variables: Nuestra estación de medición admite más conexiones para diferentes sensores. Una vez conectados simplemente debemos añadir el parámetro a nuestro código replicando lo establecido para la temperatura y humedad

- Uso de alertas: Grafana permite configurar alertas en el panel de visualización de medidas, así como el envío de correos al e-mail.

- Mantenimiento sencillo del servidor y de los dispositivos de medición: Por un lado, gracias a la sencillez de nuestra estación, la robustez de sus componentes y la cantidad de información que encontramos en Internet sobre estos, detectar y corregir un error resulta sencillo. Por otro lado, gracias al uso de microservicios que aíslan los errores en el propio contenedor afectado, y a la automatización a la hora de administrar nuestro servidor, el mantenimiento es muy simple.

- Coste reducido de los dispositivos y servidor: Los componentes de nuestra estación de medición se pueden adquirir por menos de 5 euros y nuestro servidor se puede desplegar en un ordenador personal.

Finalmente, cabe recordar que este proyecto se ha basado en la configuración y aprendizaje de dos tecnologías cuyo uso en los últimos años ha crecido de manera exponencial, gracias a las características que aportan en el mundo de las telecomunicaciones y la gestión de servidores.

A continuación, vamos a mostrar como gracias a las características y funcionalidades del protocolo de mensajería MQTT y del orquestador de contenedores Kubernetes, de manera conjunta, consiguen alcanzar los objetivos que hemos comentado al inicio del proyecto.

Escalabilidad: Gracias a los distintos modos de escalados automáticos que ofrece K8s, en todo momento vamos a disponer de la potencia computacional necesaria para responder a un aumento en el número de peticiones. Por otro lado, añadir estaciones de monitorización no supone ningún tipo de problema, ya que únicamente debemos realizar el montaje y copiar el código de Arduino. Nuestro *broker* de MQTT permite una gran cantidad de conexiones y en caso de sobrepasar el límite, nuestro escalado creará otro *pod* con esta funcionalidad.

Consumo de recursos: Gracias al escalado automático, únicamente se encontrarán activos el número de *pod*s necesarios para resolver las peticiones por parte del usuario, aportando una gran eficiencia a nuestro proyecto.Por otro lado, nuestra estación de monitorización requiere de muy poca energía para realizar su función y el protocolo MQTT supone un uso muy reducido en el ancho de banda de la red.

Puesta en producción: Mediante el uso de Helm, podemos desplegar y actualizar nuestro proyecto de manera sencilla con un solo comando. Por otro lado, Kubernetes es compatible con los principales S.O (Windows, Linux, MacOs…)

Tiempo de disponibilidad: A través de la funcionalidad *Rolling Update* de los *deployment*, podemos realizar la actualización de un contenedor al mismo tiempo que el contenedor antiguo se está ejecutando, por lo que no es necesaria una pausa del servicio.

4.1. Trabajos futuros

Es proyecto tiene una gran cantidad de funciones que podemos ampliar:

- Incluir cualquier otro tipo de sensor: CO_2, 0_2, luz, sonido….
- Incluir alertas en Grafana.
- Desplegar proyecto en un cluster en la nube.

Por otro lado, podemos aplicar esta configuración de Kubernetes a otro tipo de proyectos:

- Envió de datos desde básculas alojadas en centros sanitarios, con tal de llevar un control de los pacientes.

5. Anexos

Anexo A

Acrónimos

Siglas	Significado
CPU	*Central Processing Unit*
HPA	*Horizontal Pod Autoscaler*
I2C	*Inter-Integrated Circuit*
IoT	*Internet of Things*
IP	*Internet Protocol*
K8s	Kubernetes
M2M	*Machine-to Machine*
MQTT	*Message Queuing Telemetry Transport*
promQL	*Promtheus Query Language*
S.O	Sistema Operativo
TCP	*Transmission Control Protocol*
VM	*Virtual Machine*
YAML	*YAML Ain't Markup Language*

