Universidad Zaragoza

1542

Trabajo Fin de Grado

Implantación de un clúster Kubernetes en "bare-metal", con optimización de uso de energía renovable
Kubernetes cluster on "bare-metal" deployment, optimized to use renewable energy.

Autor/es

Roberto Jiménez López

Director/es

José Félix Serna Fortea

Escuela Universitaria Politécnica de Teruel

2021/2022

Resumen

En la actualidad, existe una gran cantidad de servicios que se ofrecen de manera digital, los cuales precisan de características como la escalabilidad, tolerancia a fallos o la optimización del coste.
Uno de los objetivos de este TFG es implantar un *cluster* Kubernetes en “bare-metal”, documentando de una manera precisa todo el proceso llevado a cabo.

Por otro lado, se va a desarrollar un prototipo de aplicación que permita priorizar la asignación de las cargas de trabajo presentes en el *cluster*, a aquellos nodos de los centros de datos que estén generando mayor cantidad de energía renovable, con el objetivo de minimizar la huella de carbono asociada a las tareas y operaciones del *cluster*.

Finalmente, se va a desarrollar una aplicación de ejemplo basada en microservicios, que será ejecutada en el *cluster* implantado.

Abstract

Nowadays, there are a large number of services that are offered digitally, which require features such as scalability, fault tolerance and cost optimization.
One of the goals of this TFG is to implement a “bare-metal“ Kubernetes cluster, documenting in a precise way all the process carried out.

Moreover, a prototype application will be developed to proritize the allocation of the workloads present in the cluster to those nodes of the data centers that are generating the largest amount of renewable energy, with the aim of minimizing the carbon footprint associated with the cluster’s tasks and operations.

Finally, an example application based on microservices will be developed and executed in the implemented cluster.

Palabras clave:
Cluster, Kubernetes, Docker, Contenedor, Software, MQTT, Arduino, Java, Python, JMeter, Optimización, Energía renovable.

Índice

1. Introducción **1**
1.1. Objetivos 1
1.2. Propósito 1
1.3. Alcance 2
1.4. Definiciones y acrónimos 3
1.5. Estructura del documento 5

2. Conceptos preliminares **6**
2.1. Contenedores 6
2.1.1. ¿Qué es un contenedor? 6
2.1.2. Diferencia Contenedores y Máquinas Virtuales 6
2.1.3. ¿Qué es un motor de contenedores (Container Engine)? . 7
2.1.4. ¿Qué es una imagen de contenedor? 8
2.1.5. Containerd vs CRI-O 8
2.1.6. ¿Qué es un *runtime* de contenedores? 9
2.1.7. Utilidades de un contenedor 9
2.1.8. Registro de contenedores 9
2.1.9. Docker 10
2.2. Kubernetes 10
2.2.1. ¿Qué es Kubernetes? 10
2.2.2. Componentes de un nodo *master* 10
2.2.3. Componentes de un nodo *worker* 11
2.2.4. Elementos de kubernetes 11
2.3. MQTT 14
2.3.1. Modelo Publish/Subscribe 14
2.3.2. Funcionamiento 15
2.3.3. Alternativas MQTT 16
2.4. Arduino 16
2.5. JMeter 16

3. Instalación de Kubernetes en "bare-metal" **17**
3.1. Requisitos previos 17
3.2. Configuración común de todos los nodos 17
3.2.1. Habilitar el tráfico en modo "bridge" 18
3.2.2. Deshabilitar espacio *swap* del nodo 18
3.2.3. Instalar un motor de contenedores 18
3.2.4. Instalar herramientas *kubeadm*, *kubelet* y *kubectl* 20
3.3. Inicializar nodo *master* 20
3.4. Inicializar nodos *worker* 21
3.5. Instalación "*plugins*" en el *cluster* 22
3.5.1. Proyecto Calico 22
3.5.2. Instalación servidor de métricas 22

4. Demostrador tecnológico **23**
4.1. Análisis 23
4.1.1. Diagramas 23
4.1.2. Publicación APIs en el sistema 28
4.2. Desarrollo/Puesta en marcha 29
4.2.1. Registro de contenedores 29
4.2.2. Base de Datos 30
4.2.3. AhorraTER_PRICES 32
4.2.4. AhorraTER_GUI 34
4.2.5. AhorraTER_MANAGER 36
4.2.6. AhorraTER_REST_K8S 38
4.2.7. AhorraTER_TELEGRAM_REST 40
4.2.8. Automatización de la puesta en marcha a través de scripts 41
4.3. Tecnologías y dispositivos utilizados 42
4.3.1. Dispositivos utilizados 42
4.3.2. Tecnologías utilizadas 42
4.4. Resultados obtenidos 45
4.4.1. Test microservicio AhorraTER_PRICES 45
4.4.2. Test microservicio AhorraTER_MANAGER 46
4.4.3. Test microservicio AhorraTER_GUI 47

5. Conclusiones y trabajo futuro **48**

1. Introducción

1.1. Objetivos

En la actualidad, existe una gran cantidad de servicios que se ofrecen de manera digital, a través de la informática se consigue dar soporte tanto a grandes como pequeñas instituciones que dependen de la fiabilidad y disponibilidad de sus sistemas informáticos. Estas aplicaciones, como norma general precisan de una serie de características:

- Escalabilidad. Existen 2 tipos: vertical (aumento de los componentes hardware sin que afecte al software) y horizontal (aumentar el número de nodos en un sistema sin que el mismo se vea afectado)
- Tolerancia a fallos: Pueden aparecer fallos a nivel de hardware (en el caso de un fallo en un disco), fallos a nivel de red (caída de la red de comunicaciones), caídas del sistema debido a fallos en la red eléctrica. El sistema debe ser capaz de mantener su operatividad pese a posibles fallos
- Optimizar la infraestructura con el fin de economizar el coste de uso, una solución utilizada en las grandes empresas actuales consiste en compartir la infraestructura entre varias aplicaciones/clientes (siendo independientes entre ellas) para evitar tener una gran infraestructura con exceso de recursos. Este concepto se conoce como "*multi-tenancy*".

Como valor adicional a estas características que son necesarias prácticamente en cualquier aplicación informática, en la actualidad, además del objetivo de optimizar costes de uso, es primordial la responsabilidad social.

Es por ello que se busca optimizar o reducir el consumo energético de los centros de datos, en mayor medida aquellos que disponen de parques de generación de energía renovable asociados a los mismos.

1.2. Propósito

Este Trabajo de Fin de Grado consiste en la implantación de un *cluster* (conjunto de nodos que trabajan en conjunto como si fuese uno) basado en la tecnología Kubernetes en "bare-metal" (dedicadas al 100 % a su trabajo en el *cluster*), de forma que permita el despliegue de todo tipo de aplicaciones en un ambiente de desarrollo y producción.

Para el desarrollo del proyecto, se asumirá que cada nodo (máquina física) perteneciente al *cluster*, tiene asociado un parque de generación de energía renovable, el cual permite suministrar energía en momentos determinados del día. Se adume también que cada nodo está geográficamente disperso.

El sistema se encargará de balancear la carga en los distintos nodos del *cluster*, en base a la energía renovable que esté produciendo cada parque asociado al nodo en el momento de despliegue de una aplicación. De forma que, se pueda asignar la carga de trabajo a los nodos que puedan consumir un mayor porcentaje de energía renovable dinámicamente.

Este sistema permite reducir, por un lado, el consumo del nodo con mayor carga de trabajo, y por otro, la huella de carbono, a partir del uso de una energía renovable.

Este Trabajo Fin de Grado viene motivado tras la realización de prácticas en la empresa NTT Data, donde se ha trabajado con algunas de las herramientas que se utilizarán para el desarrollo del proyecto.

1.3. Alcance

Los distintos aspectos a considerar en este Trabajo Fin de Grado son:

- Configuración e instalación de los paquetes necesarios para la instalación de las herramientas Kubernetes y Docker.
- Puesta en marcha de un *cluster* Kubernetes.
- Despliegue de un repositorio local de Docker para las distintas imágenes construidas.
- Análisis de los distintos resultados de disponibilidad que existen entre el despliegue de aplicaciones sobre un *cluster* con balanceo de cargas y un servidor tradicional.
- Implementación de un algoritmo que permita decidir qué nodo del *cluster* tiene unas mejores condiciones para obtener energía renovable y, destinar la carga de trabajo al que cumpla las condiciones más óptimas.
- Diseño de una aplicación basada en microservicios que sirva como demostrador tecnológico de todo lo mencionado.
- Realizar pruebas de escalabilidad con la herramienta JMeter.

1.4. Definiciones y acrónimos

Algunas definiciones de conceptos que se van a utilizar a lo largo de todo el documento:

- SSH: Secure Shell. Herramienta que permite el acceso remoto a un servidor por medio de un canal criptado.
- K8S: Kubernetes. Plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios. [1].
  K8S facilita la automatización y configuración declarativa.
- MQTT: Message Queue Telemetry Transport. Protocolo de comunicaciones desarrollado por IBM y Arcom como un mecanismo para conectar dispositivos empleados en la industria petrolera en 1999. [14]
- HTTP: Hypertext Transfer Protocol. Protocolo de comunicación que permite transferencias de información en Internet.
- MV: Máquina Virtual: Es un software que simula un sistema de computación y puede ejecutar programas como si fuese una computadora real.
- IoT: Internet of Things o Internet de las cosas, es el proceso por el cual se pueden conectar elementos físicos cotidianos a Internet, desde una bombilla hasta un dispositivo médico que permita avisar a una enfermera.
- QoS: Quality of Service: Forma de gestionar la robustez del envío de mensajes al cliente ante fallos.
- CLI: Command Line Interface. Es la interfaz de línea de comandos, permite a los usuarios escribir comandos de texto para que un computador realice una tarea específica.
- JSON. JavaScript Object Notation, es un formato de texto sencillo utilizado para el intercambio de datos.
- URI: Uniform Resource Identifier: Es un identificador que sirve para acceder a un recurso físico o abstracto por Internet, es utilizado en aplicaciones para interactuar con un recurso o consultar información sobre el mismo.
- Contenedor: Paquete de software que agrupa el código de una aplicación con las bibliotecas y archivos de configuración asociados y sus dependencias para que se ejecute, permitiendo que una aplicación pueda se ejecutada en cualquier dispositivo.
- IDE. Integrated Development Environment. Aplicación informática que facilita el desarrollo de software.
- JPA. Java Persistence API, es una API de persistencia desarrollada para Java.

- I2C. Es un bus serie de datos diseñado por Philips Semiconductors a principios de los años 80. Se utiliza para la comunicación entre partes de un circuito. [45]
- CI/CD: Continuous integration and continuous delivery.
- OCI: Open Container Initiative. Es un proyecto de la "Linux Foundation" para diseñar un estándar abierto para virtualización a nivel de sistema operativo.
- CRI: Container *Runtime* Interface. Es un *plugin* de Kubernetes que permite utilizar gran variedad de *runtime* de contenedores.
- *Web Services*: Un *web service* facilita un servicio a través de Internet, es una interfaz mediante la que dos máquinas o aplicaciones se comunican entre sí.
- Multi-tenancy: Modo de operación del software en varias instancias independientes de una o múltiples aplicaciones las cuales están operando en un entorno compartido, pero aislados entre sí.
- *Cluster*: Un *cluster* de servidores es un conjunto de varios servidores que se comunican entre sí y permiten trabajar como un único servidor.
- Docker: Es un *runtime* para contenedores (Docker como herramienta, no como empresa), se instala como un paquete en los servidores en los que se desee construir contenedores.
- Registro de contenedores: Un registro de contenedores es un repositorio (en la nube o en una máquina física) donde se almacenan imágenes de contenedores de forma privada.
- API. Application Programming Interface. Es un conjunto de funciones y mecanismos que permiten a dos componentes software comunicarse entre sí a través de un protocolo [49].
- Servidor proxy: Un servidor proxy es un programa o dispositivo que hace de intermediario en las peticiones de recursos que realiza un cliente a otro servidor.
- Microservicios: La arquitectura de microservicios es un método de desarrollo de aplicaciones que funciona como un conjunto de pequeños servicios que se ejecutan de manera independiente y autónoma, pero con toda la funcionalidad completa, proporcionando escalabilidad.
- Servidor bare-metal: Consiste en un equipo informático cuyos recursos y potencia de cálculos son para una sola finalidad, en el caso de este proyecto, cada nodo que forma el *cluster* [11].
- Protocolo TCP/IP: Es un conjunto de reglas estandarizadas que permiten a los equipos comunicarse en una red como Internet.

- Bash: Es una interfaz de usuario de línea de comandos, específicamente un shell de Unix.
- Modelo OSI: Open System Interconnection, es un modelo de referencia para los protocolos de la red que fue definido entre 1977 y 1983. Divide todas las funciones que tiene que realizar un sistema de comunicaciones en siete capas o niveles.
- Container Engines: Motor de contenedores, se trata de una aplicación que permite gestionar los contenedores, simulando un sistema operativo que permite montar contenedores sin necesidad de tener que crear un hardware virtual.
- Espacio SWAP: En Linux el espacio *swap* se utiliza cuando la RAM está llena. Si el sistema necesita más recursos de memoria y la memoria física está completa, las páginas inactivas de la memoria se mueven al espacio *swap*.
- SELinux: Security Enhanced Linux. Extensión de seguridad para el *kernel* de Linux.
- *Daemon*: Un proceso demonio es un programa que se ejecuta en segundo plano. Sus nombres suelen terminar con la letra “d”, por ejemplo sshd es el demonio que maneja las conexiones SSH.

1.5. Estructura del documento

A continuación, en el capítulo 2 (página 6), se describen los conceptos más relevantes que serán utilizados a lo largo del trabajo, así como sus alternativas, utilidades y ámbitos de trabajo.

En el capítulo 3 (página 17) se detalla cómo instalar y poner en marcha cada uno de los nodos del *cluster*.

En el capítulo 4 (página 23) se especifica el diseño e implementación llevados a cabo para construir una maqueta funcional, de una aplicación basada en microservicios y todo el proceso llevado a cabo para desplegarla sobre el *cluster*.

Por último, en el capítulo 5 (página 48) se mostrarán las conclusiones y algunas propuestas de mejoras futuras.

2. Conceptos preliminares

En este capítulo se exponen de manera detallada algunos de los conceptos más relevantes del proyecto, así como sus ámbitos de trabajo y alternativas.

2.1. Contenedores

2.1.1. ¿Qué es un contenedor?

Un contenedor software es un paquete de elementos que permite ejecutar una aplicación determinada en cualquier SO.
El objetivo de los contenedores es garantizar que una aplicación se ejecute correctamente cuando cambie su entorno.
En las situaciones que se pretende aislar de una manera más eficiente un contenedor, se utilizan herramientas como SELinux.

