Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Informática

Trabajo Fin de Grado

Estudio e implementación de escenarios con Kubernetes

Autor: Gissella Carolina Santacruz Osorio

Tutor/es: José Manuel Arco Rodríguez

2020

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

Estudio e implementación de escenarios con Kubernetes

Autor: Gissella Carolina Santacruz Osorio

Tutor: José Manuel Arco Rodríguez

Tribunal:

Presidente: Elisa Rojas Sánchez

Vocal 1º: Juan Antonio Carral Pelayo

Vocal 2º: José Manuel Arco Rodríguez

Calificación:
Fecha: septiembre 2020

Agradecimientos

Este proyecto va dedicado a todas las personas que me han apoyado al principio de este camino y a los que se han ido uniendo a él.

Muchas gracias a mi familia y amigos, especialmente a José, Ester, Leslie, Aída y George, sin vuestro apoyo no hubiese sido posible.

Resumen

En este proyecto fin de grado se va a explicar todos los conceptos básicos de Kubernetes, su arquitectura, conceptos avanzados y se van a plantear, diseñar e implementar cinco escenarios principales para demostrar su fiabilidad y escalabilidad. En estos escenarios se verá el balanceo de carga, la replicación de base de datos, un clúster hibrido en local y en la nube, persistencia y cambio de versión de aplicaciones. Se instalará un clúster en local y se desplegará también en Cloud, utilizando la plataforma AWS.

El enfoque principal es descubrir Kubernetes, dar una amplia información de sus objetos, y realizar escenarios donde se implementen algunos conceptos principales.

Palabras clave: Pods, servicios, contenedores, Docker, kops, orquestación de contenedores.

Abstract

In this BSc dissertation we are going to explain concepts of Kubernetes, the architecture, advanced concepts and going to raise, design and implement five main scenarios for demonstrate Kubernetes reliability and scalability. Where we are going to see load balancing, databases replication and a hybrid cluster in local and in the cloud, also persistent and rollouts.

The principal aim is to discover Kubernetes, give a full information of its objects and compare with other containers orchestrators. We will install Kubernetes cluster in a local machine and deploy in the cloud, using platform such an AWS, Microsoft Azure and GCP.

Keywords: Pods, services, containers, Docker, kops, container orchestration.

Contenido

Índice de figuras

Glosario de acrónimos y abreviaturas

k8s	Kubernetes
GCP	Google Cloud Platform
Helm	Gestor de paquetes para monitorización
AWS	Amazon Web Services
VPC	Virtual Private Cloud
VPN	Virtual Private Network
VPG	Virtual Private Gateway
IGW	Internet Gateway
CGW	Customer Gateway
CPU	Central Proccesing Unit
API	Application Programming Interface
PVC	Persistent Volume Claim
PV	Persistent Volume
EC2	Amazon Elastic Compute Cloud
EBS	Amazon Elastic Block Store
HPA	Horizontal Pod Autoscaling
IAM	Identity and Access Management
YAML	Lenguaje en el que se definen los objetos de Kubernetes
NFS	Network File System
GCE	Google Compute Engine
Kops	Herramienta para la creación del clúster k8s
strongSwan	Solución VPN para clúster cloud y on-premise

Resumen extendido

Este trabajo fin de grado es principal objetivo es descubrir Kubernetes y desarrollar unos escenarios que reflejen lo aprendido, utilizando los principales conceptos, ya que Kubernetes es complejo y cubre muchos casos disponibles en internet y otros desarrollados en este proyecto.

En primer lugar, se va a explicar algunos conceptos relevantes antes de adentrarnos a lo que es Kubernetes, como, por ejemplo, máquinas virtuales, contenedores, la arquitectura en microservicios y los orquestadores de contenedores que existen además de Kubernetes.

En la segunda parte se va a explicar todos los conceptos básicos y avanzados que Kubernetes ofrece, así como su arquitectura. También se realizará un ejemplo práctico de como instalar en el ordenador minikube, clúster de k8s con un único nodo y un clúster k8s con múltiples nodos en local.

También se realizará un clúster k8s en la nube AWS, utilizando una herramienta poco conocida pero muy útil, llamada Kops que despliega el clúster desde la línea de comandos.

Por último, se implementarán cinco escenarios. Autoescalado Horizontal de pods (HPA) donde se definen reglas para simular cómo Kubernetes realiza el autoescalado de acuerdo con reglas que se declaran en el manifiesto del *pod*. En el segundo escenario, se replicará la bases de datos de mongoDB haciendo uso de un objeto *Statefulsets*. Este tipo de objetos se encarga de la gestión de despliegue, escalado y garantiza el orden de un conjunto de *Pods*.

El tercer escenario que consiste en realizar un clúster híbrido, con parte en local y parte en la nube de AWS, utilizando herramientas que la plataforma ofrece para la comunicación entre ellos y también se tiene otro escenario para ver cómo compartir archivos entre el clúster.

También se demostrará el uso de *deployments* para retroceder a una versión previa de la aplicación o a una versión posterior y por último se hablará de un escenario que se realizará en el futuro que consiste en desplegar un clúster de Kubernetes en OpenStack.

Memoria del trabajo

Capítulo 1

1. Introducción

En las empresas se utilizan aplicaciones que en algunos casos pueden llegar a ser muy complejas. Por ejemplo, una aplicación que tenga muchas dependencias o que se componga de muchos servicios… ¿llegaría a correr en otro entorno de la misma forma?

Algunas medidas prácticas sería el despliegue de aplicaciones de un entorno a otro sin la necesidad de configurar sus dependencias desde cero, el escalado de sistemas de manera automática, la integración continua, la realización de despliegues automatizados. Por ello, nace el concepto de contenedor, que reúne todas las dependencias para que una aplicación sea ejecutada en cualquier entorno.

A continuación, se mostrará una breve descripción de Kubernetes: *"Kubernetes es una plataforma de código abierto, extensible y portátil para administrar cargas de trabajo y servicios en contenedores, que facilita tanto la configuración declarativa como la automatización. Tiene un ecosistema grande, de rápido crecimiento. Los servicios, el soporte y las herramientas de Kubernetes están ampliamente disponibles"* [1]

Kubernetes, al ser una plataforma de código abierto, dispone de muchos desarrolladores que van aportando funcionalidades, mejorando el programa. Con Kubernetes es posible controlar los contenedores de las aplicaciones/servicios y desplegarlos en diferentes equipos, aunque no tengan las mismas características físicas o librerías. Debido a que Kubernetes ofrece escalabilidad, esto hace que sea más fácil aumentar los recursos asignados para el sistema medida sea necesario. Además, proporciona al administrador la posibilidad de monitorizar el funcionamiento del sistema. Por ello, se está convirtiendo en el sistema más usado para la orquestación de contenedores.

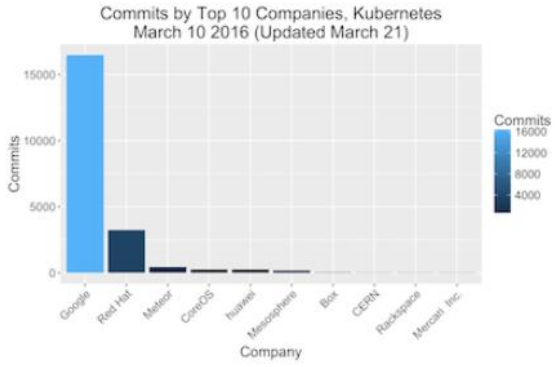


Figura 1.Número de commits realizados en el repositorio de código abierto de Kubernetes por distintas empresas durante el año 2016 (Fiz, 2017)

Se mencionarán algunas características de Kubernetes por las que está siendo elegido en la actualidad como principal sistema de orquestación de contenedores [2]:

- **Escalado y autoescalado**: permite el escalado vertical u horizontal, manual o automático, en función del uso de la CPU.

- **Descubrimiento de servicios y balanceo de carga**: Kubernetes asigna a los contenedores sus direcciones IP y DNS únicos y distribuye entre ellos la carga, por lo que no es necesario un mecanismo para el descubrimiento de servicios.
- **Auto reparación**: Cuando un nodo falla, puede reemplazarse. Cuando un contenedor da fallos se puede reiniciar e, inclusive, pararlos.
- **Despliegues y rollbacks automáticos**: Si se va a realizar un cambio en una configuración, se realiza el despliegue de forma progresiva y monitorizada, en caso de fallo, ofrece la posibilidad de rollback automático.
- **Planificación**: De acuerdo con los recursos que requiera y a otras restricciones, se decide en qué nodo se ejecutará un contenedor.
- **Gestión de la configuración**: Kubernetes ofrece un almacenaje para la información como passwords en "secrets", para no exponer información confidencial mientras se realice un despliegue o imagen.
- **Orquestación del almacenamiento**: se monta de manera automática el sistema de almacenamiento, independientemente de donde se realice (local, cloud, NFS, etc.)
- **Ejecución Batch**: se pueden gestionar cargas de trabajo batch e integración continua para ir reparando los contenedores que den fallo.

Una vez vistas estas características, ya se puede pensar en qué escenarios se podría usar Kubernetes para comprobar su fiabilidad y escalabilidad.

1.1. Objetivos y campo de aplicación

En este trabajo se va a realizar un estudio de Kubernetes, se verán las ventajas que aporta Kubernetes y se implementarán escenarios en donde se pondrá a prueba todo lo estudiado. El trabajo de fin de grado se desarrollará de tal modo que sus contenidos puedan ser empleados para la docencia en una titulación universitaria.

Los principales objetivos para este Proyecto Fin de Grado son:

- Estudio teórico de las diferentes posibilidades de uso los K8s.
- Realizar diferentes maquetas con escenarios típicos de despliegue de K8s en uno y dos servidores físicos.
- Realizar diferentes maquetas con escenarios típicos de despliegue de K8s en dos servidores físicos con un servidor de ficheros en red.
- Orquestación de servicios con Kubernetes, en local e hibrido con una nube pública.
- Comprobación de la fiabilidad y escalabilidad de Kubernetes

1.2. Descripción del trabajo

En primer lugar, se realizará una introducción explicando las necesidades que hacen que sea útil el empleo de los sistemas de gestión de contenedores y se verán las diferentes tecnologías existentes en la actualidad para ello.

También, se verán conceptos que son interesantes para ir entendiendo la terminología que se irá usando a lo largo del proyecto y se expondrán los diferentes escenarios que se quieren plantear y se realizará el diseño de los escenarios que se pretende realizar en este proyecto.

Después de realizar este análisis se procederá a la implementación de este, utilizando escenarios con servidores físicos, servidor de ficheros, servidores en la nube.

Debido a que Kubernetes no presenta muchas restricciones de uso, se pueden usar para diversos casos o plantearnos una nueva opción de uso, en cuanto a la forma de instalación del *cluster*, existen múltiples tipos de uso [3]:

- **Bare metal on-premise**: desplegar el *cluster* directamente sobre máquinas físicas y diferentes S.O.
- **Virtualización**: realizar el *cluster on-premise* pero sobre máquinas virtuales, como OpenStack, Fedora, etc.
- **Soluciones Cloud**: como pueden ser Microsoft Azure, Amazon Web Services, Google Container Engine, entre otros.

A continuación, se muestran los componentes y arquitectura básica habitual de Kubernetes:

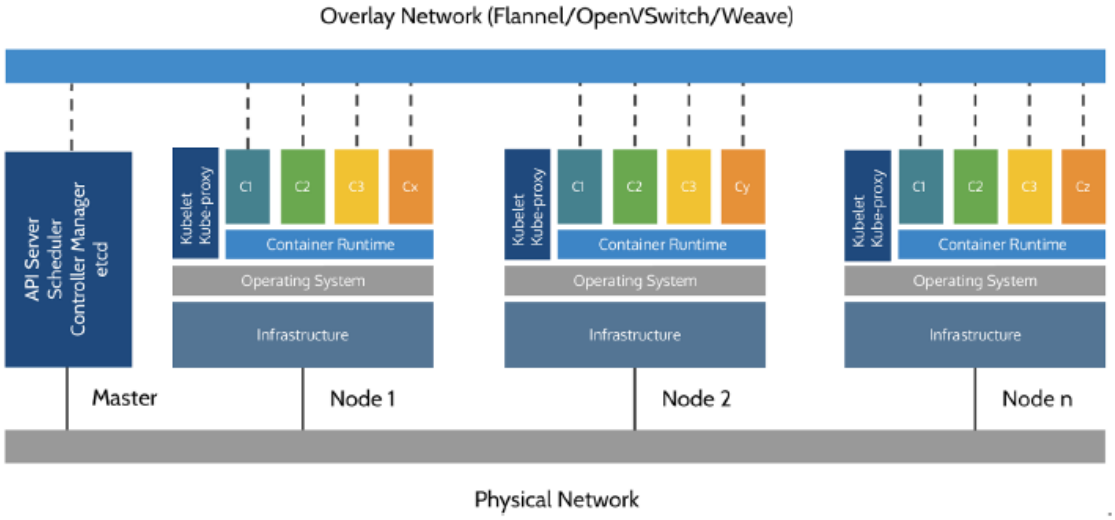


Figura 2. Cluster de Kubernetes (Matas, 2018)

Una vez realizados los escenarios de uso de Kubernetes, se procede a la documentación de este.

1.3. Estructura del trabajo

Este trabajo está estructurado en siete capítulos, el primer capítulo es introductorio, dónde se describe de forma general los objetivos y trabajo a ver en el proyecto.

El segundo capítulo tratará los fundamentos teóricos de este proyecto.

En el tercer capítulo se verá Kubernetes, todos los conceptos que ofrece y ejemplos de la mayoría de ellos.

En el cuarto capítulo se crearán clústeres reales de Kubernetes, empezando por minikube, un clúster de tres nodos en local y otro clúster en la nube de AWS. También se hablará de algunas herramientas de monitorización para Kubernetes.

El quinto capítulo se detallará la solución adoptada para los cinco escenarios principales que se han creado.

En el sexto capítulo se expondrán las conclusiones, comentado todo el trabajo realizado y algunos escenarios futuros.

En el séptimo y último capítulo, se describirá el presupuesto utilizado en la realización del proyecto.

Capítulo 2

2. Contexto teórico

Tanto los contenedores como las máquinas virtuales son dos soluciones de virtualización distintas y cada una tiene sus ventajas e inconvenientes. La decisión sobre si emplear máquinas virtuales o contenedores dependerá de cada situación.

Una máquina virtual es una aplicación que simula una máquina mediante software utilizando los recursos de la máquina física donde se hospeda (véase Figura 3). La aplicación encargada de gestionar los recursos de la máquina huésped y dedicarlos a la virtualización de la máquina simulada se conoce como hipervisor.

Un contenedor es una aplicación con sus librerías empaquetadas y todo lo necesario para ejecutarse por sí solo, en un sistema operativo en el que tenga un software runtine engine (similar a un hipervisor) para que los contenedores puedan ejecutarse.

El principal uso que se le suele dar a las máquinas virtuales es del reemplazar máquinas físicas, ya que son más ligeras, versátiles y baratas.

Antes de la concepción de los contenedores, se han utilizado máquinas virtuales como mecanismo de virtualización para completar o reemplazar a los ordenadores. Pero ahora se tiene un concepto nuevo: los contenedores.

Inicialmente, los contenedores se utilizaban únicamente en empresas como Google, Facebook, etc. Actualmente, gracias al surgimiento de Docker y software de virtualización utilizando contenedores, se ha extendido el uso de los contenedores al público general.

La ventaja que proporcionan los contenedores frente a las máquinas virtuales es que éstos no necesitan configuraciones externas a la hora de desplegar una aplicación. Esto es debido a que es posible encapsular todas las dependencias de la aplicación y, por tanto, desplegar las dependencias y la propia aplicación al mismo tiempo. Por ejemplo, son muy recomendables para las aplicaciones web.

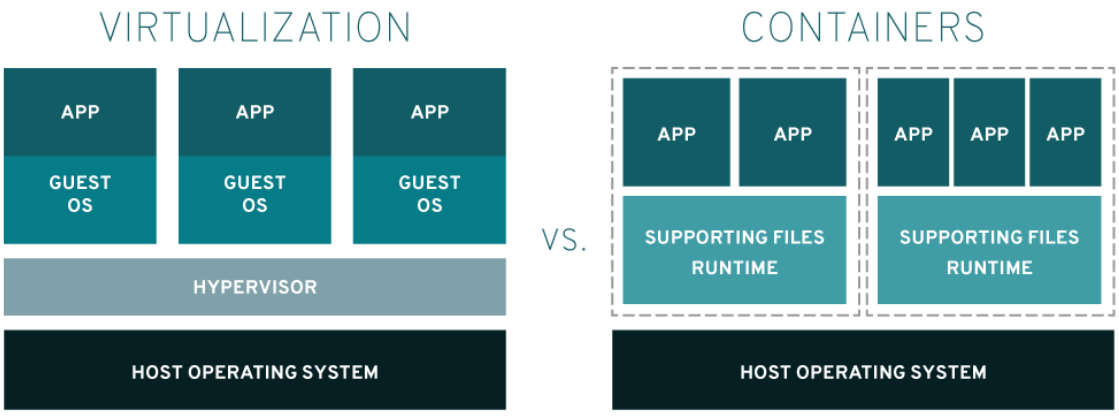


Figura 3. Comparación de la arquitectura de máquinas virtuales y contenedores [4]

2.1. Contenedores y máquinas virtuales

Cuando se piensa en estos mecanismos se puede llegar a pensar que son bastantes parecidos, ya que ambos son formas de virtualización destinadas a la creación de entornos independientes. Sin embargo, en realidad son muy diferentes y su uso dependerá de la situación.

2.1.1. Hipervisores

Cuando se tiene virtualización basada en hipervisores, se refieren a las máquinas virtuales con sistemas operativos que simular ser un ordenador físico real, permitiendo compartir el hardware entre una o más máquinas virtuales y con el sistema anfitrión (como CPU, RAM o almacenamiento en disco). Existe gran variedad de software comercial para el trabajo con máquinas virtuales, como puede ser VMWare, VirtualBox, etc.

2.1.2. Contenedores

Un contenedor es un entorno ligero donde se aíslan y agrupan aplicaciones y donde se tienen todos los archivos, variables y bibliotecas necesarias para poder ejecutar la aplicación, son portables, ya que, a diferencia de las máquinas virtuales, que proveen su propio sistema operativo, los contenedores hacen uso del sistema operativo de su anfitrión, aunque en algunos sistemas operativos, como en MAC-OS, se requiere el uso de VMs para la ejecución de contenedores.

Antes de que la tecnología de virtualización basada en contenedores sea conocida principalmente por contenedores como Rocket (rkt), CRI-O o Docker, en Linux se tiene LXC Container, que proporciona un entorno virtual para desplegar instancias de sistemas operativos aislados, se lograba este entorno haciendo uso de las características del kernel como *cgroups* o *namespaces*.

Los contenedores de Linux solo funcionan en máquinas Linux o máquinas virtuales con sistema operativo Linux. Hoy en día, los contenedores se pueden utilizar en servidores Windows mediante virtualización Hyper-V o VirtualBox requisito indispensable o en plataformas *cloud* como Azure.

2.1.2.1. Docker

En este proyecto se utiliza Docker como gestor de contenedores para desplegar las aplicaciones. Docker se define como un programa de código abierto que encapsula la aplicación y sus dependencias en un contenedor. Como se está trabajando con microservicios, es importante que un contenedor no contenga más una aplicación para no sobrecargarlo, ya que en algunos casos se puede desplegar más de una aplicación desde un contenedor.

Docker trabaja con el concepto de imagen. Una imagen es un archivo binario que contiene la aplicación, sus variables de entornos y todo lo necesario para la ejecución. Una imagen es portable y puede contener cualquier tipo de aplicación, desde una aplicación web hasta un sistema operativo. También se puede subir a Docker Hub donde se comparten imágenes entre los desarrolladores y administradores.

Docker dispone de un repositorio *Hub* en el que se pueden descargar imágenes que aporta la comunidad de usuarios para trabajar con contenedores de forma gratuita. También se puede tener un repositorio privado para uso particular o de empresa.

Dentro de repositorio de Docker, se encuentran contenedores con las siguientes aplicaciones:

- Aplicaciones web → Apache, Nginx, httpd
- SSOO → Ubuntu, Centos, Debian
- Aplicaciones BBDD → MySQL, MongoDB, PostgreSQL, Redis, SQL Server, mariadb
- Lenguajes de programación → golang, Java, Python, PHP, Ruby

Entre muchos otros, consultando la web oficial de DockerHub

https://hub.docker.com/search?image_filter=official&type=image

Un contenedor es una imagen de Docker en estado de ejecución. Es posible tener varios contenedores en ejecución a la vez que hayan sido generados a partir de una misma imagen.

Se puede instalar Docker en un ordenador Windows desde la página: https://docs.docker.com/docker-for-windows/install/.

Algunos de los comandos básicos para la gestión de contenedores con Docker se listan a continuación:

- Listar las imágenes de Docker instaladas:

```
docker image ls
```

- Descargar una imagen de un repositorio de imágenes (por ejemplo, del Docker Hub):

```
docker image pull <URL de la imagen>
```

- Ejecutar una imagen descargada previamente y configura una redirección del puerto 8080 del anfitrión al puerto 80 del contenedor (útil para servidores web):

```
docker container run -p 8080:80 -d <nombre de la imagen>
```

- Listar los contenedores en ejecución:

```
docker container ls
```

- Detener un contenedor:

```
docker container stop <id>
```

- Eliminar un contenedor y liberar los recursos reservados para el mismo:

```
docker container rm <id>
```

2.1.3. Orquestadores de contenedores

Un orquestador es la herramienta que se utiliza para gestionar no solo contenedores, sino también la infraestructura necesaria para organizar a estos contenedores en arquitecturas más complejas. Por ejemplo, un orquestador se encargaría de generar los contenedores necesarios para el funcionamiento de una aplicación web (servidor web, servidor de ficheros, base de datos, etc.) y la infraestructura de red necesaria para la comunicación entre los contenedores.

El orquestador más popular, y el más utilizado, es Kubernetes. Existen muchos orquestadores como Docker Swarm, Mesos, Helios, Marathon, Amazon (ECS), Azure (AKS), entre muchos otros más en el mercado, siendo algunos gratis y otros se pagan según los recursos que se vayan utilizando.

2.2. Arquitectura de microservicios

Es importante tener en cuenta el concepto de microservicio, ya que es la arquitectura que se suele utilizar en clústeres gestionados por orquestadores de servicios. Es una arquitectura en la que una aplicación se divide en conjuntos de servicios y cuyo objetivo es que cada parte de una aplicación (contenedor) realice solo una actividad y nada más, siendo cada parte independiente de las otras partes de la aplicación. Esto se conoce como arquitectura de microservicios. Esto facilita el reemplazo de servicios individuales o el escalado de una parte concreta de la aplicación. Esto se debe a las siguientes características de esta arquitectura:

- **Bajo acoplamiento** debido al reducido nivel de dependencia entre diferentes servicios.
- **Alta mantenibilidad** debido a que es posible gestionar cada servicio de forma independiente.
- Permite hacer **despliegues independientes** de los servicios. Esto quiere decir que es posible añadir, actualizar o eliminar un servicio sin que afecte al resto.

A partir de esta estructura se puede ir construyendo aplicaciones más complejas, rápidas y fiables.

2.3. Orquestadores de contenedores

Kubernetes es el más popular de entre los orquestadores de contenedores que existen y el más elegido por los desarrolladores para desplegar sus aplicaciones. Al principio, empezó como un proyecto de Google y éste ejecuta millones de contenedores usando Kubernetes.

Sus características principales son la escalabilidad, fiabilidad y el balanceo de carga.

Aunque este orquestador sea el que se va a ver y estudiar en este trabajo, además existen otros en el mercado, entre los cuales los siguientes son los más populares.

2.3.1. Docker Swarm

Swarm es una herramienta de Docker y por lo tanto viene incluido como el motor de aplicaciones Docker. Ofrece, al igual que Kubernetes, servicios, escalabilidad, seguridad y balaceo de carga.

En cuando a la usabilidad, Docker Swarm se considera más fácil de manejar, pero no llega a ser tan potente como Kubernetes ni tan usado por los desarrolladores ni proveedores *cloud*.

2.3.2. Mesos

Es una plataforma de código abierto que ofrece un *kernel* para clúster con datos y contenedores, se ha diseñado para gestionar múltiples máquinas dentro de un mismo entorno.

Se basa en una configuración *master-slave*, en el cual el *master* es responsable de controlar los recursos que se tiene y asignarlos dependiendo de ciertos factores como la disponibilidad. Los recursos pueden ser asignados a los diferentes *frameworks* como: Aurora, Marathon, Hadoop, etc.

El clúster de Mesos tiene también Zookeeper, un subproyecto para Hadoop que ofrece sincronización, coordinación y planificación de los procesos, así como almacenado de información de los clústeres.

Mesos ofrece tolerancia a errores. Cuando el nodo *master* falla, el sistema se organiza para elegir un nuevo nodo *master* de entre los demás nodos.

Se puede utilizar junto con los orquestadores Docker Swarm y Kubernetes.

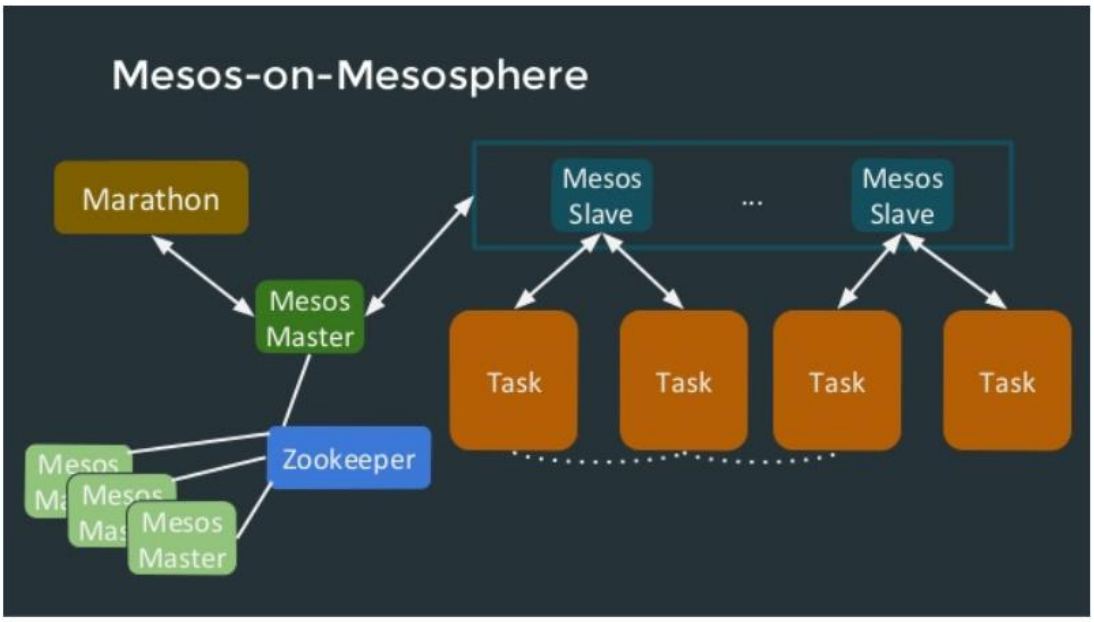


Figura 4. Componentes del clúster de Mesos [5]

Capítulo 3

3. Kubernetes

3.1. ¿Qué es Kubernetes?

Kubernetes es un orquestador de contenedores, que delega en un gestor de contenedores para la creación y gestión de los contenedores. En este proyecto, se hablará de utilizar Kubernetes con el gestor de contenedores Docker, aunque se puede implementar otros contenedores que están en el mercado.

De acuerdo con la página oficial de Kubernetes tenemos esta definición:

"Kubernetes (K8s) es una plataforma de código abierto para automatizar la implementación, el escalado y la administración de aplicaciones en contenedores."

Kubernetes fue creada y utilizada por Google y luego fue donada a la Cloud Native Computing Foundation (CNCF). Se utiliza para la creación, desarrollo, prueba y monitorización de los contenedores permitiendo tener un entorno totalmente automatizado.

Kubernetes es uno de los más destacados a la hora de elegir un orquestador.

3.2. Descripción y funcionalidades

Se debe de tener en cuenta algunos conceptos que nos ayudarán a entender mejor el ámbito donde se va a trabajar, por ello es este capítulo se hablará de conceptos de Kubernetes y sus componentes con algunos ejemplos útiles para poder aplicarlo más adelante, como creando un clúster.

Kubernetes, basándose en los microservicios agrupa los contenedores en *pods* para poder gestionarlos fácilmente. Un *pod* (apartado 3.5.2) es la unidad mínima de ejecución dentro de Kubernetes, siendo una agrupación lógica de uno o más contenedores, dependiendo de la arquitectura propia de la aplicación a desplegar.

3.3. ¿Por qué se utiliza Kubernetes?

Kubernetes permite ajustar aplicaciones a las necesidades actuales, utilizando Docker y permitiendo el orquestado de estos contenedores.

Antes de Kubernetes, Google ya estaba utilizando contenedores para la virtualización, cuando se utilizaba el predecesor de Kubernetes: Borg. Kubernetes hereda de Borg [6] los conceptos principales por los que actualmente se conoce al primero: *pods*, servicios, etiquetas, la existencia de una IP por cada *pod*.

Las características principales se han comentado antes. Para más detalles consultar la referencia (1. Introducción)

Además, Kubernetes ofrece distribución inteligente de contenedores, alta disponibilidad y monitoreo de contenedores: tanto minikube como Kubernetes ofrecen esto. El primero cuenta con el *dashboard* y el en el segundo se puede agregar algunos más sofisticados como Grafana, Prometheus, etc.

3.4. Manifiestos de Kubernetes

Los manifiestos en Kubernetes son archivos escritos en YAML y en los que se definen los diferentes elementos que se pueden desplegar en un clúster de Kubernetes.

3.5. Conceptos básicos de Kubernetes

3.5.1. Nodos

Para construir un clúster de Kubernetes se necesitan tener nodos, como mínimo un nodo *master*. Los nodos (anteriormente llamado *minions*) pueden ser máquinas virtuales o máquinas físicas y en ellos es dónde Kubernetes va a desplegar los *pods* que contienen los contenedores con la aplicación.

El nodo *master* es el encargado de distribuir los *pods* entre todos los nodos *worker* y *master* que componen el clúster.