Anexo B

Código Arduino

```
#include <ESP8266WiFi.h>                                    // Wi-Fi
ESP8266
#include <ESP8266WebServer.h>
#include <AsyncMqttClient.h>
#include <PubSubClient.h>

// DHT11
#include "DHTesp.h"
#define DHT_GPIO 5                                          // señal
GPIO5 (D1)

String sw_version = "v1.0";


String device_id = String(ESP.getChipId(), HEX);    //
versión hexadecimal de la MAC



char MQTT_server[24] = "192.168.1.53";
// IP de nuestro broker MQTT, que es la misma que la de
nuestro servidor.
uint16_t MQTT_port = 30183;
// Puerto MQTT 30183


const char* ssid     = "MOVISTAR_7FC0";                 //
Nombre SSID de la red Fi-Fi a la que nos vamos a conectar
const char* password = "KrRtiJfV3LukjbcLidNz";          //
Contraseña de la red Wi-Fi

// Loop de medidas: Tiempo entre medidas
```

```
unsigned int measurements_loop_duration = 8000;    // 8
segundos
unsigned long measurements_loop_start;             // inicio
de tiempo para loop de toma medidas

// Loop MQTT: tiempo entre el envío de medidas al broker
MQTT
unsigned int MQTT_loop_duration = 8000;            // 8
seconds
unsigned long MQTT_loop_start;                     // inicio
de tiempo para loop de envío medidas
unsigned long lastReconnectAttempt = 0;            // MQTT
reconnections

//  Loop de errores: tiempo entre recuperación de errores
unsigned int errors_loop_duration = 3000;          // 3
segundos
unsigned long errors_loop_start;                   // inicio
de tiempo para loop de errores

float myTime = 0;

WiFiClient wifi_client;


bool initialConfig = false;

// Saber si existe alguna actualización en curso
boolean updating = false;

// MQTT
char MQTT_message[256];
String MQTT_send_topic = "measurement";
PubSubClient MQTT_client(wifi_client);

// Estados de control del dispositivo

boolean err_MQTT = false;
boolean err_dht = false;


// Inicializar sensor DGT11
DHTesp dht;
float temperature = 1;      //
```

```
float humidity = 1;          //
int control = 1;




void setup() {

 //  Inicialización del puerto serial del monitor en
Arduino
  Serial.begin(115200);


  // Enable debug
  Serial.setDebugOutput(true);

  Serial.begin(115200);          // Comenzar la comunicación
Serial con el pc
  delay(10000);
  Serial.println('\n');
  Serial.println("Iniciado proceso de activación de
estación de toma de medidas. ");
  delay(2000);

  WiFi.begin(ssid, password);              // Conectar a la
red Wi-Fi
  Serial.print("Intentando conectar  con red Wi-Fi ");
  Serial.print(ssid); Serial.println("...");

  int i = 0;
  while (WiFi.status() != WL_CONNECTED) { // Esperando
conexión Wi-Fi
    delay(1000);
    Serial.print(++i); Serial.print(' ');
  }

  Serial.println('\n');
  Serial.println("Conexión Wi-Fi establecida");
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());          // Mostrar IP de
la placa ESP8266
```

```
  // Indicar activación del dispositivo
  Serial.println();
  Serial.println();
  Serial.println("XxX Dispositivo Activado XxX");

    // Conectar al broker MQTT

    Init_MQTT();

    // Inicializar sensores
    Setup_sensors();

    // Init loops
    measurements_loop_start = millis();
    MQTT_loop_start = millis();
    errors_loop_start = millis();

}

// CONTROL LOOP


void loop() {

  //
  if (MQTT_client.connected()) {

    // Medida del loop
    if ((millis() - measurements_loop_start) >=
measurements_loop_duration)
    {