Un contenedor tiene dos estados:

- Parado: Cuando un contenedor está parado, actúa como un archivo o conjunto de archivos que está almacenado en disco.
- En ejecución: Cuando un usuario ejecuta un comando para iniciar un contenedor, el motor de contenedores desempaqueta los archivos necesarios y se los manda al *kernel*. En el proceso de creación se monta una copia de los archivos del contenedor. Una vez el contenedor se está ejecutando, es un proceso más sobre una máquina.

2.1.2. Diferencia Contenedores y Máquinas Virtuales

El objetivo de los contenedores es distinto al de las MVs, en este apartado se pretende detallar la diferencia entre una máquina virtual y un contenedor.

Una máquina virtual es un software que simula un sistema de computación y puede ejecutar programas como si fuese una computadora real. Una MV tiene CPU, memoria, discos para almacenar los archivos y conexión a internet. [46]

Los contenedores funcionan de manera diferente, aunque también tratan de aislar las aplicaciones, su objetivo es generar un entorno que sea replicable en cualquier máquina, y en vez de colocar un SO completo, lo que hacen es compartir los recursos del *host* sobre el que se ejecutan.

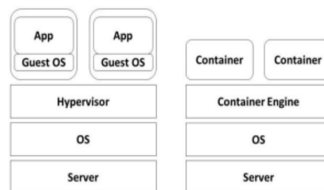


Figura 1: Diferencia entre MV y contenedor. Fuente [23]

Las principales ventajas que proporcionan los contenedores sobre las MVs son las siguientes:

- En una MV se necesitan asignar los recursos físicos que van a ser utilizados, mientras que en los contenedores no es necesario, ya que el propio motor de contenedores es capaz de asignarlo automáticamente
- Los contenedores se ejecutan de manera más eficiente y veloz que las máquinas virtuales.
- El espacio ocupado en disco es menor en los contenedores, ya que no se replica todo el SO completo.

La principal desventaja que tiene actualmente el uso de contenedores es, que no se permite desplegar un contenedor cuyo sistema operativo sea distinto al del *host*.

2.1.3. ¿Qué es un motor de contenedores (Container Engine)?

Un motor de contenedores es una pieza de software que acepta las solicitudes de los usuarios (y de la línea de comandos), obtiene las imágenes de los repositorios, y a vista del usuario, es el encargado de ejecutar el contenedor. [22]

Sin embargo, realmente el motor de contenedores no es el encargado de ejecutar un contenedor, sino que delega el trabajo en un *runtime* compatible con OCI (en la mayoría de los casos, “runc”).

Las funciones del motor de contenedores son, principalmente: manejar las entradas que provienen tanto de usuarios como de orquestadores (p.e Kubernetes), obtener las imágenes de un registro de imágenes, preparar un punto de montaje para el contenedor, enviar la información necesaria al contenedor y llamar al *runtime* para ejecutarlo.

Los principales motores de contenedores son:

- Docker: Se trata de una plataforma software que engloba todas las partes del proceso de creación de contenedores, ya sea a través de su herramienta de línea de comandos o su interfaz, Docker permite gestionar todos los contenedores de una manera óptima.

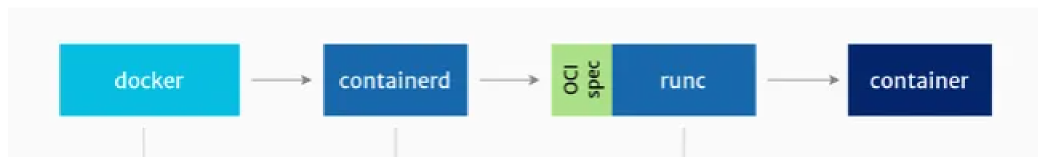


Figura 2: Esquema que muestra el proceso que sigue la herramienta docker para crear interactuar con un contenedor. Fuente: [21]

- Podman: Es una herramienta nativa de Linux, de código abierto y sin demonios que está diseñada para facilitar la creación, uso e implementación de aplicaciones utilizando OCI.
  Podman ofrece una CLI muy similar a la que ofrece Docker (permite crear un alias en Linux para utilizar podman en lugar de docker). [30]

- RKT es un motor de contenedores de aplicaciones desarrollado para entornos cloud, sigue la especificación App Container y permite la ejecución de imágenes de contenedor Docker. [31]

- Linux Containers (LXC), es una tecnología de virtualización a nivel de SO que permite crear y correr varios entornos virtuales Linux de manera aislada. Consume pocos recursos y no necesita ser ejecutada como administrador, pero su uso es complejo y su documentación es confusa. [32]

2.1.4. ¿Qué es una imagen de contenedor?

Una imagen de un contenedor es un paquete software independiente y ejecutable que incluye todo lo necesario para ejecutar una aplicación (código, *runtime*, bibliotecas, herramientas del sistema y su configuración). Este paquete se puede obtener de un registro de imágenes local, o en la nube. Esto quiere decir que se pueden crear y publicar imágenes personalizadas, permitiendo su acceso a todo el mundo o a una lista restringida de usuarios.
La imagen es utilizada localmente cuando se ejecuta un contenedor como punto de montaje para empezar.

En sus inicios, cada motor de contenedores tenía su propio formato de imagen de contenedores (LXD, RKT y Docker), actualmente casi todos los motores principales de contenedores han cambiado el formato al que define OCI [19]. Este formato principalmente define que una imagen debe estar compuesta por una serie de archivos con extensión ".tar" para las capas y un archivo JSON con los metadatos.

2.1.5. Containerd vs CRI-O

Cuando se está trabajando con contenedores y se va a utilizar Kubernetes, es necesario un *daemon* que permita controlar tanto el *runtime* del contenedor como su ciclo de vida completo.
Las 2 opciones que implementa la API de Kubernetes (CRI) son las siguientes:

- Containerd: Es el *runtime* para Linux y Windows, se encarga de administrar el ciclo de vida completo del contenedor en el sistema *host*, sus características principales son:
  - Soporta la cración/modificación/eliminación de redes.
  - Soporta *multi-tenant*.

- CRI-O: Es un *runtime* de contenedores que utiliza instancias de OCI y surgió como una implementación de la interfaz CRI. Algunas de sus características son:
  - Monitorización continua de los contenedores.
  - Interfaz de red de contenedores para crear redes.
  - Tiempo de ejecución compatible con OCI.
  - Está enfocado a Kubernetes como entorno, permite ejecutar contenedores directamente sin necesidad de ninguna herramienta adicional.

2.1.6. ¿Qué es un *runtime* de contenedores?

El *runtime* de contenedores es la herramienta que permite generar y ejecutar contenedores en tiempo de ejecución, el más común es “runc”, que fue creado por la empresa Docker y es funcional en Docker, Kubernetes y otros programas que trabajen con contenedores.
Actualmente es una especificación del proyecto OCI.

2.1.7. Utilidades de un contenedor

El uso de contenedores es una tendencia actual que permite grandes beneficios frente a las máquinas virtuales y servidores tradicionales.
Algunas de sus utilidades son:

- Aislamiento, permite desplegar una o más aplicaciones.
- Seguridad, en caso de encontrar alguna vulnerabilidad, el atacante solo podría acceder a dicho contenedor y no al resto de la máquina, limitando el daño a ese contenedor.

  Una buena práctica en relación a los contenedores y la seguridad es utilizar, en la medida de lo posible, contenedores sin privilegios, ya que en caso de ataque, el atacante tendrá menos oportunidad de dañar el sistema.
- Modularidad, uno de los enfoques principales del uso de contenedores es la abstracción por partes de una aplicación, de modo que cada parte se pueda actualizar o reparar sin deshabilitar el resto de funcionalidades.
- Restauración, un contenedor tiene la capacidad de restauración, cada imagen tiene una serie de capas que se puede restaurar, esto apoya la integración e implementación continua (CI/CD).

2.1.8. Registro de contenedores

Un registro de contenedores (Docker registry) es una aplicación que gestiona el almacenamiento y el envío de imágenes de contenedores Docker, permitiendo almacenar imágenes modificadas de manera privada.

2.1.9. Docker

Docker es un proyecto de código abierto, cuya idea principal es la de crear contenedores ligeros y que sean portables independientemente del SO instalado, creando de esta forma una capa de abstracción que permita ejecutar un software en cualquier máquina con Docker instalado.

2.2. Kubernetes

2.2.1. ¿Qué es Kubernetes?

Kubernetes es una plataforma portable y extensible de código abierto que permite administrar las cargas de trabajo y servicios, facilita la automatización y cuenta con una gran cantidad de documentación. Esta plataforma ofrece un entorno de administración centrado en contenedores, orquesta la infraestructura para que los usuarios no tengan que hacerlo.

Kubernetes utiliza una arquitectura *master-worker* en la que se distinguen dos tipos de nodos:

- *Master* : Son los responsables de gestionar el *cluster*, se encargan de repartir el trabajo entre los nodos *workers* y detectan y dan respuesta a los distintos eventos que ocurren en el sistema. En un *cluster*, puede existir 1 o más nodos *master*, para aumentar la disponibilidad del mismo.
  En el caso de que haya varios nodos *master*, el número de estos tiene que ser impar, para que pueda haber consenso entre los distintos nodos para poder acordar las actualizaciones del estado del *cluster*.
- *Worker*: Son los responsables de ejecutar las aplicaciones en pods.

2.2.2. Componentes de un nodo *master*

Los componentes que tienen los nodos *master* son los siguientes:

- API Server: Componente que expone la API de Kubernetes y donde se validan todas las solicitudes REST.
- Scheduler (Planificador): Observa los pods que se acaban de crear sin un nodo asignado y se lo asigna, en función de los recursos necesarios.
- Controller-manager: Ejecuta los controladores, que usan el API server para observar el estado del *cluster*.
- etcd: Es un almacén de datos que se utiliza para guardar la configuración compartida del *cluster* y para descubrir servicios.

2.2.3. Componentes de un nodo *worker*

Los componentes que tiene un nodo *worker* son:

- *Kubelet*: Es el servicio dentro del *worker* responsable de comunicarse con el nodo *master*, obtiene la configuración de los pods del API server y es el encargado de comprobar que los nodos están corriendo y que tienen la configuración correcta.
- *Kube-proxy*: Actúa como un proxy de red y balanceador de carga, de modo que enruta el tráfico al contenedor correspondiente en cada petición realizada.
- *Runtime* de contenedores: Para poder ejecutar los distintos contenedores que se vayan a ejecutar (capítulo 2.1.6).

2.2.4. Elementos de kubernetes

En primer lugar, destacar que todos los elementos con los que trabaja Kubernetes se despliegan sobre un espacio de nombres "namespaces". Un namespace es una forma de dividir los recursos del *cluster* entre múltiples usuarios, de forma independiente.

Kubernetes permite trabajar con una gran variedad de elementos, los más frecuentes son los siguientes:

- Pod: Es un grupo de uno o más contenedores, con almacenamiento y red compartidos y una especificación de como ejecutar los contenedores.
  Un contenedor dentro de un Pod comparte dirección IP y puerto, y se pueden encontrar a través de *localhost*. Un pod es una entidad efímera, cuando se crea un pod se le asigna un identificador único (UID) y se despliega en un nodo donde permanecen hasta su finalización, si un nodo muere, los pods mueren pasado un tiempo, y se crearán réplicas de los pods pero con otro UID.
- ReplicaSet: Es un tipo de controlador, junto a los Deployments y StatefulSet. Un ReplicaSet se define a través de campos, e incluye un selector que indica cómo identificar a los pods que puede controlar, señalando cuantos pods debe gestionar para conseguir el número de réplicas esperado. Su objetivo es crear y eliminar los pods necesarios para alcanzar un número esperado.
- Deployment: Un Deployment proporciona actualizaciones declarativas para pods y ReplicaSets, esto quiere decir que cuando se describe el estado en un Deployment, el controlador se encarga de cambiar el estado actual al deseado. Permite controlar los ReplicaSets (actúa como un elemento situado una capa por encima de estos), es recomendable no crear ReplicaSets directamente, sino crear un objeto Deployment que lo controle.
  Permite escalar horizontalmente, declarar un nuevo estado (nueva versión) o volver a una versión anterior en caso de que no sea estable.

- StatefulSet: El StatefulSet actúa del mismo modo que un Deployment, sin embargo el StatefulSet gestiona los pods que se basan en una especificación idéntica del contenedor, los StatefulSet mantienen la identidad asociada a sus pods, es decir si se crean con un identificador concreto, se mantendrá. Permite identificadores de red estable y escalado ordenado. Una limitación de los StatefulSet es que cuando se elimina un StatefulSet no tienen por qué eliminarse todos sus pods, por lo que es recomendable escalarlo a cero antes de eliminarlo.

- Jobs: Un Job consiste en la creación de uno o más pods de forma que se asegure que un número de ellos termina de forma satisfactoria, cuando se alcanza dicho número de ejecuciones satisfactorias, el Job se completa y los pods se eliminan.

- CronJobs: Un Cronjob es similar a un *crontab* de una máquina Linux, ejecuta un trabajo de forma periódica según un horario establecido por el usuario en formato Cron *"* * * * *"*

- Service: Es un objeto que describe cómo se accede a las aplicaciones, permite describir pods, puertos y balanceadores de carga. Es una abstracción que define una política que permite acceder a otros elementos de Kubernetes, es muy útil para poder acceder a los pods de un Deployment sin conocer la dirección de ninguno de estos.
  Dentro del *cluster*, si se quiere acceder a los recursos a los que apunta un servicio, se debe seguir esta estructura:
  "<service-name>.<namespace>.svc.cluster.local:<svc-port>".
  Con este formato, se puede acceder a los pods que controle un servicio.

  Hay distintos tipos de servicio en función del uso que se le quiera dar: ClusterIP que expone el servicio de forma interna en el *cluster*, NodePort que expone el servicio en un puerto estático de la máquina haciendolo visible desde fuera del *cluster*, LoadBalancer que expone el servicio de manera externa usando el balanceador de carga del proveedor de la nube y finalmente Ingress el cual no es un tipo de servicio pero actúa como el punto de entrada del *cluster*, permite exponer múltiples servicios bajo la misma IP.

Para cada uno de estos elementos puede ser necesaria configuración adicional, como el nombre de un servidor, la contraseña de un pod para la base de datos... Por ellos existen ciertos elementos de configuración que permiten añadir datos a los pods:

- ConfigMap: Un ConfigMap es un objeto de la API utilizado para almacenar datos no confidenciales en el formato clave-valor. Un pod puede utilizar un ConfigMap como variable de entorno, argumento de la línea de comandos o como ficheros de configuración en un volumen, estos datos no están encriptados, por lo que no es una buena práctica introducir datos confidenciales en un ConfigMap, como solución a este problema, existen los Secrets.
  El uso de los objetos ConfigMap es útil para correr aplicaciones que necesitan de variables de entorno y van a ser ejecutadas en dos entornos distintos para pruebas (en local y en un entorno cloud).