3.5.2. Pods

Un *pod* [7] es la unidad mínima de ejecución en Kubernetes. Pueden ejecutarse uno o más contenedores en un *pod*. Los contenedores de un *pod* comparten IP y pueden compartir almacenamiento de forma sencilla. Kubernetes garantiza que los contenedores que están en un mismo *pod* se ejecutarán en la misma máquina, es decir, en un mismo nodo. Los contenedores dentro de un *pod* se comunican por *localhost* porque comparten la misma red.

Si un *pod* tiene una carga de trabajo muy elevada, para no sobrecargar los recursos del *pod*, es recomendable crear otro *pod* con la aplicación o crear un *pod* con la réplica de la aplicación en otro nodo *worker*, pero no es recomendable replicar la aplicación en un mismo *pod*. Se pueden ejecutar múltiples contenedores en un *pod* siempre que la aplicación necesite de otro contenedor para poder funcionar. No se debe olvidar que si se sobrecarga a un *pod* con varias tareas el concepto de microservicios se perdería. Normalmente se tiene un contenedor por *pod*, pero también hay casos en que se utilizan más contenedores por *pod*.

A continuación, se presentan algunos casos de uso de *pods*, que proporcionan un entorno similar a la ejecución de múltiples procesos en una VM:

- Ejecución de servidores web que generan logs que luego se almacenan en un clúster, parecido a fuentd o logstash.
- Ejecución de servidores web que sirve datos desde un directorio y un proceso que lee datos del sistema de archivos de clúster.
- Ejecución de funciones de procesamiento definidas por el usuario incluido almacenamiento.

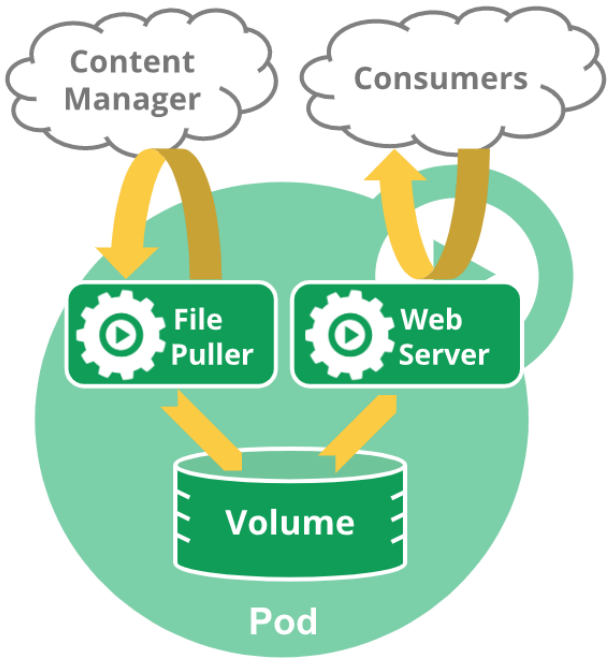


Figura 5. Un pod multi contenedor en donde se comparte el almacenamiento entre esos contenedores

En el diagrama de la Figura 5 se ve como un *pod* puede en algunos casos necesitar de un *"helper container",* en este caso *File Puller*. En este caso, estos dos contenedores nunca actuarán uno independiente del otro y por ello es conveniente agruparlos en un mismo *pod*.

Los manifiestos en Kubernetes se utilizan para definir diferentes elementos en un cluster, como puede ser un Pod. El siguiente fichero es un manifiesto para la definición de un *Pod* con una imagen de nginx. Para más detalles sobre los manifiestos, consúltese la sección 3.4.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.16
    ports:
    - containerPort: 80
```

Como se puede ver, para la definición de un pod es necesario especificar un nombre de una imagen (en el caso anterior, la imagen nginx:1.16 se descarga desde el Docker Hub). También se especifica un nombre para identificar al pod ("nginx"). Por último, es posible especificar uno o más puertos que serán expuestos en el contenedor (en el ejemplo anterior, se expone el puerto 80 de nginx).

3.5.3. ReplicaSets

Un *pod* puede fallar. En ese caso, podrá reiniciarse por Kubernetes, pero a veces no puede ser reiniciado y ese caso la aplicación dejará de funcionar y habrá que volver a desplegarlo manualmente. Por ello existe el mecanismo de *replica sets*, en el cual se puede configurar que un *pod* tenga el número de réplicas deseadas y Kubernetes se asegura de que exista ese número de réplicas en ejecución en todo momento.

Si se configura un replica set con una sola réplica, Kubernetes se asegura de que cuando el pod muera, Kubernetes va a iniciar uno nuevo para reemplazarlo. Esto último es la diferencia entre la definición un replica set de una sola réplica y de un pod aislado.

A continuación, se muestra un ejemplo de manifiesto para la definición de un replica set de tres réplicas de un servidor nginx:

```
kind: ReplicaSet
metadata:
  name: replicaset-example
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.10
```

3.5.4. Deployment

Un *deployment* es un objeto de Kubernetes que proporciona una capa de gestión por encima de los *pods*. Es la unidad de más alto nivel que se puede gestionar en Kubernetes, en un manifiesto de *deployment* se pueden indicar los pods o réplicas que se quieran desplegar en el clúster.

Los *deployments* entre una de sus funciones permite el control de la salud de los *pods*, reiniciando los que están parados o reemplazando los que están caídos, escabilidad de pods (a través de réplicas), actualizaciones continuas y despliegues automáticos, como son los Rolling updates o Rolling backs sin tiempo de latencia.

Un deployment se crea, al igual que un *pod*, *replica set* o servicio, con un manifiesto cuyo *Kind* se especifica como “Deployment”.

```
kind: Deployment
```

A continuación, se muestra un ejemplo del manifiesto YAML para la definición de un *deployment*.

```
apiVersion: v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Es este manifiesto se crea el deployment con un replicaSet asociado de la aplicación con los pods indicados, en este caso 2.

Resultado después de aplicar el anterior manifiesto de deployment.

```
$ kubectl get all
NAME                                READY   STATUS    RESTARTS   AGE
pod/nginx--8574847b85-9vlkg         1/1     Running   3          8d
pod/nginx-8574847b85-q69b8          1/1     Running   2          8d

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx          2/2     2            2           8d

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-8574847b85        2         2         2       8d
```

3.5.5. Servicios

La utilización de un servicio normalmente es para exponer un *deployment* al exterior, ya que por defecto los *pods* y *deployments* solo están accesibles desde dentro del clúster.

Los servicios permiten que los *pods* sean accesibles desde dentro o fuera del clúster, configurando el servicio por nombre DNS o por IP.

Esta conexión se realiza mediante etiquetas. Es posible asignar a cada *pod* una o más etiquetas que podrán ser usadas en el servicio para seleccionar los *pods* o *deployments* a los que aplica el servicio.

Los *pods* tienen un ciclo de vida generalmente cortos, los *pods* mueren, se paran, se vuelven a crear, son efímeros, por el contrario, los servicios son un objeto de larga duración en Kubernetes.

Los servicios tienen una IP y un puerto constante, y se utilizan para asociarlo con un *pod* o a varios, esto depende de dos valores: Los *label* ubicados en el manifiesto del *Pod* y los *selector* en el manifiesto del Servicio, como se verá en los ejemplos de cada tipo. Los *labels* son la etiqueta

que se puede añadir en el manifiesto a un *Pod* y los *selectors* están en el manifiesto del servicio y ambos deben coincidir para poder asignar un *pod* al servicio y así poder exponer el *pod*, de acuerdo con las reglas configuradas en el manifiesto.

Existen varios métodos para exponer un servicio. A continuación, se explican los tipos de servicios que Kubernetes ofrece [8] [9] .

Los servicios se especifican con el parámetro type, y puede haber cuatro tipos.

1. ClusterIP

Este método implica que el servicio solo hará accesible el pod dentro del clúster. Es decir, se utiliza cuando algunos microservicios tienen que comunicarse con otros dentro del clúster, pero no necesitan exponerse a tráfico externo.

A continuación, se muestra un ejemplo de definición de un servicio de tipo ClusterIP.

```
apiVersion: v1
kind: Service
metadata:
  name: my-internal-service
spec:
  selector:
    app: my-app
  type: ClusterIP
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
```

Los nodos que se encuentran en el interior del clúster podrán comunicarse con los pods que estén seleccionados por el selector del servicio (aquellos cuya label “app” tenga el valor “my-app”) a través del puerto 80 (port), el cual será redirigido al puerto 80 del pod (targetPort).

2. NodePort

Este método se utiliza cuando se quiere exponer al mundo una aplicación de dentro del pod. NodePort abre un puerto en los nodos del clúster y el tráfico que entre por dicho puerto se reenvía al servicio.
Restricciones para el servicio tipo NodePort:

1. Se deberá escoger un número de puerto para el NodePort, en Kubernetes este número está restringido para usarlo a partir del puerto número 30.000 en adelante hasta 32.767.
2. Solo se puede tener un servicio por puerto.
3. Si la IP del nodo cambia, se tiene que modificar para volver a exponer a todos los nodos.

A continuación, se muestra un ejemplo de definición de un servicio de tipo NodePort.

```
apiVersion: v1
kind: Service
metadata:
  name: my-nodeport-service
spec:
  selector:
    app: my-app
  type: NodePort
  ports:
  - name: http
    port: 80
    targetPort: 80
    nodePort: 30000
    protocol: TCP
```

Al definir este servicio, se expone un puerto (nodePort) en cada nodo (en este caso, el 30.000) que redirigirá al puerto 80 (port) del servicio. A su vez, este puerto del servicio se configurará para que redirija al puerto 80 (targetPort) del *pod*. Los pods seleccionados por el servicio son los que tengan la *label* "app" con el valor "my-app", tal y como se indica por el atributo "selector" del servicio.

A continuación, se presenta una figura de cómo funciona el servicio *Nodeport* del clúster que se tiene dentro una máquina virtual en el ordenador donde se realiza el proyecto.

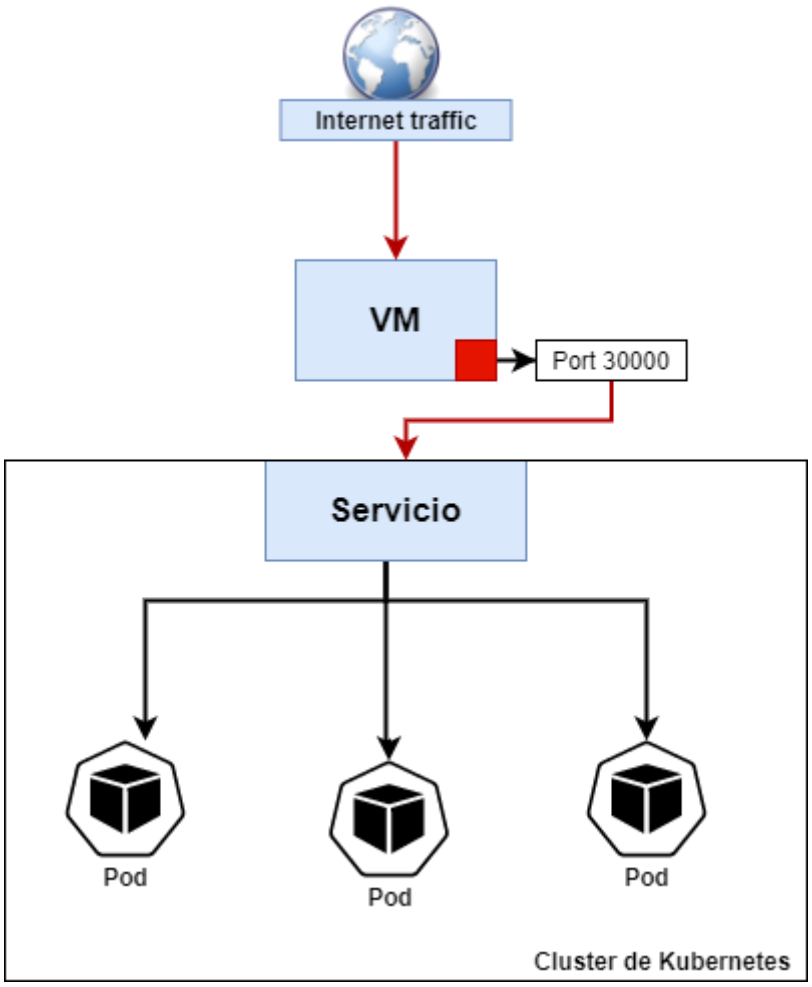


Figura 6. Servicio NodePort

3. ExternalName

Un servicio de tipo ExternalName se define con nombre externo DNS, como se muestra en el ejemplo de abajo.

YAML del tipo de servicio ExternalName.

```
kind: Service
apiVersion: v1
metadata:
  name: miServicio
spec:
  type: ExternalName
  externalName: miservicio.ejemplo.com
```

Cuando se define un *externalName*, es el caso en el que no se usa el selector, ya que se utiliza este valor definido miservicio.ejemplo.com.

4. LoadBalancer

Un servicio LoadBalancer expone un servicio a internet. Este tipo de servicio se utiliza cuando alguna plataforma cloud lo soporte. La plataforma (AWS,GKE,etc..) proporciona una IP al LoadBalancer y el tráfico que llega a ella va al servicio expuesto, según que puertos se hayan asignado, y puede enviarle cualquier tipo de tráfico HTTP,UDP,TCP, etc.

El problema con aplicar LoadBalancer a un proyecto es el coste, por ello es conveniente asegurarnos de qué servicios se quieren exponer para asegurarse de que haya los mínimos posibles.

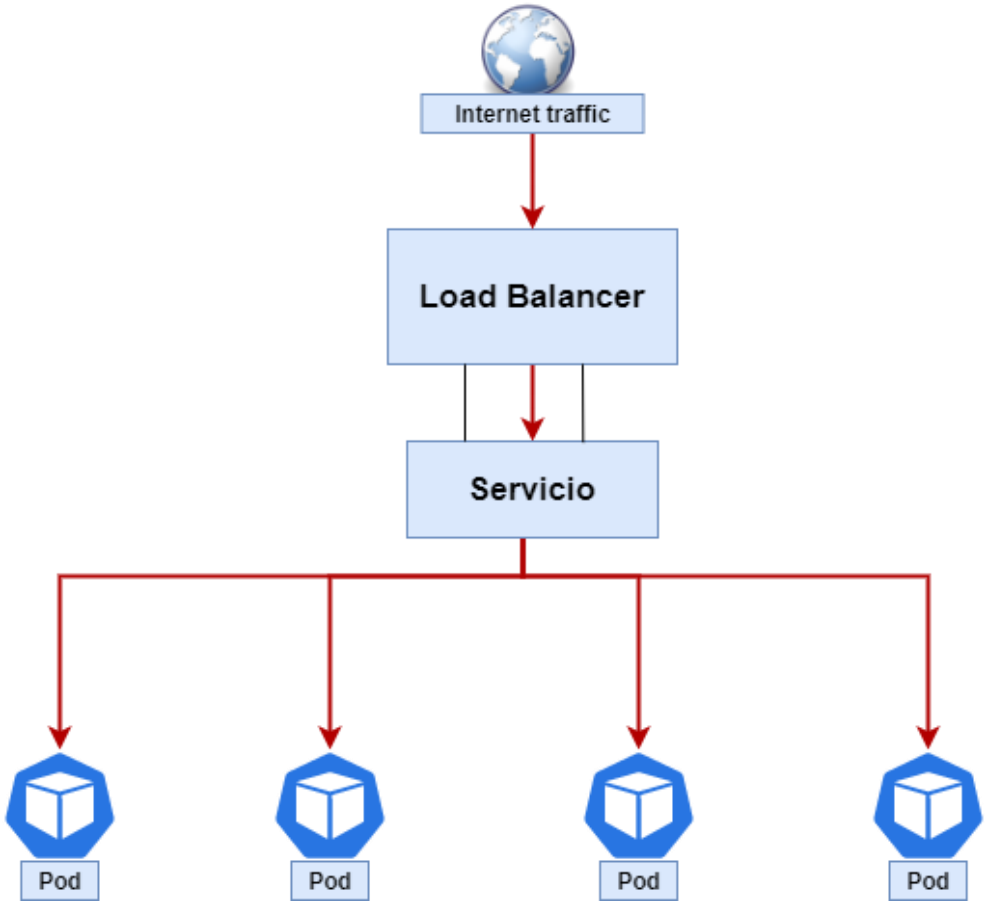


Figura 7. Se muestra el LoadBalancer en la forma estándar de exponer un servicio a internet.

Ejemplo de cómo aplicar el servicio *LoadBalancer* en el manifiesto:

```
apiVersion: v1
kind: Service
metadata:
  name: servicio-load-balancer
spec:
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

3.5.6. Conexión entre los contenedores

Los contenedores que se encuentran en un *pod* se pueden comunicar y compartir información a través del host "*localhost"*, ya que se encuentran en la misma IP.

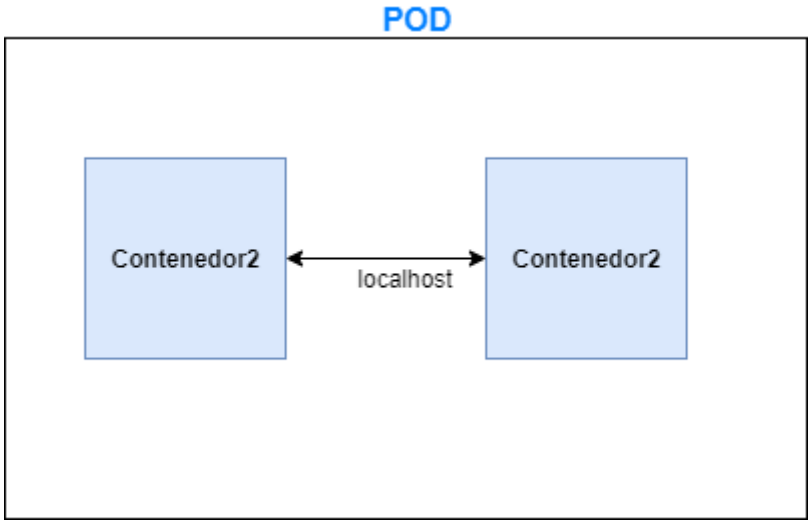


Figura 8. Conexión entre contenedores en un Pod

Aunque en la práctica es muy poco recomendable que se tengan dos contenedores o más en un *pod*, hay situaciones en las que esto puede ser inevitable. No se recomienda ya que si algún contenedor fallase sería más difícil saber cuál es la razón de ello. Debido a que Kubernetes fomenta la utilización de una arquitectura de microservicios, se recomienda mantener en pods diferentes aquellos contenedores que sean independientes, con el fin de conseguir un bajo acoplamiento.

Las aplicaciones dentro de un pod usan la misma red, es decir misma IP y mismo rango de puertos. Las aplicaciones que en ella se encuentren deberán coordinarse para el uso de estos puertos.

3.5.7. ConfigMaps

ConfigMaps en Kubernetes es como un mapa, una tabla Hash que se utiliza para pasar datos de configuración en forma de clave-valor al pod.

Existen dos formas de crear un configmap: con la línea de comando y dentro de un manifiesto del pod.

Para crear directamente objetos de Kubernetes también se puede realizar (aparte de manifiestos) el comando **Kubectl** para realizar las acciones, se explicará como instalará en el apartado 4.3.3 más adelante.

```
kubectl create configmap <configMapName> --from-literal=<key>=<value>
```

A continuación, el mismo ejemplo en formato YAML.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
```

Estas configuraciones de configMaps tienen varios usos, configurar valores para variables de entorno, definir comando que debe ejecutar un contenedor, configurar archivos de config en un volumen.

Se puede consultar los valores de un ConfigMap de esta forma:

```
$ kubectl describe configmaps <Nombre>
```

Ejemplo [10] de caso de uso del configMap del ejemplo anterior en un pod, añadiendo valores a una variable de entorno.

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      envFrom:
        - configMapRef:
            name: env-config
  restartPolicy: Never
```

Al ejecutar el *pod*, la salida mostrará esta línea:

```
log_level=INFO
```

3.5.8. Namespaces

Un *namespace* es en Kubernetes un entorno de trabajo, dónde se tienen determinados usuarios asignados a ese equipo o proyecto. Se crean en un *namespace* el clúster con los *pods*, servicios, *deployments* que se requiera para ese clúster.

Kubernetes se pueden configurar varios *namespaces*. Normalmente, se utiliza el *namespace* por defecto, llamado "default", para desplegar los *pods* pero esto se puede cambiar y tener varios *namespaces*. Esto facilitará al administrador la gestión de los elementos del clúster debido a que sería posible organizarlos de forma lógica en espacios de nombres.

Kubernetes tiene tres *namespaces* preexistentes llamados kube-system y kube-public. Estos componentes son utilizados por Kubernetes para los componentes de uso interno del cluster:

- **Default:** en este namespace se despliegan los pods, servicios, etc. Será el namespace donde se trabajará normalmente con el clúster.
- **Kube-system**: Kubernetes utiliza este *namespace* para gestionar sus componentes
- **Kube-public**: Este *namespace* es reservado para uso interno del clúster.

Los *namespaces* se pueden utilizar según la necesidad que se tenga depende del proyecto. Pueden usarse para tener, dentro del mismo clúster, elementos que correspondan a diferentes entornos (desarrollo, producción, etc.) del proyecto o tener diferentes aplicaciones dentro del mismo clúster.

Se pueden crear diferentes namespaces, principalmente en producción, para separar recursos de otros. Esta práctica es muy utilizada para no modificar accidentalmente recursos de otro namespace.

Algunos de los comandos básicos para la gestión de *namespaces* se listan a continuación:

- Lista todos los namespaces del clúster

```
kubectl get namespaces
```

- Lista todos los pods del namespace de Kubernetes llamado "kube-system"

```
kubectl get pods -n kube-system
```

- Obtiene los pods de los diferentes namespaces si hubiese.

```
kubectl get pods --all-namespaces
```

El siguiente fragmento es un manifiesto que se puede utilizar para definir un nuevo namespace en un clúster de Kubernetes:

```
apiVersion: v1
kind: Namespace
metadata:
  name: <nombreDelNamespace>
```

También se puede añadir un namespace utilizando el siguiente comando:

```
kubectl create namespace <nombre del namespace>
```

3.5.9. Variable de entorno de contenedores

En Kubernetes se pueden añadir las variables de entorno que necesite la aplicación.

Una forma de añadir las variables de entorno a un contenedor es añadiéndolo en el manifiesto:

```
spec:
   containers:
   - name:  <nombre>
     image: <URL de la imagen>
     env:
     - name: <PATH_APP>
       value: <name>
```

3.5.10. Labels y selectors

Las etiquetas o labels son pares claves/valor asociados a los pods y servicios. Se utiliza para enlazar un *pod* con un servicio, elegimos un *label* en el *pod* y un *selector* en el *pod*, estos valores deben coincidir. Un *pod* puede tener varios *labels* según se quiera añadir. Igual en los *selectors*.

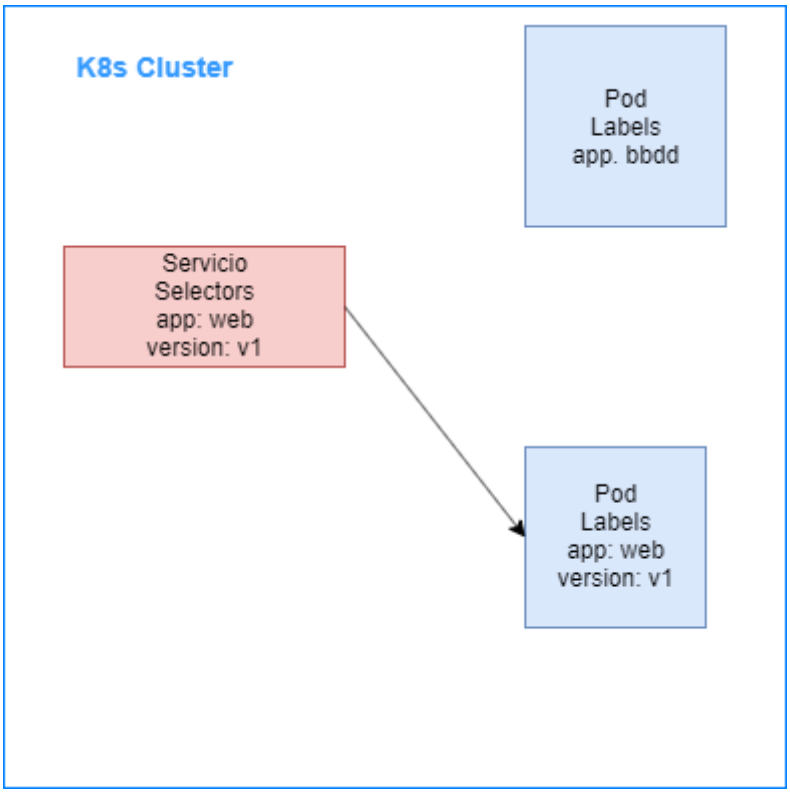
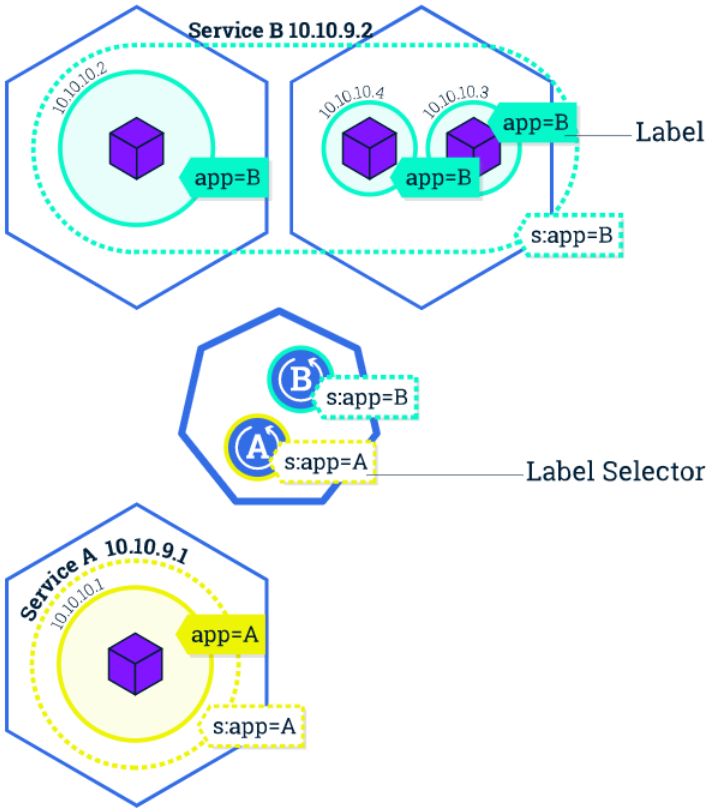


Figura 9.Diagrama donde el servicio escoge el pod, donde coinciden los labels con el selector que tiene configurado.

También se puede ver otra imagen de label y selector que se refleja en la página oficial de Kubernetes.



[11]

Figura 10. Imagen donde se refleja un servicio que se asocia a un pod con el correspondiente label.

El valor de *label o selector* se puede añadir o cambian en cualquier momento.

3.5.11. Container lifecycle hooks

Kubernetes proporciona para los contenedores el framework Container lifecycle hook para ejecutar código que se configura como eventos para que el contenedor lo realice durante su ciclo de vida. Es decir, los contenedores conocen información acerca de los eventos (tareas) que deben realizar antes de finalizar (estado terminated o completed).

Hay dos Hooks de contenedores. El prestart y el prestop [12]. El primero se ejecuta después de crear un contenedor y el prestop antes de que finalice el contenedor.

Ejemplo de aplicación en el ciclo de vida en un contenedor:

```
lifecycle:
  preStop:
    exec:
      command: [
        # Gracefully shutdown nginx
        "/usr/sbin/nginx", "-s", "quit"
      ]
```

3.5.12. Addons

Se trata de una configuración que extiende las funcionalidades de Kubernetes. Ofrece una lista de servicios que se pueden activar o desactivar, según se quiera hacer uso. La lista de las funcionalidades que ofrece minikube

Minikube es un clúster de Kubernetes de un solo nodo donde el nodo *master* hace también se nodo *worker*. Es recomendable usar este clúster especialmente para pruebas o cuando se está empezando a probar Kubernetes por primera vez.

```
$ minikube addons list
- addon-manager: enabled
- dashboard: enabled
- default-storageclass: enabled
- efk: disabled
- freshpod: disabled
- gvisor: disabled
- heapster: disabled
- ingress: enabled
- logviewer: disabled
- metrics-server: enabled
- nvidia-driver-installer: disabled
- nvidia-gpu-device-plugin: disabled
- registry: disabled
- registry-creds: disabled
- storage-provisioner: enabled
- storage-provisioner-gluster: disabled
```

Algunos de estos añadidos vienen ya desactivados por defecto, y se pueden habilitar para poder hacer uso de ellos. Se va a destacar dos de ellos: *dashboard* y *metrics-server*.

Metrics-server nos permite ver la memoria y la CPU que ocupa cada contenedor de un *pod* en cada momento (normalmente se actualiza cada 1 minuto), esto es útil para poder definir limits y requests.

Comando para ver el uso, después de habilitar metrics-server

```
kubectl top pod o kubectl top node
```

Dashboard permite tener toda la información del clúster en modo gráfico, con el siguiente comando abre un navegador, que representa el clúster con el que estamos trabajando en ese momento, se pueden ver cuando *namespaces* se tienen, *pods*, *deployments*, etc,

Con el siguiente comando se abre el *dashboard* [13] de minikube para poder monitorizar todos los recursos de este clúster.