      // Tiempo para comenzar el loop
      measurements_loop_start = millis();


      // Leer sensors
     Read_Sensors();
     myTime = millis();

    }
```

```
   // MQTT loop
    if ((millis() - MQTT_loop_start) >= MQTT_loop_duration)
    {


      MQTT_loop_start = millis();

      Send_Message_Cloud_App_MQTT();


    }

  }else{
      Init_MQTT();
      if (WiFi.status() != WL_CONNECTED){
      Serial.println("Se ha desconectado de la red WiFi.");
      delay(1000);
      WiFi.begin(ssid, password);             // Conectar a
la red Wi-Fi
      Serial.print("Intentando reconectar con la red Wi-Fi
");
      Serial.print(ssid); Serial.println("...");

      while (WiFi.status() != WL_CONNECTED) { // Esperando
conexión Wi-Fi
        delay(5000);
        Serial.println('.');
      }

      Serial.println('\n');
      Serial.println("Conexión Wi-FI establecida");
      Serial.print("IP address:\t");
      Serial.println(WiFi.localIP());
        }
      }
}
```

```
// FUNCIONES


void Init_MQTT() {
  Serial.print("Intentando conectar al broker de MQTT ");
  Serial.print(MQTT_server);
  Serial.print(":");
  Serial.print(MQTT_port);
  Serial.println("...");

  // Intento de conexión al broker
  MQTT_client.setBufferSize(512); //
  MQTT_client.setServer(MQTT_server, MQTT_port);
  MQTT_client.connect(device_id.c_str());

  if (!MQTT_client.connected()) {
    err_MQTT = true;
    MQTTReconnect();
  }
  else {
    err_MQTT = false;
    lastReconnectAttempt = 0;
    Serial.println("Conexión establecida con el broker
MQTT!!");
  }

}


void Setup_sensors() {

    // Leer DHT11
    dht.setup(DHT_GPIO, DHTesp::DHT22);

}

// Leer sensores
void Read_Sensors() {

    Read_DHT11();

}

void Read_DHT11() {
```

```
  TempAndHumidity lastValues = dht.getTempAndHumidity();

  // Leer humedad como porcentaje
  humidity = lastValues.humidity;

  // Leer temperatura en Celsius
  temperature = lastValues.temperature;

  // Comprobar si alguna lewctura ha fallado
  if (isnan(humidity) || isnan(temperature)) {
    Serial.println("Error leyendo sensor DHT!");
    err_dht = true;
    humidity = 0;
    temperature = 0;
  }
  else {
    err_dht = false;
    Serial.print("DHT11 Humedad: ");
    Serial.print(humidity);
    Serial.print(" % \n");
    Serial.print("DHT11 Temperatura: ");
    Serial.print(temperature);
    Serial.println("°C");
  }

}

// Enviar medidas al broque MQTT
void Send_Message_Cloud_App_MQTT() {

  // Print info
  Serial.print("Enviando mensaje MQTT  hacia el tópico: ");
  Serial.println(MQTT_send_topic);
  sprintf(MQTT_message, "{id: %s,control: %i,humidity:
%f,temperature: %f}", device_id.c_str(), control, humidity,
temperature);
  Serial.print(MQTT_message);
  Serial.println();

  //   enviar el mensaje
  MQTT_client.publish(MQTT_send_topic.c_str(),
MQTT_message);
```

```
}



// Función de Reconexión MQTT
void MQTTReconnect() {
  // Intentar conectara solo si han pasado mas de 10
segundos desde el último intento
  unsigned long now = millis();
  if (now - lastReconnectAttempt > 10000) {
    lastReconnectAttempt = now;

    // Intentar conectar
    if (MQTT_client.connect(device_id.c_str())) {
      err_MQTT = false;
      Serial.println("MQTT conectado");
      lastReconnectAttempt = 0;
    } else {
      err_MQTT = true;
      Serial.println("Error en la conexión con el broker
MQTT, intentando de nuevo en 10 segundos...");
      delay(10000);
    }
  }
}
```