- Secrets: Un secret es un objeto similar a un ConfigMap, pero contiene datos confidenciales como contraseñas o token. Los secretos se codifican en base64.

Las distintas aplicaciones que se desplieguen, serán funcionales, pero si requieren de persistencia, deberán utilizar volúmenes, es por ello que en Kubernetes también existe el concepto de volumen.
Los contenedores son efímeros y la información se pierde cuando un pod muere, por ello, un volumen permite resolver este problema dando persistencia a la información. Hay varios tipos de volúmenes:

- EmptyDir: Un volumen de tipo *emptyDir* es creado cuando se asigna un pod a un nodo, y existe mientras el pod se está ejecutando en el nodo, está inicialmente vacío y permite escribir y leer archivos en el volumen.

- HostPath: Un volumen de tipo *hostPath* permite persistir la información dentro del nodo, para ello, monta un volumen del contenedor en un *path* del nodo original.

La forma de utilizar estos volúmenes es a través de los recursos Peristent Volumes (PV) y Persistent Volume Claim (PVC). Por un lado el PV es un volumen que contiene datos procesados por los contenedores que no se eliminan cuando un contenedor muere, y, por otro lado los PVC son el recurso que permiten reclamar esos PV, es decir, indicar el PV que se quiere utilizar, una vez asignado un PV a un PVC, este puede ser montado por un Deployment o similar.

2.3. MQTT

Message Queue Telemetry Transport (MQTT) es un protocolo de comunicación de mensajes que se basa en una arquitectura publicación/suscripción. Está situado por encima del protocolo TCP, y requiere pocos recursos a nivel de procesamiento y ancho de banda, lo que hace que MQTT sea un protocolo muy útil para dispositivos Arduino, Raspberry PI...

MQTT tiene 3 niveles de confiabilidad (QoS) para enviar mensajes, de modo que se pueda establecer la confiabilidad de los tipos de mensajes:

- Nivel 0: Como máximo una vez, en caso de fallo puede que algún mensaje no se entregue.
- Nivel 1: Al menos una vez, el mensaje se envía hasta que se garantiza la entrega, en caso de fallo, el suscriptor puede recibir algún mensaje duplicado.
- Nivel 2: Exactamente una vez, se garantiza que cada mensaje se entrega al suscriptor, y únicamente una vez.

Cada vez que se publica un mensaje, este es enviado al *broker*. El funcionamiento de un *broker* es reenviar los mensajes recibidos a los dispositivos que se hayan suscrito a un *topic* concreto.

Los *topic* son cadenas de texto UTF-8 que distinguen entre mayúsculas y minúsculas y tiene una longitud máxima de 65535 caracteres. Los *topic* están formados por uno o más niveles separados entre sí por el símbolo "/".

En relación a los aspectos de seguridad, MQTT utiliza el cifrado TLS y autenticación de clientes mediante protocolos actuales como OAuth, además, permite cifrar las conexiones con el *broker* para que únicamente puedan publicar mensajes y suscribirse a los *topic* los usuarios que cuenten con las credenciales adecuadas.

Al tratarse de un protocolo muy ligero, permite que una mayoría de dispositivos pueda utilizar MQTT, desde máquinas completas a microcontroladores, lo que es de gran utilidad para aplicaciones IoT.

2.3.1. Modelo Publish/Subscribe

El modelo *publish/subscribe* desacopla al cliente que envía el mensaje (emisor) del cliente o clientes que van a recibir los mensajes (suscriptor), de manera que emisores y suscriptores no contactan entre ellos de ninguna manera, la conexión entre ellos es manejada a través del *broker*.

El aspecto principal de este modelo es el desacoplamiento del emisor y el receptor, en varias dimensiones:

- En relación a la ubicación, tanto emisor como receptor no necesitan conocer ninguna dirección IP o puerto para poder enviar información, salvo la del *broker*.

- Respecto al tiempo, emisor y suscriptor no necesitan ejecutarse al mismo tiempo.
- Sincronización, las operaciones que se realizan en emisor y suscriptor no son interrumpidas durante el envío o recepción del mensaje, por lo que trabajan de forma asíncrona.

2.3.2. Funcionamiento

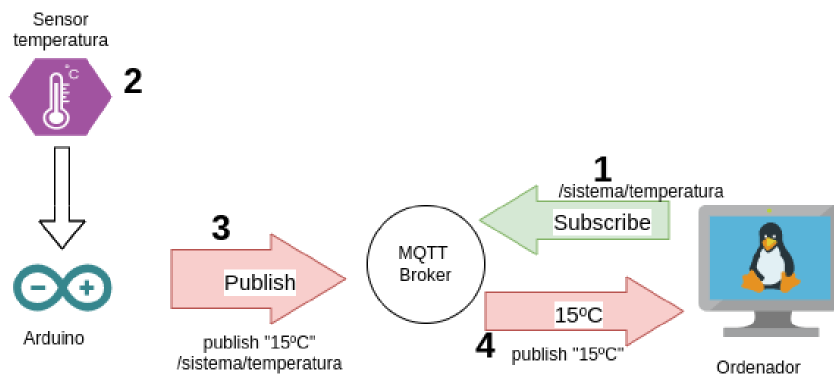


Figura 3: Ejemplo de funcionamiento del protocolo MQTT. Fuente [47]

En la figura 3 se puede observar un ejemplo simple de funcionamiento de todos los componentes de MQTT.

1. Para comenzar, un dispositivo (móvil, ordenador...) se suscribe a un *topic* a la espera de recibir información.
2. Por otro lado, un dispositivo (microcontrolador, ordenador...) recogerá, de un sensor, la temperatura actual.
3. Con la información en el dispositivo, este publica un mensaje con los valores recogidos sobre un *topic*, en este caso "sensores/temperatura/".
4. El *broker* recibe el mensaje publicado y lo reenvía a todos aquellos dispositivos que estén suscritos a ese mismo *topic*. Por tanto, el ordenador que se había suscrito, recibe la información de la temperatura.

2.3.3. Alternativas MQTT

Algunas de las alternativas que existen, a día de hoy, para el protocolo MQTT son las siguientes:

- CoAP: Constrained Application Protocol. Es un protocolo de software que se encuentra en el nivel de capa de aplicación del modelo OSI, y del mismo modo que MQTT está orientado a que se pueda ejecutar en dispositivos simples, permitiendo su comunicación en internet.

  Fue diseñado para trasladar las *requests* y *responses* de HTTP a dispositivos con recursos limitados, trabaja de forma asíncrona y trabaja sobre UDP.

- XMPP: Extensible Messaging and Presence Protocol. Es un protocolo abierto basado en XML, diseñado en su origen para mensajería instantánea.

- AMQP: Advanced Message Queuing Protocol. Es un protocolo de estándar abierto en la capa de aplicaciones de un sistema de comunicación, es un protocolo a nivel de cable.

2.4. Arduino

Arduino es una plataforma de desarrollo basada en una placa electrónica de hardware libre que incorpora un microcontrolador re-programable y una serie de sensores que permiten establecer conexiones entre el microcontrolador y distintos sensores [48].

Arduino se puede utilizar para desarrollar elementos autónomos o conectarse a otros dispositivos para interactuar con otros programas.

2.5. JMeter

JMeter es un proyecto de Apache que se utiliza como una herramienta de prueba de carga, para analizar y medir el rendimiento de una variedad de servicios, principalmente servicios web.

3. Instalación de Kubernetes en "bare-metal"

Existen varias formas de instalar un *cluster* Kubernetes y distribuciones alternativas que ofrecen aspectos similares:

- K3S: Es una versión de Kubernetes certificada. La principal diferencia con K8S es, que K3S está diseñado para ser un binario de poco tamaño que implemente completamente la API de Kubernetes, elimina algunos controladores que se sacan del núcleo principal y pueden ser instalados adicionalmente.
  Permite la instalación en dispositivos con 512MB de memoria RAM en adelante.
- K0S: Es otra distribución de Kubernetes pensada para desarrolladores que no tengan conocimientos previos, su versión de Kubernetes no es la más actual.

Para el desarrollo del proyecto se va a realizar la instalación de K8S a través de la herramienta *kubeadm*, que se utiliza para configurar un *cluster* Kubernetes. Durante la instalación, *kubeadm* realiza comprobaciones que permiten verificar que el servidor tenga todos los requisitos esenciales.

3.1. Requisitos previos

Para la instalación, existen dos requisitos previos esenciales:

- 2 nodos mínimo, uno que actuará como *master* y el otro como *worker*.
- Comunicación entre los nodos, estos deben poder comunicarse entre ellos para que funcione correctamente, se puede comprobar a través del comando *ping*.

Para este TFG se han utilizado 3 nodos, uno que actúa como *master* y los otros dos nodos que actúan como *workers*. Estos nodos pueden comunicarse entre sí, ya que están trabajando con direcciones de red públicas.

3.2. Configuración común de todos los nodos

A nivel de instalación, hay elementos comunes entre los nodos *master* y *worker*, es por ello que en esta sección se pretende explicar los paquetes y configuraciones necesarias que son iguales para el nodo maestro y el trabajador.

3.2.1. Habilitar el tráfico en modo "bridge"

Primeramente, se debe habilitar el tráfico en modo *bridge* a través de la herramienta iptables.

El tráfico en modo *bridge* permite crear una única red a partir de múltiples redes de comunicación o segmentos de red de forma independiente del protocolo. De esta forma los paquetes se reenvían en función de la dirección Ethernet (en vez de la dirección IP como en un router). Para activar este modo, se deberán ejecutar los siguientes comandos:

```
1   cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
2   br_netfilter
3   EOF
```

A través del comando "tee" se escribe en la entrada estándar del fichero k8s.conf el texto "br_netfilter", que es un módulo de iptables que permite habilitar el enmascaramiento transparente [3].

Del mismo modo, se ejecuta para el fichero k8s.conf de la carpeta sysctl.d el siguiente comando:

```
1   cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
2   net.bridge.bridge-nf-call-ip6tables = 1
3   net.bridge.bridge-nf-call-iptables = 1
4   EOF
5   sudo sysctl --system
```

Como resultado de la ejecución de estos comandos, se activa completamente el modo *bridge* en el nodo.

3.2.2. Deshabilitar espacio *swap* del nodo

A continuación, se debe deshabilitar el espacio *swap* (capítulo 1.4) en los nodos, aunque no es obligatorio, es recomendable por dos motivos: Primero, Kuernetes no soporta *swap* en algunas de sus versiones y, segundo, uno de los objetivos de Kubernetes es tener todos los archivos en disco, y en caso de no poder almacenarlos, mandarlos a otro nodo.

Para deshabilitar el *swap* en un nodo se deben ejecutar los comandos:

```
1   sudo swapoff -a
2   sudo sed -i '/ swap / s/^\(.*\)$/\#\1/g'
    /etc/fstab
```

3.2.3. Instalar un motor de contenedores

El siguiente paso consiste en instalar un motor de contenedores a los distintos nodos del *cluster*. Para esta instalación se ha escogido Docker dentro de las distintas alternativas que se han comentado en el capítulo 2.1, ya que cuenta con una documentación muy detallada y ya se había trabajado con ella previamente.

Para instalar Docker en los nodos, se deben ejecutar los siguientes comandos:

```
sudo apt-get update -y
sudo apt-get install -y apt-transport-https
ca-certificates curl gnupg lsb-release
curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
sudo gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg
echo "deb [arch=amd64
signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu  $(lsb_release
-cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null

```

Como primer paso, se deben añadir a la máquina la lista de repositorios que permiten añadir las dependencias necesarias para instalar Docker.
Una vez añadidos a la lista de repositorios, con el gestor de paquetes de la máquina, se puede instalar Docker con los siguientes comandos:

```
sudo apt-get update -y
sudo apt-get install docker-ce docker-ce-cli
containerd.io -y
```

En este punto, en la máquina ya se tiene instalado la herramienta Docker, ahora el siguiente paso consiste en crear un fichero de configuración para el *daemon* de Docker, en el cual se especifican los registros de contenedores a los cuales la máquina va a querer conectarse para publicar o descargar imágenes. Para crear este fichero (en formato JSON), se ejecuta el siguiente comando:

```
cat <<EOF | sudo tee /etc/docker/daemon.json
{
        "exec-opts": ["native.cgroupdriver=systemd"],
        "log-driver": "json-file",
        "insecure-registries":["155.210.71.117:5000"],
        "log-opts": {
                "max-size": "100m"
        },
        "storage-driver": "overlay2"
}
```

Dentro del fichero, la línea que contiene la clave “insecure-registries” es la que contiene la lista de direcciones en las que existe un registro de contenedores. Para este caso, el registro de contenedores está situado en el nodo *master*.
En este punto se reinicia el *daemon* de Docker para actualizar los cambios:

```
sudo systemctl enable docker
sudo systemctl daemon-reload
sudo systemctl restart docker
```

3.2.4. Instalar herramientas *kubeadm*, *kubelet* y *kubectl*

Para seguir con la instalación, se instalan las herramientas que permiten la comunicación con la API de Kubernetes: *kubeadm*, *kubelet* y *kubectl* :

```
1 sudo apt-get update
2 sudo apt-get install -y apt-transport-https
  ca-certificates curl
3 sudo curl -fsSLo
  /usr/share/keyrings/kubernetes-archive-keyring.gpg
  https://packages.cloud.google.com/apt/doc/apt-key.gpg
4
5 echo "deb
  [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg]
  https://apt.kubernetes.io/ kubernetes-xenial main" | sudo
  tee /etc/apt/sources.list.d/kubernetes.list
```

Con estos comandos, se añade a la lista de repositorios de la máquina, los enlaces necesarios para poder localizar las herramientas que se quieren instalar. Una vez se han añadido, se debe actualizar la lista de repositorios e instalar las herramientas:

```
1 sudo apt-get update -y
2 sudo apt-get install -y kubelet kubeadm kubectl
```

A modo de instalador, todos estos comandos se han añadido a un script en Bash que permite inicializar todos los nodos con la ejecución de un solo comando.

3.3. Inicializar nodo *master*

En esta sección, se va a detallar la instalación de los paquetes necesarios para inicializar el nodo *master*.
Para empezar, se han declarado dos variables de entorno que indican el nombre *host* y la dirección IP para facilitar la sintaxis de los comandos que se van a ejecutar:

```
1 IPADDR=$(ip -o -4 addr list eth0 | awk '{print $4}' | cut
  -d '/' -f1)
2 NODENAME=$(hostname -s)
```

Una vez se han declarado las variables, a través de la herramienta *kubeadm* y el comando *init* se inicializa el nodo:

```
1 sudo kubeadm init --apiserver-advertise-address=$IPADDR
  --apiserver-cert-extra-sans=$IPADDR --node-name $NODENAME
  --ignore-preflight-errors Swap
```

En este punto de la instalación, la salida de este comando genera un fichero “config”, el cual permite trabajar con el *cluster* desde cualquier máquina que

pueda acceder a la dirección del nodo y tenga instalada la herramienta *kubectl*. El comando también mostrará en su salida por pantalla, un *token* de acceso, el cual es necesario para que todos los nodos *worker* se unan al *cluster*.