```
minikube dashboard
```

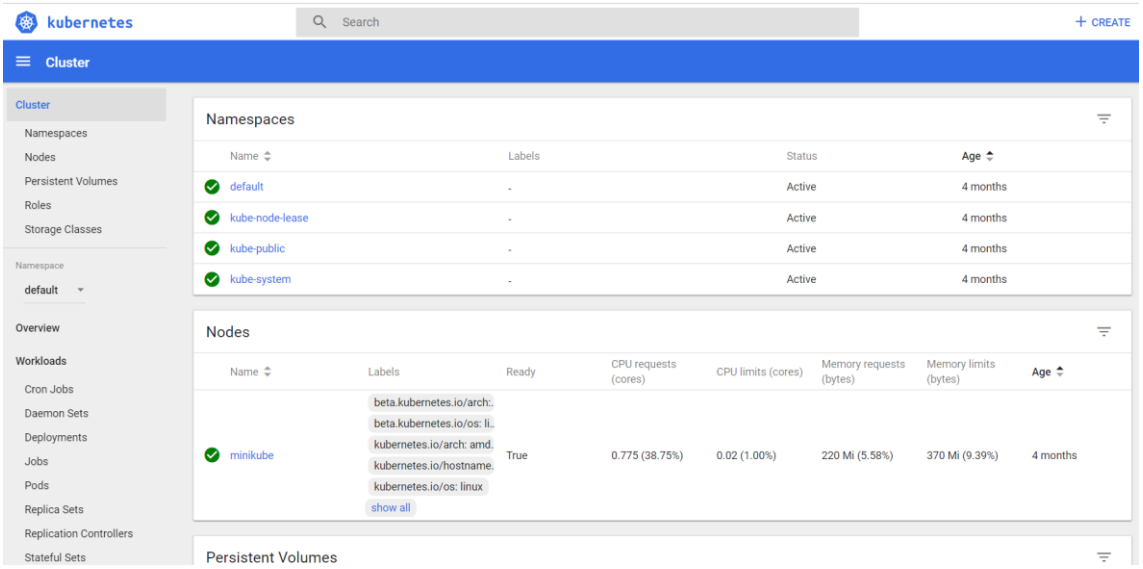


Figura 11. Dashboard en minikube

Se puede apreciar de manera gráfica todos los recursos del clúster y si algunos de sus componentes están teniendo una sobrecarga de memoria, un *pod* está en fallo, servicios, es decir, se puede monitorizar el clúster de minikube.

3.6. Conceptos avanzados de Kubernetes

Esta sección trata de conceptos más allá de los básicos de Kubernetes como son los *pods*, servicios y *deployments*. A continuación, se van a mencionar los conceptos más relevantes.

3.6.1. DaemonSets

Un DaemonSet es una característica que ofrece Kubernetes y que garantiza que existan copias de los pods ejecutándose en los nodos. Si el clúster va añadiendo nodos, también se crearán copias de los pods en los nuevos nodos y si el clúster va eliminando nodos, las copias también de eliminarán. Algunos casos de uso de los DaemonSet:

Se utilizan para implementar tareas continuas en segundo plano que se necesite ejecutar en todos los nodos o en casi todos y que no requieran intervención del usuario. Por ejemplo:

1. Un proceso de almacenamiento en el clúster, como glusterd, ceph, en cada nodo.
2. Un proceso de recolección de logs en cada nodo, como fluentd o logstash.
3. Un proceso de monitorización de nodos en cada nodo, como Prometheus Node Exporter, Sysdig Agent, collectd, etc.

Manifiesto de tipo DaemonSet. [14]

```
apiVersion: v1/beta2
kind: DaemonSet
metadata:
  name: fluentd
spec:
  selector:
      matchLabels:
        name: fluentd -> se selecciona el pod donde se ejecutará el DaemonSet
  template:
    metadata:
      labels:
        name: fluentd
    spec:
      nodeSelector:
        type: prod
      containers:
      - name: fluentd
        image: gcr.io/google-containers/fluentd-elasticsearch:1.20
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi                                                    [15]
```

Explicación del manifiesto de este ejemplo de DaemonSets.

Los DaemonSet con similares a los deployments, ya que se encargan de mantener pods ejecutándose, pero se recomienda el uso de deployments para servicios sin estado como las interfaces de usuario, el escalado de réplicas, etc, utiliza un DaemonSet cuando un pod se debe ejecutar en cada nodo y se necesite que arranque antes del resto de pods.

En el ejemplo, se realiza la recolección de logs en los nodos usando fluentd.

Se crea un DaemonSet con el nombre de fluentd y un pod con la etiqueta(label) de fluentd.

Un selector de nodo de *type: prod* donde se ejecutará el DaemonSet. Se puede omitir este campo, esto significaría que el DaemonSet se ejecutaría en todos los nodos del clúster.

Se ejecutar *fluentd-elasticsearch* en la versión 1.20

El contenedor solicita 100 m de CPU y 200 Mi de memoria, y se autolimita a 200 Mi totales de uso de memoria.

3.6.2. Hpa

Horizontal Pod Autoscaling se refiere a ir añadiendo más nodos/pods al clúster, mientras que el escalado vertical se refiere a ir añadiendo más capacidad asignada.

Como de trata de un escalado de forma automática, existen métricas que se puede utilizar y gestionar para realizar el auto escalado como pueden ser la CPU, memoria, espacio requerido.

Este auto escalado se puede utilizar también se forma que los nodos vayan disminuyendo según las reglas que se apliquen, o cuando el siguiente caso, el uso de CPU, haya disminuido una cantidad significativa.

Ejemplo de un autoescalado horizontal en un YAML

```
kind: HorizontalPodAutoscaler
metadata:
  name: deployment-hpa
spec:
  maxReplicas: 8
  minReplicas: 2
  scaleTargetRef:
    kind: Deployment
    name: gcs-project
  targetCPUUtilizationPercentage: 50
```

En este ejemplo se tiene la regla de autoescalado en el manifiesto del pod, que al alcanzar un uso superior de CPU del 50% se van creando un máximo de 8 instancias del pod y un mínimo de 2 instancias.

A continuación, se realiza el mismo ejemplo del YAML, en formato consola.

```
kubectl autoscale deployment deployment-hpa --cpu-percent=50 --min=2 --max=8
```

Por regla general, no se autoescalan pods directamente, se hace uso de deployments o replicaSets que a su vez cambiarán el número de réplicas de los pods.

Se pueden ver los HPA según cuantas reglas se hayan definido en el clúster con el comando:

```
kubectl get hpa
kubectl describe hpa <NombredelAutoscale>
```

3.6.3. Ingress

Ingress es un punto de entrada a un clúster, es decir, es como servicio que nos permite acceder al clúster desde fuera de él. Expone HTTP y HTTPS y el tráfico se puede controlar mediante reglas que se definan en el manifiesto de Ingress. Si se desea utilizar otro protocolo, se debe utilizar los servicios NodePort o LoadBalancer.

Se va a diferenciar dos conceptos recuso Ingress y controlador Ingress.

1. **El recurso *ingress*** es una definición de reglas que indica como se enrutan los servicios hacia el exterior.

   Reglas de entrada para *ingress*

   a. Un host opcional. En este ejemplo, no se especifica ningún host, por lo que la regla se aplica a todo el tráfico HTTP entrante a través de la dirección IP especificada. Si se proporciona un host (por ejemplo, fleetman.com), las reglas se aplican a ese host.
   b. Una lista de rutas (por ejemplo, /testpath), cada una de las cuales tiene un backend asociado definido con una serviceName y servicePort. Tanto el host como la ruta deben coincidir con el contenido de una solicitud entrante antes de que el equilibrador de carga dirija el tráfico al Servicio al que se hace referencia.
   c. Un backend es una combinación de Servicio y nombres de puerto. Las solicitudes HTTP (y HTTPS) al Ingress que coinciden con el host y la ruta de la regla se envían al backend listado.

   ```
   backend:
       serviceName: testServicio
       servicePort: 80
   ```

2. **El controlador ingress** es un contenedor que enruta las peticiones hacia los servicios correspondientes en base a la definición del recurso ingress en el manifiesto. [16]

Ejemplo de Ingress usando minikube (clúster de un solo nodo)

Con el siguiente comando se activa uno de la lista de addons que tiene minikube para utilizar (ver apartado 3.5.12).

```
addons ingress enabled
```

Para definirlo se hace uso de un manifiesto donde se escriben las reglas para este *ingress controller* que se pondrá de ejemplo.

Un ejemplo de tipo *Ingress* a continuación:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  rules:
  - http:
      paths:
      - path: /testpath
```

```
        backend:
          serviceName: test
          servicePort: 80
```

En este ejemplo, la regla es redireccionar cualquier petición del puerto 80 del servicio test a la carpeta (testpath).

Para obtener Ingress que se está corriendo en el clúster, es necesario el comando:

```
kubectl get ingress
```

3.6.4. Ingress control

Un recurso *ingress* declarado por su cuenta no tiene efectoen minikube, para ello se debe de activar el controlador *ingress*. Existe varios controladores de *ingress*, el más utilizado es *ingress-nginx* para *cloud*.

En minikube un *ingress control*, es un *addons*, y se activa de la siguiente manera:

```
$ minikube addons list
- ingress: disabled
$ minikube addons enable ingress [17]
* ingress was successfully enabled
```

A continuación, comprobar que se ha creado un pod para el ingress en el namespace kube-system (se crean en ese namespace por defecto) y que se encuentre en estado running.

```
$ kubectl get po -n kube-system

default-http-backend-6864bbb7db-857nc       1/1     Running
nginx-ingress-controller-586cdc477c-lrgp7   1/1     Running
```

A continuación, un listado de algunos Ingress controllers para plataformas cloud.

- AWS ALB este es el Ingress Controller que habilita el Ingress usando Application Load Balancer de Amazon.
- AKS Application Gateway Ingress Controller es el ingress controller que habilita ingress a clústeres de AKS usando Azure Application Gateway.
- Istio based ingress controller Control Ingress Traffic.
- NGINX, Inc. Ofrece soporte y mantenimiento para Nginx Ingress Controller para Kubernetes.

Para listar el número de *ingress* que se tiene en el clúster y ver las reglas que cada una define:

```
kubectl get ingress --all-namespaces
kubectl describe ingress <nombre>
```

El uso de Ingress en una plataforma cloud

Cuando se tiene un clúster desplegado en una plataforma *cloud* como AWS, GCP, Azure, etc *Ingress* ofrece una solución que consiste en utilizar *load balancers (*ver más adelante en el apartado 3.6.8*)*, ofreciendo mayor disponibilidad, resiliencia y tolerancia a fallos.

En el siguiente diagrama, se utiliza un *ingress controller* para hacer un enrutamiento de dónde dirigir el tráfico que llega en el *load balancer* a varios servicios, esos servicios que se enlazan al *ingress* se han definido en el manifiesto del recurso *Ingress*.

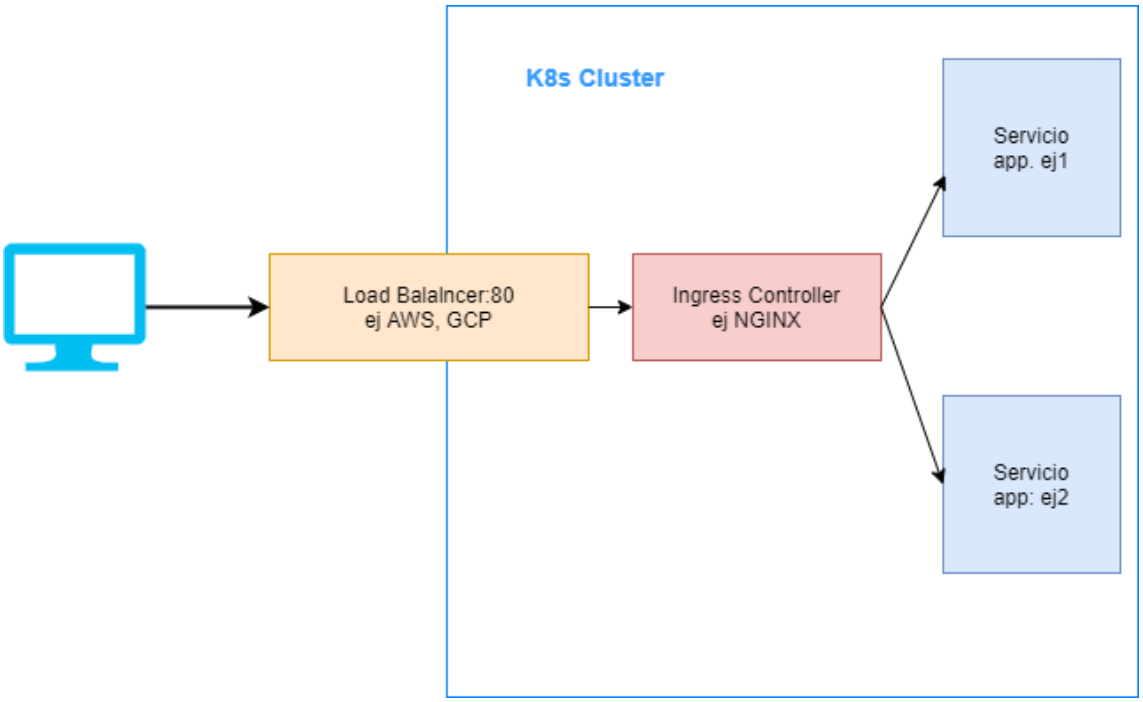


Figura 12. Imagen donde se muestra la utilidad de ingress, cuando te tienen varios servicios con salida a internet.

Con *Ingress* se puede reducir el coste de crear varios *load balancers*, creando solo uno para dar salida a internet, ya que *ingress* puede enrutar a varios servicios, se puede tener varios *ingress* para un mismo clúster.

3.6.5. Jobs

Un *job* en Kubernetes es un controlador que se encarga de que uno o más *pods* duren un corto período de tiempo, es decir, se asegura que un grupo de *pods* que contengan alguna tarea finalicen. Dada la naturaleza de los *pods* que se vuelven a reiniciar una vez finalizados hasta que se eliminen de forma manual, un *job* se encarga de que no vuelva a reiniciarse, quedando el estado del *pod* en "*completed*". Para poder asegurarse de que un *pod* no vuelva a reiniciase una vez terminada la actividad, existe en el manifiesto un valor por defecto, *restartPolicy=Always*. Este valor podemos cambiarlo a *Never* o *On failure* [18].

Los Jobs se pueden ejecutar en múltiples *pods* añadiendo en la cantidad de *pods* que deben realizar la tarea, por defecto los *pods* se van creando secuencialmente.

Un ejemplo de manifiesto de un *Job* con un *pod.*

```
kind: Job
metadata:
  name: suma
spec:
  template:
    spec:
      containers:
      - name: suma-matematica
        image: ubuntu
```

```
        command: ["expr", "3", "+", "2"]
      restartPolicy: Never
```

Este ejemplo de Job realiza una suma, se ejecuta con el comando:

```
kubectl create -f nombre-manifiesto.yaml
```

Y luego, para visualizarlo

```
kubectl get jobs
```

Para ver el resultado que deja, se puede consultar el log del pod, que este caso será “5”.

```
kubectl get logs <nombre-pod>
```

Por último, una vez completada la tarea, se puede borrar el Job, al eliminar un job también se elimina el pod que creó para ello.

```
kubectl delete job <nombre-job>
```

En casos reales de producción, los Jobs se utilizan para, por ejemplo, procesar imágenes, generar informes, enviar emails, realizar cálculos analíticos, etc.

Otra característica para tener en cuenta en que los pods se pueden iniciar paralelamente, sin esperar que un pod finalice para empezar el trabajo el siguiente pod, es con la propiedad que ofrece dentro del manifiesto.

```
spec
   completions=3   -> número de pods
   parallelism=3
```

3.6.6. Cronjobs

Un cronjob es un job que puede ser programado, como un crontab en Linux. La diferencia es que, un job se ejecuta cuando creas el job y cronjob puede ser programado para ejecutarse periódicamente.

Manifiesto de un cronjob

```
kind: CronJob
metadata:
  name: report
spec:       -> spec del cronjob
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:         -> spec del job
       completions=3   -> número de pods
       parallelism=3
       template:
           spec:        -> spec del pod
             containers:
              - name: reporting-tool
                image: reporting-tool
                command: ["expr", "3", "+", "2"]
                restartPolicy: Never
```

El ejemplo es el mismo que el del *job*, adaptado para que sea de tipo *cronjob*.

El comando para ver el *cronjob* es:

```
kubectl get cronjob
```

En este ejemplo de *cronjob*, se crea un *job* y tres *pods* en un tiempo especificado.

3.6.7. Secrets

Secrets utilizado para almacenar información delicada como contraseñas, y los guarda en un formato encriptado.

Existen múltiples formas de crear *secrets*, desde la línea de comando, pasando un archivo cuando se quiera almacenar más de una contraseña o usando manifiestos.

```
kubectl create secret generic ssh-key-secret --from-file=ssh-
privatekey=/path/to/.ssh/id_rsa --from-file=ssh-
publickey=/path/to/.ssh/id_rsa.pub
```

Encriptar la información que se quiere guardar usando base64

```
echo -n ‘mysql’ | base64
kubectl get secrets
```

Después de crear, vamos a añadirlo a un pod, con el siguiente fragmento de yaml.

```
envFrom:
  -secretRef:
     name: app-secret
```

De esta forma, estamos añadiendo secret al pod, como si fuese una variable de entorno.

3.6.8. Loads balancers

El load balancer son los encargados de distribuir tráfico entre los distintos nodos, proporcionan un nombre de DNS estable, apuntan generalmente a la instancia del *master*.

Con el *load balancer* se expone el servicio a internet y distribuye la carga del clúster. Es preferible tener un *load balancer* por proyecto, es decir, un *load balancer* para un servicio que cubra todo el proyecto, ya que exponer un servicio puede resultar caro, especialmente si se utiliza las diferentes plataformas cloud que se tiene en el mercado.

Ejemplo de un servicio LoadBalancer:

```
apiVersion: v1
kind: Service
metadata:
  name: my-internal-service
spec:
  selector:
    app: my-app
  type: LoadBalancer
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
```

3.6.9. Recursos requeridos: Requests y Limits

Requests es el espacio de memoria y CPU que necesita el contenedor y *limits* son los límites de memoria y CPU que puede hacer uso, cuando se dan esos datos del pod, Kubernetes (*scheduler*) lo usa para decidir en qué nodo es conveniente desplegar el *pod* y kubelet (ver apartado 3.9) se encarga de que el contenedor no utilice más de los límites fijados en el manifiesto.

Un request se utiliza para que el clúster de Kubernetes tenga en cuenta lo que un contenedor especifico necesita en términos de memoria o CPU para que pueda ejecutarse sin llegar a tener problemas. Estas características las define el programador, ya que sabe qué tipo de datos contiene ese contenedor (bases de datos, app, servicios, etc). En orden a lo que sabemos del contenedor podemos dar una estimación de qué necesita para que se implemente sin sufrir falta de memoria o CPU por ejemplo.

```
spec:
     containers:
     - name: queue
       image: release2
       resources:
           requests:
               memory: 300Mi  -> Megabytes
               cpu: 100m  -> milicores
           limits:
               memory: 500Mi
               cpu: 200m
```

En cuanto a los Limits, es importante destacar que si el uso de **memoria** de un contenedor se excede al límite fijado en el fichero .yaml el contenedor va a ser borrado (*killed*). El *pod* al contrario seguirá e intentará reiniciar ese contenedor muerto cada vez que se exceda el uso de memoria. Por el contrario, si un contenedor excede el uso de **CPU** establecido, este contenedor no será borrado, el contenedor seguirá corriendo, pero sujeto a una limitación de uso de CPU que no sobrepase el límite establecido, no se reiniciará solo se limitará.

3.6.10. Rollbacks y rolling updates

Un deployment es una unidad de alto nivel para gestionar las replicasSet y además ofrece una característica que permite el uso de rollback y Rolling updates.

Por ejemplo, el *rollback* se utilizará solo en caso de emergencia, cuando un error haya ocurrido y no se tenga otra opción que volver hacia atrás a la última versión que estuvo en funcionamiento.

Por otro lado, también se puede ir a alguna versión superior, a esto se le llama Rolling update.

Los siguientes comandos se pueden utilizar para consultar el estado de los rolling updates de los *deployments* del clúster.

```
Kubectl get deployment
kubectl rollout status deployments nombreDelDeployment
kubectl rollout history deploy nombreDelDeployment
kubectl rollout undo deploy nombreDelDeployment --to-revision=2
```

La forma que almacena Kubernetes las replicasSets es de crear una replicaSet según cada versión de la aplicación.

Con el comando `kubectl get all`, se puede ver las replicasSets que se tienen almacenadas y la que se está usando actualmente, será la que tiene alguna cantidad de *pods* corriendo (distinto a ceros como los demás).

Es muy importante tener el manifiesto del *deployment* siempre actualizado, ya que al hacer los *rollouts* manualmente, éste no modifica el .yaml y con el tiempo se perderá el seguimiento de la situación actual del sistema.

Se realizará un escenario acerca de este concepto en el apartado 5.5.

3.6.11. Readiness and liveness probes

Este concepto en Kubernetes es muy importante a la hora de evitar posibles retrasos en nuestras aplicaciones y que los usuarios no puedan percibirlo a la hora de interactuar con las aplicaciones. La manera de configurarlo es muy sencilla y se verán ejemplos de estas.

Kubernetes, al ser una herramienta que se centra en la orquestación de contenedores, no conoce los detalles acerca del funcionamiento de las aplicaciones que se estén desplegando en estos contenedores. Por tanto, Kubernetes no sabrá si una aplicación dentro de un contenedor está funcionando correctamente o si, por el contrario, aún se está iniciando (por ejemplo, un servidor de aplicaciones web puede necesitar minutos para iniciarse) o si la aplicación ha fallado de una forma en la que no puede recuperarse (debido a un error o excepción no controlada).

Kubernetes, por defecto, asume que la aplicación está lista para aceptar trabajo si su contenedor está en ejecución. No obstante, en algunos casos esto puede ser problemático. En los siguientes subapartados se hablará de los problemas mencionados y las herramientas que ofrece Kubernetes para solucionarlos.

Liveness probes

El proceso Kubelet va realizando revisiones periódicas de los *pods* que se encuentran en el clúster y si encuentra que un *pod* no está ejecutándose, los reinicia. No obstante, y como se ha mencionado antes, Kubernetes no puede saber por sí solo si la aplicación que se ejecuta dentro del contenedor ha fallado inesperadamente y no puede aceptar trabajo. Si esto ocurre, el servicio puede estar reenviando solicitudes a un pod que no sea funcional, ocasionando tiempos de espera elevados al usuario.

Para evitar este problema, al igual que en las *readiness probes*, es necesario indicar instrucciones a Kubernetes para que pueda comprobar periódicamente si la aplicación sigue funcionando correctamente. Estas comprobaciones periódicas se conocen como *liveness probes*. Si una *liveness probe* falla, es un indicativo de que la aplicación no está funcionando correctamente y Kubernetes, en este caso, reiniciará el contenedor y con él, la aplicación.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
```

```
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5 -> tiempo que tarda, antes de ejecutar el
livenessProbe
      periodSeconds: 5  -> tiempo en que se ejecute el liveness probe,
cada 5 segundos
```

En este ejemplo, se utiliza una imagen de busy box y se crea una carpeta /tmp/healthy, sleep, se borra la carpeta creada y se vuelve a dormir nuevamente. Mientras esto ocurre el chequeo del *liveness probe*, consiste ver si la carpeta creada existe, pero como el comando del *pod* consiste en borrarla, esto daría error y el *pod* sería reiniciado todo el tiempo ya que no pasa el chequeo del *liveness probe.*

Readiness probes

En algunos casos, como por ejemplo cuando se implementa un escalado horizontal automático, en los que Kubernetes necesita crear nuevos *pods* de manera automática. El servicio que gestiona el balanceo de carga tendrá que distribuir la carga de trabajo (por ejemplo, peticiones HTTP) entre los nuevos *pods*. No obstante, es posible que uno o más de los contenedores de los nuevos *pods* no hayan terminado de iniciarse cuando Kubernetes redirija las solicitudes a estos, lo cual puede ocasionar tiempos de espera elevados o *timeouts* en las solicitudes. Esto impactará sobre la experiencia del usuario.

Como se ha mencionado antes, Kubernetes no tiene forma de saber si una aplicación se ha iniciado, por lo que será necesario indicar "instrucciones" para que Kubernetes pueda saber si un contenedor está listo realmente. Estas instrucciones se ejecutarán, de manera periódica, por medio de las *readiness probes*. Cuando un contenedor no está listo, Kubernetes no envía tráfico a ese contenedor y si no está vivo con el liveness probe los reinicia.

```
spec:
   containers:
     - name: nginx
       image: nginx
       ports:
       - containerPort:80
       readinessProbe:
        httpGet:
           path: /
           port: 80
        initialDelaySeconds: 5 -> tiempo que tarda, antes de ejecutar
el readinessProbe
        periodSeconds: 10 -> cada 10 segundos se ejecuta el readiness
probe.
```

En este ejemplo, se realiza una petición a / para ver si está listo el *pod* nginx.

3.6.12.Role-based access control (RBAC)

RBAC consiste en dar privilegios de clúster, el administrador crea varios roles para poder tener un control de acceso a un Proyecto determinado.

Un administrador puede dar estos roles a los diferentes usuarios, RBAC se presupone que viene habilitado por defecto en el clúster, se puede comprobar con el comando:

```
Kubectl api-versions
```

Con esta función habilitada se puede hacer que usuarios tengan acceso a grupos, servicios, *pods namespaces* o todo el clúster.

En RBAC se utilizan estos cuatro conceptos para crear un rol.

Roles: estos se refieren a establecer permisos dentro de un *namespace*, se debe especificar entre los *namespaces* que están por defecto o los creados por el administrador.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

En este ejemplo [19] se crea un rol para dentro del namespace default, que otorga permisos para la lectura de *pods*.

Cluster roles: Este tipo ClusterRol tiene varios usos, entre ellos, dar permisos a recursos para todo el ámbito del clúster, sin *namespaces*, definir permisos sobre recursos con *namespaces* y ser otorgados dentro de *namespaces* individuales, por último, definir permisos sobre recursos con *namespaces* y ser otorgados en todos los *namespaces*.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" se omite, ya que ClusterRoles, se refiere a todos los
namespaces del clúster
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

En este ejemplo [19], se tiene un clusterRole que otorga acceso de lectura sobre el recurso *secrets* en todos los *namespaces* del clúster.

Clúster Role Binding: otorga permisos definidos en el *ClusterRole* en todo un clúster.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

En este ejemplo, se permite a cualquier usuario del grupo *manager* la lectura de *secrets* en cualquier *namespace*.

Role bindings: este concepto asocia un determinado usuario/grupo a un rol.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: Leslie
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

El RoleBindings está otorgando permisos de lectura de *pods* al usuario Leslie, en el namespace determinado anteriormente para pod-reader, *default*.

Service accounts; consiste en dar permisos de usuario a un pod, deployment, etc. asociándole un service account.

```
subjects:
- kind: ServiceAccount
  name: dns-system
  namespace: kube-system
```

Se escribe en el manifiesto del pod la entrada del ejemplo, para indicarle que use *service account* creado y posteriormente crear el Role binding o cluster role bindig para dar permisos al *service account*.

```
ServiceAccountName: dns -system
```

Entrada añadida al final del manifiesto del *pod*.

3.6.13. StatefulSets

Al crear múltiples réplicas de un pod utilizando un ReplicaSet, el nombre asignado a cada pod por parte de Kubernetes consiste en un nombre proporcionado por el usuario sufijado por una serie de caracteres aleatorios. Utilizando ReplicaSets, la forma en la que un cliente podría acceder a los pods sería a través de un servicio. El hecho de que los nombres de los pods, en este caso, sean aleatorios, es irrelevante ya que es el servicio el encargado de distribuir la carga entre los pods del ReplicaSet. Por tanto, el cliente no necesita saber los nombres de cada pod individual.

No obstante, hay ocasiones en las que será necesario tratar cada pod de forma individual, proporcionándoles nombres predecibles. El propósito de los StatefulSets es permitir la replicación de uno o más pods de forma que a cada uno se le proporcione un nombre de pod predecible. Esto es necesario cuando un cliente necesite acceder a los pods del StatefulSet individualmente.

Uno de los usos de StatefulSets puede ser para la configuración de una base de datos replicada. Si, por ejemplo, se configura un MongoDB mediante un ReplicaSet, se tendrían en realidad tantas bases de datos independientes como número de réplicas creadas. En cambio, con un StatefulSet, se puede definir un nombre predecible para cada una de las réplicas y permitir que cada una ellas sean accesibles por separado mediante un **headless service**. Que cada réplica sea un pod independiente también facilita la distinción, por parte del cliente, del nodo primario de la base de datos (en el cual, normalmente, es donde se tienen que hacer las escrituras) del resto de nodos (secundarios) de la base de datos.

En definitiva, la utilización de un StatefulSet frente a un ReplicaSet tiene las siguientes particularidades:

1. Los pods dentro de un StatefulSet siempre tendrán un nombre previsible. Por ejemplo: pod-1, pod-2, pod-3.
2. Los pods siempre se iniciarán secuencialmente, es decir, hasta que no se inicie el pod-1, no se iniciará el pod-2 y así sucesivamente.
3. Los clientes (usado el headless service) pueden llamar a los pods por su nombre.

Se realizará un ejemplo práctico de esto en el apartado de Escenario 2.

3.6.14. Headless Service

Cuando no se quiere asignar una IP a un servicio o reenviar tráfico, es porque se quiere comunicar con los pods directamente y sabiendo con cuál se está interactuando, este tipo de servicio se llama Headless Service.

Se utilizan en conjunción con los StatefulSets. Se define en el manifiesto de la misma forma que un servicio normal. Un StatefulSet se usa para gestionar aplicaciones con estado.

Sintaxis de llamada ejemplo: mongo-0.mongo

El formato utilizado para referenciar a un pod de un StatefulSet a través de un HeadlessService: Nombre del Pod del StatefulSet + Nombre del Servicio Headless

Como se puede ver en el ejemplo, tiene la misma sintaxis que un servicio. Para Kubernetes, un servicio es un HeadlessService si se asocia con un StatefulSet.

Servicio que selecciona un pod con el label mongo.

```
apiVersion: v1
kind: Service
metadata:
 name: mongo
 labels:
   name: mongo
spec:
 ports:
 - port: 27017
   targetPort: 27017
 clusterIP: None  → Esto no indica que es HeadlessService
 selector:
   role: mongo
```

3.7. Persistencia

Como en cualquier aplicación, algunos *pods* pueden llegar a terminar inesperadamente y pueden llevarse la información con ellos. Por ello existe en Kubernetes formas de persistencia.

3.7.1. Almacenamiento en pods/contenedores

Kubernetes ofrece varias formas de implementar el almacenamiento en los pods, pero se van a detallar tres tipos principales y se mencionarán más tipos.

Sin volumen

Usando el almacenamiento local del contenedor. El almacenamiento se destruye cuando se destruye el contenedor.

Es decir, lo que se guarda en el contenedor tiene una vida ligada a la del contenedor, éste de elimina y la información se destruye. En este caso se carece de persistencia.

Volumen ligado al pod

Está ligado al ciclo de vida del pod. Se borra su contenido cuando el pod deje de estar asignado al nodo (o cuando este se borra). Se define con la opción “emptyDir”.

Ejemplo de aplicación en el manifiesto:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-volumen1
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

En este ejemplo, un pod con un contenedor que monta un volumen en /cache y en emptyDir de nuestro nodo donde se despliega el pod.

Volumen mapeado contra un directorio del worker

Se hace corresponder un directorio local del worker con un directorio del contenedor. El contenido del volumen persiste más allá de la vida del pod pero si el pod es recreado en otro worker, no se tiene acceso a los datos almacenados en el otro worker. Se define con type: Directory y hostpath type: DirectoryOrCreate (crea el dir si no existe).

Ejemplo YAML usando hostpath

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-volumen2
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
      type: DirectoryOrCreate
```

En el ejemplo se tiene el volumen /test-pd que se monta en el directorio /data del nodo donde se ejecuta el pod.