Anexo C

Documentos YAML de configuración y despliegue del Servidor de Kubernetes

Values

```yaml
#IPV4 de nuestra VM
publicIP: 192.168.1.53

#IPV4 de VM de backup (por el momento no se ha implementado
secondaryPublicIP: 192.168.1.53


#Contraseña del user admin
grafanaAdminPass: admin

#Configuración para el uso de TLS( en este proyecto no se
ha implementado)
tls: false

smtp_enabled: true
smtp_host: smtp.gmail.com:587
smtp_user: microk8s.avisos@gmail.com
smtp_pass: Al3j4ndr14
smtp_from: Grafana


#Recursos computacionales para los pods.
  mqttbroker:
    requests:
      memory: "40Mi"
      cpu: "80m"
    limits:
      memory: "50Mi"
      cpu: "100m"
  mqttforward:
    requests:
```

```
      memory: "80i"
      cpu: "80m"
    limits:
      memory: "100Mi"
      cpu: "100m"
  pushgateway:
    requests:
      memory: "80i"
      cpu: "80m"
    limits:
      memory: "100Mi"
      cpu: "100m"
  prometheus:
    requests:
      memory: "400Mi"
      cpu: "400m"
    limits:
      memory: "500Mi"
      cpu: "500m"
  grafana:
    requests:
      memory: "200Mi"
      cpu: "400m"
    limits:
      memory: "250Mi"
      cpu: "500m"
  grafanaimagerenderer:
    requests:
      memory: "800Mi"
      cpu: "400m"
    limits:
      memory: "1Gi"
      cpu: "500m"
  apiserver:
    requests:
      memory: "80Mi"
      cpu: "80m"
    limits:
      memory: "100Mi"
      cpu: "100m"
  cleanercronjob:
    requests:
      memory: "80Mi"
      cpu: "80m"
```

```
    limits:
      memory: "100Mi"
      cpu: "100m"
  nginx:
    requests:
      memory: "80Mi"
      cpu: "80m"
    limits:
      memory: "100Mi"
      cpu: "100m"
```

Chart

```
apiVersion: v2

name: tfgproyect

description: Helm

type: application

version: 1.0.1

appVersion: 1.0.1
```

Pushgateway-Cleaner

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: pushgateway-cleaner
spec:
  schedule: "*/5 * * * *"
  jobTemplate:
    spec:
      ttlSecondsAfterFinished: 100
      template:
```

```
        spec:
          containers:
          - name: pushgateway-cleaner
            image: python
            command: ["/bin/bash"]
            args: ["-c","/config/run.sh"]
            volumeMounts:
              - name: cleaner
                mountPath: /config/
            resources:
              {{- toYaml .Values.resources.cleanercronjob |
nindent 14 }}
          restartPolicy: OnFailure
          volumes:
          - name: cleaner
            configMap:
              name: cleaner
              defaultMode: 0777
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: cleaner
data:
  run.sh: |
    #!/bin/bash
    pip install requests
    echo 'running: /config/cleaner.py'
    python /config/cleaner.py pushgateway-np:9091

  cleaner.py: |
    import requests
    import json
    import time
    import sys

    max_age = 5 #maximum age in minutes
    PUSHGATEWAT_URL = sys.argv[1]


    def main():
      response = requests.request('GET', 'http://' +
PUSHGATEWAT_URL + '/api/v1/metrics')
      metrics = json.loads(response.text)
```

```
      now = time.localtime()
      for data in metrics['data']:
        co2 = data.get('CO2', False)
        if co2:
          last_time = time.strptime(co2['time_stamp'][:16],
'%Y-%m-%dT%H:%M')
          if (int((time.mktime(now) -
time.mktime(last_time)) / 60) > max_age):
            url = 'http://' + PUSHGATEWAT_URL +
'/metrics/job/' + co2['metrics'][0]['labels']['job']
            print('DELETE ' + url)
            response = requests.request('DELETE', url)