Para poder trabajar con el fichero "config", se debe copiar a la carpeta "$HOME/.kube", para que la herramienta *kubectl* pueda localizar el fichero sin necesidad de indicarlo como un parámetro, ejecutando los siguientes comandos:

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Tras ejecutar estos comandos, el nodo *master* está completamente iniciado.

3.4. Inicializar nodos *worker*

Para inicializar cada nodo *worker*, utilizando el *token* que el *master* genera al iniciarse, a través del comando *kubeadm*, se puede inicializar el nodo *worker*. Para obtener de nuevo el *token*, se ejecuta dentro del nodo *master*, el siguiente comando:

```
kubeadm token create --print-join-command
```

Generando una salida similar a esta:

Figura 4: Token generado por el nodo *master* que permite unirse a todos los *worker*

Ejecutando el comando que se indica en la figura 4, el *worker* se unirá al *cluster*. Para comprobar el estado de los distintos nodos del *cluster*, se ejecuta el siguiente comando

```
kubectl get nodes
```

Generando la siguiente salida:

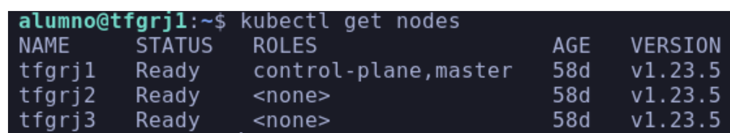


Figura 5: Captura que muestra todos los nodos que pertenecen al *cluster*

3.5. Instalación "*plugins*" en el *cluster*

Hasta el momento, se tiene un *cluster* funcional en el que se pueden desplegar aplicaciones, sin embargo, no se tienen instaladas algunas funciones importantes como una política de segmentación de la red, o un servidor de métricas.
Como solución a este problema, se instalan manualmente.

3.5.1. Proyecto Calico

El proyecto Calico es un motor de políticas de red para Kubernetes, las políticas de red de Calico permiten implementar segmentación de red y aislamiento de usuarios, lo que permite que miembros de diferentes empresas puedan acceder a un *cluster* de forma autónoma y sin ver la información del resto de usuarios que trabajen sobre el mismo *cluster*. De igual manera sirve si se pretende tener varios entornos en el mismo *cluster* como por ejemplo un entorno de desarrollo y otro de producción [26].

Para instalar el *plugin*, se despliega el fichero proporcionado por la propia organización:

```
1   kubectl apply -f
    https://docs.projectcalico.org/manifests/calico.yaml
```

3.5.2. Instalación servidor de métricas

El servidor de métricas permite obtener información relacionada con el estado de los nodos y sus recursos (CPU, memoria...), para poder monitorizar y controlar tanto el estado como la salud del *cluster*, es conveniente tener instalado un servidor de métricas.

Para este TFG, se va a instalar el servidor de métricas oficial que proporciona Kubernetes [27], desplegando los componentes a través de un manifiesto que proporciona Kubernetes:

```
1   kubectl apply -f
    https://raw.githubusercontent.com/scriptcamp/kubeadm-scripts/main/
    manifests/metrics-server.yaml
```

Una vez instalado, se puede comprobar el funcionamiento a través de *kubectl*:

```
NAME     CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
tfgrj1   286m         14%    2130Mi          56%
tfgrj2   194m         9%     2259Mi          59%
tfgrj3   246m         12%    2914Mi          76%
```

Figura 6: Salida generada por el comando kubectl que muestra métricas sobre los nodos

4. Demostrador tecnológico

Como complemento a la instalación del *cluster* Kubernetes (capítulo 3), se ha implementado un prototipo software y una maqueta que simulan una situación real a pequeña escala de un escenario real.

En esta sección se van a detallar tanto la estructura que va a adoptar la maqueta, como la estructura del prototipo software. Se va a precisar la metodología adoptada, así como también las tecnologías y dispositivos utilizados a lo largo de la implementación del proyecto.

Para la fase de análisis se han elaborado unos diagramas que facilitan la comprensión de la estructura del sistema.

Para el prototipo software se va a implementar una aplicación basada en microservicios que utilice la infraesutrctura creada en el capítulo 3.

4.1. Análisis

4.1.1. Diagramas

El primer diagrama incluye todos los elementos que componen el sistema:

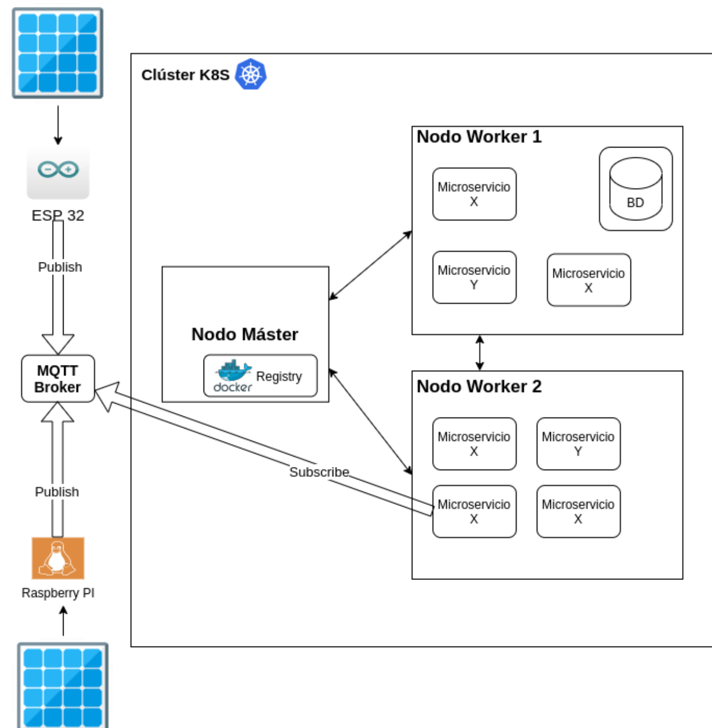


Figura 7: Diagrama de la infraestructura que se va a desplegar para el demostrador tecnológico

Algunos elementos importantes a destacar son:

- Para comenzar, el elemento principal que forma el sistema es el *cluster*, el encargado de comunicar los nodos y permitir el despliegue de aplicaciones.
- Fuera del *cluster*, se tiene 2 dispositivos que recogen información de un panel solar, cada uno de ellos (ESP32 y Raspberry PI) representan un parque de generación de energía renovable asociado a cada nodo.
- Estos dispositivos recogen los valores de intensidad y tensión de la placa solar y la envían a través de MQTT a un *broker* situado fuera del *cluster*. Cada uno de los dispositivos realiza la misma tarea, medir la tensión e intensidad del panel solar y publicar la información en el *broker* MQTT.
- Dentro del nodo *master* se tiene un registro de contenedores (Docker registry) que aloja las imágenes que se utilizan para el prototipo software.
- Dentro de uno de los nodos *worker*, se tiene un volumen que persiste la información en caso de que mueran los contenedores asociados al volumen.

En cuanto al prototipo software que se ha desarrollado, se ha implementado un diagrama que incluye los microservicios que conforman el sistema:

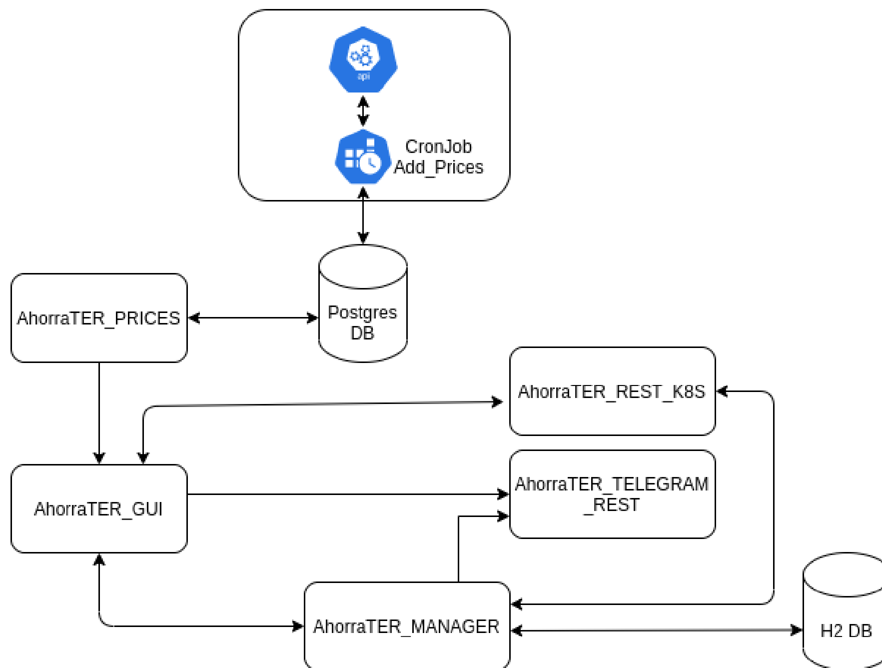


Figura 8: Diagrama de la estructura de la aplicación

Los distintos microservicios que forman el sistema son son:

- AhorraTER_GUI: Este microservicio es el encargado de comunicar al usuario con el sistema, en esta interfaz se van a poder consultar, por un lado los distintos precios de los combustibles en las estaciones de servicio españolas, y por otro lado, el estado de los nodos del *cluster* (métricas de CPU y memoria).
  Además, se puede obtener en tiempo real la los valores de energía que están generando las placas solares (intensidad y tensión).
- AhorraTER_MANAGER: Este microservicio se encarga de controlar la comunicación MQTT del sistema, suscribirse a los *topic* correspondientes y tratar los mensajes recibidos que contengan información sobre la intensidad y tensión de las placas solares, almacenando en una BBDD toda la información recogida.
  En base a la información que recolecta a través de MQTT, AhorraTER_MANAGER es el encargado de decidir en qué momento se debe trasladar la carga de trabajo de un nodo a otro.
- AhorraTER_REST_K8S: Este microservicio se encarga de desplegar una API REST que permite interactúar con el *cluster* Kubernetes, es capaz de realizar operaciones de borrado de pods o etiquetar/desetiquetar nodos.
- AhorraTER_PRICES: Este microservicio despliega una API REST que permite interactuar con la BBDD que contiene la información de los comubustibles y sus estaciones de servicio, así como toda la información relacionada con los mismos.
  Es el encargado de proporcionar la información necesaria sobre la BBDD al resto de microservicios, de tal forma que no necesiten conocer la ubicación de la misma.
- AhorraTER_TELEGRAM_REST: Este microservicio se encarga de ofrecer una API REST que permita el envío de mensajes de texto por Telegram. Ofrece la posibilidad de exportar los datos relacionados con las métricas del *cluster* y enviarlos a modo de gráfica.

Como se observa en la figura 8, la aplicación necesita almacenamiento persistente para mantener los datos relacionados con el combustible y sus precios en las estaciones de servicio españolas.
El diagrama E/R de esta BBDD se corresponde con el de la siguiente figura:

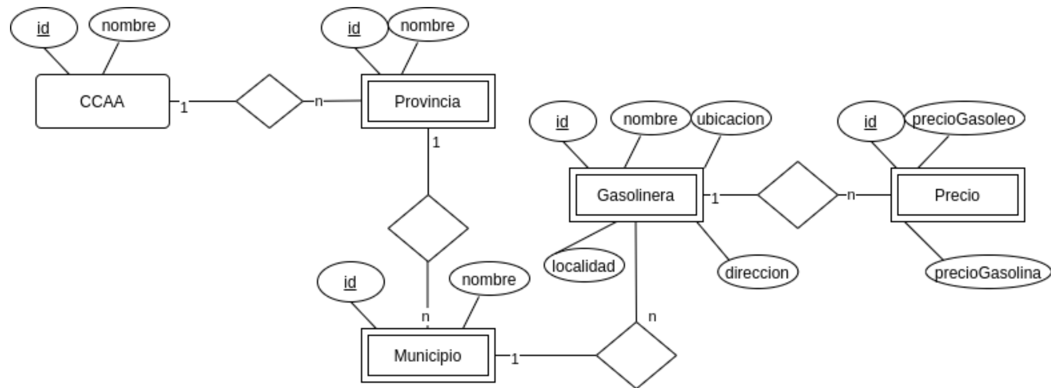


Figura 9: Diagrama Entidad-Relación de la BBDD sobre combustibles

Para la recolección de los datos relacionados con la intensidad y tensión de los circuitos eléctricos de las placas solares, se tendrá una única entidad que almacene la fecha y los valores obtenidos.

El siguiente diagrama se corresponde con un diagrama de actividades asociado a la acción de balancear la carga en función de los valores que recogen los dispositivos ESP32 y Raspberry PI respectivamente. Esta operación se realizará cada dos segundos *in aeternum*.