Volúmenes persistentes

Espacio proporcionado por el clúster de Kubernetes a los pods. En este caso se tienen dos definiciones que hace uso para poder crear un volumen persistente [20].

Persistent Volume: es el espacio que provisiona el clúster, en un manifiesto de persistent volumen se pueden ofrecer varios tipos de almacenamiento para cada persistentVolumeClaim como se tenga.

Persistent Volume Claim: es una petición de volumen, que se define en este manifiesto. Puede especificar el tipo de almacenamiento, cuánto espacio desea utilizar o el tipo de acceso (lectura/escritura).

Ejemplo de un pod que hace uso de los volúmenes persistentes

```
spec:
      containers:
      - name: mongodb
        image: mongo:3.6.5-jessie
```

```
        volumeMounts:
          - name: mongo-persistent-storage
            mountPath: /data/db
      volumes:
        - name: mongo-persistent-storage

          persistentVolumeClaim:
            claimName: mongo-pvc
```

En este ejemplo, se tiene un volumen /data/db que utiliza un persistent volume claim llamado mongo-pvc.

Kubernetes ofrece varias opciones de dónde se quiere implantar, volúmenes en local, cloud, entre otros.

A continuación, se van a nombrar algunas de ellas. [21]

- awsElasticBlockStore: define un volumen EBS(Elastic Block Store) en Amazon Web Services en un *pod*. Cuando un *pod* es eliminado, este volumen no se pierde, solo se desmonta, esto significa que cuando se vuelve a crear otro pod los datos del volumen pueden transferirse a ese pod. AWS tiene algunas restricciones como pueden ser:
    - Cada nodo es un EC2, por lo que el volumen solo admite esa única instancia EC2.
    - Las instancias EC2, deben ser de la misma región y zona de disponibilidad.
- azureDisk y azureFile: Azure ofrece AKS (*Azure Kubernetes Service*) donde se asigna volumen y persistencia según las necesidades del clúster. *AzureDisk* se puede usar para crear un recurso de Kubernetes *DataDisk*, este tipo de almacenamiento solo está disponible para un único *pod*, para el almacenamiento compartido entre *pods* se utiliza azureFile.

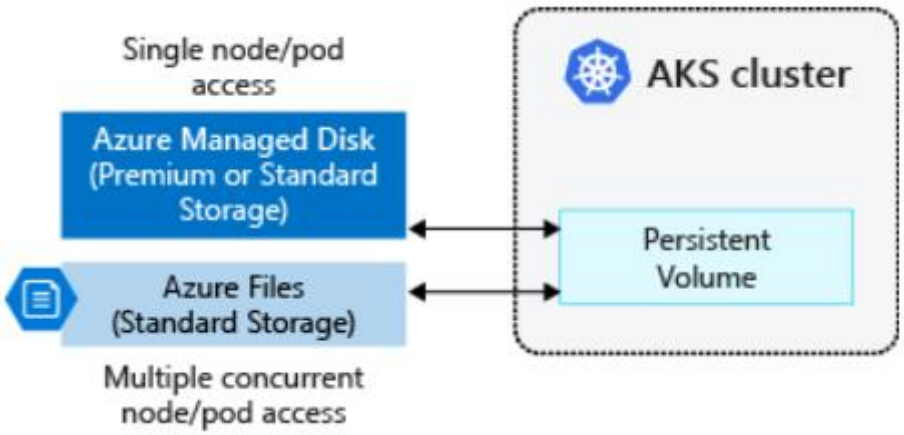


Figura 13. Se muestra la diferencia entre azureDisk y azure files que ofrece Azure para el almacenamiento persistente en un clúster de Kubernetes.

- glusterfs: Para poder hacer uso de este volumen se debe de tener previamente este sistema de archivos en red en ejecución. Este permite crear un volumen Glustefs, dónde los datos se pueden conservar después de haber eliminado el pod y desmontarlo después de esto.
- hostPath: este volumen monta un archivo o directorio del sistema de archivos del host del *pod*.

- cephfs: esto permite que un volumen CephFS existente se pueda montar en el *pod* del clúster. La información del volumen se conserva y solo se desmonta. Puede ser montado por varios pods.
- cinder: se necesita previamente Kubernetes con Openstackcon un proveedor *cloud* configurado. Cinder se utiliza para montar *Openstack Cinder Volume* en un *pod*.
- iscsi: permite montar un volumen iSCSI en un *pod*. Un volumen iscsi se puede rellenar previamente con datos y esos datos se pueden "transferir" entre *pods* y cuando estos se eliminen el volumen se conserva y solo se desmonta. Es necesario tener previamente un servidor propio iscsi ejecutándose para poder hacer uso de este volumen.
- Local: un volumen local se refiere a usar el almacenamiento local y montarlo como un disco, directorio o una partición. Un volumen local todavía no es posible el aprovisionamiento dinámico, solo estático con un PersitentVolume creado.

- nfs: esto permite que un volumen NFS existente se pueda montar en el *pod* del clúster. La información del volumen se conserva y solo se desmonta.
- downwardAPI: este tipo de volumen se utiliza para que los datos del downward API estén disponibles para las apps. Monta un directorio y escribe los datos solicitados.
- gcePersistentDisk: este volumen monta un disco persistente de Google Compute Engine (GCE) en el *pod*. La información del volumen se conserva y solo se desmonta.
  Para este volumen los nodos en donde se encuentran los pods deben ser máquinas virtuales GCE y deben estar en el mismo proyecto y zona horaria.
- rbd: permite montar un volumen Rados Block Device. Este volumen se puede rellenar previamente con datos y esos datos se pueden "transferir" entre *pods* y cuando estos se eliminen el volumen se conserva y solo se desmonta. Es necesario tener previamente una instalación Ceph ejecutándose para poder hacer uso del volumen rbd.
- scaleIO
- secret: este tipo de volumen se utiliza para pasar información confidencial entre los pods. Se pueden guardar en la API de Kubernetes y posteriormente pasar a los *pods* como un archivo, este volumen está respaldado por el sistema de archivos tmpfs.
- storageos: se despliega en Kubernetes como si fuese un contenedor, proporcionando acceso a todo el clúster, mediante un sistema de archivos. Requiere de un Linux de 64 bits y ofrece una licencia gratuita para desarrolladores.

Como se ha mencionado antes, los PersistentVolumen (PV) son los recursos del clúster y los PersistentVolumenClaim (PVC) son las peticiones para esos recursos. Existen dos formas de provisionar a esas peticiones, estática o dinámicamente.

Estática: se puede especificar a qué usuarios del clúster van relacionadas esas PVs con toda la información detallada del almacenamiento real del clúster.

Dinámica: cuando se tiene una petición, el clúster debe ir provisionando dinámicamente de ese volumen al PVC.

Estados que puede presentar un volumen(PV):

- Available: Disponible para reclamación (PVC).
- Bound: espacio ya asociado a un PVC, por lo que no está disponible.
- Released: El PVC del volumen se ha eliminado y está en modo de espera por otro PVC.
- Failed: En estado de fallo (no se ha conseguido asociar a un PVC, por ejemplo: por falta de espacio).

Estos espacios de almacenamiento se implementan en el en *master*.

Ejemplo de cómo funciona la dinámica en Kubernetes

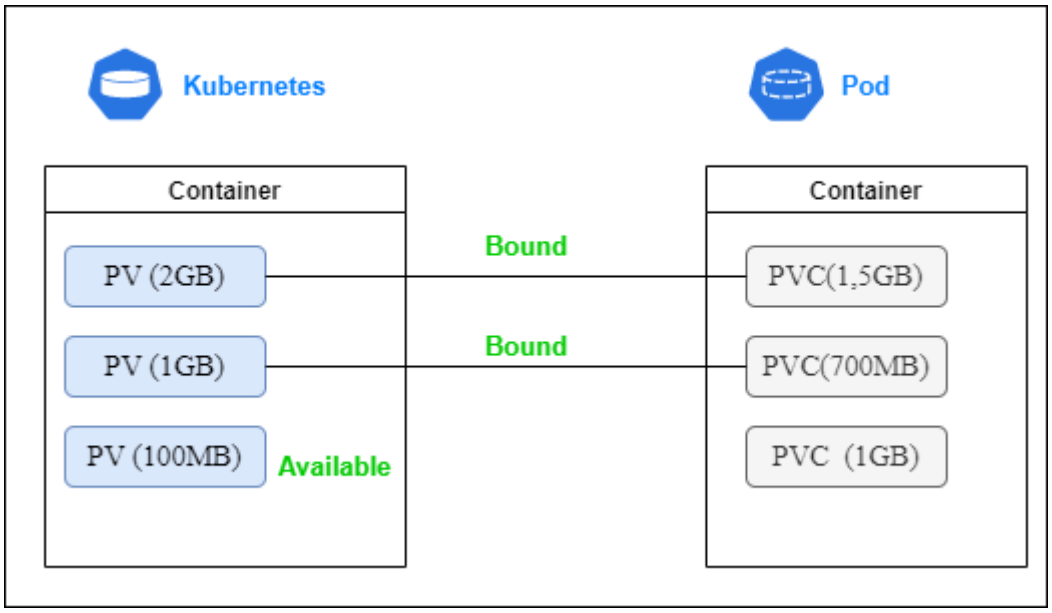


Figura 14. Ejemplo de asociación entre PV y PVCs en Kubernetes.

En la *figura 14* se refleja el uso de PersistentVolumeClaims, en dónde los dos primeros son asociados con el PersistentVolume, porque cumplen los requisitos de espacio y el último PVC no se asigna ya que el nodo ya no tiene espacio, pero dispone de 100MB disponibles para otro PVC con esa característica.

Dos formas de crear un *persistent volume claim:*

- Automático: pvc.yaml, nginx-pvc-pod.yaml
- Manual: pvc.yaml, pv.yaml, nginx-pvc-pod.yaml

```
# Requerimiento del almacenamiento
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pvc
spec:
  storageClassName: mylocalstorage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
---
# Como se implementa
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-storage
spec:
  storageClassName: mylocalstorage
  capacity:
    storage: 20Gi
  accessModes:
```

```
    - ReadWriteOnce
  hostPath:
    path: "/mnt/some new/directory/structure/"
    type: DirectoryOrCreate
```

Manifiesto del pod que tiene la BBDD con el volumen.

```
    metadata:
      labels:
        app: mongodb
    spec:
      containers:
      - name: mongodb
        image: mongo:3.6.5-jessie
        volumeMounts:
          - name: mongo-persistent-storage
            mountPath: /data/db
      volumes:
        - name: mongo-persistent-storage
          persistentVolumeClaim:
            claimName: mongo-pvc
```

En este ejemplo, el *pod* que contiene la base de datos mongoDB, necesita almacenar su información, por ello se utiliza un persistentVolumeClaim

Manifiestos para reclamo y asignación de volumen persistente. El proceso de "*binding*" del *PersistentVolumeClaim* con el PersistentVolume consiste en coincidir en encontrar un *PersistentVolume* de todo el clúster que tenga la capacidad que se pide, el modo de acceso que se describe en el manifiesto y la etiqueta, que es mylocalstorage en este caso.

3.8. Escalado de pods

En Kubernetes se permite realizar un escalado horizontal de pods de forma automática. Esta funcionalidad se conoce como *Horizontal Pod Autoscaling* (abreviado comúnmente por las siglas HPA). El mecanismo HPA se basa en la evaluación de reglas que permiten a Kubernetes saber si debe o no crear más réplicas de un *pod* en base a los recursos Algunos comandos para visualizar, crear, eliminar, describir o listar ese HPA (Horizontal Pod Autoescaling) como se haría con los pods.

```
kubectl create
kubectl get hpa
kubectl describe hpa
kubectl delete hpa
```

Kubectl nos ofrece un comando adicional para crear reglas de autoescalado.

```
kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --
max=10
```

Esto crearía el autoescalado para php-apache cuando el porcentaje de uso de CPU sea mayor al 50%, habiendo un mínimo de 1 réplica y un máximo de 10 réplicas, pero también se pueden usar el escalado en memoria y otras métricas personalizadas.

Estas métricas personalizadas son **métricas del *pod***, que consiste en describir *pods* y determinar un valor objetivo para el control de replicas.

Métricas de objetos describen un objeto diferente en el mismo espacio de nombres, en lugar de describir pods.

Estas métricas que se pueden usar para lanzar el escalado se refieren al consumo de memoria y CPU que utilizan los *pods*, para ello es necesario configurar en los manifiestos los *request (ver apartado 3.6.9*) del recurso que se quiere configurar para la regla del auto escalado.

3.9. Arquitectura de Kubernetes

Kubernetes está compuesto por un nodo *master* y uno o más nodos *workers*, los cuales generalmente tienen corriendo contenedores dentro de cada uno, estos nodos eran conocidos generalmente como *minions*.

Las aplicaciones se ejecutan mediante *pods* y en los nodos *workers* se despliegan los *pods* y cada *worker* puede ejecutar múltiples *pods*.

El máster es el encargado de asignar a los nodos *workers* los contenedores que van a ejecutarse en ellos, servicios de monitorización y chequeo de los estados de salud del clúster, entre otros. Este conjunto de máquinas se conoce como clúster.

Es necesario a la hora de crear un clúster, configurar un nodo *master* y un nodo *worker*, es aconsejable crear más de un nodo *worker* así se puede prever si un nodo falla utilizar el otro que está disponible. En general el nodo *master* no ejecuta contenedores, pueden hacerlo, pero no es un escenario deseable, ya que es el que gestiona el clúster y éste debe de componerse de al menos un nodo *master* y un nodo *worker*, esto es un escenario básico.

El nodo *master* se encarga de los puntos principales que vamos a detallar a continuación: [22]

1. **Servidor API**: Es uno de los elementos que se instala cuando instalamos Kubernetes. El *master* hace público este servidor para los *workers* y los clientes del clúster para comunicarse. Es el servidor API el que nos permite interactuar con el clúster.
2. **Servicio etcd**: es un servicio cuyo trabajo es mantener la configuración actual del clúster. Es el almacenamiento clave- valor donde se tiene toda la información de los nodos *workers* y *masters*.
3. **Asignación de trabajo y control** (*Scheduler and controller manager*): El *scheduler* es el que se encarga de distribuir el trabajo o los contenedores a los múltiples nodos *workers*. Cuando un nuevo *pod* se ha creado los asigna a los nodos. El *controller* es el demonio responsable de que todo el tiempo se tenga el número deseado de instancias de los contenedores y notificarlos cuando hay algún contenedor en mal estado. Es un bucle de control que observa el clúster desde el apiserver y realiza cambios como reiniciar un nuevo contenedor cuando estos no están ejecutándose, para que el clúster esté en el estado deseado siempre.

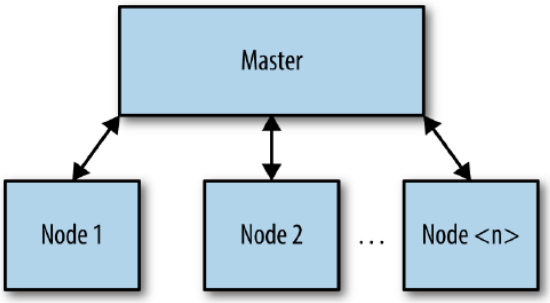


Figura 15.Clúster básico en Kubernetes

Por otro lado, los nodos *workers* también se componen de elementos que vamos a ver a continuación:

1. **Kubelet**: es el proceso más importante que se asegura de que los contenedores se están ejecutando y si no es así, los reinicia, el número de contenedores que se ejecutan por nodo debe coincidir con el manifiesto de cada *pod*.
2. **Kube-proxy**: red proxy que gestiona IPs virtuales de los nodos.
3. **Container runtime**: es el software que se utiliza por debajo para ejecutar contenedores, en el trabajo se utiliza Docker, pero existen otras opciones como CRI-O, containerd y las implementaciones de Kubernetes CRI, descritas en este enlace https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md

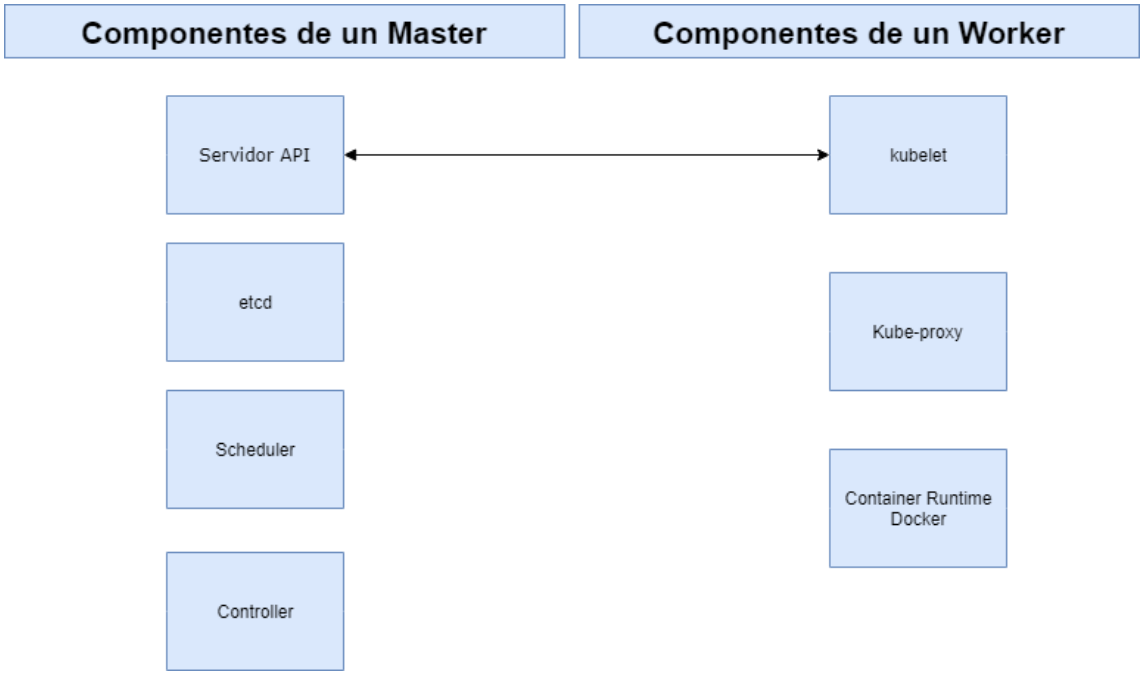


Figura 16. Componentes de los nodos del Kubernetes

Los tres componentes que son imprescindibles en un nodo de Kubernetes de cualquier tipo son:.

- Kubeadm: Herramienta para manejar clusters, la configuración mínima necesaria para utilizar los comandos para el clúster.
- Kubelet: es el proceso que se asegura de ejecutar todo el tiempo lo que está programado en el fichero .yaml.

- Kubectl: es la herramienta de la línea de comandos de Kubernetes para desplegar y gestionar aplicaciones.

Para la creación de un *Pod* primero necesitamos una red para que pueda haber comunicación entre los *pods*. Para ello, Kubernetes utiliza la Interfaz de red de contenedor (CNI), se ha investigado de que existen varias herramientas que nos permiten hacer esto:

Flannel: ofrece una subred a cada *host* para su uso con Kubernetes.

Weave: proporciona una red para la comunicación entre *pods*.

Calico: facilita la política de red y el enrutamiento en Kubernetes en la nube. Calico hace uso de una red IP de nivel superior para facilitar el rendimiento, y flexibilidad.

OVN (Open Virtual Networking): es un sistema para soportar la abstracción de redes virtuales.

Tras valorar los distintos tipos, se ha elegido Calico porque es una red de código abierto seguro para los contenedores.

3.8. Kubernetes en las distintas plataformas cloud

Providers	Managed	Turnkey cloud	On-prem datacenter	Custom (cloud)	Custom (On-premises VMs)	Custom (Bare Metal)
Agile Stacks		✓	✓			
Alibaba Cloud		✓				
Amazon	Amazon EKS	Amazon EC2				
AppsCode	✓					
APPUiO	✓	✓	✓			
Banzai Cloud Pipeline Kubernetes Engine (PKE)		✓		✓	✓	✓
CenturyLink Cloud		✓				
Cisco Container Platform			✓			
Cloud Foundry Container Runtime (CFCR)				✓	✓	
CloudStack					✓	
Canonical		✓		✓	✓	✓
Containership	✓	✓				
Digital Rebar						✓
DigitalOcean	✓					
Docker Enterprise		✓	✓			✓
Fedora (Multi Node)					✓	✓
Fedora (Single Node)						✓
Gardener		✓		✓		
Giant Swarm	✓	✓	✓			
Google	Google Kubernetes Engine (GKE)	Google Compute Engine (GCE)	GKE On-Prem			
IBM	IBM Cloud Kubernetes Service		IBM Cloud Private			
Ionos	Ionos Managed Kubernetes	Ionos Enterprise Cloud				
Kontena Pharos		✓	✓			
Kubermatic	✓	✓	✓			
Kubespray				✓	✓	✓
Kublr	✓	✓	✓	✓	✓	✓
Microsoft Azure	Azure Kubernetes Service (AKS)					
Mirantis Cloud Platform			✓			
Nirmata		✓	✓			
Nutanix	Nutanix Karbon	Nutanix Karbon			Nutanix AHV	
OpenShift	OpenShift Dedicated and OpenShift Online		OpenShift Container Platform		OpenShift Container Platform	OpenShift Container Platform
Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE)	✓	✓				
oVirt					✓	
Pivotal		Enterprise Pivotal Container Service (PKS)	Enterprise Pivotal Container Service (PKS)			
Platform9	✓	✓	✓		✓	✓
Rancher		Rancher 2.x		Rancher Kubernetes Engine (RKE)		k3s
StackPoint	✓	✓				
Supergiant		✓				
SUSE		✓				
SysEleven	✓					
Tencent Cloud	Tencent Kubernetes Engine	✓	✓			✓
VEXXHOST	✓	✓				
VMware	VMware Cloud PKS	VMware Enterprise PKS	VMware Enterprise PKS	VMware Essential PKS		VMware Essential PKS

Figura 17. Servicio de Kubernetes en plataformas cloud. [23]

En la imagen anterior, se muestra las soluciones que cada proveedor Cloud ofrece para hacer uso de Kubernetes.

Por ejemplo, Google ofrece para GKE (Google Kubernetes Engine) para creación del clúster en la nube, GCE (Google Kubernetes Engine) y GkE On-prem.

Azure tiene la herramienta AKS (Azure Kubernetes Service.

Kubernetes también es compatible con Opensift y en la figura se puede ver las herramientas disponibles para la creación del clúster.

Capítulo 4

4. Instalación de Kubernetes y configuración de clústeres

4.1. Clúster de un único nodo con Minikube en Windows

Minikube se define como un clúster con un único nodo, específicamente un nodo máster. Se utiliza para empezar a practicar con Kubernetes y así evitar cometer un error en un entorno de producción. Tiene todas las herramientas necesarias para ir tomando contacto con los conceptos utilizados en Kubernetes, tiene una instalación muy sencilla y casi todos suelen iniciarse con minikube antes de crear un cluster con múltiples nodos.

Los pasos a seguir para la instalación son los siguientes, para cuando estamos en Windows 10 Home:

4.1.1. Instalar virtual box de la página oficial

https://www.virtualbox.org/wiki/Downloads

4.1.2. Instalación de kubectl

Desde la página oficial de Kubernetes se debe instala el ejecutable de kubectl y lo guardamos en un directorio.

```
https://kubernetes.io/es/docs/tasks/tools/install-kubectl/
```

A continuación, se debe de agregar el directorio del ejecutable en la variable de entorno: path, se edita añadiendo la ruta del directorio.

Para comprobar que se ha instalado correctamente, bastaría con escribir kubectl en la terminal de Windows y recibir la lista de comandos.

4.1.3. Instalar minikube

Descargar el ejecutable de esta pagina:

https://github.com/kubernetes/minikube/releases/tag/v1.5.2

Seleccionar el **minikube-windows-amd64** y luego una vez descargado, renombrarlo a **minikube.exe** y a continuación añadir al path la ruta donde se encuentra.

4.1.4. Iniciación de minikube

```
minikube start                                  minikube status
```

4.2. Instalación de minikube para Linux

Instalar virtual box de la página oficial

https://www.virtualbox.org/wiki/Linux_Downloads

4.2.1. Instalación de kubectl

Desde la página oficial de Kubernetes se debe instala el paquete binario de kubectl

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.16.0/bin/linux/amd64/kubectl
```

Luego se debe de hacer que el binario tenga los permisos necesarios para la ejecución y por último moverlo en el directorio del path.

```
chmod +x ./kubectl           sudo mv ./kubectl /usr/local/bin/kubectl
```

Para asegurarse de que tenemos la última versión y bien instalado

```
kubectl version
```

Debe aparecer la versión de Kuberctl y que está conectado al cliente y servidor. Algo similar a esto:

```
Client Version: version.Info{Major:"1", Minor:"14",
GitVersion:"v1.14.0",
GitCommit:"641856db18352033a0d96dbc99153fa3b27298e5",
GitTreeState:"clean", BuildDate:"2019-03-25T15:53:57Z",
GoVersion:"go1.12.1", Compiler:"gc", Platform:"windows/amd64"}
Server Version: version.Info{Major:"1", Minor:"14",
GitVersion:"v1.14.2",
GitCommit:"66049e3b21efe110454d67df4fa62b08ea79a19b",
GitTreeState:"clean", BuildDate:"2019-05-16T16:14:56Z",
GoVersion:"go1.12.5", Compiler:"gc", Platform:"linux/amd64"}
```

4.2.2. Instalar minikube

Descargar el ejecutable, añadir permisos de ejecución y moverlo al directorio que se indica en el comando:

```
curl -Lo minikube
https://storage.googleapis.com/minikube/releases/latest/minikube-
linux-amd64 \
  && chmod +x minikube
cp minikube /usr/local/bin && rm minikube
```

4.2.3. Iniciación de minikube

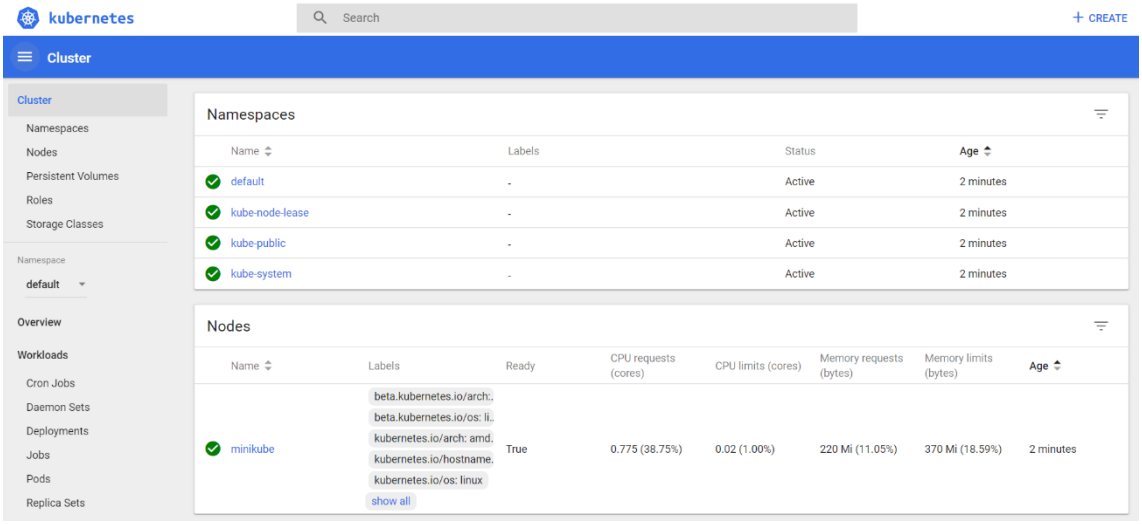
```
minikube start
```

Comandos

Algunos comandos útiles para ir manejando Kubernetes

Minikuke también ofrece un dashboard para ver el estado del cluster utilizando el comando

```
minikube dashboard
```



Dashboard de Kubernetes en minikube.

En la imagen se tiene el clúster de minikube con los namespaces que vienen creados por defecto y el panel donde se puede acceder a los diferentes elementos del clúster.

4.3. Clúster de múltiples nodos en local

4.3.1. Creación y configuración de las máquinas virtuales

Para la configuración del clúster se ha utilizado tres máquinas virtuales de VirtualBox con sistemas operativos Ubuntu 16.04 Se le han proporcionado 16GB de almacenamiento a cada una.

Estas máquinas virtuales serán los nodos del clúster, siendo una el nodo maestro y los otros dos nodos *workers*.

4.3.2. Instalación de Docker

Se ha instalado Docker en el nodo máster y los dos nodos workers utilizando el gestor de paquetes de Ubuntu y los repositorios de Docker.

```
apt-get update


apt-get install -y apt-transport-https ca-certificates curl software-
properties-common


curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
add-apt-repository "deb https://download.docker.com/linux/$(. /etc/os-
release; echo "$ID") $(lsb_release -cs) stable"


apt-get update && apt-get install -y docker-ce=$(apt-cache madison
docker-ce | grep 17.03 | head -1 | awk '{print $3}')
```

4.3.3. Instalación de Kubeadm, Kubelet and Kubectl

Se añade el repositorio de paquetes necesario para instalar estos tres componentes y después se procede a la instalación de paquetes desde el repositorio.

```
apt-get install -y apt-transport-https curl
    curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |
apt-key add -
    cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
    deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
    apt-get update
    apt-get install -y kubelet kubeadm kubectl
    apt-mark hold kubelet kubeadm kubectl
```

4.3.4. Desactivación el Swap

Desde la versión 3.3 de Kubernetes se espera que el swap esté desactivado. Si no se desactiva, el servicio kubelet no podrá utilizarse en el *master* y nodos *workers*.

A continuación, ejecutamos el comando.