    if __name__ == "__main__":
      main()
```

Grafana


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      labels:
        app: grafana
    spec:
      containers:
        - name: grafana
          image: grafana/grafana:7.4.1
          env:
          - name: GF_AUTH_ANONYMOUS_ENABLED
            value: "true"
          - name: GF_SECURITY_ADMIN_PASSWORD
```

```
            value: "{{ .Values.grafanaAdminPass }}"
          - name: GF_SERVER_ROOT_URL
          {{- if .Values.secondaryPublicIP }}
            value: "http://{{ .Values.secondaryPublicIP }}"
          {{- else }}
            value: "http://192.168.1.53"
          {{- end }}
          - name: GF_RENDERING_SERVER_URL
            value: "http://grafana-image-renderer-
cip:8081/render"
          - name: GF_RENDERING_CALLBACK_URL
          {{- if .Values.secondaryPublicIP }}
            value: "http://{{ .Values.secondaryPublicIP
}}/"
          {{- else }}
            value: "http://192.168.1.53"
          {{- end }}
          - name: GF_LOG_FILTERS
            value: rendering:debug
          - name: GF_INSTALL_PLUGINS
            value: "cloudspout-button-panel"
          {{- if eq .Values.smtp_enabled true }}
          - name: GF_SMTP_ENABLED
            value: "true"
          - name: GF_SMTP_HOST
            value: "smtp.gmail.com:587"
          - name: GF_SMTP_USER
            value: "microk8s.avisos@gmail.com"
          - name: GF_SMTP_PASSWORD
            value: "Al3j4ndr14"
          - name: GF_SMTP_FROM_ADDRESS
            value: "microk8s.avisos@gmail.com"
          - name: GF_SMTP_FROM_NAME
            value: "Grafana"
          {{- end }}

          args:
          volumeMounts:
            - name: grafana-storage-volume
              mountPath: /var/lib/grafana/
            - name: grafana-config
              mountPath:
/etc/grafana/provisioning/datasources/datasource.yaml
              subPath: datasource.yaml
```

```
          resources:
            {{- toYaml .Values.resources.grafana | nindent
12 }}
      volumes:
        - name: grafana-storage-volume
          hostPath:
            path: /data/grafana
            type: Directory
        - name: grafana-config
          configMap:
            name: grafana-config
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: grafana-config
data:
  datasource.yaml: |
    apiVersion: 1
    deleteDatasources:
      - name: Prometheus
        orgId: 1
    datasources:
      - name: Prometheus
        type: prometheus
        access: proxy
        url: http://prometheus-cip:9090
        version: 1
        editable: true
        isDefault: true
---
apiVersion: v1
kind: Service
metadata:
  name: grafana-np
spec:
  type: NodePort
  selector:
    app: grafana
  ports:
  - protocol: TCP
    port: 3000
    nodePort: 30300
  - protocol: TCP
```

```
    port: 3000
    nodePort: 30300
```

Grafana-image-renderer

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana-image-renderer
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grafana-image-renderer
  template:
    metadata:
      labels:
        app: grafana-image-renderer
    spec:
      containers:
        - name: grafana-image-renderer
          image: grafana/grafana-image-renderer:2.0.0-beta1
          env:
          - name: ENABLE_METRICS
            value: 'true'
          resources:
            {{- toYaml
.Values.resources.grafanaimagerenderer | nindent 12 }}
---
apiVersion: v1
kind: Service
metadata:
  name: grafana-image-renderer-cip
spec:
  type: NodePort
  selector:
    app: grafana-image-renderer
  ports:
  - protocol: TCP
```

```
    port: 8081
    targetPort: 8081
```

MQTT - Broker

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mqttbroker
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mqttbroker
  template:
    metadata:
      labels:
        app: mqttbroker
    spec:
      containers:
        - name: mqttbroker
          image: eclipse-mosquitto:1.6.13
          volumeMounts:
            - name: mqtt-storage-volume
              mountPath: /mosquitto/
            - name: mqtt-config
              mountPath: /mosquitto/config/mosquitto.conf
              subPath: mosquitto.conf
          resources:
            {{- toYaml .Values.resources.mqttbroker |
nindent 12 }}
      volumes:
        - name: mqtt-storage-volume
          hostPath:
            path: /data/mosquitto
            type: Directory
        - name: mqtt-config
          configMap:
            name: mqttbroker-config
---
```