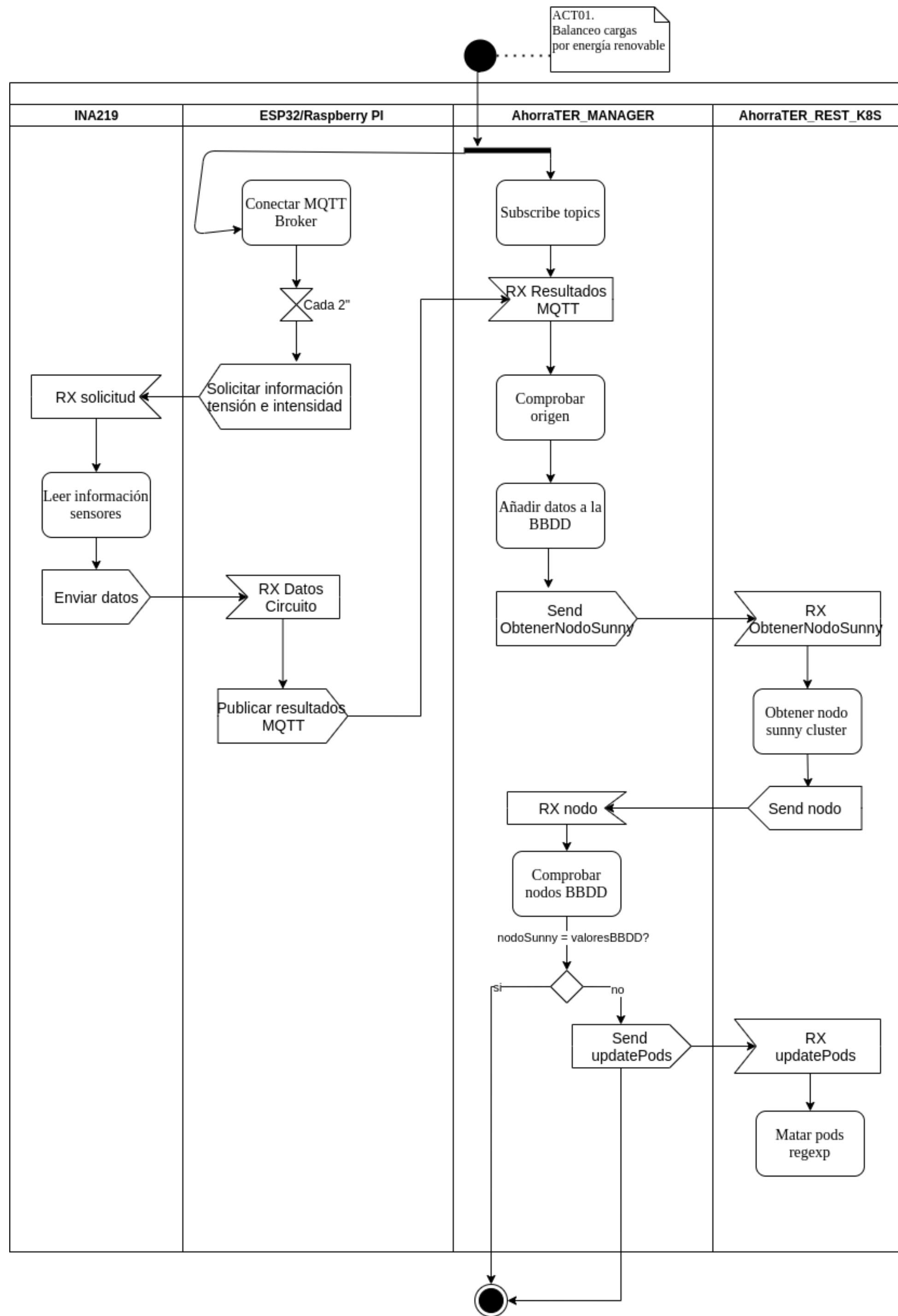


Figura 10: Diagrama de actividades del balanceo de cargas en función de la energía renovable.

4.1.2. Publicación APIs en el sistema

Durante la primera etapa del proyecto, una vez definidos los distintos microservicios, se ha realizado una breve especificación de los métodos necesarios para implementar las distintas API REST.

		AhorraTER REST Path	
AhorraTER_PRICES			
.../rest/locate/	GET	ccaa/	Devuelve una lista de todas las CCAA de España
	GET	ccaa/{idCCAA}	Devuelve una lista de todas las provincias pertenecientes a la CA que se pasa como parámetro
	GET	provincia/	Devuelve una lista de todas las provincias de España
	GET	provincia/{idProv}	Devuelve una lista de todas los municipios pertenecientes a la provincia que se pasa como parámetro
	GET	municipio/	Devuelve todos los municipios de España
	GET	municipio/{idMuni}	Devuelve todas las estaciones de servicio pertenecientes al municipio que se pasa como parámetro
.../rest/price/	GET	gasoleoa/{idEESS}	Devuelve una lista de precios del gasoleo en la estación de servicio pasada como parámetro
	GET	gasoleoa/{idEESS}/{fechaInicio}/{fechaFin}	Devuelve una lista de precios del gasoleo en la estación de servicio pasada como parámetro entre 2 fechas indicadas por parámetro.
	GET	gasoleo95/{idEESS}	Devuelve una lista de precios de la gasolina en la estación de servicio pasada como parámetro
AhorraTER_REST_K8S			
.../rest/	GET	listNodes	Devuelve la lista de nodos worker que forman el clúster
	GET	getSunnyNode	Devuelve el nodo del clúster que tiene la mayor energía solar disponible.
	GET	getNodePod	Devuelve una lista de pods asociados al nodo en el que están desplegados
	GET	getMetricsNodes	Devuelve una lista de nodos con sus valores de CPU y memoria
	POST	updatePodsSun/{nodo}	Despliega los pods en el nodo que tenga mayor cantidad de energía renovable, el nodo se pasa por parámetro.
	POST	unlabelNodes	Desetiqueta todos los nodos de la etiqueta "sunny"
	POST	labelNode/{nodo}	Etiqueta al nodo pasado por parámetro con la etiqueta "sunny"
AhorraTER_MANAGER			
.../rest/metrics	GET	getValuesPlacas	Devuelve la lista de nodos junto a la intensidad y tensión actuales
.../rest/metrics/media	GET	getMediaValuesPlacas	Devuelve la lista de nodos junto a la intensidad y tensión medios
AhorraTER_TELEGRAM_REST			
.../rest/sendMessage	POST	send_message(String json)	Envía el mensaje pasado como parámetro en el body de la petición
.../rest/exportData	POST	export_data(String json)	Convierte a gráfico los datos enviados como parámetro en el body

Figura 11: Definición de los métodos de los microservicios AhorraTER_PRICES y AhorraTER_REST_K8S

El formato de envío de la información se realiza en todos los microservicios en formato JSON.

4.2. Desarrollo/Puesta en marcha

En esta sección se explica la metodología y procedimiento llevado a cabo para la puesta en marcha del prototipo software, explicando de cada uno de los microservicios, su implementación, construcción de imagen y despliegue en el *cluster*.

4.2.1. Registro de contenedores

En este punto se describe como instalar un registro de contenedores para poder almacenar las imágenes Docker que se vayan construyendo de cada uno de los microservicios en un registro privado.

El registro de contenedores, en este prototipo software, se crea dentro del nodo *master* del *cluster*, sin embargo, podría estar ubicado en cualquier otra máquina que fuese accesible desde todos los nodos del *cluster*.

Creación del registro

El primer paso para construir un registro de contenedores es tener Docker instalado en la máquina que va a contenerlo.

A continución, es necesario obtener del repositorio de imágenes de Docker [40], la imagen "registry" que permite crear un registro de contenedores.

Una vez se tiene de forma local, la imagen Docker, será necesario ejecutar el comando docker correspondiente, indicando el puerto sobre el que se quiere trabajar y, el directorio en el que se almacenarán las distintas imágenes.
El comando que se ejecuta es el siguiente:

```
1   docker run -d --name nombreRegistro -p 5000:5000 -v
    /rutaRegistro:/var/lib/registry registry
```

De este modo se habrá creado en la ruta indicada, con el puerto indicado, el registro de contenedores.

Configuración de una máquina para su acceso

Una vez creado el registro de contenedores, por defecto en las máquinas que quieran consumir o utilizar dicho registro, deben modificar el fichero "daemon.json" situado en la carpeta "/etc/docker", añadiendo al fichero la siguiente línea:

```
1   "insecure-registries":["155.210.71.117:5000"]
```

Una vez modificado el fichero es necesario reiniciar el servicio de Docker para poder actualizar los cambios, tras la actualización, el cliente ya puede interactuar con el registro creado.

Para publicar o obtener las imágenes es necesario utilizar los comandos *docker push* y *docker pull* (del mismo modo que una imagen obtenida del repositorio de Docker) pero indicando la dirección y puerto del registro a utilizar. Un ejemplo sería:

```
1   docker pull 155.210.71.117:5000/imagen-test
```

4.2.2. Base de Datos

Como se ha comentado anteriormente, para la implementación del prototipo software, se requerían varias BBDD. En este apartado se va a tratar de explicar la puesta en marcha de la BBDD relacionada con el precio del combustible.

Se ha escogido PostgreSQL como sistema gestor de bases de datos por 2 motivos principalmente: La experiencia previa con el sistema gestor y conocimiento sobre su funcionamiento y la posibilidad de replicación de la información. PostgreSQL permite, si se hacen ciertas modificaciones replicar la información de la BBDD, lo que sería interesante para un trabajo futuro (capítulo 5).

En cuanto a las tablas, siguiendo el diagrama ER de la figura 9, a través del lenguaje SQL se ha implementado un script que crea las distintas tablas que dan soporte a la información que se quiere almacenar, con las restricciones que se indican en el diagrama.

Construcción de la imagen Docker y subida al registro de contenedores

Después de tener un script SQL que crea las tablas, es necesario construir una imagen de Docker que contenga dicho script y lo ejecute en la fase posterior al arranque, creando una nueva imagen que deberá ser almacenada en el registro de contenedores (capítulo 4.2.1).

Como imagen base para la construcción del *Dockerfile*, se utilizará la imagen oficial de PostgreSQL del repositorio de Docker [40]. A la imagen se le añaden dos ficheros, por un lado el script SQL que crea las tablas, y por otro lado un script Bash que permita crear la BBDD y un rol para poder acceder a las nuevas tablas.
Una vez construida la imagen, se sube al registro de contenedores.

Tratamiento en Kubernetes e inserción de los datos

Una vez se ha construido la imagen de la BBDD, es necesario desplegarla dentro del *cluster*. Para desplegar la BBDD de forma correcta se han necesitado varios elementos de Kubernetes, que se incluyen en un fichero con extensión ".yaml":

- Secret: Que incluye la información relacionada con usuario y contraseña de la BBDD de Postgres.
- ConfigMap: Que incluye la información relacionada con el nombre de la BBDD y el puerto de conexión.
- Service: Uno de los objetivos de Kubernetes es el aislamiento de pods, por ello es conveniente, cuando se quieren comunicar distintos pods, crear un servicio que permita comunicar pods sin necesidad de conocer su ubicación ni su dirección de red. Por tanto, para poder acceder dentro del *cluster* a la BBDD, se crea un servicio.

- StatefulSet: En este caso se utiliza para mantener el nombre de los pods una vez mueren. Otros elementos como el Deployment cambian su nombre cada vez que un pod muere, en este caso se deben persistir para que los volúmenes puedan funcionar correctamente.

- Job: En este caso se ha utilizado para la inserción de los datos. Una vez se ha creado el contenedor con la BBDD se lanza un Job que permite conectarse a la BBDD e insertar los datos.
  La forma de obtener los datos es a través de la API REST del Gobierno de España [41]. Una vez se han insertado los datos, el contenedor muere y se elimina de la lista de pods.

- Volumen: Para persistir los datos se ha utilizado el componente “Volume” de Kubernetes. Aunque hay distintos tipos de volúmenes como se explica en el capítulo 2.2.4, para esta implementación se ha creado un volumen de tipo “HostPath” para persistir la información en uno de los nodos.
  Una posible mejora sería utilizar un volumen en la nube (capítulo 5) y tener replicación de la BBDD.

  Para montar el volumen, se ha creado en primer lugar un PV, que permite indicar el tipo de volumen que se quiere crear (hostPath) y el tipo de acceso que se quiere, lectura y escritura una vez o varias a la vez.
  Tras crear el PV, se ha creado un PVC que permite reclamar el volumen creado, de modo que cuando el PVC esté creado y asociado al PV, se puede montar el volumen referenciando al objeto PVC creado.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: vol-db1
  namespace: app-combustible
  labels:
    tipo: local1
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/home/alumno/database"
```

(a) Creación de un PV en un manifiesto YAML

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-vol-db1
  namespace: app-combustible
spec:
  storageClassName: manual #
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  selector:
    matchLabels:
      tipo: local1
```

(b) Creación de un PVC en un manifiesto YAML

Figura 12: Implementación del volumen para el servicio de BBDD.

4.2.3. AhorraTER_PRICES

Este microservicio es el encargado de implementar una API REST con los métodos indicados en el capítulo 4.1.2 para trabajar contra la BBDD creada en el capítulo 4.2.2.

El formato utilizado para la implementación de esta API, como para el resto, es el formato JSON. Como servidor web se ha utilizado Payara. [42].

Para implementar este microservicio, se ha necesitado crear en primer lugar un *pool* de conexiones a la BBDD. Para poder conectarse, es necesario tener la información del servicio de la misma, por ello es necesario añadir tanto en el fichero Dockerfile como en el manifiesto de Kubernetes, las variables de entorno que indiquen los valores del puerto, nombre del servicio y namespace en el que se ubica.

Con la conexión a la BBDD realizada, se han implementado los métodos que obtienen de la BBDD la información necesaria que se indica en la documentación de la API: obtener las comunidades autónomas, provincias, municipios, estaciones de servicio y el precio de los carbuantes.

Para implementar la API REST, se crea un *Web Service* que publica los resultados obtenidos de la BBDD, se muestra un ejemplo del resultado obtenido en formato JSON.


```
{"1":"Andalucia","2":"Aragón","3":"Asturias","4":"Baleares","5":"Canarias","6":"Cantabria",
"7":"Castilla la Mancha","8":"Castilla y León","9":"Cataluña","10":"Comunidad
Valenciana","11":"Extremadura","12":"Galicia","13":"Madrid","14":"Murcia","15":"Navarra","1
6":"País Vasco","17":"Rioja (La)","18":"Ceuta","19":"Melilla"}
```


Figura 13: Resultado obtenido en la operación GET sobre el Path "/rest/locate/ccaa"

Construcción de la imagen Docker y publicación en el registro de contenedores

Para la construcción de esta imagen, en primer lugar, se utiliza la imagen de "payara/micro", puesto que no es necesario obtener la imagen completa, ya que su tamaño es elevado y la imagen micro contiene las operaciones que se necesitan. Sin embargo, para crear el *pool* de conexiones y el recurso para acceder al *pool*, deben crearse a través de la línea de comandos de Payara.

La imagen de Payara ofrece la posibilidad de ejecutar comandos tras la creación del contenedor, por ello, se declara un fichero con los comandos de payara necesarios para crear el *pool* de conexiones y el recurso. Cuando el contenedor se lanza, se crean los recursos tras el *boot* del contenedor, y la aplicación funciona correctamente.

Para indicar que se deben ejecutar dichos comandos tras el *boot* del contenedor, se puede hacer a través de la instrucción "CMD" de Docker, o, a través de la operación "command" en el manifiesto Kubernetes.

Para esta implementación, se ha optado por ejecutarlo desde Kubernetes, para tener la opción de modificar las variables de entorno desde el manifiesto y poder trabajar con la misma imagen y diferentes variables.

Al tratarse de una BBDD PostgreSQL, la imagen de Payara, no cuenta con el driver correspondiente para la creación de los recursos, por lo que se añade al construir el contenedor, para que la pueda encontrar a la hora de lanzar la imagen.

```
FROM payara/micro

COPY AhorraTER_PRICES.war $DEPLOY_DIR

ADD postgresql-42.3.5.jar /opt/payara/appserver/glassfish/lib/
ADD create-pool.txt /opt/payara/
```

Figura 14: Archivo Dockerfile para la construcción de la imagen Docker del microservicio AhorraTER_PRICES

Una vez se tiene la imagen lista, a través de un script (capítulo 4.2.8), se sube al registro de contenedores.

Tratamiento en Kubernetes

Este microservicio va a servir como prueba del algoritmo de balanceo de cargas, es decir, será el microservicio que se irá desplazando de un nodo a otro en función de la energía renovable en ese momento.
Los distintos elementos de Kubernetes que se han necesitado son los siguientes:

- Secret y ConfigMap: Del mismo modo que en el microservicio de la BBDD, se ha necesitado la información de un secreto que proporcione las variables de usuario y contraseña de la BBDD, y del ConfigMap para obtener el nombre y puerto de conexión.