```
sudo swapoff -a
```

O podemos desactivarlo permanentemente de la siguiente forma:

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

4.3.5. Inicialización del nodo *master*

Se inicializa el *master* de Kubernetes con el siguiente comando, indicando la dirección IP del nodo *master*, *hostname*, rango de red para la comunicación entre los *pods*.

```
kubeadm init --apiserver-advertise-address=192.168.56.4 --apiserver-
cert-extra-sans=192.168.56.4 --node-name $(hostname -s) --pod-network-
cidr=10.0.0.0/16
```

Al finalizar el init, se ejecuta los comandos para permitir a un usuario regular poder usar los comandos de kubectl.

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

Con esta configuración, se comprueba que se tiene el clúster con el master corriendo.

```
Kubectl get nodes
```

```
debian@k8s-master:~$ kubectl get nodes
NAME          STATUS   ROLES    AGE     VERSION
k8s-master    Ready    master   7d20h   v1.14.0
```

Figura 18. Nodo master en estado activo

4.3.6. Instalación de Calico para comunicación entre pods

Se instala Calico en el clúster, mediante la aplicación de sus ficheros de configuración (obtenido de su web oficial)

```
install Calico pod network addon
export KUBECONFIG=/etc/kubernetes/admin.conf
kubectl apply -f https://docs.projectcalico.org/v3.1/getting-
started/kubernetes/installation/hosted/rbac-kdd.yaml
kubectl apply -f https://docs.projectcalico.org/v3.1/getting-
started/kubernetes/installation/hosted/kubernetes-datastore/calico-
networking/1.7/calico.yaml
```

Luego de instalarlo, se debe comprobar de que esté en estado Running.

```
Kubectl get pods -A
```

```
debian@k8s-master:~$ kubectl get pods -A
NAMESPACE       NAME                                       READY   STATUS    RESTARTS   AGE
default         hello-world-78f8bf5c5c-pzs2d               1/1     Running   3          3d16h
default         hello-world-78f8bf5c5c-qv9cw               1/1     Running   3          3d16h
default         nginx-deployment-54f669bb6c-7csqb          1/1     Running   4          4d20h
default         nginx-deployment-54f669bb6c-gd4tb          1/1     Running   4          4d20h
default         serve-hostname-bcdf65f64-b7fwp             1/1     Running   4          4d17h
default         serve-hostname-bcdf65f64-hxsw7             1/1     Running   4          4d18h
default         serve-hostname-bcdf65f64-ps25m             1/1     Running   4          4d17h
default         serve-hostname-bcdf65f64-w6qjl             1/1     Running   4          4d18h
ingress-nginx   nginx-ingress-controller-5694ccb578-pjw4w   1/1     Running   3          3d16h
kube-system     calico-node-dgc67                          2/2     Running   10         7d20h
kube-system     calico-node-fqcfv                          2/2     Running   10         7d20h
kube-system     calico-node-vzv97                          2/2     Running   10         7d20h
kube-system     coredns-fb8b8dccf-dbxgw                    1/1     Running   5          7d20h
kube-system     coredns-fb8b8dccf-m8hft                    1/1     Running   5          7d20h
kube-system     etcd-k8s-master                            1/1     Running   5          7d20h
kube-system     kube-apiserver-k8s-master                  1/1     Running   5          7d20h
```

Figura 19.Listado que muestra los pods de Calico en ejecución.

4.3.7. Conexión de los nodos worker al clúster

Una vez que el *master* está listo, los nodos *workers* pueden unirse al clúster, cuyo token ha sido facilitado por el *master* a la hora de iniciar el *master*.

```
kubeadm join 192.168.56.4:6443 --token xu00av.knwjyd87ksb3qrv6 \
    --discovery-token-ca-cert-hash
sha256:2623d86c3b1ae547bfc951ded84e2be4358c9e46a6413b076d556135bfe568c
9
```

Se añaden los dos workers y confirmamos que los tres están arrancados.

```
debian@k8s-master:~$ kubectl get nodes
NAME          STATUS   ROLES    AGE     VERSION
k8s-master    Ready    master   7d20h   v1.14.0
k8s-worker1   Ready    <none>   7d20h   v1.14.0
k8s-worker2   Ready    <none>   7d20h   v1.14.0
```

Figura 20. Se muestra el clúster en activo.

Una vez se tiene el clúster arrancado, se crea el fichero nginx.yaml, que será el manifiesto para poder crear el *pod*.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

A continuación, se aplica la configuración de este manifiesto para crear el pod.

```
Kubectl apply -f nginx.yaml
```

Posteriormente, se comprueba el estado del pod para verificar que ha sido creado.

```
Kubectl get pods -o wide
```

```
debian@k8s-master:~$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE     IP          NODE          NOMINATED NODE   READINESS GATES
hello-world-78f8bf5c5c-pzs2d        1/1     Running   3          3d16h   10.0.1.56   k8s-worker1   <none>           <none>
hello-world-78f8bf5c5c-qv9cw        1/1     Running   3          3d16h   10.0.2.33   k8s-worker2   <none>           <none>
nginx-deployment-54f669bb6c-7csqb   1/1     Running   4          4d20h   10.0.1.54   k8s-worker1   <none>           <none>
nginx-deployment-54f669bb6c-gd4tb   1/1     Running   4          4d20h   10.0.1.58   k8s-worker1   <none>           <none>
serve-hostname-bcdf65f64-b7fwp      1/1     Running   4          4d17h   10.0.2.30   k8s-worker2   <none>           <none>
serve-hostname-bcdf65f64-hxsw7      1/1     Running   4          4d18h   10.0.1.55   k8s-worker1   <none>           <none>
serve-hostname-bcdf65f64-ps25m      1/1     Running   4          4d17h   10.0.2.31   k8s-worker2   <none>           <none>
serve-hostname-bcdf65f64-w6qjl      1/1     Running   4          4d18h   10.0.1.57   k8s-worker1   <none>           <none>
```

Figura 21. Listado de pods en el clúster y en qué nodo se ejecutan.

Con esta configuración realizada, se tendría creado un clúster de múltiples nodos en Kubernetes.

4.3.8. Prueba de acceso a contenedor

A los contenedores creados en los *pods* anteriormente, se puede acceder al Pod desde k8s-*master*:

```
$ curl http://<pod IP>
```

Muestra la siguiente información

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

```
HTTP/1.1 200 OK
Server: nginx/1.18
Content-Type: text/html
Content-Length: 612
Connection: keep-alive
Accept-Ranges: bytes
```

Creamos un pod en donde se pueda correr una Shell en él. Usando el manifiesto de busybox, que se enseña a continuación.

Ejemplo busybox-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
name: busybox
spec:
containers:
- name: busybox
image: busybox
command:
- sleep
- "3500"
```

Se puede acceder al pod ejecutando el commando y nos devuelve una terminal (prompt):

```
kubectl exec -it busybox -- /bin/sh
```

4.4. Clúster en la nube de AWS

Para empezar a desplegar el clúster de Kubernetes en AWS, primero se debe de crear una cuenta. Y posteriormente, se tiene que instalar Kubernetes, para ello existen dos formas de utilizar Kubernetes en esta plataforma, primero la manual utilizando instancias, creando recursos, loadbalancers, etc. y la segunda es utilizando una herramienta que ayude con la configuración del clúster, existen varias en el mercado, pero la más popular se llama Kops, que nos ayudará a crear el clúster más fácilmente. Se utilizará la segunda forma.

Kops es una herramienta oficial que se puede encontrar en GitHub, ya que la propietaria es Kubernetes https://github.com/kubernetes/kops.

4.4.1. Instalación de Kops

La definición oficial Kops es el que sigue y es una herramienta para crear clústeres en Amazon Web Services.

"Ayuda a crear, destruir, actualizar y mantener el clúster de Kubernetes de alta producción desde la línea de comandos"

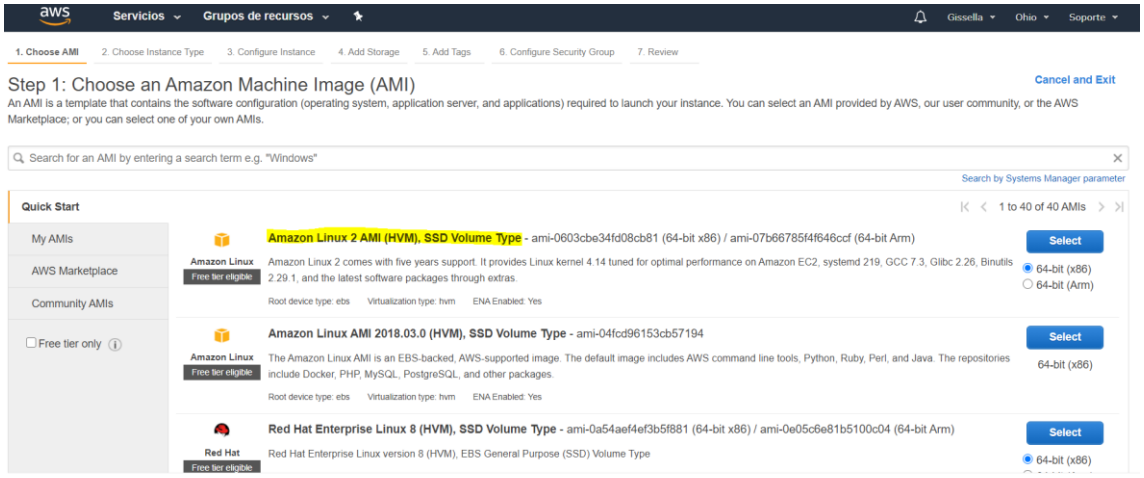
Aws es compatible con esta herramienta y esta es la mejor herramienta para conseguir que el clúster de Kubernetes esté listo y corriendo. También es compatible con Google Cloud Engine en versión beta y VMWare vSphere en versión alfa.

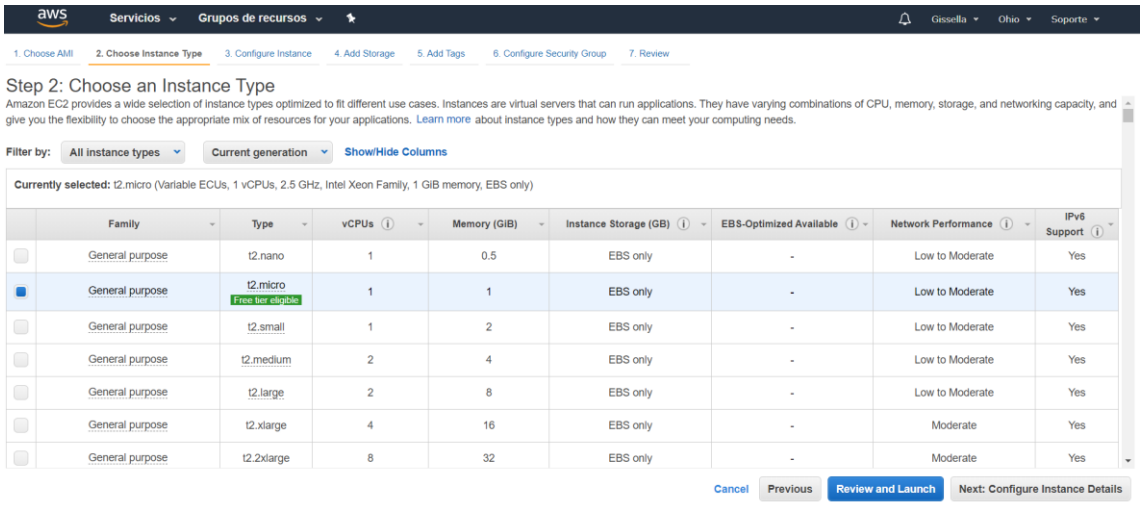
GitHub nos provee información de como instalar kops en AWS y crear la configuración de nuestro clúster en esta plataforma en la siguiente URL:

https://github.com/kubernetes/kops/blob/master/docs/getting_started/aws.md#install-kops

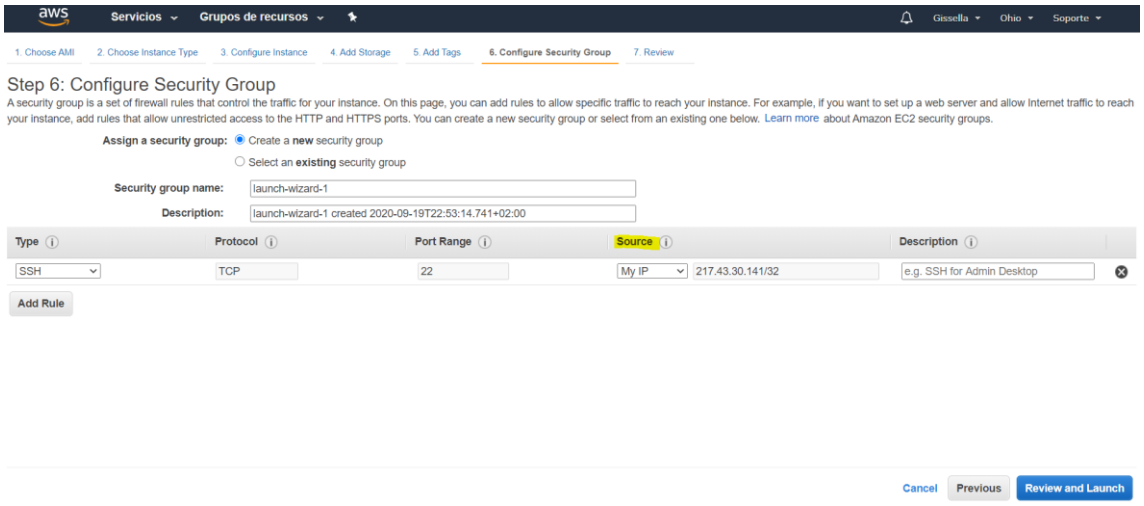
Una instancia en AWS, es un servidor virtual, dónde se instalará las dependencias de Kubernetes para el componente del clúster que se va a crear, se necesitará crear una instancia para poder crear en clúster desde ella, utilizándola como Bootstrap (componente base para el arranque).

Una vez, entramos con nuestra cuenta en AWS, se debe de ir al apartado EC2→Launch Instance → Amazon Linux 2 AMI (HVM), SSD Volume Type → t2.micro (tier elegible)





→ saltar a Step 6 : Configure Security Group → Source : myIP *por seguridad, para limitar acceso solo desde nuestro PC*



→ Asegurarnos antes de lanzar→ Launch

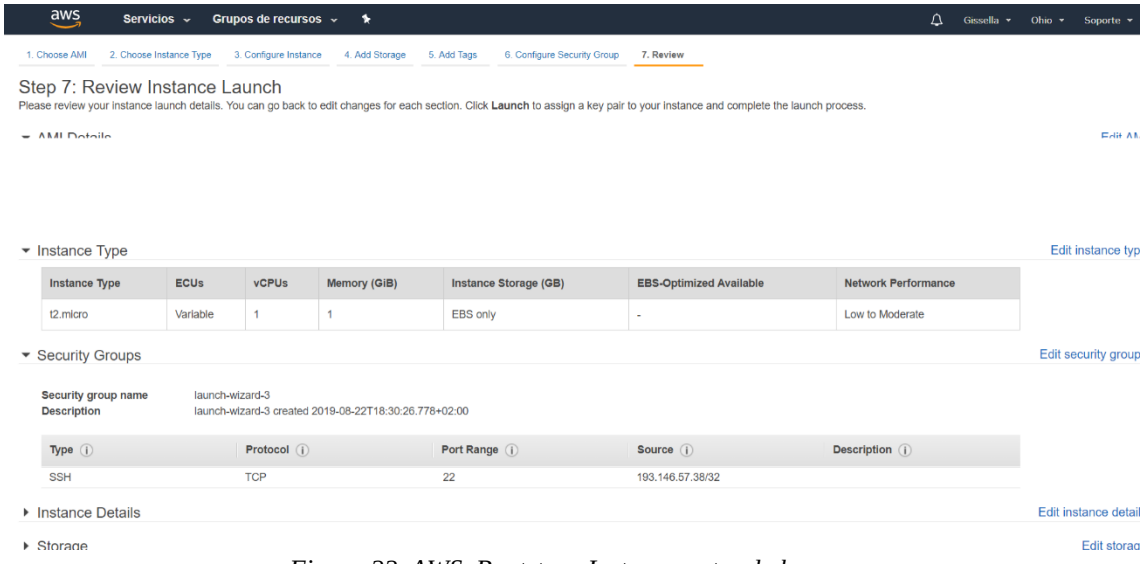


Figura 22. AWS. Bootstrap Instance antes de lanzar.

Create a new key-pair y seleccionamos **descargar** antes de dar aceptar para poder conectarnos al clúster desde nuestro PC.

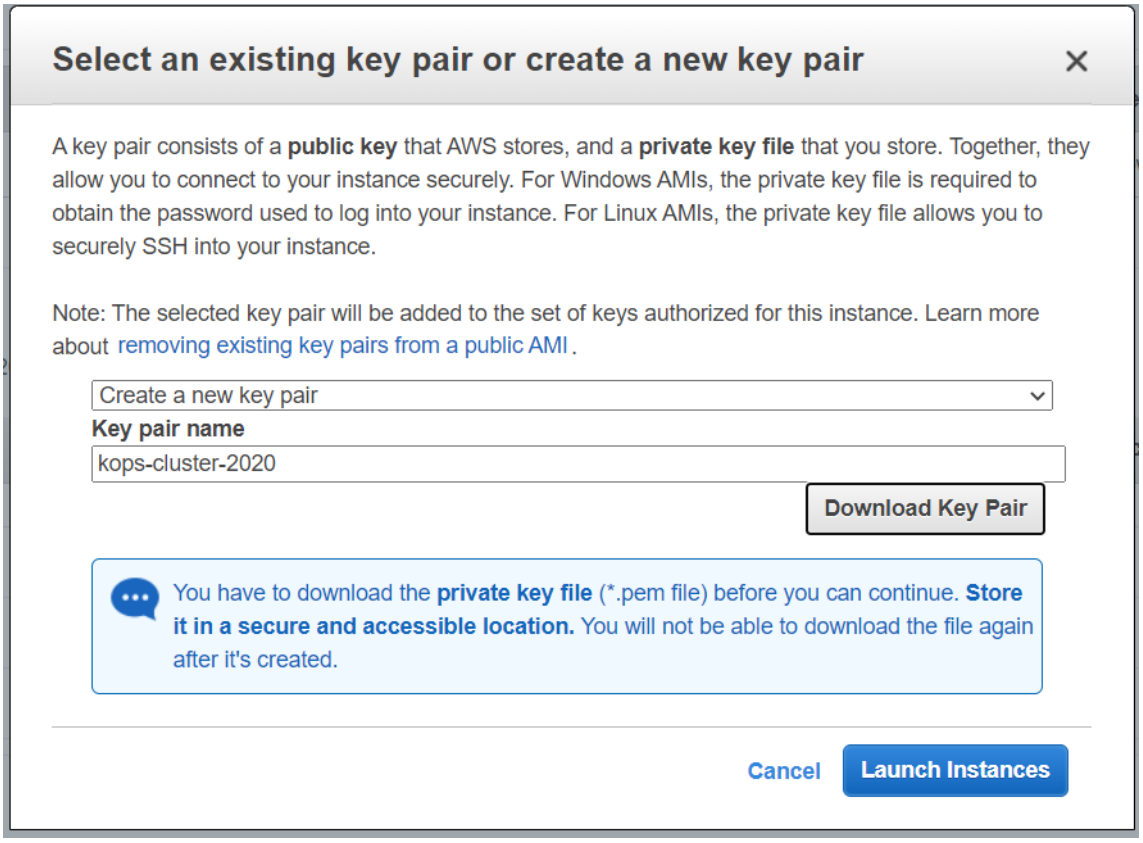


Figura 23. Key Pair que se crea al lanzar la instancia, para conectarse a ella desde la terminal.

Una vez que la instancia se encuentre en estado running, podemos acceder desde nuestro equipo al Bootstrap con el comando

```
ssh -i kops-cluster-2020.pem ec2-user@IP de la instancia
```

Una vez dentro, se procede a descargar Kops [24] en nuestra instancia de EC2:

```
curl -Lo kops https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag_name | cut -d '"' -f 4)/kops-linux-amd64

chmod +x ./kops

sudo mv ./kops /usr/local/bin/
```

A continuación, se debe instalar la herramienta Kubectl [24]con estos tres comandos para Linux: en la terminal de EC2

```
curl -Lo kubectl https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl

chmod +x ./kubectl

sudo mv ./kubectl /usr/local/bin/kubectl
```

Se necesita crear un grupo IAM con los siguientes permisos:

```
AmazonEC2FullAccess

AmazonRoute53FullAccess

AmazonS3FullAccess

IAMFullAccess

AmazonVPCFullAccess
```

Para ello se ejecuta estos comandos para crear el grupo Kops con los permisos mencionados.

Servicios → IAM (Identity and Access Management) → Grupos → crear nuevo grupo → kops → paso siguiente → buscar los 5 permisos mencionados y seleccionarlos → Revisar → crear Grupo

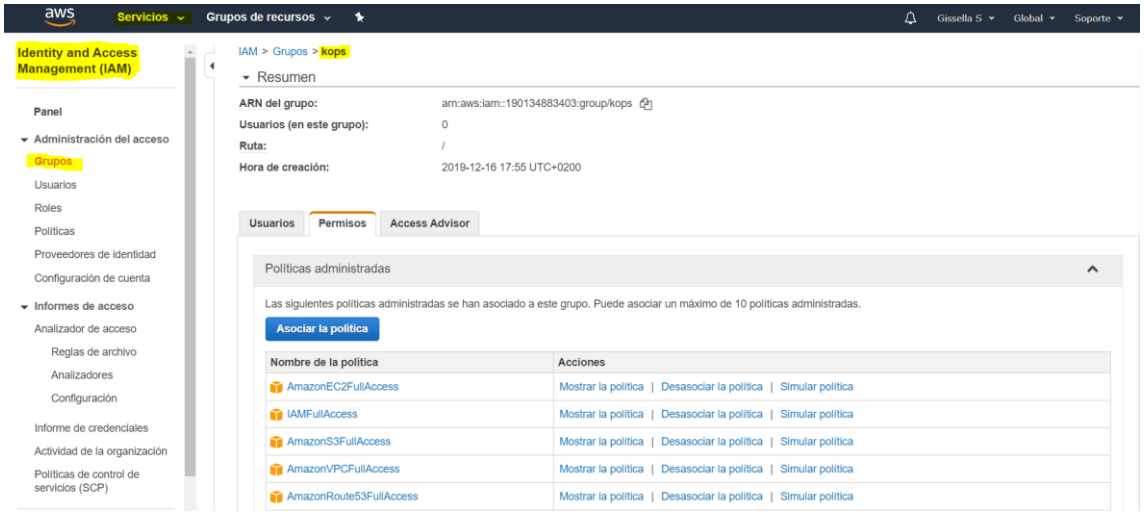


Figura 24. Pasos seguidos para crear un grupo IAM.

Posteriormente se crea un usuario y se añade a ese grupo creado anteriormente.

Usuarios → Añadir usuario → Nombre de usuario: kops → Tipo de acceso: acceso mediante programación → Siguiente → Añadir un usuario al grupo → Seleccionar kops -→ Revisar → Crear un usuario.

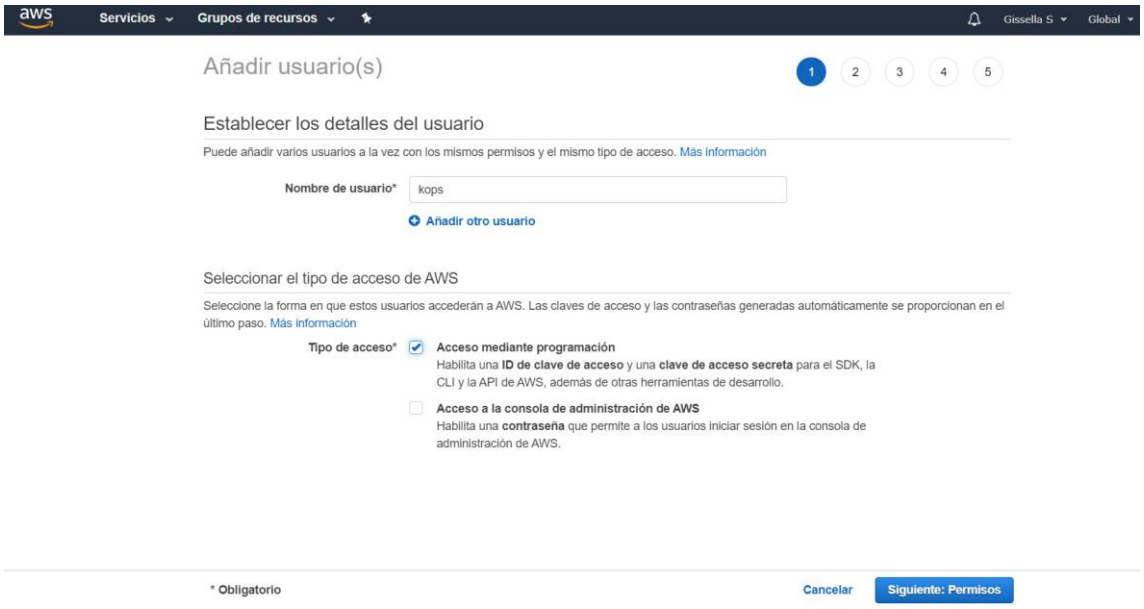


Figura 25. Paso 1. Crear un usuario IAM en AWS.

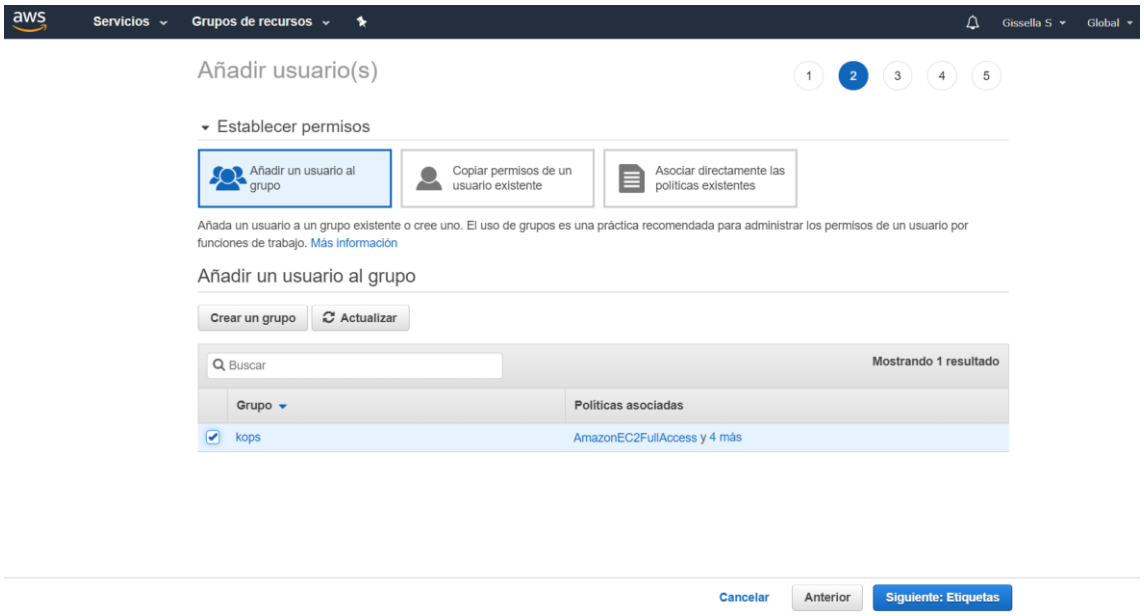


Figura 26. Paso 2. Añadir el usuario al grupo creado anteriormente (Kops).

También se puede hacer los mismo, desde la línea de comandos. Crear grupo y usuarios de Kops.

```
aws iam create-group --group-name kops


aws iam attach-group-policy --policy-arn
arn:aws:iam::aws:policy/AmazonEC2FullAccess --group-name kops

aws iam attach-group-policy --policy-arn
arn:aws:iam::aws:policy/AmazonRoute53FullAccess --group-name kops

aws iam attach-group-policy --policy-arn
arn:aws:iam::aws:policy/AmazonS3FullAccess --group-name kops
```

```
aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/IAMFullAccess --group-name kops

aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/AmazonVPCFullAccess --group-name kops


aws iam create-user --user-name kops


aws iam add-user-to-group --user-name kops --group-name kops


aws iam create-access-key --user-name kops
```

Se debe de guardar el Access Key ID y Secret Accesss key, para usarlo posteriormente cuando se vaya a conectar con el clúster de AWS.

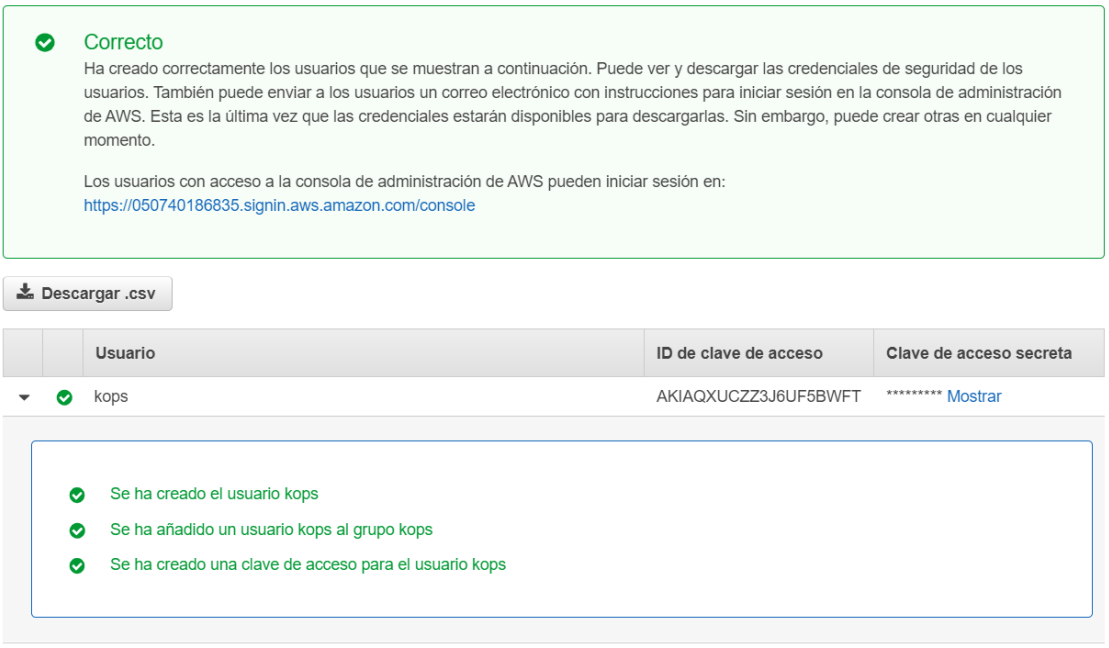


Figura 27. Creación de usuario y grupo kops en AWS con su clave de acceso creado.

```
ssh -i kops-cluster-2020.pem ec2-user@IP de la instancia

aws configure

(pedirá credenciales)

Default region name [Ohio]: Ohio

Default output format [us-east-2]: us-east-2

aws iam list-users --output table
```

```
export AWS_ACCESS_KEY_ID=$(aws configure get aws_access_key_id)

export AWS_SECRET_ACCESS_KEY=$(aws configure get aws_secret_access_key)
```

4.4.2. Creación del clúster con Kops

Para seguir configurando el clúster dentro de AWS se necesita además crear un almacenamiento llamado S3 bucket, que es un sistema de archivos distribuido en esta plataforma y donde Kops guardará los datos.