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: mqttbroker-config
data:
  datasource.yaml: |
    persistence true
    persistence_location /mosquitto/data/
    log_dest file /mosquitto/log/mosquitto.log
---
apiVersion: v1
kind: Service
metadata:
  name: mqtt-np
spec:
  type: NodePort
  selector:
    app: mqttbroker
  ports:
  - protocol: TCP
    port: 1883
    nodePort: 30183
```


MQTT – Forwarder

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mqttforward
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mqttforward
  template:
    metadata:
      labels:
        app: mqttforward
    spec:
      containers:
        - name: mqttforward
```

```
          image: python
          command: ["/bin/bash"]
          args: ["-c","/config/run.sh"]
          volumeMounts:
            - name: mqttforward
              mountPath: /config/
          resources:
            {{- toYaml .Values.resources.mqttforward |
nindent 12 }}
      volumes:
        - name: mqttforward
          configMap:
            name: mqttforward
            defaultMode: 0777
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: mqttforward
data:
  run.sh: |
    #!/bin/bash
    pip install paho-mqtt
    pip install requests
    pip install pyyaml
    while ((1))
    do
      python /config/pushgateway_forwarder.py pushgateway-
np mqtt-np
      echo "ERROR: Script failed"
    done

  pushgateway_forwarder.py: |
    #!/usr/bin/env python
    import sys
    import requests
    from paho.mqtt import client as mqtt
    import yaml

    HEADERS = {'X-Requested-With': 'Python requests',
'Content-type': 'text/xml'}


    def pushdata(DATA, URL):
```

```
      try:
        #print(DATA)
        response = requests.post(url=URL,
data=DATA,headers=HEADERS)
        print(response.content)
      except requests.exceptions.RequestException as e:  #
This is the correct syntax
        raise SystemExit(e)

    def on_connect(client, userdata, flags, rc):
        # connect mqtt broker
        client.subscribe([("measurement", 0)])

    def on_message(client, userdata, msg):
        payload = msg.payload.decode("utf-8")
        payload = yaml.safe_load(msg.payload.decode("utf-
8"))

        device_id = payload.pop('id', None)
        if device_id:
            msg = 'CO2 ' + str(payload.pop('CO2', 0)) +
'\n' + \
                  'Temperature ' +
str(payload.pop('temperature', 0)) + '\n' + \
                  'Humidity ' + str(payload.pop('humidity',
0)) + '\n'
            URL  = 'http://' + sys.argv[1] +
':9091/metrics/job/' + str(device_id)
            pushdata(msg, URL)
        else:
            print( 'ERROR: Malformed message' +
str(payload))


    def main():
      if (len(sys.argv) != 3):
          print('Usage: '+sys.argv[0]+' <pushgateway IP>
<mqttbroker IP>. Found '+len(sys.argv)+' arguments.')
          exit()

      client = mqtt.Client()
      client.connect(sys.argv[2], 1883)
      client.on_connect = on_connect
      client.on_message = on_message
```

```
        client.loop_forever()

    if __name__ == "__main__":
        main()
```

Nginx

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.19.6-alpine
          {{- if eq .Values.tls true }}
          command: ["sh"]
          args: ["-c","/scripts/run.sh"]
          {{- end }}
          volumeMounts:
            - name: confd
              mountPath: /etc/nginx/conf.d/
            - name: letsencrypt-volume
              mountPath: /etc/letsencrypt
            {{- if eq .Values.tls true }}
            - name: nginx-run
              mountPath: /scripts
            {{- end }}
          resources:
            {{- toYaml .Values.resources.nginx | nindent 12
}}
            requests:
```

```
          ports:
          - containerPort: 80
            hostPort: 80
            protocol: TCP
          - containerPort: 443
            hostPort: 443
            protocol: TCP
      initContainers:
        - name: copyfile
          image: nginx:1.19.6-alpine
          volumeMounts:
            - name: confd
              mountPath: /confd
            - name: nginx-configmap
              mountPath: /configmap
          command:
          - cp
          - /configmap/default.conf
          - /confd/default.conf
      volumes:
        - name: letsencrypt-volume
          hostPath:
            path: /data/letsencrypt
        {{- if eq .Values.tls true }}
        - name: nginx-run
          configMap:
            name: nginx-run
            defaultMode: 0777
        {{- end }}
        - name: nginx-configmap
          configMap:
            name: nginx-configmap
        - name: confd
          emptyDir: {}
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-run
data:
  run.sh: |
    apk add certbot certbot-nginx
    echo "5 0 * * 6 certbot renew >/dev/null 2>&1" >>
/etc/crontabs/root
```

```
    crond
    nginx -g 'daemon off;'
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configmap
data:
  default.conf: |
    upstream grafana {
      server         192.168.1.53:30300;
    }

    server {
        {{- if eq .Values.tls true }}
        listen [::]:443 ssl ipv6only=on; # managed by
Certbot
        listen 443 ssl; # managed by Certbot
        ssl_certificate
/etc/letsencrypt/live/192.168.1.53/fullchain.pem; # managed
by Certbot
        ssl_certificate_key
/etc/letsencrypt/live/192.168.1.53/privkey.pem; # managed
by Certbot
        include /etc/letsencrypt/options-ssl-nginx.conf; #
managed by Certbot
        ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; #
managed by Certbot
        {{- else }}
        listen       80;
        listen  [::]:80;
        {{- end }}

        server_name  192.168.1.53;

        location / {
            proxy_pass http://grafana;
            proxy_redirect     off;
            proxy_set_header   Host $host;
            #root   /usr/share/nginx/html;
            #index  index.html index.htm;
        }

        location ~ ^/(panel|live|kiosk)  {
```

```
            rewrite ^(.*)$ https://$host?kiosk break;
        }

        location ~
^/(sensor|device|dispositivo|medidor|view|unidad|unit) {
            rewrite /(.*)/(.*)/(.*)
https://$host/d/lastvalue?var-uid=$2&var-name=$3&kiosk
break;
            rewrite /(.*)/(.*)
https://$host/d/lastvalue?var-uid=$2&kiosk  break;
        }

        location ~ ^/(admin|detalle|detail|edit|editor) {
            rewrite /(.*)/(.*)/(.*)
https://$host/d/lastvalue?var-uid=$2&var-name=$3  break;
            rewrite /(.*)/(.*)
https://$host/d/lastvalue?var-uid=$2  break;
        }

        #error_page  404              /404.html;

        # redirect server error pages to the static page
/50x.html
        #
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   /usr/share/nginx/html;
        }
    }
    {{- if eq .Values.tls true }}
    server {
        if ($host = 192.168.1.53) {
            return 301 https://$host$request_uri;
        } # managed by Certbot

        {{- if .Values.secondaryPublicIP }}
        if ($host = "192.168.1.53") {
            return 301 https://$host$request_uri;
        } # managed by Certbot
        {{- end }}

        listen [::]:80;
        listen 80;
```

```
        server_name   192.168.1.53;
        return 404; # managed by Certbot
    }
    {{- end }}
```

Prometheus

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-server-conf
  labels:
    name: prometheus-server-conf
data:
  prometheus.yml: |-
    global:
      scrape_interval: 2s
      evaluation_interval: 15s

    scrape_configs:
      - job_name: 'pushgateway'

        static_configs:
          - targets: ['pushgateway-np:9091']
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prometheus
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      containers:
```

```
        - name: prometheus
          image: prom/prometheus
          args:
            - "--
config.file=/etc/prometheus/prometheus.yml"
            - "--storage.tsdb.path=/prometheus/"
          volumeMounts:
            - name: prometheus-config-volume
              mountPath: /etc/prometheus/
            - name: prometheus-storage-volume
              mountPath: /prometheus/
          resources:
            {{- toYaml .Values.resources.prometheus |
nindent 12 }}
      volumes:
        - name: prometheus-config-volume
          configMap:
            defaultMode: 420
            name: prometheus-server-conf
        - name: prometheus-storage-volume
          hostPath:
            path: /data/prometheus
            type: Directory
---
apiVersion: v1
kind: Service
metadata:
  name: prometheus-cip
spec:
  type: ClusterIP
  selector:
    app: prometheus
  ports:
  - protocol: TCP
    port: 9090
    targetPort: 9090
```

Pushgateway

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: pushgateway
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pushgateway
  template:
    metadata:
      labels:
        app: pushgateway
    spec:
      containers:
        - name: pushgateway
          image: prom/pushgateway
          command: ["/bin/pushgateway"]
          args: ["--
persistence.file=/pushgateway/pushwateway.data","--
persistence.interval=10s"]
          volumeMounts:
            - name: pushgateway-storage-volume
              mountPath: /pushgateway/
          resources:
            {{- toYaml .Values.resources.pushgateway |
nindent 12 }}
            requests:
      volumes:
        - name: pushgateway-storage-volume
          hostPath:
            path: /data/pushgateway
            type: Directory
---
apiVersion: v1
kind: Service
metadata:
  name: pushgateway-np
spec:
  type: NodePort
  selector:
    app: pushgateway
  ports:
  - protocol: TCP
    port: 9091
    nodePort: 30991
```

Anexo D

Lista de comandos de Kubernetes empleados:

`microk8s kubectl autoscale deployment/grafana-image-renderer --cpu-percent=60 --min=1 --max=5` – creación de escalado horizontal automático.

`microk8s kubectl delete hpa grafana-image-renderer` – eliminación de escalado horizontal automático.

`microk8s kubectl delete rs (nombre replicaset)` – eliminación de *replicaset*.

`microk8s kubectl describe deployments grafana` – descripción de un *deployment*.

`microk8s kubectl describe node pepelu-virtualbox` – descripción de un nodo.

`microk8s kubectl describe pod grafana` – descripción de un *pod*.

`microk8s kubectl get hpa/grafana-image-renderer -owide` – descripción de un escalado horizontal automático.

`microk8s kubectl get all --all-namespaces` – descripción de todos los objetos de nuestro cluster.

`microk8s kubectl get deployments` – listado de *deployments*.

`microk8s kubectl get pods` – listado de *pods*.

`microk8s kubectl get replicasets` – listado de *replicasets*.

`microk8s kubectl scale deployment/apiserver --replicas=1` – configurar número de réplicas de un *deployment*.

`microk8s kubectl top node` – informe consumo de recursos de los nodos.

`microk8s kubectl top pod` – informe de consumo de recursos de los *pods*.

`microk8s start` – arrancar servidor de Kubernetes.

`microk8s stop` – apagar servidor de Kubernetes.

6. Bibliografía

AnaireOrg - Asociación sin ánimo de lucro para el desarrollo de tecnologías abiertas que beneficien a la sociedad.

Aplyca. (s.f.). Obtenido de https://www.aplyca.com/blog/grafana-y-prometheus-para-monitoreo-de-contenedores

Aprender Big Data - Introducción a MQTT y Mosquitto. (s.f.). Obtenido de https://aprenderbigdata.com/mqtt-mosquitto/

Clouding.io. (s.f.). Obtenido de https://help.clouding.io/hc/es/articles/360010704900-Introducci%C3%B3n-a-Eclipse-Mosquitto

Docker - Use containers to Build, Share and Run your applications. (s.f.). Obtenido de https://www.docker.com/resources/what-container/

GitHub - Prometheus Pushgateway. (s.f.). Obtenido de https://github.com/prometheus/pushgateway

Grafana. (s.f.). Obtenido de https://grafana.com/

Helm. (s.f.). Obtenido de https://v2.helm.sh/docs/developing_charts/

InfoWorld - 4 reasons you should use Kubernetes. (s.f.). Obtenido de https://www.infoworld.com/article/3173266/4-reasons-you-should-use-kubernetes.html

Itnext. (s.f.). Obtenido de https://itnext.io/k8s-vertical-pod-autoscaling-fd9e602cbf81

Kubernetes - Production-Grade Container Orchestration. (s.f.). Obtenido de https://kubernetes.io/

Matthewpalmer. (s.f.). Obtenido de https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-deployment-tutorial-example-yaml.html

Microk8s. (s.f.). Obtenido de https://microk8s.io/

NetApp - Containers vs. Virtual Machines (VMs). (s.f.). Obtenido de https://www.netapp.com/blog/containers-vs-vms/#:~:text=Virtual%20machines%20and%20containers%20differ,to%20run%20multiple%20OS%20instances.

Prometheus. (s.f.). Obtenido de https://prometheus.io/docs/prometheus/latest/getting_started/

Randomnerdtutorials. (s.f.). Obtenido de https://randomnerdtutorials.com/esp8266-nodemcu-mqtt-publish-dht11-dht22-arduino/

Red Hat - Containers vs VMs. (s.f.). Obtenido de https://www.redhat.com/en/topics/containers/containers-vs-vms

Serrrocal.medium - Un vistazo a Kubernetes. (s.f.). Obtenido de https://serrodcal.medium.com/un-vistazo-a-kubernetes-8f448512ffbb

Serverfault. (s.f.). Obtenido de https://serverfault.com/questions/1045712/how-to-setup-mosquitto-mqtt-broker-in-kubernetes

YT - TechWorld with Nana. (s.f.). Obtenido de https://www.youtube.com/c/TechWorldwithNana/featured

YT - Ubuntu. (s.f.). Obtenido de https://www.youtube.com/c/UbuntuOS/videos