- Service: Al tratarse de un microservicio que se va a comunicar con el resto de elementos del sistema, es necesario crear un servicio que permita el acceso únicamente desde dentro del *cluster*, por tanto se crea un servicio de tipo ClusterIP.

- Deployment: Para controlar los pods que se crean, se crea un deployment que controle las réplicas de los distintos pods.
  Es en la declaración del Deployment, donde se han añadido las operaciones necesarias para que el pod se desplace de nodo.
  Kubernetes permite etiquetar un nodo a través de *labels*. Dentro de un manifiesto cuando se va a desplegar un pod, existe el atributo “nodeAffinity” que permite asociar una mayor afinidad a un nodo que cumpla cierta condición, en este caso, que contenga la etiqueta “sunny=1”.
  Con esta implementación, cada vez que se vaya a desplegar un pod controlado por este Deployment, comprobará que nodo contiene la etiqueta

"sunny=1" y, siempre que los recursos del nodo sean suficientes, desplegará el pod en ese nodo.

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: sunny
            operator: In
            values:
            - "1"
```

Figura 15: Fragmento de la declaración del Deployment del servicio PRICES que especifica la afinidad de un nodo en función de su *label*.

Como se puede observar en la figura anterior, en el manifiesto se especifica que los nodos que contengan la etiqueta "sunny=1", tendrán mayor afinidad para desplegar los pods controlados por el nuevo Deployment. La forma de etiquetar y desetiquetar los nodos, se explica en los microservicios AhorraTER_MANAGER y AhorraTER_REST_K8S (capítulos 4.2.6 y 4.2.5).

4.2.4. AhorraTER_GUI

Este microservicio se ha implementado para mostrar la información con la que se trabaja en el resto de microservicios, así como de toda la infraestructura desplegada.

AhorraTER_GUI permite filtrar los precios de los combustibles, en función de su estación de servicio y origen, los cuales provienen del servicio AhorraTER_PRICES. También muestra las métricas (CPU y memoria) de los nodos que componen el *cluster*, la información en este caso proviene del servicio AhorraTER_REST_K8S.

Se muestran también los valores de intensidad y tensión en tiempo real, que se obtienen del microservicio AhorraTER_MANAGER.
Finalmente, se permite exportar todos estos datos a una gráfica en Telegram a través de un botón en la interfaz de usuario.

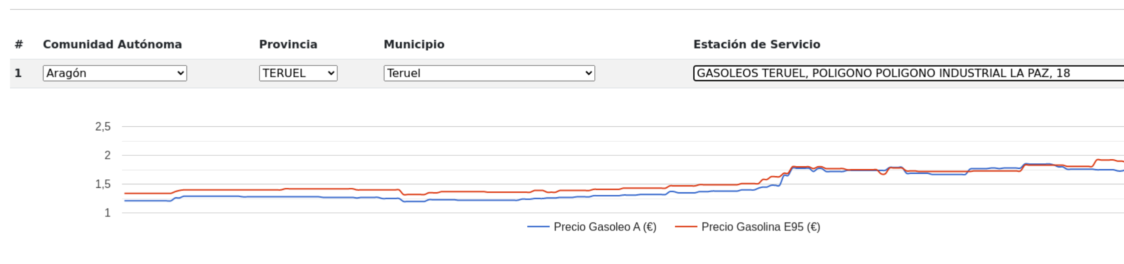
Este microservicio implementa un manejador de conexiones a través de un WebSocket, lo que permite mandar en tiempo real información desde el *backend* a la interfaz de usuario.

La forma de obtener la información es la siguiente:

- Para obtener las métricas del servicio AhorraTER_REST_K8S, se ha implementado un timer, que se ejecuta cada dos segundos y hace una petición al método GET "getMetricsNode" que devuelve el estado de cada nodo del *cluster*.

- Para obtener los precios de los combustibles, conforme el usuario va seleccionando los distintos valores que se muestran por la interfaz relacionados con la ubicación (CCAA, provincia, municipio y estación de servicio) se van realizando peticiones a la API del microservicio PRICES y va devolviendo toda la información actualizada, una vez seleccionada la estación de servicio, se dibuja una gráfica comparativa entre dos combustibles.
- Para obtener los valores de la intensidad y tensión que están circulando por las placas solares, del mismo modo que para obtener las métricas, se debe realizar una petición a la API REST, pero en este caso del microservicio MANAGER a través de un timer, que devuelve la información actualizada y la muestra.

Para exportar la información a Telegram, basta con realizar una llamada a la operación "exportData" de la API REST que implementa el microservicio AhorraTER_TELEGRAM_REST.



Nodo con mayor energía renovable: tfgrj2

Figura 16: Captura de pantalla de la GUI

Construcción de la imagen Docker y publicación en el registro de contenedores

Para la construcción de esta imagen, solamente es necesario incluir el archivo ".war" en la imagen de "Payara/micro" en el contenedor.

Una vez construida la imagen, se sube al registro de contenedores, y a través de un script, se lanza y se despliega dentro del *cluster*.

Tratamiento en Kubernetes

La configuración de Kubernetes que ha necesitado este microservicio es la siguiente:

- En primer lugar un "ConfigMap" que incluye toda la información relacionada con los nombres de los servicios desplegados en el *cluster*, así como los puertos y el espacio de trabajo en el que están situados.
- Deployment, al que se le pasan los valores creados del "ConfigMap" para que se tengan como variables de entorno del sistema.
- Service, en este caso, si que se quiere que la interfaz de usuario sea accesible desde fuera del *cluster*, por ello el servicio que se crea es de tipo NodePort, que permite mapear un puerto en el rango 30000-32767 al puerto del pod que se indique, en este caso el 8080.
  La dirección para acceder a la interfaz es: 155.210.71.117:32465/AhorraTER_GUI

4.2.5. AhorraTER_MANAGER

Este microservicio se ha implementado para poder manejar, junto al servicio AhorraTER_REST_K8S (capítulo 4.2.6), las propiedades de los nodos del *cluster*, y modificarlas en función de los recursos energéticos de los que disponga cada nodo en todo momento.

Como ya se ha comentado en los capítulos anteriores, la forma que se ha utilizado para simular la energía renovable asociada a cada nodo es a través de dispositivos de bajo consumo, en este caso ESP32 y Raspberry PI, los cuales tienen asociada una pequeña placa solar de la que obtienen sus valores de intensidad y tensión a través del sensor INA 219. La información recogida del sensor se envía a través del protocolo MQTT (capítulo 2.3).

Este microservicio es el encargado de leer los datos recibidos y almacenarlos en una BBDD. Para la implementación llevada a cabo, se ha creado una BBDD dentro del propio servicio, debido a que para este escenario la información se está actualizando en todo momento en la BBDD, además de estar constantemente enviando información a la interfaz de usuario. Para tener una velocidad de acceso mayor, se ha considerado una mejor opción tener la información dentro del microservicio.

Para la implementación de la BBDD en este caso, se ha utilizado H2 como sistema administrador, ya que el servidor web Payara incluye los drivers en la propia imagen, y se ha utilizado JPA para el tratamiento de la información almacenada.

Se ha creado una entidad "HistoricoLuz" que contiene los datos que provienen de la información enviada por MQTT.

Para manejar la información que se envía a través de los dispositivos por MQTT, se ha implementado un manejador de conexiones "MQTT Manager" que cuenta con un *listener* que recibe los mensajes de manera asíncrona, lo que permite controlar la distinta información que vaya recibiendo y almacenándola en la BBDD.

Una vez tiene la información actualizada, la utiliza para decidir si un pod debe ser desplegado en otro nodo o no, en función de los últimos 10 valores obtenidos de la BBDD.

Construcción de la imagen Docker y publicación en el registro de contenedores

Para la construcción de la imagen Docker, como se ha comentado en la sección anterior, se tiene una BBDD que permite almacenar el histórico de los datos que obtiene de los dispositivos vía MQTT. Para ello, es necesario crear un *pool* de conexiones a la BBDD. A través de la opción "–postbootcommandfile" que ofrece Payara, se crea el *pool* de conexiones tras arrancar el contenedor.
Una vez se ha creado el fichero Dockerfile incluyendo el archivo .war, se sube al registro de contenedores a través de un script en bash (capítulo 4.2.8).

Tratamiento en Kubernetes

Los componentes que se han necesitado desplegar para poner en marcha este microservicio, han sido los siguientes:

- Deployment: Las variables de entorno que se necesitan para poder localizar el resto de microservicios, se obtienen del ConfigMap que se ha creado para el microservicio de la interfaz de usuario.
- Service: Es un servicio interno del *cluster* que no debe ser accesible desde fuera, por tanto es un servicio de tipo ClusterIP.

4.2.6. AhorraTER_REST_K8S

Este microservicio se ha implementado para poder interactuar con el *cluster* a través de una API REST, y realizar algunas de las operaciones necesarias para poder distribuir la carga de trabajo en función de la energía renovable. Este microservicio se ha implementado en el lenguaje Python.

Los métodos que implementa esta API, se han especificado en la sección 4.1.2. Como ya se ha comentado a lo largo del capítulo 4, uno de los objetivos del demostrador es poder simular el balanceo de cargas en función de la cantidad de energía renovable que tenga cada nodo.

En la figura 15, se indica cómo definir qué nodo va a tener un valor de afinidad más alto, y en base a ese criterio desplegar los pods controlados por el Deployment en dicho nodo.
Los métodos de esta API REST, permiten:

- Desetiquetar todos los nodos (POST): Cada vez que se vaya a etiquetar un nodo, se desetiquetan todos y solamente se etiqueta uno, de modo que siempre uno de los nodos tenga la etiqueta puesta, se realiza una operación POST sobre la URL "/rest/unlabelNodes".

- Obtener el nodo con mayor energía renovable (GET): Permite comprobar,si es necesario, que el *cluster* tenga un nodo con la etiqueta puesta, en caso de no tener un nodo etiquetado, se deberá etiquetar (durante el arranque del servicio AhorraTER_MANAGER). Se realiza una operación GET sobre la URL "/rest/getSunnyNode"

- Etiquetar un nodo (POST): A través del microservicio MANAGER, se llamará a esta operación cada vez que haya un cambio en los valores de intensidad y tensión obtenidos por las placas solares, se realiza una operación POST sobre la URL "/rest/updatePodsSun/nodo".

Para realizar las operaciones dentro del *cluster*, se han realizado llamadas al sistema desde Python. Una vez se ha implementado todo el código, se exporta el entorno con el que se ha trabajado para poder incluirlo en la construcción de la imagen Docker, que se explica en el siguiente capítulo.

Para hacer llamadas a la API REST dentro del *cluster*, es necesario tener el nombre del servicio al que pertenece y el namespace, con esos dos valores se puede crear una URL que permite acceder a cualquiera de los pods a los que apunta el servicio.

Este microservicio, junto al microservicio MANAGER, permiten dar soporte a toda la parte del algoritmo de detección solar.

Construcción de la imagen Docker y publicación en el registro de contenedores

En relación a la construcción de la imagen, en primer lugar la imagen utilizada es "Alpine", una distribución de Linux muy ligera pero funcional. Sobre esta imagen se ha instalado "Miniconda" [43] una herramienta que permite crear entornos virtuales de Python, y que permite importar entornos creados previamente, en este caso se importa a la imagen Docker el fichero con todas las dependencias de la aplicación desarrollada en Python.

Para la instalación de "Miniconda", se ha implementado un script en bash que instala todas las dependencias necesarias de la aplicación e inicializa el servicio con el entorno importado.

Además, para poder trabajar contra el *cluster*, es necesario tener la herramienta kubectl instalada, por ello, a través de una instrucción RUN, se instala la herramienta y se añade al PATH de la máquina para que cuando se realicen las llamadas al sistema, pueda encontrar la ruta del archivo.

Finalmente, se añade el fichero "config" del *cluster* y se indica como variable de entorno la ruta, ya que kubectl por defecto busca el fichero en el directorio " /.kube".

Una vez realizados estos pasos, con el comando CMD se indica la ruta del fichero a ejecutar y la ruta de la ubicación del entorno creado con las dependencias instaladas y se construye la imagen. En este punto la imagen se sube a través de un script (explicación capítulo 4.2.8) al registro de contenedores

Tratamiento en Kubernetes

Para este microservicio, los componentes Kubernetes que se han necesitado para su despliegue son los siguientes:

- Deployment: Para controlar el número de réplicas de la aplicación.
- Service: Esta API REST solo debe ser accesible por los distintos microservicios de dentro del *cluster*, por eso se ha creado un servicio ClusterIP que permita que el resto de microservicios del *cluster* puedan acceder a el.

4.2.7. AhorraTER_TELEGRAM_REST

Este microservicio implementa una API REST que permite enviar mensajes y gráficas a través de Telegram. Está desarrollado en Python y contiene únicamente dos métodos que se especifican en el capítulo 4.1.2, cuyos objetivos son:

1. Cuando la intensidad que recoja el servicio AhorraTER_MANAGER por un nodo sea mayor que la del otro, notificar a través de un mensaje en Telegram, que se va a cambiar la carga de trabajo de un nodo a otro.

2. Cuando el usuario, desde el servicio AhorraTER_MANAGER interactúe con el botón "Exportar datos a Telegram", obtener los valores actuales, a modo de instantánea del sistema, de las métricas del servidor y de los últimos valores obtenidos de intensidad y tensión de cada nodo, y enviarlos a través de una imagen por Telegram.

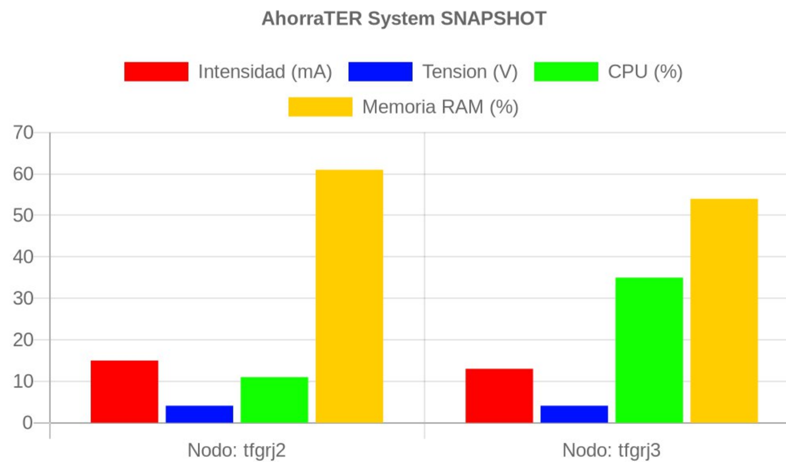


Figura 17: Mensaje enviado por el bot de Telegram.

Para esta implementación se ha utilizado la librería que proporciona Python para la implementación de bots en Telegram.

Construcción de la imagen Docker y publicación en el registro de contenedores

Una vez se ha implementado la aplicación, para construir la imagen Docker, se ha seguido el mismo procedimiento que en el microservicio AhorraTER_REST_K8S (capítulo 4.2.6).
Se ha utilizado la imagen "Alpine" como base para empezar a instalar paquetes. Posteriormente se han instalado la imagen de "Miniconda" a través de un script en Bash, y una vez instalado se ha importado el entorno Python y el fichero que contiene el código para poder ejecutar la aplicación.
Una vez se ha construido la imagen, se sube al registro de contenedores con el comando *push* de Docker.

Tratamiento en Kubernetes

Para este microservicio, se han necesitado desplegar dos componentes de Kubernetes:

- Deployment: Para controlar las réplicas que se tiene de la aplicación, se creado con 1 réplica dentro del *cluster*.
- Service: De tipo ClusterIP ya que solo debe ser accesible desde dentro del *cluster*.

4.2.8. Automatización de la puesta en marcha a través de scripts

A lo largo de la implementación es necesario construir varias imágenes Docker, lanzar distintos manifiestos de Kubernetes para los distintos microservicios y componentes. Por ello, se han implementado una serie de scripts en Bash que permiten automatizar el proceso de despliegue de los componentes.

Constantemente es necesario actualizar las imágenes del registro de contenedores y eliminar las que se están ejecutando de forma local en los distintos nodos.
Para solucionar este problema, el script, en primer lugar elimina todos los contenedores que tengan la imagen que se quiere actualizar. Una vez eliminados, a través de una conexión SSH (acceso con claves públicas), se borra la imagen local que se tiene descargada del registro de contenedores y se publica la nueva imagen actualizada.
Al terminar, se vuelve a lanzar el manifiesto que contiene los componentes a desplegar.

De esta forma se automatiza el proceso de borrado de los pods del *cluster*, construcción de la nueva imagen y despliegue de la misma en el *cluster*.

Como posible trabajo futuro, sería interesante poder hacer *rollout* y *rollback* de las distintas versiones de cada microservicio. Estas operaciones consisten en ir actualizando las imágenes a partir del manifiesto creado manteniendo un porcentaje de las imágenes antiguas para evitar posibles errores (capítulo 5).

4.3. Tecnologías y dispositivos utilizados

En este capítulo se van a detallar las diversas tecnologías y dispositivos que se han utilizado para la implementación de este demostrador tecnológico, desde la parte de la infraestructura y comunicación hasta la parte de programación.

4.3.1. Dispositivos utilizados

Para la correcta implementación del demostrador tecnológico, se han utilizado una serie de dispositivos y sensores que han permitido lograr el correcto funcionamiento del mismo, los dispositivos que se han utilizado son:

- Placas solares: Para el demostrador se han utilizado un par de placas solares, cada placa simula el parque de generación de energía asociado a cada nodo.
- Sensor INA 219: Es un sensor que permite medir tensión (hasta 26 V), intensidad (hasta 3.2A) y potencia (hasta 83.2W) en circuitos eléctricos, la comunicación con el sensor se realiza a través de I2C.
- ESP32: Es un microcontrolador de bajo costo y consumo de energía, tiene un procesador dual core Xtensa LX6 y su fabricante es Espressif [44]. Incluye tecnologías Wi-Fi y Bluetooth.
  El ESP32, conectado al sensor INA 219, a través de I2C, consigue los valores de la intensidad y tensión que están circulando por las placas solares en ese momento. A través de MQTT, el ESP32 se conecta con el *broker* MQTT y publica un mensaje con los datos obtenidos.
- Raspberry PI. El uso que tiene la Raspberry en el demostrador es similar al ESP32, se encarga de enviar la información obtenida del sensor INA 219, está continuamente enviando los datos relacionados con la tensión e intensidad generadas en el circuito asociado a cada placa solar.

4.3.2. Tecnologías utilizadas

En esta sección se van a describir las tecnologías utilizadas a lo largo de todo el proyecto, distinguiendo entre lenguajes de programación y tecnologías utilizadas:

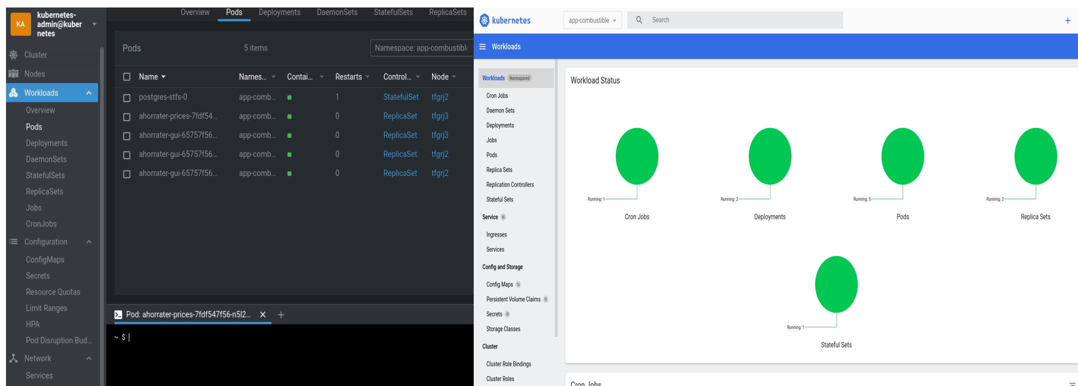
- Para comenzar, para el ESP 32, el lenguaje de programación que se ha utilizado es Arduino, se ha implementado un cliente MQTT que publica constantemente mensajes a un *broker* MQTT.

  Por otro lado, para la implementación del cliente MQTT en la Raspberry PI se ha utilizado Python como lenguaje de programación.
- Para la comunicación MQTT, se ha utilizado Eclipse Mosquitto [37], es un intermediario de mensajes de código abierto (*broker* MQTT), es muy ligero y permite ser ejecutado en computadores con pocos recursos o servidores completos.

- Para las distintas bases de datos se han utilizado, dos tipos diferentes de base de datos: para la BBDD que almacena la información de las estaciones de servicio y su precio, se ha utilizado PostgreSQL, mientras que para la información que se almacena para el cálculo de la energía solar se ha utilizado la base de datos H2.

- Para el acceso al *cluster* se han utilizado 3 tecnologías o herramientas diferentes:

  1. CLI: A través de la línea de comandos y la herramienta *kubectl* para operaciones de consulta de información a lo largo del periodo desarrollo.
  2. *Dashboard* Kubernetes. Se trata de una aplicaicón web que ofrece Kubernetes y que permite desplegar un *dashboard* para trabajar con el *cluster* a través de una GUI. Para su ejecución, basta con desplegar el manifiesto que se indica en la documentación de Kubernetes.
  3. Lens: Es un IDE que permite trabajar con el *cluster* de manera muy intuitiva, es similar a la interfaz web que proporciona Kubernetes, pero tiene algunas ventajas como el "*forwarding*" de los distintos elementos de Kubernetes, que da una gran utilidad para hacer pruebas durante el periodo de desarrollo.



(a) Imagen del panel principal de la herrmienta Lens

(b) Imagen de la pantalla Home del Dashboard de Kubernetes

Figura 18: Comparativa de las dos herramientas de acceso al *cluster*

- Para la programación de la API REST que trabaja contra el *cluster*, redesplegando pods o etiquetando los nodos, se ha utilizado el lenguaje de programación Python.

- Para la programación de los servicios de interfaz de usuario y de obtención de precios de la BBDD se ha utilizado Java como lenguaje de programación y Payara como servidor web.

- La implementación de los scripts que permiten automatizar el despliegue de componentes está escrita en Bash.
- Para implementar la API REST de Telegram, se ha consumido el cliente REST que ofrece Telegram y permite enviar tanto mensajes como imágenes.
- Como herramienta de control de versiones se ha utilizado Github.

Finalmente, se incluye una imagen del montaje de la maqueta, la cual contiene los siguientes elementos:

- Simulando el parque de energía renovable asociado al nodo 2 del *cluster*, está el dispositivo ESP32 conectado a un sensor INA 219, que permite obtener la información del circuito eléctrico de la placa solar.
  El circuito conecta una placa solar a un LED con una resistencia.
- Para simular el parque de energía renovable del nodo 3, se tiene una Raspberry PI, la cual está conectada del mismo modo que el ESP 32 al sensor INA 219 a través de I2C, y el sensor obtiene los valores de intensidad y tensión del circuito eléctrico.

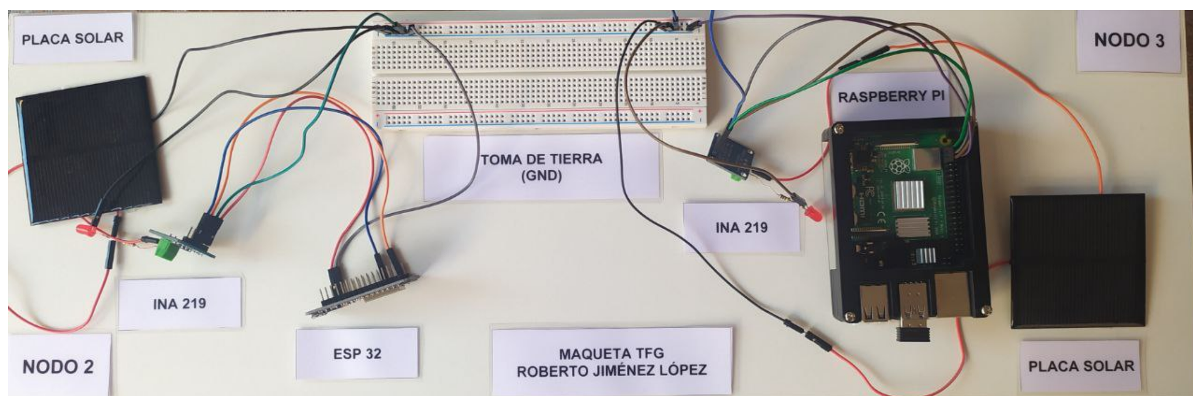


Figura 19: Maqueta elaborada para el demostrador tecnológico

Destacar que la toma de tierra (GND) en el circuito es común a todos los elementos de cada nodo: ESP32, INA 219 y placa solar por un lado, y por otro, Raspberry PI, INA 219 y placa solar.
La toma de tierra debe de ser la misma ya que si no los valores de tensión que se obtienen en el sensor INA 219 son erróneos, debido a que la diferencia de potencial se está midiendo con dos tomas de tierra diferentes.

4.4. Resultados obtenidos

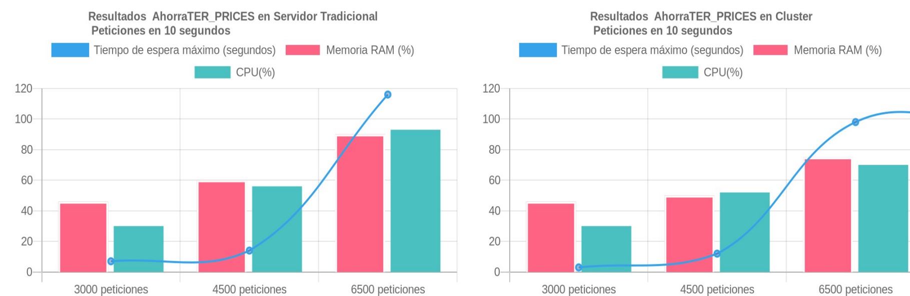
Como apartado final para el demostrador tecnológico, se han realizado una serie de test con la herramienta JMeter, en los que se ha simulado una carga de trabajo elevada para comparar la eficiencia de un *cluster* frente a la de un servidor tradicional.

Como servidor tradicional se ha utilizado cada microservicio con una única réplica dentro del *cluster*, mientras que para las pruebas de *cluster*, se han tenido los microservicios replicados en cada nodo del *cluster*.

4.4.1. Test microservicio AhorraTER_PRICES

El primer test se ha realizado con el microservicio AhorraTER_PRICES, el cual, como ya se ha comentado a lo largo de este capítulo, tiene una BBDD sin replicación, por lo que se deberían obtener resultados similares en las pruebas a realizar.

Se van a lanzar peticiones frente a una de las operaciones GET de la API REST del microservicio, concretamente la operación de obtener las comunidades autónomas de la BBDD.
Los resultados obtenidos son los siguientes:



(a) Resultados obtenidos en los test realizados sobre un servidor tradicional.

(b) Resultados obtenidos en los test realizados sobre un *cluster*.

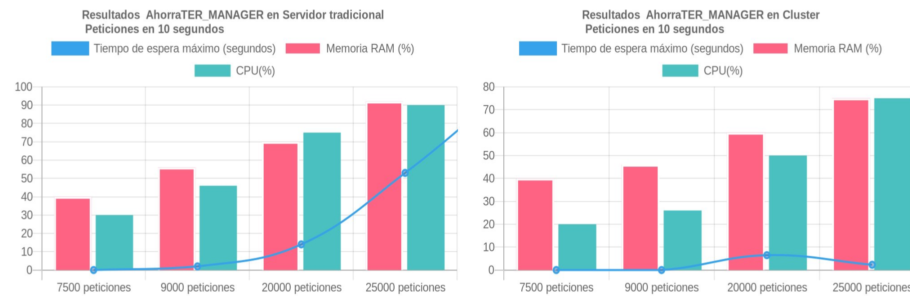
Figura 20: Comparativa de los resultados obtenidos para el microservicio AhorraTER_PRICES

Tras realizar el test, tal y como se esperaba los resultados obtenidos son similares. Por un lado, los tiempos de espera dentro del *cluster* son menores, y la carga de CPU y memoria obtiene valores más bajos. Por otro lado, los valores son relativamente similares ya que al tener la BBDD en un único pod, la información está centralizada, y por más réplicas que se tengan de la API REST, la información no está duplicada.

4.4.2. Test microservicio AhorraTER_MANAGER

El siguiente test se ha realizado con el microservicio AhorraTER_MANAGER, con la operación GET que obtiene la intensidad y tensión de cada uno de los nodos.

Para este test, se espera obtener una diferencia notoria, tanto en las métricas como en los tiempos de espera, ya que en este caso la información si que está replicada, por tanto, se debería repartir la carga de trabajo.
Los resultados obtenidos son los siguientes :



(a) Resultados obtenidos en los test realizados sobre un servidor tradicional.

(b) Resultados obtenidos en los test realizados sobre un cluster.

Figura 21: Comparativa de los resultados obtenidos para el microservicio AhorraTER_MANAGER

Los resultados obtenidos muestran en primer lugar, que la carga de trabajo CPU y memoria RAM se reduce considerablemente en el test realizado sobre el *cluster*, y el tiempo de espera que se obtiene en el caso del servidor tradicional es 23 veces más alto que en el *cluster*.

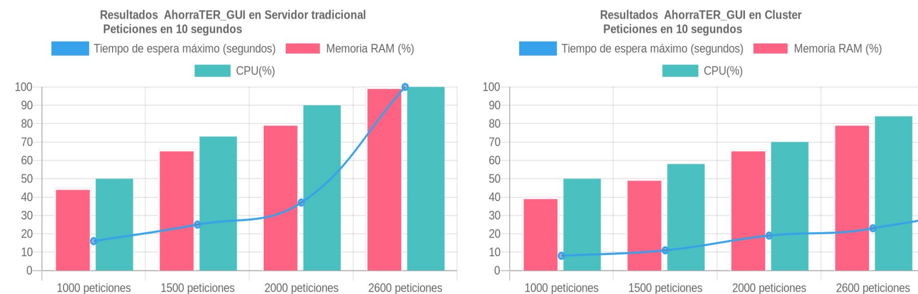
Destacar que en la prueba realizada con 25000 operaciones, se han detectado un porcentaje elevado de errores, en el caso del servidor tradicional un 30 % de las peticiones han generado error y en el caso del *cluster*, un 39.04 %. El resto de peticiones han sido exitosas.

4.4.3. Test microservicio AhorraTER_GUI

Para concluir, el último test se ha realizado con el microservicio que contiene la interfaz de usuario.

Al tratarse de un microservicio que contiene la interfaz de usuario y que obtiene información del resto de microservicios, se espera que pueda soportar menos operaciones que el resto.

Los resultados obtenidos son los siguientes:



(a) Resultados obtenidos en los test realizados sobre un servidor tradicional.

(b) Resultados obtenidos en los test realizados sobre un *cluster*.

Figura 22: Comparativa de los resultados obtenidos para el microservicio AhorraTER_GUI

Tal y como se esperaba, en este test, los resultados son muy favorables hacia el *cluster*, en primer lugar en el último test, el servidor tradicional no puede hacer frente a 2600 peticiones y acaba saltando una excepción en el servidor. Por otro lado, el tiempo de espera que necesita el *cluster* para hacer frente a las peticiones es muy inferior al que necesita el servidor tradicional.

5. Conclusiones y trabajo futuro

A modo de cierre, para concluir este TFG, se puede señalar que el estudio de la implantación de un *cluster* Kubernetes en entornos tanto de desarrollo como de producción, puede plantearse como una buena opción ya que ofrece ventajas sobre la implantación de un servidor tradicional.

El resultado más relevante ha sido la posibilidad de utilizar como variable principal para el balanceo de cargas de un *cluster* una fuente de energía renovable, pues permite reducir el gasto y consumo de un nodo aprovechando los recursos energéticos naturales y contribuyendo a reducir la huella de carbono.

Por otro lado, es importante destacar las ventajas que ofrece la implantación de un *cluster* Kubernetes, como se ha podido observar en el capítulo 4.4, un *cluster* soporta una gran cantidad de peticiones al distribuir la carga de trabajo en los nodos disponibles, cosa que ocurre en menor medida en un servidor tradicional, debido a que todos los recursos provienen de un mismo nodo.

Sin embargo, el trabajo deja una línea de posibilidades abierta a mejoras en las que seguir trabajando, algunas de estas podrían ser:

- Replicación en BBDD. Como ya se ha comentado en el capítulo 4.2.2, para el demostrador tecnológico llevado a cabo solo utiliza un Volumen en Kubernetes de tipo "HostPath", lo que quiere decir que si el pod asociado a la BBDD muere y se despliega en otro nodo, la información se perdería, ya que solo persiste en uno de los nodos disponibles.
  Una opción de mejora es añadir replicación a la BBDD, a través de un esquema maestro-esclavo en el que se tienen varias réplicas del pod de la BBDD y cada vez que se interactúa con el sistema, toda la información se replica en el resto de contenedores.
  Otro tipo de solución al problema planteado es el uso de volúmenes en la nube, lo que permitiría abstraer el volumen de los nodos del *cluster*, de modo que cualquier nodo pudiese acceder a la información, solo bastaría con tener varios pods que permitiesen controlar el acceso a la misma.

- *Rollout* y *rollback*. Es una opción que proporciona Kubernetes y sería interesante aplicar para una versión futura, el *rollout* y *rollback* permiten, a la hora de desplegar los distintos componentes sobre el *cluster*, por ejemplo un Deployment, actualizar las imágenes que ya están corriendo sobre el propio *cluster*, obteniendo en ese caso, un número de réplicas con la nueva imagen, y el resto con la imagen funcional antigua, de modo que se puedan realizar test para comprobar el funcionamiento. De igual modo, en caso de que la nueva imagen tuviese cualquier tipo de error, Kubernetes permite volver a las imágenes antiguas a través de un rolback.
  Para este demostrador tecnológico, no se han utilizado estas dos herramientas ya que el espacio de memoria del nodo donde se almacenan las imágenes no es ilimitado, y se han realizado una gran cantidad de pruebas, lo que hubiese generado demasiadas imágenes y saturado el almacenamiento de la máquina.

- Energía renovable. En este aspecto, como ya se ha comentado durante todo el trabajo, en la maqueta realizada se ha simulado que el valor que se obtenía de las placas solares era el valor disponible para un nodo directamente cuando no es así, una solución a esta simulación podría ser el uso de baterías, las cuales se cargasen con la energía que suministran las placas solares y poder obtener la energía disponible de una batería, generando una maqueta más realista.
- Microservicio Telegram Bot. Una posible mejora para el sistema podría ser una implementación de un microservicio que contenga un bot de Telegram, el cual permita interactuar con él para obtener información relacionada con el *cluster*, así como las métricas o los valores obtenidos por el microservicio AhorraTER_MANAGER relacionados con la intensidad y tensión de las placas solares.
- En relación a la interfaz de usuario, una mejora que permitiría al usuario tener más información sobre lo que ocurre en el *backend*, sería algún tipo de indicador que permitiese comprobar si los dispositivos encargados de enviar la información, están activos o no.

Referencias

[1] ¿Qué es Kubernetes? | Kubernetes ¿Qué es Kubernetes? | Kubernetes. (2022) Retrieved 26 May 2022, from https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/.

[2] K3s vs k8s – What's the difference between K8s and k3s - Civo.com. (2022) Retrieved 26 May 2022, from https://www.civo.com/blog/k8s-vs-k3s.

[3] 2.3.5.3 br_netfilter Module. (2022) Retrieved 26 May 2022, from https://docs.oracle.com/en/operating-systems/olcne/1.1/start/netfilter.html.

[4] 830-1998 - IEEE Recommended Practice for Software Requirements Specifications | IEEE Standard | IEEE Xplore. (2022) Retrieved 27 May 2022, from https://ieeexplore.ieee.org/document/720574.

[5] Contenedores de Docker | ¿Qué es Docker? | AWS. (2022) Retrieved 30 May 2022, from https://aws.amazon.com/es/docker/.

[6] Qué es un contenedor? | Microsoft Azure. (2022) Retrieved 30 May 2022, from https://azure.microsoft.com/es-es/overview/what-is-a-container/#overview.

[7] Web services | Definición Web Services - IONOS. (2022) Retrieved 30 May 2022, from https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/web-services/.

[8] Qué es Multi-Tenancy - Evaluando Cloud. (2022) Retrieved 30 May 2022, from https://evaluandocloud.com/que-es-multi-tenancy/#:~:text=Multi%2DTenancy%20(Multiusuario)%20hace,aislados%20l%C3%B3gicamente%2C%20pero%20f%C3%ADsicamente%20integradas.

[9] Clusters de servidores | Seguridad Informática. (2022) Retrieved 30 May 2022, from https://infosegur.wordpress.com/unidad-2/clusters-de-servidores/.

[10] ¿Qué es el registro de contenedores?. (2022) Retrieved 30 May 2022, from https://www.redhat.com/es/topics/cloud-native-apps/what-is-a-container-registry#:~:text=El%20registro%20de%20contenedores%20es,de%20las%20aplicaciones%20en%20contenedores.

[11] Bare Metal: ¿qué són y para qué sirven los servidores bare metal?. (2022) Retrieved 31 May 2022, from https://www.servnet.mx/blog/bare-metal-que-son-y-para-que-sirven-los-servidores-bare-metal.

[12] Apache JMeter - Apache JMeter™. (2022) Retrieved 31 May 2022, from https://jmeter.apache.org/.

[13] ¿Qué es TCP/IP? | Cómo funcionan el modelo y los protocolos | Avast (2022) Retrieved 31 May 2022, from https://www.avast.com/es-es/c-what-is-tcp-ip.

[14] MQTT - The Standard for IoT Messaging. (2022) Retrieved 01 Jun 2022, from https://mqtt.org/.

[15] ¿Qué es el URI? Definición y explicación del identificador - IONOS. (2022) Retrieved 02 Jun 2022, from https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/uri-identificador-de-recursos-uniformes/.

[16] CoAP — Constrained Application Protocol | Overview. (2022) Retrieved 02 Jun 2022, from https://coap.technology/.

[17] Goto IoT | Introducción a CoAP. (2022) Retrieved 02 Jun 2022, from https://www.gotoiot.com/pages/articles/coap_intro/index.html.

[18] How To Setup Kubernetes Cluster Using Kubeadm - Easy Guide. (2022) Retrieved 02 Jun 2022, from https://devopscube.com/setup-kubernetes-cluster-kubeadm/.

[19] Open Container Initiative - Open Container Initiative Menu Open. (2022) Retrieved 02 Jun 2022, from https://opencontainers.org/.

[20] CRI-O: Container Runtime Interface para Kubernetes - IONOS. (2022) Retrieved 02 Jun 2022, from https://www.ionos.es/digitalguide/servidores/know-how/que-es-cri-o/#:~:text=CRI%2DO%20es%20una%20implementaci%C3%B3n,en%20la%20primavera%20de%202019..

[21] The differences between Docker, containerd, CRI-O and runc - Tutorial Works. (2022) Retrieved 03 Jun 2022, from https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/#docker.

[22] A Practical Introduction to Container Terminology | Red Hat Developer. (2022) Retrieved 03 Jun 2022, from https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction.

[23] Performance comparison between Linux containers and virtual machines | IEEE Conference Publication | IEEE Xplore. (2022) Retrieved 03 Jun 2022, from https://ieeexplore.ieee.org/abstract/document/7164727.

[24] ¿Qué diferencia hay entre contenedores y Máquinas Virtuales? (2022) Retrieved 03 Jun 2022, from https://www.campusmvp.es/recursos/post/que-diferencia-hay-entre-docker-contenedores-y-maquinas-virtuales.aspx.

[25] ¿Qué es un proceso daemon en Linux? - CompuHoy.com. (2022) Retrieved 04 Jun 2022, from https://www.compuhoy.com/que-es-un-proceso-daemon-en-linux/.

[26] Project Calico. (2022) Retrieved 05 Jun 2022, from https://www.tigera.io/project-calico/.

[27] GitHub - kubernetes-sigs/metrics-server: Scalable and efficient source of container resource metrics for Kubernetes built-in autoscaling pipelines.. (2022) Retrieved 05 Jun 2022, from https://github.com/kubernetes-sigs/metrics-server.

[28] ¿Qué son los contenedores de software y por qué utilizarlos?. (2022) Retrieved 05 Jun 2022, from https://profile.es/blog/contenedores-de-software/#:~:text=El%20motor%20de%20contenedores%20es,m%C3%A1s%20popular%20actualmente%20ser%C3%ADa%20Docker.

[29] networking:bridge [Wiki]. (2022) Retrieved 11 Jun 2022, from https://wiki.linuxfoundation.org/networking/bridge.

[30] Podman. (2022) Retrieved 11 Jun 2022, from https://podman.io/.

[31] What is RKT?. (2022) Retrieved 11 Jun 2022, from https://www.redhat.com/en/topics/containers/what-is-rkt.

[32] Linux Containers. (2022) Retrieved 11 Jun 2022, from https://linuxcontainers.org/.

[33] Top 5 mejores alternativas a Docker - Guiadev. (2022) Retrieved 11 Jun 2022, from https://guiadev.com/top-5-alternativas-a-docker/.

[34] Arquitectura de Kubernetes - aprenderDevOps. (2022) Retrieved 11 Jun 2022, from https://aprenderdevops.com/arquitectura-de-kubernetes/.

[35] Componentes de Kubernetes | Kubernetes Componentes de Kubernetes | Kubernetes. (2022) Retrieved 12 Jun 2022, from https://kubernetes.io/es/docs/concepts/overview/components/.

[36] Conceptos | Kubernetes. (2022) Retrieved 12 Jun 2022, from https://kubernetes.io/es/docs/concepts/_print/.

[37] Eclipse Mosquitto. (2022) Retrieved 12 Jun 2022, from https://mosquitto.org/.

[38] Servidor proxy - Wikipedia, la enciclopedia libre. (2022) Retrieved 11 Jun 2022, from https://es.wikipedia.org/wiki/Servidor_proxy.

[39] I2C - Puerto, Introducción, trama y protocolo - HETPRO/TUTORIALES. (2022) Retrieved 12 Jun 2022, from https://hetpro-store.com/TUTORIALES/i2c/.

[40] Docker Hub. (2022) Retrieved 14 Jun 2022, from https://hub.docker.com/_/postgres.

[41] API | datos.gob.es. (2022) Retrieved 14 Jun 2022, from https://datos.gob.es/es/apidata.

[42] Payara Micro Enterprise ; Payara Services Ltd. (2022) Retrieved 14 Jun 2022, from https://www.payara.fish/products/payara-micro/.

[43] Miniconda ; Conda documentation. (2022) Retrieved 16 Jun 2022, from https://docs.conda.io/en/latest/miniconda.html.

[44] Wi-Fi & Bluetooth MCUs and AIoT Solutions I Espressif Systems. (2022) Retrieved 17 Jun 2022, from https://www.espressif.com/.

[45] I²C - Wikipedia, la enciclopedia libre. (2022) Retrieved 19 Jun 2022, from https://es.wikipedia.org/wiki/I%C2%B2C .

[46] Qué; es una máquina virtual y cómo funciona | Microsoft Azure. (2022) Retrieved 19 Jun 2022, from https://azure.microsoft.com/es-es/overview/what-is-a-virtual-machine/#overview.

[47] El protocolo MQTT: impacto en España - Security Art Work. (2022) Retrieved 20 Jun 2022, from https://www.securityartwork.es/2019/02/01/el-protocolo-mqtt-impacto-en-espana/.

[48] ¿Qué es Arduino? | Arduino.cl. (2022) Retrieved 20 Jun 2022, from https://arduino.cl/que-es-arduino/.

[49] ¿Qué es una API? - Guía sobre las API para principiantes - AWS. (2022) Retrieved 20 Jun 2022, from https://aws.amazon.com/es/what-is/api/.