`Services → S3 → create bucket → crear un nombre único → crear`

Ahora finalmente se procede a crear el clúster, para ello se utiliza:

```
export NAME=mycluster.k8s.local
export KOPS_STATE_STORE=s3://gissellasantacruz-state-storage
```

Creación de la configuración del clúster

```
aws ec2 describe-availability-zones --region us-east-2 --output text
```

Se crea configuraciones para el clúster

```
kops create cluster --zones us-east-2a,us-east-2b,us-east-2c ${NAME}
```

Resultado de comando:

```
SSH public key must be specified when running with AWS (create with
`kops create secret --name mycluster.k8s.local sshpublickey admin -i
~/.ssh/id_rsa.pub`)
```

Una vez ejecutado este comando, debe informar que se ha aplicado la configuración, pero aún queda por realizar algunos pasos previos: generar claves públicas ssh y añadirlo a la configuración de kops.

Para generar la clave pública y privada se aplica el siguiente comando:

```
ssh-keygen -b 2048 -t rsa -f ~/.ssh/id_rsa
```

Añadimos esa clave generada a nuestro clúster, con kops

```
kops create secret --name ${NAME} sshpublickey admin -i
~/.ssh/id_rsa.pub
```

Ahora se podrá comprobar que toda la configuración se encuentra correcta utilizando el editor de kops, donde se puede cambiar las subredes si se quiere ampliar la IP o cambiar alguna otra configuración disponible.

```
kops edit cluster ${NAME}
```

Se abre un editor de vi por defecto. Con esta información del clúster a crear:

```
apiVersion: kops.k8s.io/v1alpha2
kind: Cluster
metadata:
  creationTimestamp: "2020-09-19T21:53:30Z"
  name: mycluster.k8s.local
spec:
  api:
    loadBalancer:
      type: Public
  authorization:
    rbac: {}
  channel: stable
  cloudProvider: aws
  configBase: s3://gissellasantacruz-state-storage/mycluster.k8s.local
  containerRuntime: docker
  etcdClusters:
  - cpuRequest: 200m
    etcdMembers:
    - instanceGroup: master-us-east-2a
      name: a
    memoryRequest: 100Mi
    name: main
  - cpuRequest: 100m
    etcdMembers:
    - instanceGroup: master-us-east-2a
      name: a
    memoryRequest: 100Mi
    name: events
  iam:
    allowContainerRegistry: true
    legacy: false
  kubelet:
    anonymousAuth: false
  kubernetesApiAccess:
  - 0.0.0.0/0
  kubernetesVersion: 1.18.8
  masterInternalName: api.internal.mycluster.k8s.local
  masterPublicName: api.mycluster.k8s.local
  networkCIDR: 172.20.0.0/16
  networking:
    kubenet: {}
  nonMasqueradeCIDR: 100.64.0.0/10
  sshAccess:
  - 0.0.0.0/0
  subnets:
  - cidr: 172.20.32.0/19
    name: us-east-2a
    type: Public
    zone: us-east-2a
```

También se puede editar las instancias, es decir los nodos del clúster.

IG indica instancia de grupos de nodos

```
kops edit ig nodes --name ${NAME}
```

Se puede cambiar el tipo de las máquinas, cuántos nodos como máximo o como mínimo.

```
apiVersion: kops.k8s.io/v1alpha2
kind: InstanceGroup
metadata:
  creationTimestamp: "2020-09-19T21:53:30Z"
  labels:
    kops.k8s.io/cluster: mycluster.k8s.local
  name: nodes
spec:
  image: 099720109477/ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-20200817
  machineType: t3.medium
  maxSize: 2
  minSize: 2
  nodeLabels:
    kops.k8s.io/instancegroup: nodes
  role: Node
  subnets:
  - us-east-2a
  - us-east-2b
  - us-east-2c
```

```
kops get ig --name ${NAME}
```

Lista los nodos *workers* y el *master que ha creado en AWS*.

Para editar el *master* podemos usar el comando:

```
 kops edit ig nombre_master_de_la_lista --name ${NAME}
```

A continuación, se creará el clúster en AWS.

```
kops create secret --name ${NAME} sshpublickey admin -i
~/.ssh/id_rsa.pub
kops update cluster ${NAME} --yes
kops validate cluster
kubectl get nodes --show-labels
```

Una vez estén todos nuestros nodos a *true* se puede decir que los nodos *workers* se han unido al *master* y que el nuevo clúster está en estado *Ready*.

```
[ec2-user@ip-172-31-36-250 ~]$  kops validate cluster
Using cluster from kubectl context: mycluster.k8s.local

Validating cluster mycluster.k8s.local

INSTANCE GROUPS
NAME                    ROLE    MACHINETYPE     MIN     MAX     SUBNETS
master-us-east-2a       Master  t2.micro        1       1       us-east-2a
nodes                   Node    t2.micro        1       3       us-east-2a,us-east-2b,us-east-2c

NODE STATUS
NAME                                            ROLE    READY
ip-172-20-50-242.us-east-2.compute.internal     node    True
ip-172-20-56-63.us-east-2.compute.internal      master  True

Your cluster mycluster.k8s.local is ready
```

Una vez se tiene el clúster listo en la plataforma de AWS. Se accede al nodo maestro para poder desplegar una aplicación, con el comando:

```
ssh ubuntu@3.131.37.188
```

Una vez dentro se crea un *pod* con la imagen redis, de DockerHub.

```
ubuntu@ip-172-20-59-90:~$ kubectl create deployment redis --image=redis
deployment.apps/redis created
```

Con el comando *`kubectl get all`* se obtiene todos los objetos del clúster.

```
ubuntu@ip-172-20-59-90:~$ kubectl get all
NAME                         READY   STATUS    RESTARTS   AGE
pod/redis-85d47694f4-b4gvn   1/1     Running   0          4m35s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   100.64.0.1   <none>        443/TCP   119m

NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/redis   1/1     1            1           4m35s

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/redis-85d47694f4   1         1         1       4m35s
```

Al crear un *deployment*, también se crean *replicaSet* y el *pod* asociados.

4.4.3. Eliminación del clúster con Kops

La mejor manera de borrar un clúster creado con *kops* es entrar en una instancia y utilizar este comando:

```
kops delete cluster ${NAME} --yes
```

Esto borra todos los componentes que creó para el clúster, no se necesita ir uno por uno y borrarlo desde la interfaz de AWS.

4.5. Herramientas de monitorización

Tener una visión general y gráfica es muy importante a la hora de mantener un clúster o varios clústeres, Kubernetes en general ofrece varias opciones para poder mejorar la interfaz que ofrece para la monitorización de un cluster. Por otro lado, minikube como se ha visto ofrece addons para el dashboard de Kubernetes para visualizar el estado del clúster. En este apartado se va a describir cómo utilizar Grafana y Prometheus para esta monitorización mejorada que ofrecen estas herramientas.

Es conveniente y casi obligatorio en los entornos de producción revisar el clúster para ver si los nodos están ejecutándose correctamente o están parados, si están consumiendo mucha CPU, si hay que agregar más nodos al clúster, etc.

4.5.1. Helm

Helm es un gestor de paquetes que se puede usar para utilizar otras herramientas que como pueden ser Grafana y Prometheus. Helm [25] es una herramienta de Kubernetes para manejar las aplicaciones, es decir, pods, servicios, etc. Para la previa instalación de Prometheus y Grafana se va a instalar Helm.

Pasos para la instalación de HELM

Descargar Helm en el EC2 (bootstrap) que se tiene en AWS

https://github.com/helm/helm [26]→ the Releases page → Helm 3.0.3

```
wget https://github.com/helm/helm/releases/tag/v3.0.3/helm-v3.0.3-linux-amd64.tar.gz
```

Descomprimir el archivo

```
tar zxvf helm-v3.0.3-linux-amd64.tar.gz
```

Cambiar de ubicación a /usr/local/bin

```
mv /Helm /usr/local/bin
```

Se inicia Helm

```
helm init
```

Comprobamos que está instalado y qué versión es

```
helm version
```

Helm se encontrará en el namespace de kube-system

```
kubectl get pods -n kube-system
```

Se puede ver que el nombre del pod es tiller

Primero se actualiza el repositorio para luego hacer la instalación

```
helm repo update

helm install –name monitoring stable/prometheusoperator --nameespace monitoring
```

4.5.2. Prometheus

Prometherus es una herramienta de monitorización de código abierto creada por la Cloud Native Computing Foundation. Ofrece una interfaz muy básica por lo que normalmente se suele integrar con otra herramienta para ofrecer una monitorización más sofisticada, en este caso se ha elegido Grafana.

Para poder instalar Prometheus [27], se va a seguir los pasos de la página https://github.com/helm/charts/tree/master/stable/prometheus-operator, donde el primer comando para instalar, se utiliza Helm.

```
$ helm install--name monitoring --namespace monitoring
stable/prometheus-operator
```

Se debe proporcionar un nombre para evitar nombres largos generados automáticamente y también es conveniente crear un namespace para ello, ya que se crean varios objetos como pods, servicios, daemonSet, deployments, replicaSets y statefulSets.

Prometheus estará integrado con Grafana para ofrecer todos los detalles del clúster de una manera visual y prolija, aunque Grafana sea la interfaz, Prometheus es el encargado de obtener las metricas y juntar toda la información por debajo, también ofrece una interfaz más sencilla en la que se puede acceder.

4.5.3. Grafana

Grafana [28] es una herramienta de monitorización que se puede integrar con hasta 44 fuentes, entre ellas Prometheus. Se va a utilizar por la interfaz sofisticada que ofrece y por la sencilla integración con Prometheus,

Para poder activar Grafana, una vez instalado Prometheus. Se debe editar el pod de grafana y cambiar el tipo de servicio, anteriormente ClusterIP a LoadBlancer, una vez aplicado los cambios se puede ver que asigna un puerto, con el nombre y el puerto en el navegador, ya se tiene la interfaz de Grafana.

Figura 28. Ventana de Login de Grafana

Para entrar se debe de autenticarse, se tiene “admin” como username y “prom-operator” como contraseña.

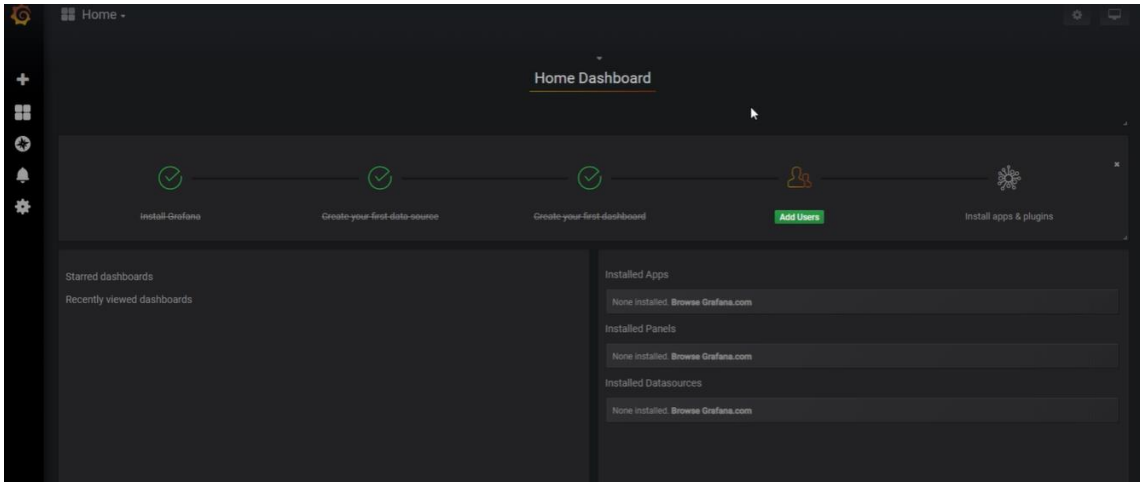


Figura 29. Página Principal de Grafana

Ya una vez dentro de la interfaz de Grafana se puede monitorizar los *pods*, *deployments*, servicios, seleccionar *namespaces* o verlo en forma de gráfico. Aunque tenga una interfaz más avanzada que Prometheus, éste sigue funcionando por debajo para obtener las métricas.

Capítulo 5

5. Diseño e implementación de escenarios con Kubernetes

En este apartado se van a desarrollar cinco escenarios donde se podrá ver la flexibilidad, comprobar la fiabilidad y escalabilidad de Kubernetes.

Se dividen en estos escenarios:

Escenario 1. Utilización de autoescalado (aumentar y disminuir) y load-balancer.

Escenario 2. Réplica de bases de datos (mongoDB en un pod).

Escenario 3. Clúster híbrido, clúster en la nube y nodos conectándose desde otra red al master.

Escenario 4. Rollouts utilizando deployments (ir versiones superiores y volver a versión anterior en caso de error)

Escenario 5. Almacenamiento compartido entre máquinas (diferente ubicación) utilizando NFS.

5.1. Introducción al proyecto fleetman

Este es un proyecto desarrollado por virtualProgrammers [29] y disponible en el repositorio de github y de Docker. Se trata de la localización de furgonetas a través de GPS y conocer tanto la situación actual como el histórico. Se va a mostrar un diagrama de la distribución los microservicios que la componen.

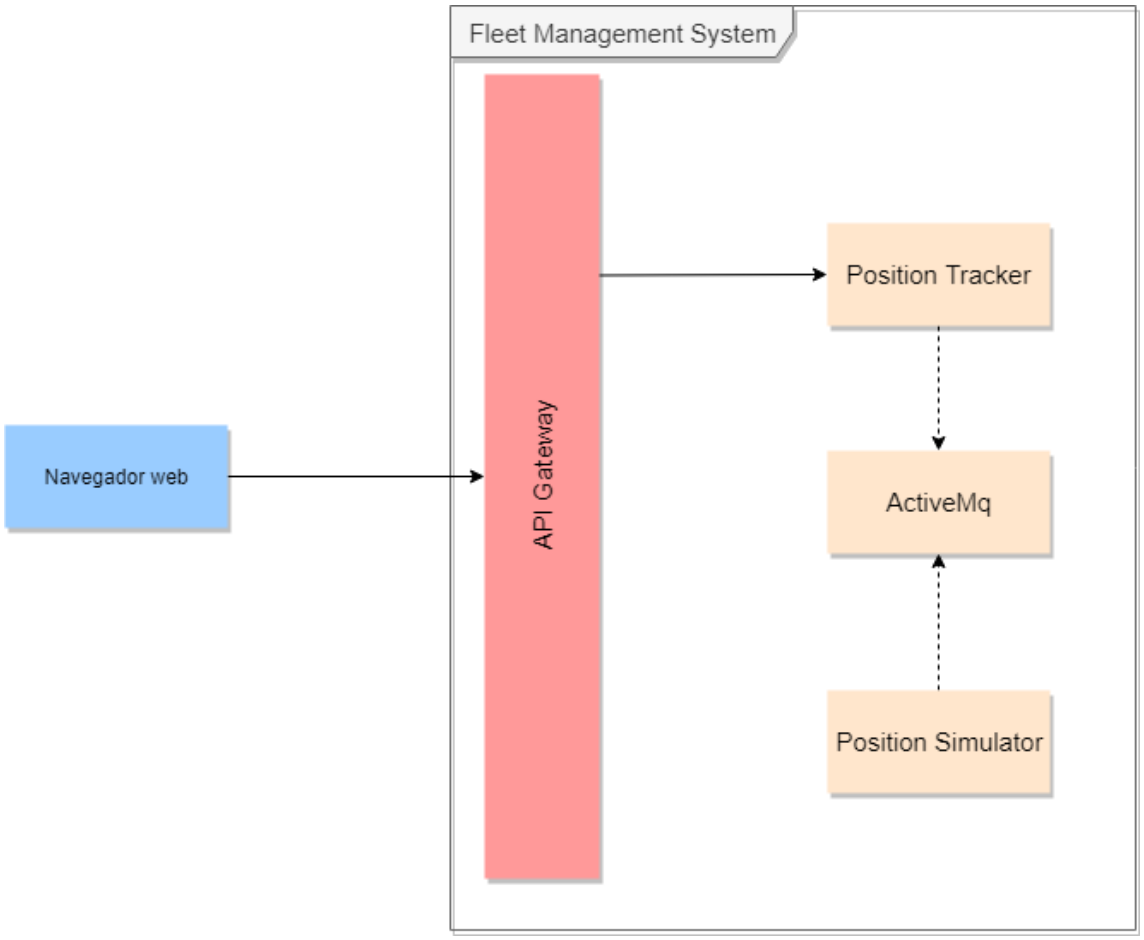


Figura 30. Componentes principales de la aplicación Fleetman

Position simulator: este microservicio simula posiciones de las furgonetas que se tendrá en el mapa. Se compone de una aplicación java que lee varios ficheros con información de latitud y longitudes.

ActiveMq: las posiciones que genera el microservicio anterior se envían a este microservicio, es un dockerfile que contiene un apache ActiveMq.

Position tracker: este microservicio es el encargado de calcular la velocidad de las furgonetas y almacenar el histórico de los mismos.

API-Gateway: este microservicio se encarga de recibir las solicitudes y delegar la llamada al microservicio al que se solicita.

5.2. Escenario 1: Autoescalado HPA

Es este escenario se va a realiza una simulación de tráfico web y añadiendo unas reglas de request y limits (ver apartado 3.6.9) para que se pueda ejecutar y ocurrir el autoescalado.

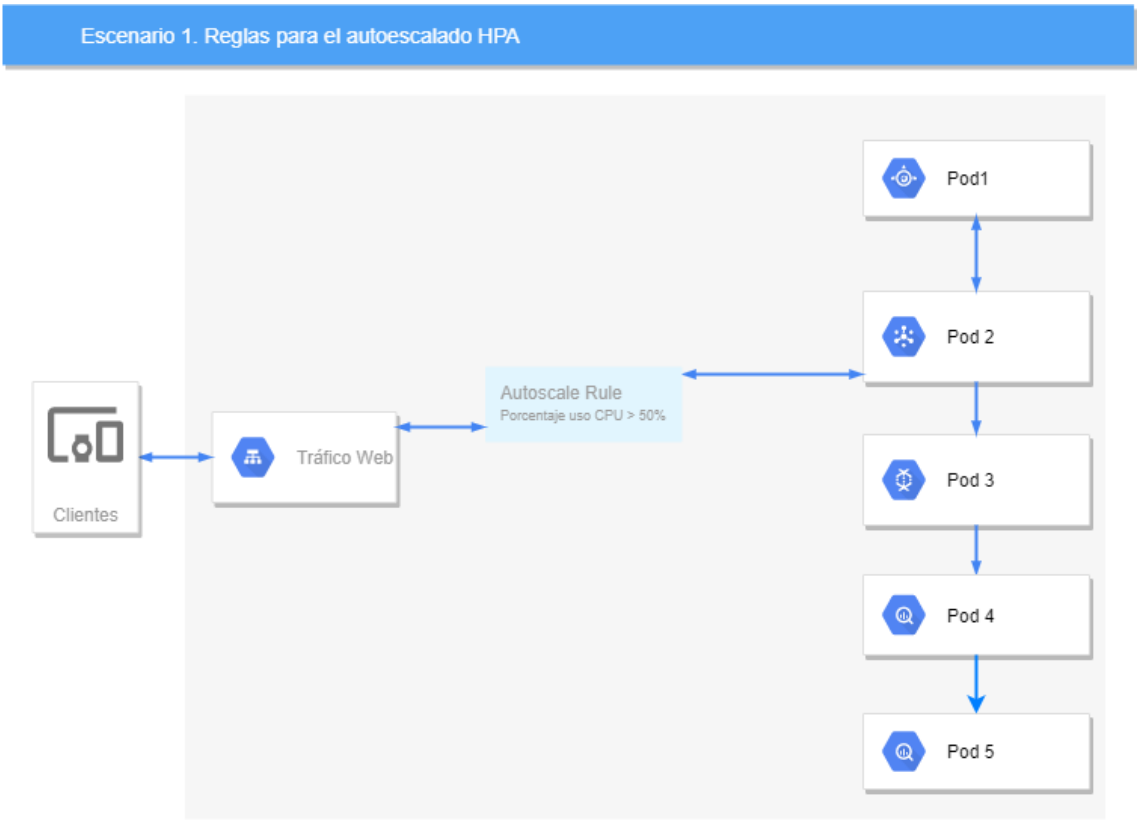


Figura 31. Escenario 1. Autoescalado de Pods

Configurar el *Balancer* para que según se realicen más peticiones HTTPs, se creen más nodos o se vayan eliminando.

En primer lugar, se tiene una aplicación de código abierto que se va a desplegar con Kubernetes llamado Fleetman.

Para poder simular el autoescalado horizontal de Pods, se van a definir un conjunto de reglas que se irán aplicando hasta lograr el autoescalado. En este caso lo que se va a autoescalar es el contenedor *api-gateway* y la regla que se usará es que, si el contenedor utiliza más del 50% de CPU, esté va a generar replicas automáticamente para poder atender esto y a medida que se vaya liberando CPU se irá bajando el número de nodos hasta llegar al inicial número de réplicas (cero). Se puede aplicar reglas para autoescalar los pods o para destruirlos si es que no están siendo utilizados.

Para poder ver cuanta CPU y memoria están utilizando los contenedores se puede ejecutar el comando:

```
$ kubectl top pod
NAME                                CPU(cores)   MEMORY(bytes)
api-gateway-667cc58746-t28v7        18m          164Mi
mongodb-57f6bf75cc-7xgbw            17m          195Mi
position-simulator-74559c676f-f5xj8 49m          144Mi
position-tracker-54b746d759-tdsm5   29m          201Mi
queue-5b67bdfc68-m855m              67m          424Mi
```

```
webapp-58755f59db-hcwhb                0m           7Mi
```

Así se tiene una idea más precisa de qué se está utilizando para poder aplicar la regla del autoescalado. Una vez se tenga aplicado un *request* y un limit al candidato a escalar, se puede aplicar la regla de autoreplicarse cuando alcance un límite permitido de uso de CPU.

Cómo se puede observar el consumo de CPU del api-gateway es 18 milicores si aumenta a 200 milicores que significaría 4 veces más de lo que está pidiendo el request y cuando llegue a 600 milicores, crearía tres replicas.

Para simular que hay tráfico continuamente en el api-gateway, se utilizará la ventana de pánico (panic) que ofrece la aplicación y se refiere a ir refrescando la página que se describe más abajo, con esto simula que hay más entradas y va incrementando el uso de la CPU para que pueda producirse este autoescalado.

Ventana de panic de la aplicación fleetman:

```
192.168.99.109:30080/api/panic
```

Una vez hecho esto, podemos comprobar que el uso ha aumentado con el comando

`Kubectl top pod` y para asegurarnos que se han producido replicas con los comandos:

```
kubectl describe hpa api-gateway
kubectl get all
```

Request actual del pod

```
  requests:
    memory: 200Mi
    cpu: 50m
```

Reglas de auto escalado para el HPA

```
apiVersion: v1
kind: HorizontalPodAutoscaler
metadata:
  name: api-gateway
  namespace: default
spec:
  maxReplicas: 5
  minReplicas: 1
  scaleTargetRef:
    apiVersion: v1
    kind: Deployment
    name: api-gateway
  targetCPUUtilizationPercentage: 400
```

También se puede realizar la misma regla con este comando:

```
kubectl autoscale deployment api-gateway --cpu-percent 400 --min 1 --
max 5
```

Cuando el tráfico vuelva a ser normal, los pods irán eliminándose hasta volver a ser un único pod, esto ocurrirá después de algunos minutos de actividad normal.

Para poder ver esto, se utiliza el comando `kubectl describe hpa api-gateway`

```
$ kubectl describe hpa api-gateway
Name:                                                  api-gateway
Namespace:                                             default
Labels:                                                <none>
Annotations:                                           kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"autoscaling/v1","kind":"HorizontalPodAutoscaler","meta
data":{"annotations":{},"name":"api-gateway","namespace":"default"},"spec":{"maxRe...
CreationTimestamp:                                     Sat, 23 Feb 2019 21:22:57 +0000
Reference:                                             Deployment/api-gateway
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):  10% (5m) / 400%
Min replicas:                                          1
Max replicas:                                          4
Conditions:
  Type            Status  Reason              Message
  ----            ------  ------              -------
  AbleToScale     True    ReadyForNewScale    recommended size matches current size
  ScalingActive   True    ValidMetricFound    the HPA was able to successfully calculate a replica count from cpu resource utilization (percentage of request)
  ScalingLimited  False   DesiredWithinRange  the desired count is within the acceptable range
Events:
  Type    Reason             Age   From                       Message
  ----    ------             ----  ----                       -------
  Normal  SuccessfulRescale  8m    horizontal-pod-autoscaler  New size: 3; reason: cpu resource utilization (percentage of request) above target
  Normal  SuccessfulRescale  1m    horizontal-pod-autoscaler  New size: 2; reason: All metrics below target
  Normal  SuccessfulRescale  41s   horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

Figura 32. Imagen donde se muestra el autoescalado de pods, cuando va bajando las el uso de la CPU.

5.3. Escenario 2: Réplica de bases de datos

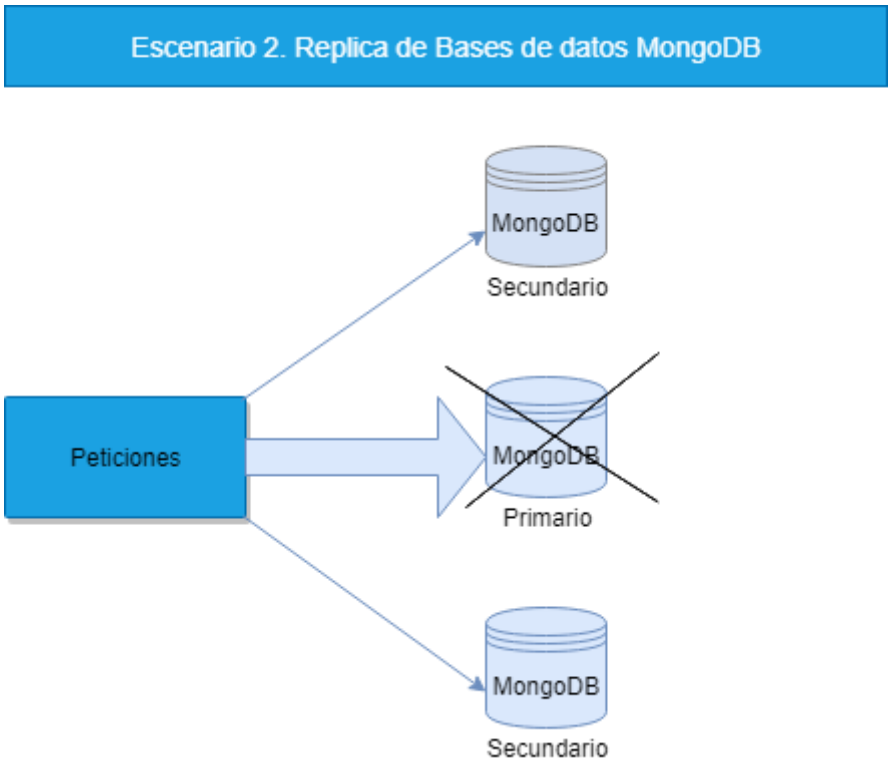


Figura 33. Escenario 2. Escenario donde se replican las bases de datos y se ofrece tolerancia a fallos

Para poder realizar este escenario se tiene que hacer uso de StatefulSet de Kubernetes, explicado en el apartado 3.6.13.3.6.13

Las bases de datos no son normalmente replicadas haciendo uso de replicaSets o deployments, para ello hace falta más y es cuando ponemos a práctica definiciones avanzadas de Kubernetes.

Para replicar las bases y hacerlas autoescalables se requiere el uso de statefulSet, esto es aplicable a varias bases de datos como MySql, Mongo, Redis, etc.

Se tiene una base de datos mongoDB que se quiere replicar, es importante considerar que las bases de datos necesitan compartir información.

Se debe de tener un servidor primario y al menos 2 servidores secundarios en este escenario que se quiere implementar.

Al crear tres BBDDs, también se crean tres PVCs, cada uno de 7GB.

```
kubectl get pvc
```

```
$ kubectl get pvc
NAME                                 STATUS   VOLUME                                     CAPACITY   ACCESS MOD
ES   STORAGECLASS   AGE
mongo-persistent-storage-mongo-0     Bound    pvc-4bf6c1b9-00fc-11eb-91ae-0800271dec51   7Gi        RWO
     standard       7m10s
mongo-persistent-storage-mongo-1     Bound    pvc-66362e3e-00fc-11eb-91ae-0800271dec51   7Gi        RWO
     standard       6m26s
mongo-persistent-storage-mongo-2     Bound    pvc-6cccee6e-00fc-11eb-91ae-0800271dec51   7Gi        RWO
     standard       6m15s
```

```
$ kubectl get all
NAME                                       READY   STATUS    RESTARTS   AGE
pod/api-gateway-8574847b85-jgq9k           1/1     Running   2          3d5h
pod/mongo-0                                2/2     Running   2          3d5h
pod/mongo-1                                2/2     Running   2          3d5h
pod/mongo-2                                2/2     Running   2          3d5h
pod/position-simulator-dfdbf6c6b-2r88n     1/1     Running   2          3d5h
pod/position-tracker-744859667d-lggcc      1/1     Running   1          3d5h
pod/queue-5c66494499-95zxw                 1/1     Running   1          3d5h
pod/webapp-ghkwd                           1/1     Running   5          3d5h

NAME                                  TYPE         CLUSTER-IP        EXTERNAL-IP
PORT(S)                          AGE
service/fleetman-api-gateway          NodePort      10.109.92.99      <none>
8080:30020/TCP                   3d5h
service/fleetman-position-tracker     ClusterIP     10.100.118.249    <none>
8080/TCP                         3d5h
service/fleetman-queue                NodePort      10.97.96.174      <none>
8161:30010/TCP,61616:32668/TCP   3d5h
service/fleetman-webapp               NodePort      10.104.40.180     <none>
80:30080/TCP                     3d5h
service/kubernetes                    ClusterIP     10.96.0.1         <none>
443/TCP                          3d5h
service/mongo                         ClusterIP     None              <none>
27017/TCP                        3d5h

NAME                          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
NODE SELECTOR   AGE
daemonset.apps/webapp   1         1         1       1            1           <none>
3d5h

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/api-gateway           1/1     1            1           3d5h
deployment.apps/position-simulator    1/1     1            1           3d5h
deployment.apps/position-tracker      1/1     1            1           3d5h
deployment.apps/queue                 1/1     1            1           3d5h

NAME                                             DESIRED   CURRENT   READY   AGE
replicaset.apps/api-gateway-8574847b85           1         1         1       3d5h
replicaset.apps/position-simulator-dfdbf6c6b     1         1         1       3d5h
replicaset.apps/position-tracker-744859667d      1         1         1       3d5h
replicaset.apps/queue-5c66494499                 1         1         1       3d5h

NAME                        READY   AGE
statefulset.apps/mongo      3/3     3d5h
```

La entidad que se comunica con mongoDB es position-tracker y es el responsable de conectarse con la bases de datos primaria, esto se va a poder ver desde los logs del pod cuando esté arrancándose.

Logs del position-tracker:

```
kubectl logs -f position-tracker-744859667d-lggcc
```

```
2019-09-02 16:50:28.974 INFO 1 --- [nio-8080-exec-2] org.mongodb.driver.connection : Opened connection [connectionId{localValue:5, serverValue:37}] to mongo-0.mongo.default.svc.cluster.local:27017
```

En el log del position-tracker se puede ver arriba, que la base de datos primaria es mongo-0 en este momento.

Para probar que tenemos esta fiabilidad/alta disponibilidad, se procederá a eliminar la bases de datos primaria y veremos cómo, desde el log que la bases de datos primaria pasa a ser otro de los secundarios.

Los pods que tenemos hasta el momento son:

```
$ kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
api-gateway-8574847b85-jgq9k          1/1     Running   2          3d5h
mongo-0                               2/2     Running   2          3d5h
mongo-1                               2/2     Running   2          3d5h
mongo-2                               2/2     Running   2          3d5h
position-simulator-dfdbf6c6b-2r88n    1/1     Running   2          3d5h
position-tracker-744859667d-lggcc     1/1     Running   1          3d5h
queue-5c66494499-95zxw                1/1     Running   1          3d5h
webapp-ghkwd                          1/1     Running   5          3d5h
```

```
$ kubectl delete pod mongo-0
pod "mongo-0" deleted
```

Obtener logs del position tracker para visualizar los cambios.

```
kubectl logs -f position-tracker-744859667d-lggcc
```

```
2019-09-02 17:20:05.932 INFO 1 --- [ter.local:27017] org.mongodb.driver.cluster : Exception in monitor thread while connecting to server mongo-0.mongo.default.svc.cluster.local:27017

2019-09-02 17:20:06.912 INFO 1 --- [ter.local:27017] org.mongodb.driver.cluster : Monitor thread successfully connected to server with description ServerDescription{address=mongo-2.mongo.default.svc.cluster.local:27017, type=REPLICA_SET_PRIMARY, state=CONNECTED, ok=true, version=ServerVersion{versionList=[3, 6, 5]}, minWireVersion=0, maxWireVersion=6


2019-09-02 17:20:06.914 INFO 1 --- [ter.local:27017] org.mongodb.driver.cluster : Discovered replica set primary mongo-2.mongo.default.svc.cluster.local:27017
```

Se puede apreciar que en cuestión de milisegundos se ha escogido un nuevo primario, es el mongo-2 y se ha conectado a él, esto ni siquiera se puede apreciar en la interfaz de la aplicación.

```
$ kubectl get pvc
NAME                                 STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   AGE
mongo-persistent-storage-mongo-0   Bound    pvc-269d546f-cb18-11e9-
b318-080027fd98d4   7Gi        RWO            standard       172d
mongo-persistent-storage-mongo-1   Bound    pvc-8c8ebecc-cb18-11e9-
b318-080027fd98d4   7Gi        RWO            standard       172d
mongo-persistent-storage-mongo-2   Bound    pvc-91012d6f-cb18-11e9-
b318-080027fd98d4   7Gi        RWO            standard       172d
```

La persistencia no se elimina.

5.4. Escenario 3: Clúster híbrido, nodos en la nube y en local

Un clúster híbrido, en este proyecto, se refiere a nodos que se encuentran en diferentes servidores físicos, redes, pero que están conectados. Para simular esto, en este escenario se ha decidido realizar de la siguiente forma: crear un clúster en la nube AWS y otro en local que se implementará en una máquina virtual usando sistema operativo Ubuntu y para la conexión una VPN que hará de túnel entre las máquinas. Escenario similar explicado en [30].

A continuación, se describirán algunos conceptos para entender mejor con lo que se va a trabajar en AWS [31].

VPN: red privada virtual en internet, que permite la conexión entre ordenadores, usando como protocolo de seguridad IPSec.

VPC: proporciona a los usuarios una red privada virtual, aísla lógicamente una sección de la nube.

VPG [32]: es un repetidor de red privada virtual, es el extremo AWS del túnel VPN.

IGW: es una componente que permite la conexión entre el VPC e internet. Esta puerta de enlace enruta el tráfico por internet según esté en la tabla VPC.

CGW: es una puerta de enlace del cliente.

La conexión VPN que se crea AWS consiste en dos túneles para una alta disponibilidad para la VPC. (solo se configura un túnel)

Una conexión VPN puede ser establecida usando VPG en la parte final de AWS y un router en la parte local, llamado CGW o nuestro caso NAT.

El tráfico que se va a configurar en las dos máquinas será de SSH, TCP y ICMP. Como se ve en la Figura 40 .

Type	Protocol	Port Range	Source
All TCP	TCP	0 - 65535	10.200.0.0/16
SSH	TCP	22	79.108.121.243/32
All ICMP - IPv4	All	N/A	10.200.0.0/16

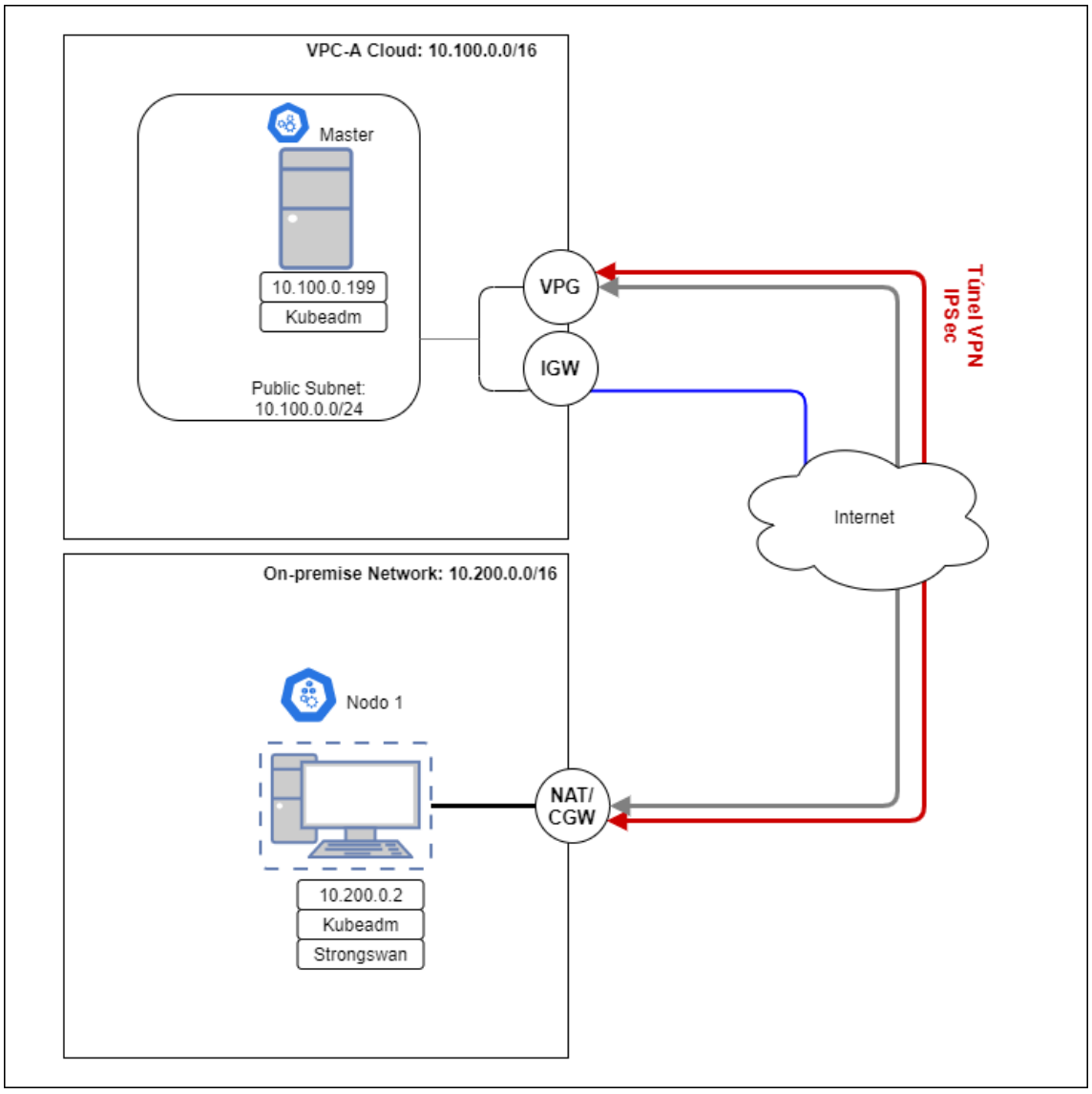


Figura 34. Clúster Hibrido en AWS y en local [33]

Como se puede observar en la figura, se ha creado un clúster cuyo master se encuentra en la nube como instancias de EC2, instalando en ellas los componentes de Kubernetes. En local, simulando una red on-premise se ha creado una máquina que hará el papel de worker.

En la parte de AWS se han realizado estos pasos:

1. Crear una VPC (Virtual Private Cloud) VPC-A-Cloud (CIDR 10.100.0.0/16)
2. Crear in Internet Gateway con el nombre VPC-A-IGW y posteriormente adjuntarlo con el VPC creado anteriormente
3. Se crea una subnet pública VPC-A-Public-Subnet con IPv4 10.100.0.0/24 como se muestra en la imagen

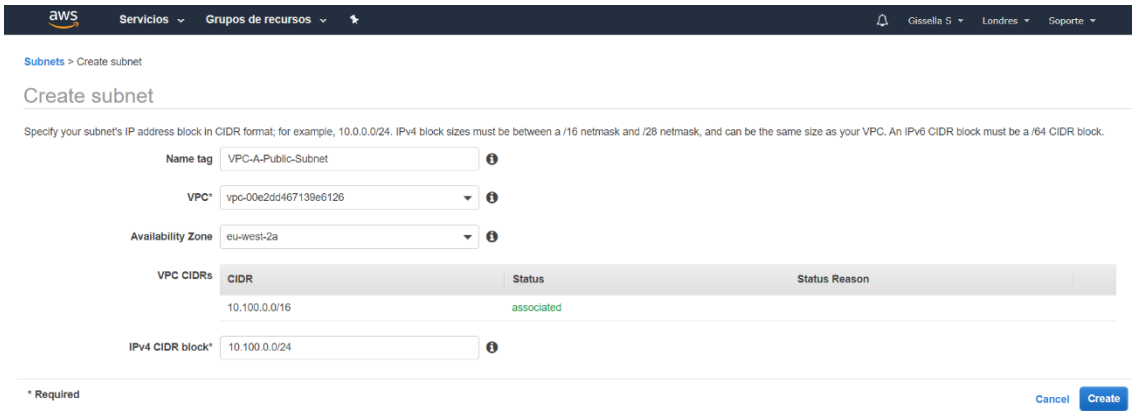


Figura 35. Se crea la Subnet Pública para la red virtual privada en la nube

4. Se define una route table VPC-A-Public-RT

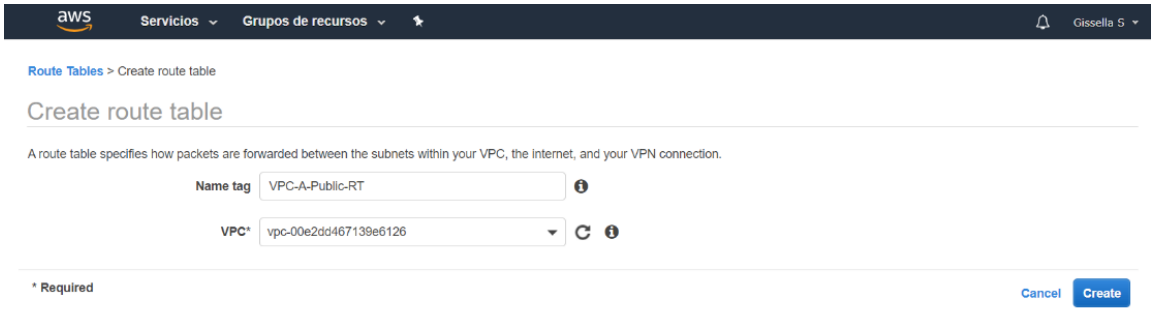


Figura 36. Al crear la tabla, se debe asociar con el VPC creado anteriormente.

A continuación, se edita la tabla añadiendo acceso a internet a través de la Internet Gateway creada anteriormente. Después se debe editar la subnet associations de la tabla, seleccionarlo la subnet creada anteriormente y guardar los cambios.

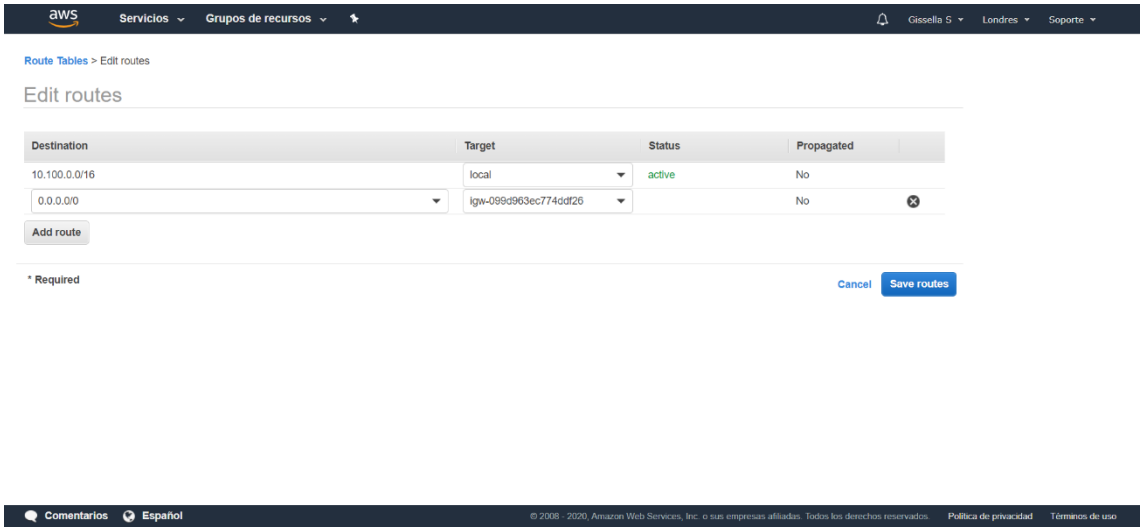


Figura 37. Añadir la ruta a internet y asociarlo con la internet Gateway que va a exponer

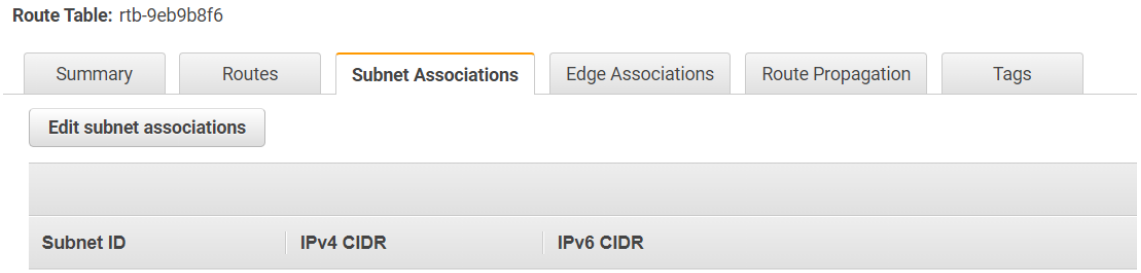


Figura 38. Edición de la subnet associations de Route table.

5. Se crea una máquina EC2 de tamaño c2.medium que va a utilizarse como master y otro de tamaño c2.micro para el worker, modificando los siguientes atributos.

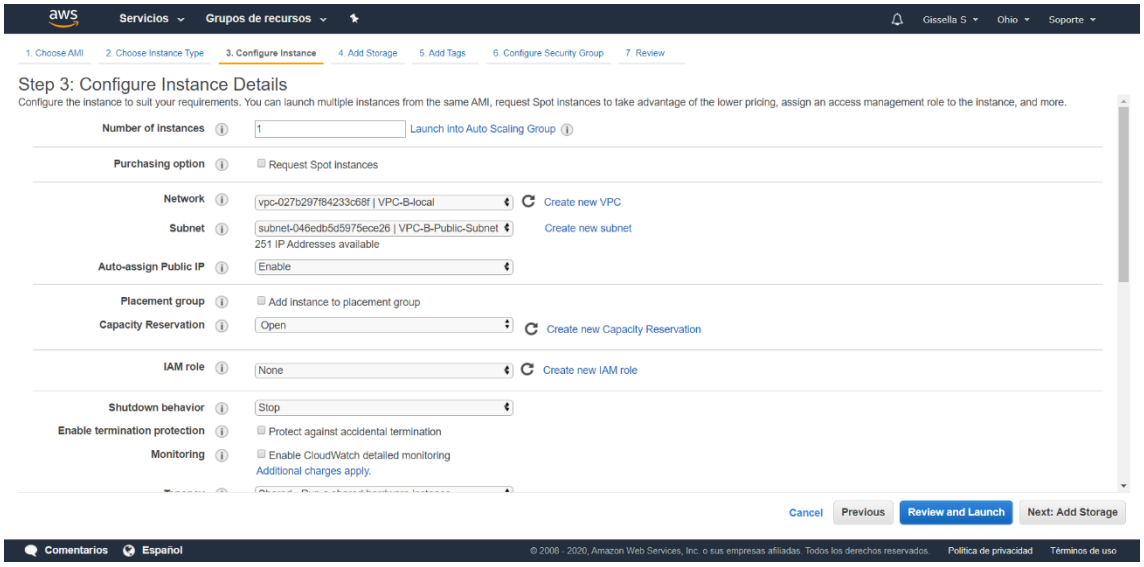


Figura 39. Al crear las máquinas, se asocian con la Subnet Pública y VPC directamente

Cuando todavía se está creando las máquinas se crea un nuevo security group con las siguientes reglas. Acceso a través de SSH, TCP e ICMP.

Type	Protocol	Port Range	Source
All TCP	TCP	0 - 65535	10.200.0.0/16
SSH	TCP	22	79.108.121.243/32
All ICMP - IPv4	All	N/A	10.200.0.0/16

Figura 40 . Security group con las reglas de acceso.

6. Se crea el Customer Gateway añadiendo la IP del equipo en local que se utiliza como VPN

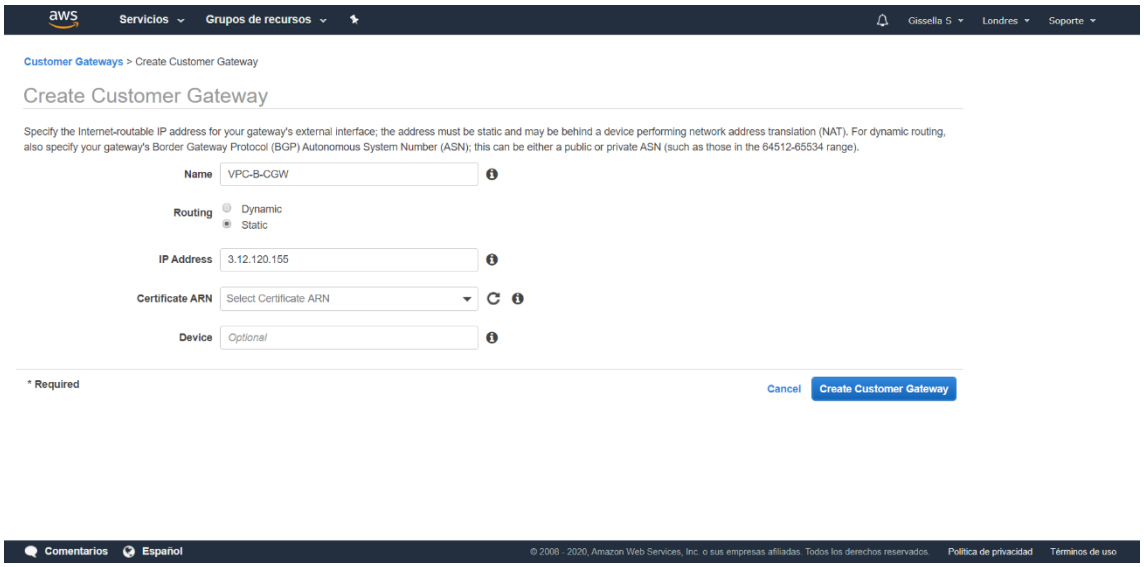


Figura 41. Customer Gateway con IP del equipo on-premise

7. Crear una Virtual Private Gateway y asociarlo con el VPC creado anteriormente

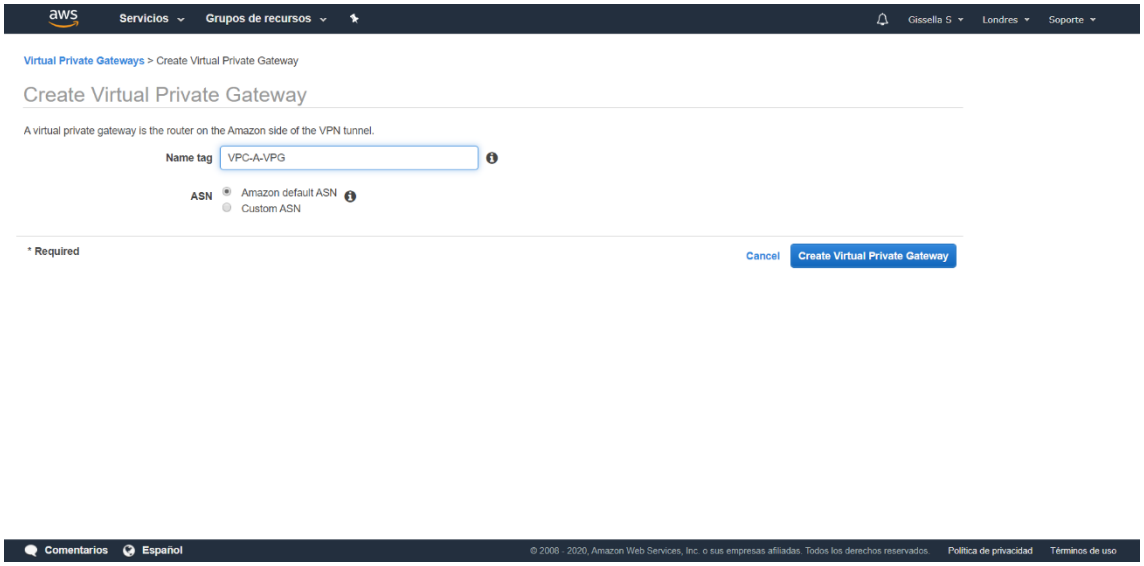


Figura 42. Se crea el Gateway del lado de la nube.

8. Crear la conexión VPN con las siguientes características, cuando esté disponible descargar la configuración, que posteriormente se realizara en la máquina local para configurar el túnel.

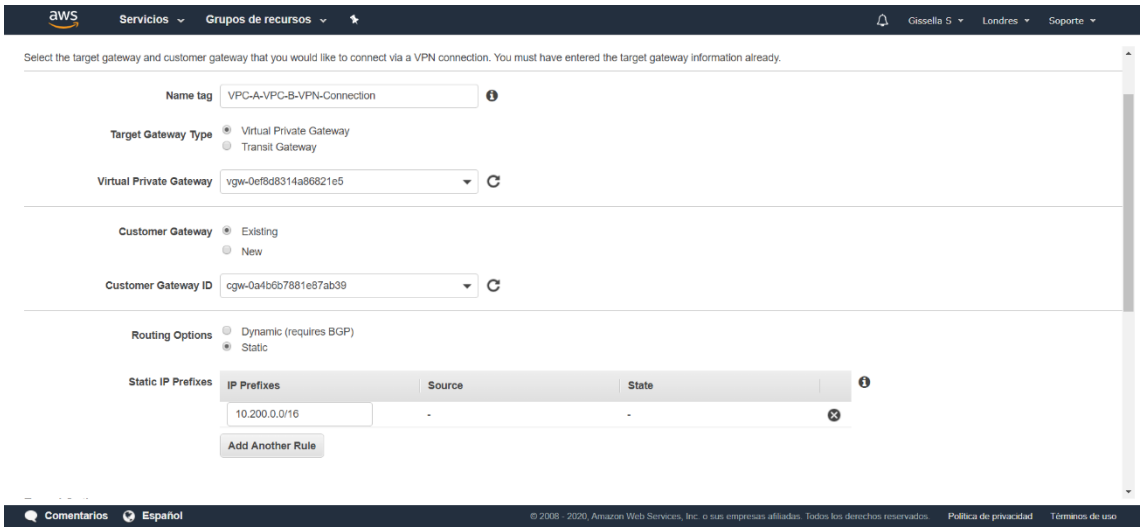


Figura 43. Proceso de creación de la VPN, seleccionando la VPG y Customer Gateway creados.

Por último, comprobar que la opción de route propagation esté a "yes" para que se pueda añadir la ruta 10.200.0.0 del VPN creado, así no habría que añadir la ruta manualmente, cuando el túnel esté "up" se añadirá automáticamente.

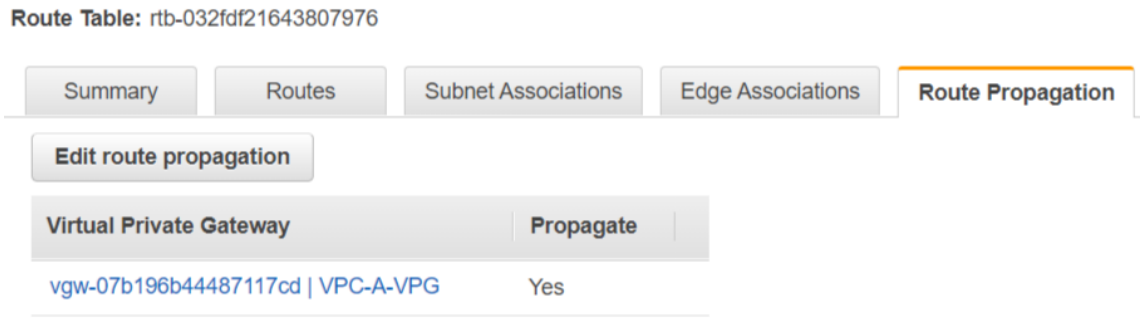


Figura 44. Opción para editar en el apartado de Route Tables.

Una vez se tenga todo de arriba, se accede al nodo maestro y se instala lo necesario para Kubernetes, se han seguido estos pasos [34].

En la máquina en local:

Antes de seguir, se va a definir Strongswan, ya que se hará uso de ello en el nodo 1 de la parte local.

Strongswan es la solución *open source* VPN para crear túneles IPSec, que AWS ofrece para las conexiones *on-premise* y en *cloud* [35].

1. Nodos On premise (CIDR 10.200.0.0/16)
   Por otro lado, se tiene el nodo worker con el servidor VPN al que se conectará
   a. Nodo worker con strongswan
   b. Se definen esta máquina con una ip pública y privada y una subnet (10.200.0.0/16)
   c. Se añaden unas reglas para el reenvío de puertos (NAT) [36]

Figura 45. Puertos 8500 y 8501 del router local. [36]

2. Se instalan Docker, y las demás herramientas de Kubernetes.
3. Descargar strongswan
   apt install strongswan -y [37]
   $ cat >> /etc/sysctl.conf << EOF
   echo net.ipv4.ip_forward = 1
   net.ipv4.conf.all.accept_redirects = 0
   net.ipv4.conf.all.send_redirects = 0
   EOF

   $ sysctl -p

4. Se siguen los pasos del documento descargado de la VPN de AWS

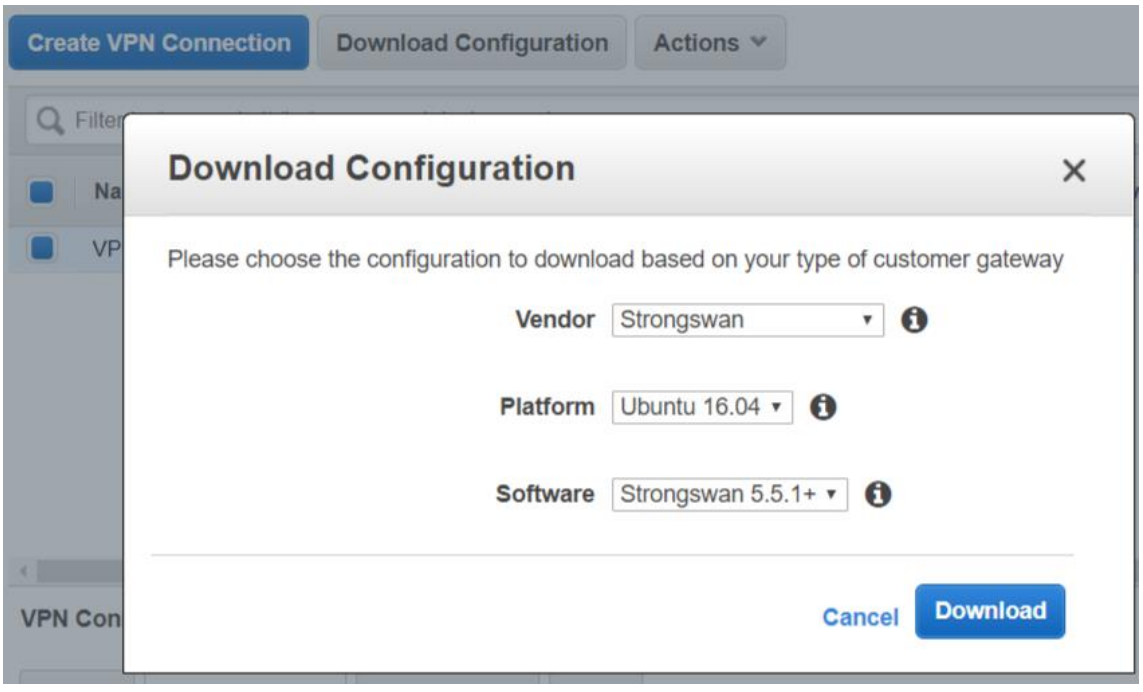


Figura 46. Descarga del documento de configuración para la VPN.

A continuación, se realizará los pasos IPSec para la configuración entre el AWS VPC y la red local con enrutamiento estático (Static Routing)

Se crea un archivo llamado /etc/ipsec.conf con los siguientes parámetros:

```
conn Tunnel1
      auto=start
      left=%defaultroute
      leftid=3.135.200.192
      right=3.10.239.92
      type=tunnel
      leftauth=psk
      rightauth=psk
      keyexchange=ikev1
      ike=aes128-sha1-modp1024
      ikelifetime=8h
      esp=aes128-sha1-modp1024
      lifetime=1h
      keyingtries=%forever
      leftsubnet=0.0.0.0/0
      rightsubnet=0.0.0.0/0
      dpddelay=10s
      dpdtimeout=30s
      dpdaction=restart
      mark=100
      leftupdown="/etc/ipsec.d/aws-updown.sh -ln Tunnel1 -ll
169.254.80.114/30 -lr 169.254.80.113/30 -m 100 -r <VPC CIDR>"
```

Se comparte la pre-shared key, para ello se crea un archivo /etc/ipsec.secrets, con un contenido parecido a este:

```
3.135.200.192 3.10.239.92 : PSK "PMm8K4lHqGF9ycVBWjUK7tcLwDi4KDck"
```

Para que los cambios sean persistentes se crea un script que realiza la configuración de la interfaz del túnel, configura la iptables con las reglas necesarias para aceptar los paquetes en la nueva interfaz lógica. Además, realiza unas pequeñas modificaciones en el kernel /etc/sysctl.conf con la nueva interfaz y las aplica. Para una configuración persistente después de los reinicios se encarga de asegurar que la iptables no se pierde, añadiendo enlaces y crea la interfaz lógica (Tunnel1) con la configuración requerida.

Por último, se reinicia el ipsec y si el túnel está disponible con los siguientes comandos:

```
ipsec restart
ipsec status
Security Associations (1 up, 0 connecting):
     Tunnel1[1]: ESTABLISHED 1 second ago,
10.200.0.2[79.108.121.243]...3.11.175.41[3.11.175.41]
     Tunnel1{1}:  INSTALLED, TUNNEL, reqid 1, ESP in UDP SPIs:
c6aeabfd_i a3ac2bf3_o
     Tunnel1{1}:   0.0.0.0/0 === 0.0.0.0/0
```

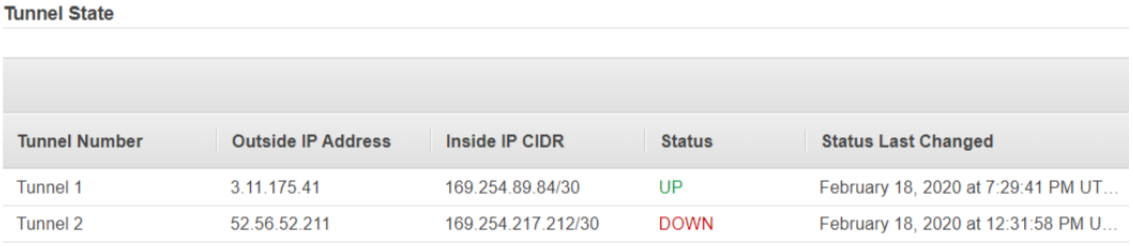


Figura 47. Imagen en AWS del túnel creado en estado "UP"

Se muestra en la imagen dos túneles, esto se debe a que AWS crea éstos para ofrecer alta disponibilidad pero en este caso solo está configurado uno de ellos, aunque solo uno de los está "up" a la vez.

Una vez establecido el túnel se comprueba con ping si las dos partes obtienen respuesta y ahora ya se puede unir al clúster con el token que el master haya generado. En esta referencia se describe algunos pasos a seguir para garantizar el trafico en el Tunnel si es que no llega a hacer ping [38].

```
kubeadm join 10.100.0.199:6443 --token yhl6af.s3es6npsu3s90g6l \
    --discovery-token-ca-cert-hash
sha256:2bee65c0b0a4d5c6eefaf1a17301da0fe94228cff507ae479e2251705e97067
b
```

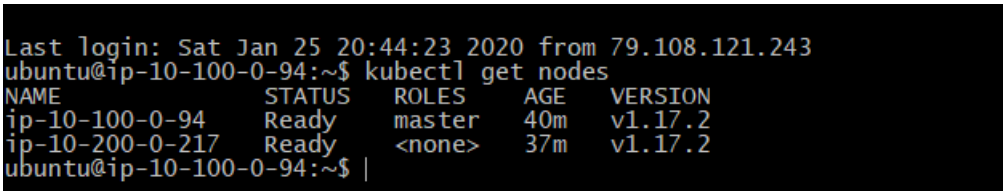


Figura 48. Clúster híbrido

Figura dónde se muestra el clúster unido por una parte en la nube de AWS y otra en local en una máquina virtual.

5.5. Escenario 4: Rollouts con deployments

Como ya se ha explicado, en el apartado 3.4.10. del documento sobre los rollbacks y rollings updates que ofrece Kubernetes en este escenario se mostrara una forma de hacer un rollback cuando una versión nueva presenta un problema y cómo volver a la última que estaba en funcionamiento.

Kubernetes ofrece la posibilidad de hacer rollouts de una versión que se tenga utilizando deployments, permite hasta 10 versiones, es decir tiene un almacenamiento para 10 replicasSet que tenga versiones diferentes.

Para este escenario, se utilizarán únicamente los microservicios webapp y queue, cuyo manifiesto se lista a continuación:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  selector:
    matchLabels:
      app: webapp
  replicas: 2
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: richardchesterwood/k8s-fleetman-webapp-
angular:release0-5

---
apiVersion: v1
kind: Pod
metadata:
  name: queue
  labels:
    app: queue
spec:
  containers:
  - name: queue
    image: richardchesterwood/k8s-fleetman-queue:release1
```

A continuación, se muestran los manifiestos necesarios para la creación de los servicios que exponen los microservicios anteriores al exterior:

```
apiVersion: v1
kind: Service
metadata:
  name: fleetman-webapp
spec:
    selector:
    app: webapp
```

```
  ports:
    - name: http
      port: 80
      nodePort: 30080

  type: NodePort
---
apiVersion: v1
kind: Service
metadata:
  name: fleetman-queue

spec:
  selector:
    app: queue

  ports:
    - name: http
      port: 8161
      nodePort: 30010

  type: NodePort
```

Una vez se aplica este escenario con estos pods y servicios, el comando que permite ver el estado de un rollout (para el deployment “webapp”) es:

```
$ kubectl rollout status deploy webapp
deployment "webapp" successfully rolled out
```

Una forma de realizar un *rollback* manual es cambiando en el manifiesto de release 0-5 (la versión actual) a release 0, aplicar los cambios y volver a consultar el estado del *rollout*.

El cambio de versión no se puede apreciar en la terminal, pero sí en la interfaz de la aplicación. Esta imagen es el release 0 de la aplicación.

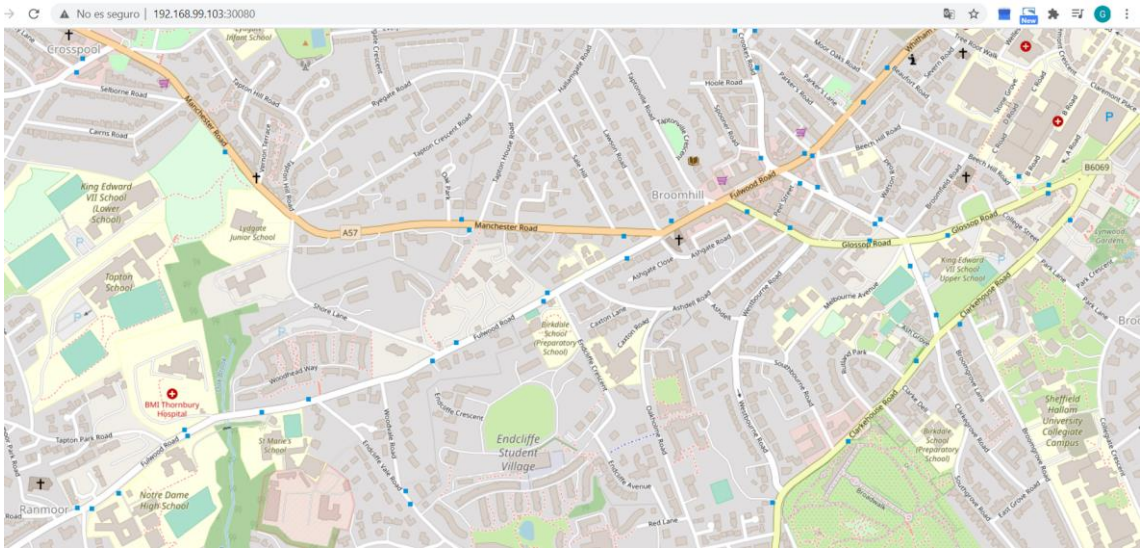


Figura 49. Release 0 del proyecto, primera versión.

Para un proyecto pequeño esto se puede utilizar, pero para un proyecto con mayor dimensión no es recomendable y se desperdiciaría mucho tiempo. La forma automática de ir o volver a diferentes versiones de la aplicación también se puede hacer con:

```
kubectl rollout history deploy webapp
deployment.extensions/webapp
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
```

El siguiente comando se puede utilizar para volver a una versión anterior del deployment. Por defecto, se volverá a la versión inmediatamente anterior, aunque si se proporciona el parámetro opcional "--to-revision" es posible indicar el número de la revisión a la que se desea volver:

```
kubectl rollout undo deploy webapp --to-revision=2
deployment.extensions/webapp rolled back
```

Es importante añadir que Kubernetes no vuelve a crear unas nuevos *replicaSet*, utiliza los mismos ya que sabe que estamos volviendo a distintas versiones de la aplicación y con el comando *history* de puede ver que se pueden tener hasta 10 versiones.

Una vez más se aprecia en la interfaz, el release 0-5 de la aplicación.

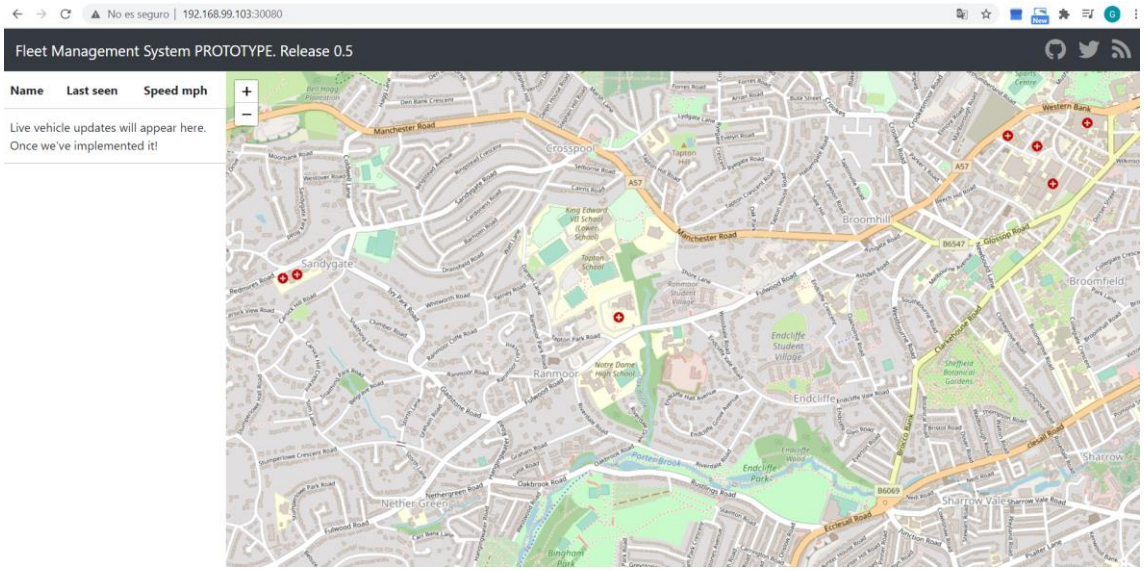


Figura 50. Release 0-5 desde la interfaz

Al realizar el *rollout* automático, el manifiesto de *pods* puede quedarse desactualizado. Hay que tener cuidado de no confundirse de versión y tener los manifiestos siempre actualizados con lo que se tiene en el sistema. Los *rollbacks* se utilizan solo en caso de emergencia, si alguna versión nueva ha producido fallo o no se encuentra la solución en un corto periodo de tiempo.

Por ejemplo, si una imagen que se tiene en el manifiesto que puede contener un error en la imagen, éste no se podría descargar, pero al tener las réplicas, Kubernetes utilizaría los anteriores *replicaSets* hasta que el nuevo *replicaSet* esté listo.

5.6. Escenario 5: Almacenamiento compartido entre nodos usando NFS

Este escenario pretende demostrar que varios nodos pueden compartir información no importan donde estén (nube, local, máquina virtual) con el master.

Para ello se ha utilizado el clúster creado en el apartado Clúster de múltiples nodos en local y se ha instalado en el master el servidor de NFS.

Configuración en el master

```
mkdir -p /var/compartido
```

Luego se describe en el archivo /etc/exports a qué equipo o rango de IPs se va a compartir esa carpeta y con qué permisos

```
/var/shared 10.200.0.0/16(rw, sync, no_subthree_check)
```

Una vez guardado el fichero se ejecuta el comando exportfs -a para exportar el directorio compartido. Luego se reinicia el servicio para que aplique los cambios

systemctl restart nfs-kernel-server.service

El siguiente comando sirve para comprobar qué carpeta se está compartiendo y a qué rango

```
showmount -e 127.0.0.1
Export list for 127.0.0.1:
/var/shared 10.200.0.0/16
```

También es importante permitir el tráfico a este rango de ips para que se pueda acceder al contenido de la carpeta compartida.

```
ufw allow from 10.200.0.0/16 to any port nfs
```

Verificar que el firewall este desactivo. Obteniendo el estado: inactive

```
ufw status
```

De este modo se tiene configurado en el master listo para compartir.

Configuración en los nodos

Se descarga el cliente NFS [39]

```
apt-get install nfs-common
```

Se crea una carpeta para acceder al contenido compartido por el servidor host (master)

```
mkdir -p /var/compartido-cliente
```

A continuación, se montan la carpeta compartido desde el host a esta carpeta que se acaba de crear

```
mount -t nfs4 10.100.0.94:/var/compartido /var/compartido-cliente
```

Una vez configurado esto, ahora ya se pueden crear los volúmenes en el master

Se crea el fichero persistencia.yaml para el PersistentVolume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
spec:
  capacity:
    storage: 4Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    path: /var/compartido
    server: 10.200.0.2
```

Luego se crea el PersistentVolumeClaim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Con el comando

```
kubectl get pv,pvc
```

Si se puede observar con el comando, los pvs en estado bound, dónde se puede decir que el PersistentVolumen está asociado con el PersistentVolumeClaim

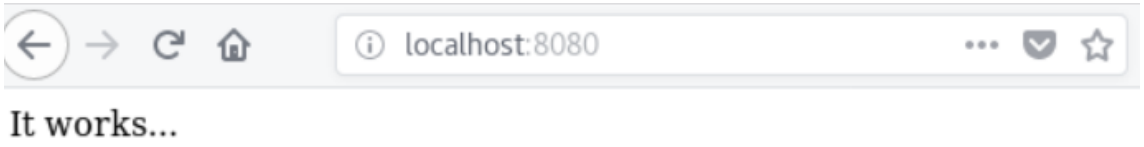
Por último, es importante crear un pod que haga uso de este volumen

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-volumen
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
      - mountPath: /usr/share/nginx/html
        name: nfs-vol
  volumes:
    - name: nfs-vol
      persistentVolumeClaim:
        claimName: nfs-pvc
```

El pod no tiene index, para ello se puede crear un index.html en la carpeta compartida del master, así todos los nodos pueden tener acceso a ella.

Se verifica que el index está, entrando de nuevo al pod.

```
curl http://localhost:8080
It works...
```



It works...

Se comprueba que el volumen se ha creado en el pod, accediendo al pod y ejecutando:

```
df -h
10.0.0.4:/var/shared   20G  4.8G   15G  26% /usr/share/nginx/html
```

Se pueden ir añadiendo más pods y no importan donde se ejecuten, porque todos los nodos conectados al clúster tienen la misma información y si se borrara el pod tampoco habría problema, ya que la información es persistente, bastaría con crear un nuevo pod.

Capítulo 6

6. Conclusiones y trabajo futuro

Se ha comprobado que Kubernetes está de moda y que toda la virtualización está en auge. Lo cierto es que además de ello es muy útil para múltiples casos, muchos otros más de los escenarios que se muestran en este proyecto. El uso de contenedores está muy extendido por muchas razones: coste, espacio, mantenibilidad y fácil de controlar. Aunque existen varios tipos de contenedores, en este trabajo y el mundo laboral Docker sale a relucir porque es que más se utiliza junto con otros orquestadores, pero Kubernetes es el más popular.

Con este trabajo se ha aprendido mucho, el potencial que tiene Kubernetes. En cada escenario se ha visto diferentes usos, cada escenario ha requerido tiempo para aprender utilidad y cómo implantarlo, ya que la documentación de Kubernetes es bastante extensa y compleja.

Uno de los principales objetivos de este trabajo es el estudio teórico de todo lo que ofrece Kubernetes y al haber terminado he salido de tener un conocimiento escaso a tener un conocimiento general mucho más profundo, sabiendo los principales conceptos y como utilizarlos, este conocimiento siempre va a ir a más con la experiencia especialmente para manejar los conceptos avanzados, ya que se puede tener escenarios infinitos en el mundo laboral con Kubernetes.

Trabajando como punto central ha sido la creación de los escenarios, de los cuales varios de ellos se han desarrollado utilizando máquinas virtuales simulando servidores físicos individuales como ha sido el escenario de almacenamiento compartido de ficheros.

Se ha puesto a prueba la escalabilidad y fiabilidad con Kubernetes utilizando HPA y deployments garantizando siempre la replicación de *pods* por si llegan a fallar.

Se han desarrollado cinco escenarios, pero sin lugar a duda el más retador y el que mas tiempo ha llevado por la complejidad de los conceptos que abarca es el escenario 3, que siempre puede ser mejorable añadiendo más nodos al clúster.

En el escenario 2, aunque la implantación de una base de datos en un pod está desaconsejada, es importante demostrar que Kubernetes puede hacerlo haciendo usos de otras características que ofrece que como es la persistencia, muy importante a la hora de replicar bases de datos.

Por último, hay que destacar los escenarios que me habría gustado desarrollar, ya que son tecnologías muy a la orden del día Openstack y Openshift.

Openstack que una plataforma *cloud* en local y compatible con Kubernetes y el clúster de Openshift para la gestión y administración de clústeres de Kubernetes.

Este escenario futuro que se plantea consiste en utilizar una máquina con sistema operativo Ubuntu con requisitos muy alto de memoria, dónde se instalaría microstack [40], gestor de la nube para crear el clúster de Kubernetes.

En la nube local Openstack, se crean las instancias master y *workers* y se conectan como un clúster de K8s. Para poner a prueba Openstack, lo siguiente que estaba pensado hacer es crear clústeres y que Openstack los pueda gestionar como otro clúster más.

Openshift, por su lado no se ha podido desarrollar como me habría gustado, debido a los conceptos muy avanzados que implica, que no se ha podido profundizar debido al tiempo y falta de servidor

que pueda soportar esa tecnología, por una parte y a la situación actual que se está viviendo por la pandemia global Covid-19.

Es importante añadir que Kubernetes siempre va a ir creciendo y añadiendo conceptos y tecnologías que se irán uniendo a él para poder tener un sistema de microservicio mucho más eficiente y menos acoplado, por lo que es un orquestador de contenedores que tendrá muchos años más de uso dentro de la grandes y medianas empresas.

Capítulo 7

7. Presupuesto

En este apartado se determinará el presupuesto necesario para llevar a cabo este proyecto. Algunos de los valores serán estimaciones lo más aproximadas posible.

7.1. Hardware

Se ha utilizado para creación de los clústeres del proyecto un ordenador portátil, con las siguientes características:

Componente	Precio
Portátil HP Pavilion 14-ce0xxx	800€
Procesador: Intel® Core™ i5-8250U CPU @ 1.60GHz 1.80GHz **Memoria instalada (RAM): 8,00GB** **Tipo de sistema: Sistema operativo de 64 bits, procesador x64**	--

7.2. Software

Se ha utilizado varios proveedores de Cloud de Kubernetes y algunas herramientas en local

AWS	**300€**
GCP	0€
AZURE	0€
Microsoft office 2016 (*student version*)	0€
VirtualBox	0€
Licencia SO Windows 10 Home Edition	145€

7.3. Presupuesto de recursos humanos

En este apartado se refleja todas las fases por la que ha pasado el proyecto y las horas dedicadas.

FASE	TASA	HORAS	TOTAL
Estudio	50 €/h	150	7500
Análisis	50 €/h	70	3500
Diseño	50 €/h	10	500
Implementación	50 €/h	70	3500
Prueba y error	50 €/h	100	5000
Documentación	50 €/h	150	7500
Revisión	50 €/h	40	2000
Total		590	29500

7.4. Presupuesto total del proyecto

Es la suma de todos los presupuestos mencionados anteriormente.

$\boldsymbol{Presupuesto} = P_{HW} + P_{SW} + P_{RH} =$ 800+445+29500=30.745€

Bibliografía

Bibliografía

[1] «What is Kubernetes?,» The Linux Foundation, 22 febrero 2019. [En línea]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Último acceso: 3 marzo 2019].

[2] J. M. Fiz, «Por qué todos apuestan por Kubernetes,» Paradigma Digital SL, 6 noviembre 2017. [En línea]. Available: https://www.paradigmadigital.com/techbiz/por-que-todos-apuestan-por-kubernetes/. [Último acceso: 3 marzo 2019].

[3] A. C. Matas, «Desarrollo de escenarios para la orquestación de funcionalidades de red y aplicación en redes 5G,» Universidad de Alcalá, Madrid, 2018.

[4] V. Cuervo, 20 02 2019. [En línea]. Available: http://www.arquitectoit.com/docker/contenedores-vs-maquinas-virtuales/.

[5] Ackstorm, «Comparativa orquestadores: Mesos vs Kubernetes vs Swarm,» 02 06 2016. [En línea]. Available: https://www.ackstorm.com/kubernetes-vs-docker-vs-mesos/.

[6] T. K. Authors, «Borg: The Predecessor to Kubernetes,» 23 04 2015. [En línea]. Available: https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/.

[7] The Kubernetes Authors, «Kubernetes Concepts,» 03 02 2020. [En línea]. Available: https://kubernetes.io/docs/concepts/services-networking/service/.

[8] K. T. Mumshad Mannambeth, «Kubernetes Certified Application Developer (CKAD) with Tests,» 01 2020. [En línea]. Available: https://www.udemy.com/course/certified-kubernetes-application-developer/learn/lecture/12299468#announcements/3036080.

[9] The Kubernetes Authors, «SERVICE APIs,» 10 01 2020. [En línea]. Available: https://v1-16.docs.kubernetes.io/docs/reference/generated/kubernetes-api/v1.16/#service-v1-core. [Último acceso: 28 09 2020].

[10] Read the Docs, «Configure pod container,» 2020. [En línea]. Available: https://unofficial-kubernetes.readthedocs.io/en/latest/tasks/configure-pod-container/configmap/.

[11] The Kubernetes Authors, «Uso de un servicio para exponer su aplicación,» 15 03 2020. [En línea]. Available: https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/. [Último acceso: 15 06 2020].

[12] The Linux Foundation, «Container Lifecycle Hooks,» 10 10 2019. [En línea]. Available: https://kubernetes.io/es/docs/concepts/containers/container-lifecycle-hooks/.

[13] Ichasco, «Kubernetes: Primeros pasos con Minikube,» 04 06 2018. [En línea]. Available: https://blog.ichasco.com/kubernetes-primeros-pasos-con-minikube/.

[14] Creative Commons Attribution, Apache 2.0, «Google cloud,» 18 10 2019. [En línea]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/daemonset?hl=es-419.

[15] The Kubernetes Authors, «Conceptos. DaemonSet,» 30 05 2020. [En línea]. Available: https://kubernetes.io/es/docs/concepts/workloads/controllers/daemonset/. [Último acceso: 15 06 2020].

[16] etomas, «Kubernetes (3) – Controladores Ingress,» 03 01 2018. [En línea]. Available: https://geeks.ms/etomas/2018/01/03/kubernetes-3-controladores-ingress/. [Último acceso: 22 06 2020].

[17] MkDocs and Material for MkDocs, «Nginx Ingress Controller,» 2020. [En línea]. Available: https://kubernetes.github.io/ingress-nginx/deploy/#minikube.

[18] The Kubernetes Authors, «Kubernetes,» 15 01 2020. [En línea]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/.

[19] The Linux Foundation ®, «Using RBAC Authorization,» 11 02 2020. [En línea]. Available: https://kubernetes.io/docs/reference/access-authn-authz/rbac/.

[20] J. D. Muñoz, «Pledin 3.0,» 19 03 2019. [En línea]. Available: https://www.josedomingo.org/pledin/2019/03/almacenamiento-kubernetes/.

[21] The Kubernetes Authors, «Volumes,» 08 09 2020. [En línea]. Available: https://kubernetes.io/docs/concepts/storage/volumes/. [Último acceso: 18 09 2020].

[22] D. K. Rensin, Kubernetes: Scheduling the Future at Cloud Scale, CA: O'Reilly Media, 2016, pp. 1-36.

[23] The Kubernetes Authors, «Setup,» 2020. [En línea]. Available: https://kubernetes.io/de/docs/setup/.

[24] GitHub, Inc., «Installing kops (Binaries),» 16 11 2018. [En línea]. Available: https://github.com/kubernetes/kops/blob/master/docs/install.md. [Último acceso: 19 09 2020].

[25] Cloud Native Foundation Project, «The package manager for Kubernetes,» 2020. [En línea]. Available: https://helm.sh/.

[26] GitHub, Inc., «The Kubernetes Package Manager,» 2020. [En línea]. Available: https://github.com/helm/helm.

[27] Cloud Native Computing Foundation, «From metrics to insight,» 2020. [En línea]. Available: https://prometheus.io/.

[28] Grafana Labs, «The open observability platform,» 2020. [En línea]. Available: https://grafana.com/.

[29] V. P. P. Richard Chesterwood, «Kubernetes Hands-On - Deploy Microservices to the AWS Cloud,» 08 2019. [En línea]. Available: https://www.udemy.com/kubernetes-microservices/learn/lecture/14727678#questions.

[30] Amazon Web Services, Inc. , «Simulating Site-to-Site VPN Customer Gateways Using strongSwan,» 2020. [En línea]. Available: https://aws.amazon.com/es/blogs/networking-and-content-delivery/simulating-site-to-site-vpn-customer-gateways-strongswan/. [Último acceso: 23 09 2019].

[31] J. M. González, «Todo lo que deberías saber sobre las redes virtuales en Amazon AWS,» 9 10 2018. [En línea]. Available: https://www.josemariagonzalez.es/amazon-web-services-aws/todo-lo-que-deberias-saber-sobre-las-redes-virtuales-en-amazon-aws.html.

[32] Aviatrix Systems, «What is a Virtual Private Gateway (VGW)?,» 2020. [En línea]. Available: https://a.aviatrix.com/learning/glossary/vgw.php.

[33] AWS Training Center, «AWS Setup Site to Site VPN Connection,» 02 2019. [En línea]. Available: https://awstrainingcenter-test.s3-us-west-2.amazonaws.com/10+-+Setup+Site+to+Site+VPN+Connection+in+AWS.pdf. [Último acceso: 10 2019].

[34] D. Igbe, «Kubeadm on AWS,» 20 01 2018. [En línea]. Available: https://www.cloudtechnologyexperts.com/kubeadm-on-aws/.

[35] strongswan.org, «strongSwan,» 29 07 2020. [En línea]. Available: https://www.strongswan.org/. [Último acceso: 09 23 2020].

[36] Amazon Web Services, Inc. or its affiliates, «AWS. How can I configure NAT on my VPC CIDR for traffic traversing a VPN connection?,» 17 07 2018. [En línea]. Available: https://aws.amazon.com/es/premiumsupport/knowledge-center/configure-nat-for-vpn-traffic/.

[37] Ruan, «Setup a Site to Site IPSec VPN with Strongswan on Ubuntu,» 12 02 2018. [En línea]. Available: https://sysadmins.co.za/setup-a-site-to-site-ipsec-vpn-with-strongswan-on-ubuntu/.

[38] Amazon Web Services, Inc. or its affiliates, «AWS,» 15 11 2017. [En línea]. Available: https://aws.amazon.com/es/premiumsupport/knowledge-center/vpn-cgw-vpg-traffic/.

[39] D. Naranjo, «Instala NFS en Ubuntu y comparte tus archivos en red con este protocolo,» [En línea]. Available: https://ubunlog.com/instala-nfs-en-ubuntu-y-comparte-tus-archivos-en-red-con-este-protocolo/.

[40] Canonical Ltd., «Get started with MicroStack,» 2020. [En línea]. Available: https://ubuntu.com/tutorials/microstack-get-started#2-install-microstack. [Último acceso: 01 09 2020].

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR