ULPGC
Universidad de
Las Palmas de
Gran Canaria

eii
ESCUELA DE
INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

Análisis de rendimiento y escalabilidad entre Docker Swarm y Kubernetes para el despliegue de servicios de Inerza

TITULACIÓN: Grado en Ingeniería Informática

AUTOR: Jose Manuel Hernández Aparicio

TUTORIZADO POR:
Nelson Manuel Monzón López
Carmelo León Suárez

9 de julio de 2024

Agradecimientos

En primer lugar, me gustaría agradecer a los tutores, Nelson Monzón y Carmelo León por ayudarme y resolver mis dudas siempre que lo he necesitado y con especial celeridad.
También me gustaría agradecer a todos mis amigos por hacer este viaje más ameno y a mi familia por apoyarme y confiar en mi todos estos años, en especial, durante las etapas más duras.

Resumen

Este Trabajo Fin de Título persigue el estudio de dos de las tecnologías más punteras del sector en lo relativo a los contenedores, su orquestación y el despliegue de aplicativos de microservicios. Se describirá la creación de un entorno virtualizado compuesto de varias MVs para poder desplegar un clúster de Kubernetes, instalando y configurando los servicios necesarios para la creación del mismo usando la herramienta Kubeadm.

Acto seguido se explica la configuración, testeo y modificación de varios de los componentes del mismo hasta llegar a un estado ideal en el que se ejecutaron pruebas de carga y escalado bastante simples pero funcionales con un aplicativo de microservicios.

Abstract

This Final Degree Project pursues a preliminary study of the cutting-edge technologies in the sector related to containers, their orchestration, and the deployment of microservices applications. It will describe the creation of a virtualized environment composed of several VMs to deploy a Kubernetes cluster, installing and configuring the necessary services for its creation using the Kubeadm tool.

Next, the configuration, testing, and modification of several components will be explained until reaching an ideal state in which fairly simple but functional load and scaling tests were performed with a microservices application.

Índice general

1. Introducción **1**
1.1. Contexto . . . 1
1.2. La Orquestación . . . 3
1.3. Origen de este trabajo . . . 4
1.3.1. Inerza . . . 4
1.3.2. Contexto del Trabajo . . . 5
1.4. Objetivos del proyecto . . . 5
1.5. Presupuesto . . . 6
1.5.1. Recursos de Hardware . . . 6
1.5.2. Recursos Humanos . . . 6
1.5.3. Recursos de Software . . . 6
1.5.4. Presupuesto Final . . . 7
1.6. Competencias Específicas y Aportaciones . . . 7

2. Estado del arte y objetivos iniciales **9**
2.1. Docker . . . 9
2.2. Docker Swarm . . . 11
2.3. Kubernetes . . . 12
2.4. Terraform . . . 13
2.5. Alternativas a la Orquestación directa . . . 14
2.5.1. Azure Container Instances . . . 14
2.5.2. AWS Fargate . . . 15

3. Desarrollo **16**
3.1. Metodología . . . 16
3.1.1. Kanban . . . 16
3.2. Creación del Clúster . . . 18
3.2.1. Herramientas disponibles . . . 18
3.2.2. Creación de la infraestructura . . . 19
3.2.3. Elección e Instalación de un CRI (Container Runtime Interface . . . 21
3.2.4. Instalación de Kubeadm e Inicialización del clúster . . . 25
3.2.5. Instalación de un Pod Network add-on . . . 28

3.2.6. Configuración de Almacenamiento 31
3.3. Conversión de aplicativo en Docker a Kubernetes 43
3.4. Configuración de Escalado Automático 49
3.4.1. HPA y Metrics Server 49
3.4.2. Instalación Metrics Server 50
3.4.3. Creación de un HPA 52

4. Conclusiones y trabajo futuro **59**
4.1. Conclusiones 59
4.2. Trabajo Futuro 62

Índice de figuras

3.1. Verificación de módulos cargados . . . 23
3.2. Verificación de parámetros kernel . . . 24
3.3. Output kubeadm init . . . 26
3.4. Nodos disponibles en el clúster . . . 27
3.5. Output pods de Calico . . . 30
3.6. Output Instalación OpenEBS . . . 33
3.7. Output Pods OpenEBS . . . 33
3.8. Output Get SC . . . 34
3.9. Output Get PVC . . . 35
3.10. Output Get JivaVolumes . . . 35
3.11. Output Get Rook-Ceph Pods . . . 39
3.12. Output Get Rook-Ceph Pods . . . 40
3.13. Output Get Rook-Ceph SC . . . 42
3.14. Output Metrics Server Deployment . . . 51
3.15. Output Top Nodes . . . 52
3.16. Output Get HPA . . . 56
3.17. Inicialización de Locust . . . 57
3.18. Interfaz Gráfica Locust . . . 58
3.19. Locust Ongoing . . . 58
3.20. Watch HPA . . . 58

Capítulo 1

Introducción

Per aspera ad astra

Pierce Brown - Amanecer Rojo

1.1. Contexto

A lo largo de la historia de la informática, siempre han existido problemas en la ejecución de software en diferentes sistemas. La principal causa de estos problemas ha sido la disparidad entre las versiones en las que se desarrolló el software y las del sistema anfitrión.

Eran frecuentes las situaciones en las que, dentro de un equipo de desarrollo, surgían conflictos debido a que algunos miembros podían tener instalada una versión diferente de ciertas dependencias del proyecto o emplear sistemas operativos distintos. Esto podía resultar en un rendimiento inferior al esperado o en errores catastróficos durante la ejecución del software.

En informática, se entiende por "contenedor" [5] [34] un paquete que incluye todos los elementos necesarios para ejecutar de forma aislada cualquier tipo de software, independientemente del entorno.

Los contenedores se diferencian de las máquinas virtuales en varios aspectos. Para em-

pezar, la mayoría de los sistemas de virtualización más potentes emplean un hipervisor para virtualizar recursos de hardware. En cambio, los contenedores virtualizan el sistema operativo, compartiendo el núcleo del mismo, pero aislándose a nivel de proceso. Esto hace que las máquinas virtuales consuman muchos más recursos que las tecnologías de contenedores, volviendo estas últimas una opción mucho más atractiva para el despliegue de aplicaciones y microservicios.

Podría decirse que el concepto surgió de forma primitiva en 1982, cuando Bill Joy, uno de los padres de Unix, motivado por la inestabilidad que suponía desarrollar y hacer pruebas en el mismo entorno, creó la llamada al sistema chroot(), la cual permitía cambiar el directorio raíz de un proceso y sus hijos a uno nuevo, proporcionando un nivel primitivo de aislamiento.

Sin embargo, el primer sistema de contenedores real fue “FreeBSD Jails”, desarrollado en 1999 y liberado en 2000. Consistía en una virtualización a nivel de sistema operativo que brindaba una forma robusta de ejecutar procesos en un entorno aislado, permitiendo la segmentación del sistema en subconjuntos con sus propios espacios de usuario y direcciones IP.

Aunque el concepto se asemeja al que tenemos actualmente, podríamos afirmar que una aproximación bastante similar fue LXC (Linux Containers), creado en 2008. LXC fue el primer software de gestión de contenedores. Utilizaba control groups, una característica del núcleo de Linux introducida en la versión 2.6.24 en 2007, que permitía organizar y limitar los recursos del sistema.

LXC hacía uso de este y otros recursos nativos del núcleo de Linux para proporcionar integración directa con el mismo, sin necesidad de ”parchearloçomo hacían otras tecnologías de la época. Esto permitió a LXC proporcionar contenedores ligeros y eficientes sin necesidad de capas de abstracción pesadas.

No obstante, todas estas tecnologías comparten algo en común: todas son relativamente complejas de entender y utilizar para el desarrollador común y estaban limitadas en muchos aspectos. Todo eso cambió en 2013 con la llegada de Docker, una plataforma de software que permite crear, desplegar y gestionar aplicaciones conterizadas de una forma mucho más fácil, cómoda y universal que catapultó el uso de los contenedores en la industria.

1.2. La Orquestación

Con el auge del uso de los contenedores y de las arquitecturas de microservicios, las organizaciones se han enfrentado al desafío de gestionar y mantener de manera eficiente un número creciente de contenedores, los cuales necesitan ser desplegados, escalados y administrados de manera coherente y segura. [8] [6]

Al desplegar una aplicación, se vuelve necesario crear y configurar los balanceadores de carga, los servicios y el comportamiento que tendrán los contenedores en ellos. Además, a medida que aumenta el número de contenedores gestionados, se hace más difícil de controlar por parte de un equipo. Es común que, a medida que aumenta la demanda de una aplicación, sea necesario incrementar el número de contenedores dedicados a la ejecución de un microservicio. Añadir o reducir manualmente el número de estos según la demanda no es viable.

La orquestación de contenedores surge como respuesta para suplir todas estas necesidades, automatizando todas estas tareas en base a un archivo declarativo, similar a un contenedor de Docker, donde se define la configuración deseada y el sistema de orquestación elegido usará una lógica opaca al desarrollador para llegar a esa configuración elegida.

En dicho archivo, se define qué imágenes de contenedor componen la aplicación, se especifica de qué sistemas de almacenamiento y otros recursos dispondrá, y se establecen las conexiones de red oportunas. Dependiendo del sistema elegido, también se define el número mínimo de contenedores de cada servicio deseado, así como los métodos que utilizará el orquestador para comprobar el estado actual del contenedor.

La herramienta de orquestación seleccionará el host en el que alojar los contenedores en función de los recursos disponibles y de las limitaciones impuestas. Asimismo, facilitará la disposición de sistemas redundantes en caso de que alguno falle y gestionará las actualizaciones a nuevas versiones sin que haya interrupciones en los servicios.

Además, gestionará la escalabilidad de los servicios, ya sea hacia arriba o hacia abajo de forma segura, el equilibrio de carga y la asignación de recursos dentro de cada host.

Otra ventaja que ofrecen los sistemas de orquestación es la recopilación y almacenamiento

unificado de logs del sistema, así como la telemetría utilizada para monitorizar las llamadas a cada servicio y su rendimiento.

Según un estudio de IBM en 2020, el 70 % de los desarrolladores que trabajan con contenedores utilizan alguna solución de orquestación para gestionarlos. Hoy podemos afirmar que la orquestación de contenedores se ha vuelto obligatoria para despliegues en producción de aplicaciones complejas.

1.3. Origen de este trabajo

1.3.1. Inerza

Inerza [9] es una consultoría TIC de origen Canario y con varios años de experiencia, se especializa en tecnología y servicios informáticos, ofreciendo soluciones que abarcan desde el desarrollo de software, alojamiento de servicios, gestión de infraestructura y el asesoramiento tecnológico a todos los niveles, en la cual el co-tutor, Carmelo León, ocupa un cargo de responsabilidad dentro del departamento de Ingeniería y Sistemas.

Si bien también ofrece servicios a entidades privadas, la mayor parte de sus clientes son organismos públicos, los cuales tienden a ser muy recelosos con su información y la integridad de la misma, esto hace que Inerza deba minimizar el uso de servicios externos, como pueden ser proveedores de servicio en la nube, por tanto, requieren de su propio Centro de Procesamiento de Datos, en el que actualmente tienen desplegado un clúster de Docker Swarm.

Inerza mantiene un estrecho lazo de cooperación con la ULPGC, realizando cursos de formación totalmente gratuitos y brindando un programa de prácticas externas curriculares bastante amplio, del cual nace este proyecto.

1.3.2. Contexto del Trabajo

El contexto general de este trabajo se enmarca dentro de una colaboración como parte del programa de prácticas con la empresa Inerza. Actualmente, Inerza dispone de un Centro de Procesamiento de Datos (CPD) propio en el que tiene desplegado un clúster de Docker Swarm, la intención de la empresa era proporcionarme una formación en todo lo relativo a Sistemas, Contenedores, DevOps y Orquestación en escenarios realesꝛ culminar este proceso realizando un análisis de viabilidad entre un clúster de Docker Swarm y uno de Kubernetes en calidad de TFT.

Para poder realizar una comparativa real, se procedió a la creación de un clúster de Kubernetes completamente funcional en local, labor que fue suficiente como para acabar componiendo la mayor parte de este trabajo.

El objetivo de este TFT consistía en realizar una comparativa directa, basada métricas medibles, entre Docker Swarm y Kubernetes, no obstante, conforme se fue avanzando en el proyecto, se llegó a la conclusión de que **no tiene sentido** realizar dicha comparativa.

Si bien se explicará en detalle más adelante, conviene destacar en este punto que ambos orquestadores son totalmente diferentes y no sería adecuado realizar una comparativa en ese sentido ya que no están al mismo nivel.

A continuación, se procederá a describir el estado actual del sector en lo relativo a estas tecnologías, continuará con una explicación detallada sobre la creación de un clúster de Kubernetes y se terminará con unas conclusiones sobre ambas tecnologías y una valoración a nivel personal sobre si le conviene o no a Inerza abandonar Docker Swarm para implementar un clúster de Kubernetes

1.4. Objetivos del proyecto

Lo objetivos definidos en el TFT01 consistían en la comparación directa entre Docker Swarm y Kubernetes, como ya ha sido explicado previamente, esa comparación carece de sentido debido a que si bien son herramientas del mismo estilo, suplen necesidades diferentes y no sería correcto compararlas al mismo nivel.

Se podría afirmar que los objetivos finales de este trabajo han consistido en crear un

clúster de Kubernetes para poder presentarle a Inerza una valoración actualizada sobre su complejidad, ventajas y desventajas.

1.5. Presupuesto

En esta sección se procederá a valorar los costes de diferentes aspectos del proyecto

1.5.1. Recursos de Hardware

El elevado número de máquinas virtuales ejecutándose simultáneamente, así como los recursos asignados a las mismas, requirió de un ordenador de sobremesa de altas prestaciones:

Recurso	**Coste**
Ordenador de Sobremesa	3000 €
Total	3000 €

1.5.2. Recursos Humanos

Este apartado comprende tanto el tiempo invertido del estudiante como el de los tutores, recordemos que con Carmelo León se realizaban reuniones diarias de corta duración.

Personal	**Nº Horas**	**Coste/Hora**	**Coste Total**
Estudiante	270	15 €	4050 €
Tutor académico	25	50 €	1250 €
Tutor empresa	25	50 €	1250 €
Total			6550 €

1.5.3. Recursos de Software

Todos los recursos de software empleados son de código abierto y por tanto gratuitas (VirtualBox, Visual Studio Code, etc), con la excepción de Terminus, si bien no ha sido necesario pagar debido a que ceden la licencia a estudiantes de forma gratuita.

1.5.4. Presupuesto Final

El presupuesto final del proyecto sería el siguiente:

Tipo de Recurso	Coste Final
Recursos de Hardware	3000 €
Recursos Humanos	6550 €
Recursos de Software	0 €
Coste Total del Proyecto	9550 €

1.6. Competencias Específicas y Aportaciones

- **CI1** Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente.

- **CI2** Capacidad para planificar, concebir, desplegar y dirigir proyectos, servicios y sistemas informáticos en todos los ámbitos, liderando su puesta en marcha y su mejora continua y valorando su impacto económico y social.

- **CI5** Conocimiento, administración y mantenimiento de sistemas, servicios y aplicaciones informáticas.

- **CI8** Capacidad para analizar, diseñar, construir y mantener aplicaciones de forma robusta, segura y eficiente, eligiendo el paradigma y los lenguajes de programación más adecuados

- **CI10** Conocimiento de las características, funcionalidades y estructura de los sistemas operativos y diseñar e implementar aplicaciones basadas en sus servicios.

- **CI11** Conocimiento y aplicación de las características, funcionalidades y estructura de los sistemas distribuidos, las redes de computadores e Internet y diseñar e implementar aplicaciones basadas en ellas.

Este TFT aporta a Inerza y a la Comunidad Universitaria un análisis en profundidad sobre Kubernetes, sus componentes y recursos más importantes, además de una valoración actualizada sobre sus puntos fuertes, débiles y esfuerzo necesario para la

implementación y mantenimiento del mismo.

Además, este documento constituye una guía detallada de los pasos necesarios para la creación de un clúster totalmente funcional de este.

Capítulo 2

Estado del arte y objetivos iniciales

Como ya mencionamos en la introducción, los contenedores y la orquestación de los mismos son una parte vital para el funcionamiento de la industria tecnológica, en cuanto a los contenedores, el estándar de facto del sector es, indiscutiblemente Docker. En lo relativo a la orquestación, las dos principales alternativas son Docker Swarm y Kubernetes, no obstante, existen ciertas alternativas en lo relativo a la elección de éstos.

Todos estos aspectos serán vistos a lo largo de este capítulo, así como los objetivos iniciales de este TFG.

2.1. Docker

Docker [31] fue creado en 2013 por Solomon Hykes como un proyecto de código abierto, permitiendo a los desarrolladores automatizar el despliegue de aplicaciones en contenedores de manera eficiente y sencilla. Sus características distintivas se han convertido en un estándar en el ámbito de los contenedores. Tal es su impacto, que el término çontenedores"se asocia comúnmente con Docker, aunque no es la única tecnología de contenedores en uso actualmente.

Inicialmente, Docker utilizaba LXC como su plataforma de ejecución predeterminada, pero en 2014 la reemplazó con su propia biblioteca, libcontainer, escrita en Go, un lenguaje de programación concurrente desarrollado por Google, que ofrece seguridad de memoria y recolección de basura.

Docker sobresalió frente a otras tecnologías por su facilidad de uso, portabilidad y eficiencia. Sus contenedores son significativamente más eficientes y ligeros que sus contrapartes en el momento de su popularización. Además, introdujo una forma sencilla de distribución de contenedores, empaquetando toda la lógica en dos tipos de archivos, Dockerfile y docker-compose, que se detallarán posteriormente.

Estos archivos permiten automatizar la instalación y configuración de los contenedores, simplificando su despliegue y escalabilidad. Junto con repositorios para compartir imágenes de contenedores en línea, como Docker Hub, facilitan enormemente la gestión de la infraestructura de aplicaciones de manera declarativa y versionable.

Los contenedores de Docker utilizan el kernel del sistema operativo del host, sin depender de las particularidades del sistema operativo o de su configuración específica. Este nivel de aislamiento garantiza la correcta ejecución del código independientemente del entorno, facilitando enormemente el despliegue de las aplicaciones en entornos de desarrollo, prueba o producción.

La combinación de estas características, junto con una comunidad activa y proyectos de código abierto, ha establecido a Docker como la solución líder en el mundo de la contenerización, marcando un antes y un después en el desarrollo y despliegue de aplicaciones.

Los dos componentes principales de Docker son las imágenes y los contenedores. Las imágenes son plantillas de solo lectura que representan un sistema de archivos y parámetros listos para ejecutar, generalmente basadas en sistemas tipo Linux y que contienen tanto la aplicación a ejecutar como sus dependencias. Estas imágenes se construyen a partir de archivos Dockerfile, que son archivos de texto plano en los que se especifica cómo debe construirse la imagen. Suelen partir de una imagen base, como un sistema tipo Ubuntu u otra imagen personalizada, y detallan las instrucciones a seguir, como la instalación de software adicional y la copia de archivos entre el host y el contenedor.

Un contenedor de Docker es una instancia de esa imagen. Técnicamente, es un directorio dentro del sistema que funciona de manera similar a las ”jail root”. Se puede ver como una ejecución en tiempo real en la que se añade una capa de escritura sobre la imagen inmutable, permitiendo la ejecución y modificación de archivos durante su ciclo de vida sin alterar la imagen original.

Los contenedores son efímeros y aislados; pueden ser iniciados, detenidos y eliminados. Una vez eliminados, los datos que han manipulado se pierden, por lo que es necesario almacenarlos en un “volumen” externo, otra característica de Docker. Su ejecución se realiza de forma totalmente aislada respecto a otros contenedores, aunque pueden comunicarse entre sí compartiendo volúmenes o a través de definiciones de red.

2.2. Docker Swarm

Docker Swarm [4] fue el primer orquestador diseñado específicamente para Docker, introducido en 2014 como parte de su ecosistema para proporcionar capacidades de orquestación de forma nativa a los usuarios de Docker. Ofrece funcionalidades que facilitan el escalado de servicios, el balanceo de carga y el mantenimiento de los contenedores, aunque de manera menos potente en comparación con otras tecnologías de orquestación.

Podríamos decir que una de las principales ventajas de Docker Swarm es su integración nativa y su compatibilidad directa con la interfaz de comandos de Docker, lo que permite a los usuarios configurar y gestionar clústeres usando comandos ya familiares. Esto resulta en una curva de aprendizaje más amigable en comparación con otras tecnologías de orquestación, convirtiéndolo en una solución más accesible para aquellos que solo están familiarizados con Docker.

El uso de toda la infraestructura existente de Docker convierte a Docker Swarm en una de las alternativas de orquestación más ligeras y eficientes, aunque esto implica ofrecer menor funcionalidad en comparación con sus competidores.

Docker Swarm distingue entre dos tipos de nodos: los nodos *manager*, que se encargan de gestionar el estado del clúster y tomar decisiones propias del orquestador, y los nodos *worker*, cuya función es ejecutar los servicios asignados por los nodos *manager* y no tienen ningún tipo de decisión en el funcionamiento del clúster; simplemente ejecutan tareas.

Para comprender el funcionamiento de este y otros sistemas de orquestación, es esencial entender los conceptos de servicio y tarea.

Un **servicio** es una definición de alto nivel sobre cómo se debe ejecutar una aplicación, incluyendo la imagen del contenedor a usar, su configuración y el número de instancias mínimas deseadas, entre otros parámetros. Representa un estado deseado, y Docker Swarm se encarga de alcanzar y mantener ese estado gestionando automáticamente las instancias necesarias.

Los servicios pueden ser de dos tipos:

- **Servicios Replicados**: En estos, se especifica el número de réplicas idénticas del contenedor que se desean tener en ejecución en el clúster, dejando a Docker Swarm la distribución y el mantenimiento de estas.

- **Servicios Globales**: Aseguran que una única instancia del contenedor se ejecute en cada nodo *worker* del clúster, útil para tareas de monitorización o seguridad.

Las **tareas**, por su parte, representan una instancia de un contenedor que forma parte de un servicio y constituyen la unidad más pequeña de despliegue gestionada por Docker Swarm. Si se especifica que un servicio debe tener cinco réplicas, Docker Swarm creará y gestionará cinco tareas, cada una representando una instancia del contenedor definido en el servicio.

Cuando Docker Swarm escala un servicio, ya sea de forma manual por parte del administrador o mediante sus limitadas capacidades de autoescalado, añadirá o eliminará tareas hasta alcanzar el número de réplicas deseado. Además, si alguna tarea falla, Docker Swarm la detectará mediante los *health checks* configurados y creará otra automáticamente en uno de los nodos disponibles.

Docker Swarm surgió como una tecnología de orquestación ideal para usuarios y organizaciones que buscaban una solución fácil de entender y gestionar. No obstante, estas cualidades tienen una contrapartida: para muchos, Docker Swarm es demasiado simple y limitado. Por ello, está siendo superado en popularidad por Kubernetes, una alternativa mucho más potente y compleja.

2.3. Kubernetes

Kubernetes [16], también conocido como K8s, es una tecnología de orquestación de contenedores de código abierto, revolucionaria por la manera en la que despliega, escala y gestiona las aplicaciones contenerizadas. Fue desarrollado originalmente por Google y utilizado de manera interna bajo el nombre oficial de Project Seven (aunque, para abreviar, se le llamaba Borg). Posteriormente, fue donado a la Cloud Native Computing Foundation (parte de la Linux Foundation).

Se popularizó rápidamente debido a su capacidad para automatizar el despliegue de aplicaciones, escalarlas automáticamente según la demanda u otros parámetros, y gestionar el estado deseado de las aplicaciones para asegurar su disponibilidad y resiliencia. Se podría decir que su característica más importante es su modularidad, es decir, el administrador de un clúster de Kubernetes puede modificar o intercambiar la inmensa mayoría de sus componentes parar adecuarse a las necesidades específicas.

La unidad más pequeña y básica que se puede crear en Kubernetes es el Pod. Un Pod

representa una instancia de una aplicación y puede contener uno o varios contenedores en ejecución, los cuales comparten recursos de ejecución y dirección IP es decir, se pueden comunicar entre ellos mediante localhost. También comparten volúmenes y son gestionados por entidades superiores como si fueran una sola entidad.

Los Pods son efímeros por naturaleza, creados y destruidos dinámicamente para adaptarse a las necesidades del momento.

La arquitectura de Kubernetes se basa en el despliegue de uno o varios clusters. Cada clúster se compone, como mínimo, de un Control Plane y un nodo.

El concepto de Control Plane es similar al de los nodos manager de Docker Swarm. Es responsable de la toma de decisiones globales sobre el clúster, así como de la detección y respuesta a determinados eventos que pueden suceder en él. Cualquier host del clúster puede albergar el Control Plane; sin embargo, este se inicia por defecto en el host que inicia el clúster y evita ejecutar ajenos a la gestión del clúster para asegurar la mayor cantidad de recursos posibles.

Los nodos worker, como su propio nombre indica, son los nodos responsables de ejecutar el grueso de las necesidades propias de cada despliegue de aplicativos.

2.4. Terraform

Terraform [29] es una herramienta de infraestructura como código desarrollado por HashiCorp, se le podría definir como un orquestador de infraestructura. Permite definir y aprovisionar la infraestructura de sistemas mediante un lenguaje de configuración declarativo para que resulte mucho más amigable.

Esto permite desarrollar infraestructura de una forma “versionable”, consiguiendo que se pueda reutilizar y compartir en caso de ser necesario.

El funcionamiento se basa en la creación de un plan de ejecución, el cual determina qué acciones son necesarias para llegar a un estado deseado, luego, ejecuta todas las acciones especificadas hasta llegar a dicho estado. Esto facilita enormemente la tarea de los administradores de sistemas a la hora de configurar las infraestructuras.

Al igual que Docker Swarm y Kubernetes, almacena el estado deseado en la configura-

ción y mapea continuamente los recursos desplegados, hace un seguimiento de estos, evita duplicaciones y corrige desviaciones en caso de que algún componente falle.

Si bien Terraform es antiguo, últimamente se está volviendo extremadamente popular en el sector y tiene muy buena compatibilidad con Kubernetes.

2.5. Alternativas a la Orquestación directa

Las opciones de orquestación que hemos visto son las formas más ”tradicionales”de gestionar contenedores, no obstante, con el auge de los proveedores de servicios en la nube, se están popularizando alternativas diferrentes, algunas de las más usadas son las siguientes:

2.5.1. Azure Container Instances

Azure Container Instances (ACI)[2] es un servicio proporcionado por Microsoft Azure, la plataforma de computación en la nube de Microsoft. Este servicio permite ejecutar contenedores directamente en la nube de Azure sin la necesidad de gestionar máquinas virtuales o adoptar servicios adicionales de orquestación como Kubernetes.

ACI está diseñado específicamente para ser fácil de usar. Permite a los desarrolladores proporcionar una imagen de contenedor, y ACI se encarga del despliegue siguiendo las directrices de manera abstracta para el desarrollador.

Como es de esperar, ACI está integrado con otros servicios de Microsoft, como Azure Functions, un servicio serverless similar a Amazon Lambda, que permite a los desarrolladores ejecutar funciones o pequeños scripts en respuesta a eventos. También está integrado con Azure Event Grid, que es similar a Azure Functions pero utiliza contenedores en lugar de pequeños bloques de código.

ACI es ideal para la ejecución de aplicaciones sin estado, en las que no es necesario mantener información entre ejecuciones, como pueden ser tareas de transformación de datos o tareas de CI/CD (integración continua y entrega e implementación continua), es decir, la ejecución de diferentes pruebas sobre un código nuevo. Por lo tanto, es una herramienta ideal en entornos de desarrollo y pruebas.

Además, sigue un modelo de pago por uso, en el que el desarrollador solo paga por los recursos consumidos (CPU, memoria y tiempo de ejecución), lo que lo convierte en una alternativa interesante desde una perspectiva de coste-eficiencia.

ACI está orientado a escenarios que requieren una solución simple y en los que no se desee invertir mucho tiempo en la gestión de contenedores, es la alternativa ideal en caso de no requerir un despliegue de producción complejo y exigente.

2.5.2. AWS Fargate

AWS Fargate [28] es un servicio de contenedores que permite a los desarrolladores ejecutar contenedores en Amazon Web Services sin necesidad de administrar servidores o clústeres. Fargate elimina completamente la necesidad de aprovisionar, configurar y escalar clústeres para ejecutar contenedores sin tener que configurar un orquestador de manera independiente, lo que simplifica enormemente el proceso de despliegue de aplicaciones contenerizadas. Sería el equivalente de Amazon al ACI de Microsoft.

Evidentemente, AWS Fargate se integra con el resto de las herramientas y servicios de AWS, proporcionando un nivel de flexibilidad con el que pocas alternativas en el mercado pueden competir. Al igual que con ACI, el cliente solo paga por el uso de los recursos.

Capítulo 3

Desarrollo

3.1. Metodología

3.1.1. Kanban

La metodología usada durante el desarrollo de este trabajo ha sido Kanban [1]. Ésta está enmarcada dentro de las metodologías ágiles, las cuales son un conjunto de técnicas usadas en ciclos de trabajo para aumentar su flexibilidad, adaptabilidad, colaboración y la entrega incremental de productos o servicios, aumentando la eficiencia del trabajo.

La elección de una metodología ágil era evidente debido a la incertidumbre inicial en relación a la capacidad de crear un clúster de Kubernetes, duda más que justificada debido a la casi nula experiencia en lo relativo a contenedores, haciéndolo un proyecto con un final incierto. Si bien originalmente se planteó el uso de Scrum, se decidió cambiarlo por Kanban en las etapas iniciales del proyecto.

Kanban nace a finales de 1940 de la mano de Taiichi Ohno [33], ingeniero industrial japonés y empleado de Toyota. Taiichi buscaba aumentar la eficacia de la fábrica donde trabajaba, para ello, desarrolló un método llamado “Just In Time” o “Jit”, un sistema de producción en el que la comunicación jugaba un papel vital, para lo cual Taiichi desarrolló otro método sencillo de señales que bautizo como Kanban. A principios del siglo XXI, la

comunidad del desarrollo del software se dio cuenta de que éste método podía llegar a ser muy útil en los procesos de desarrollo.

La palabra Kanban está formada por otras dos palabras japonesas, Kan (visual) y Ban (Tablero), lo cual describe a la perfección la característica más importante de éste, un tablero en el que se acoplan tarjetas mediante las cuales se pueden visualizar de forma clara y rápida el flujo de trabajo de las tareas. En Kanban se entiende por flujo de trabajo como el proceso por el que pasan las tareas desde su inicio hasta su finalización, este flujo es el que se ve representado en el tablero, y pueden ser, por ejemplo "Por hacer", "En progreso", "Revisión" y "Completado"

Cabe destacar que las tareas son modificables y el número de éstas puede variar conforme se va desarrollando el proceso, por ejemplo, al empezar una tarea nueva, puede revelarse que era más compleja de lo que inicialmente estaba previsto, haciendo necesario descomponerla en tareas más pequeñas. También pueden ser descartadas al ver que ya no son necesarias.

En lo relativo a este Trabajo de Fin de Grado, el grueso del uso de Kanban ha sido llevado a cabo por el estudiante y el co-tutor Carmelo León, el cual ha llevado a cabo reuniones diarias de corta duración para monitorizar el progreso del trabajo y hacer de guía aportando consejos gracias a su gran experiencia en el sector. Además, el tutor Nelson Monzón ha llevado una supervisión general del proyecto gracias a unas reuniones de mayor duración que originalmente eran mensuales pero que fueron aumentando de frecuencia conforme avanzaba el proyecto.

Como ya ha sido mencionado, los dos elementos necesarios para implementar esta metodología es el tablero Kanban y las tarjetas de tareas. En este caso la mayor parte de las tareas originalmente consistían en conocimientos específicos necesarios para trabajar con Kubernetes y Docker Swarm, los cuales el estudiante carecía por completo. La tarea más importante era la de crear un clúster de Kubernetes, la cual tuvo que ser descompuesta en varias tareas al iniciarse y ver que era más complejo de lo previsto.

3.2. Creación del Clúster

3.2.1. Herramientas disponibles

En informática, se entiende por clúster a un conjunto de ordenadores interconectados que trabajan para cumplir el mismo propósito, a cada ordenador del clúster se le denomina como ”nodo”

La creación de un clúster de Kubernetes representa una serie de retos sumamente complejos que puede implicar la reticencia a su implementación por parte del sector frente a otras alternativas.
Esto ha incentivado la proliferación de herramientas que automatizan en mayor o menor medida la creación y configuración de estos y que con el paso de los años se han vuelto cada vez más populares.

Se podría decir que las herramientas más usadas para la creación de un clúster **on premise** (en local) serían las siguientes:

- MicroK8s: Desarrollado por Canonical, se describen a sí mismos como la forma más rápida y sencilla de desplegar un clúster de Kubernetes, tanto para experimentación como para despliegues de producción. Se instala fácilmente con Snap y ofrece actualizaciones automáticas de forma segura sin que el administrador se tenga que preocupar por incompatibilidades. Tiene varias opciones a la hora de instalarlo para habilitar los servicios del clúster deseados.

- K3s: Similar a MicroK8s, originalmente diseñado por RANCHER y actualmente bajo gestión del Cloud Native Computing Foundation, ofrece una solución rápida y ligera para el despliegue de un clúster. K3s mantiene la mayoría de características de Kubernetes pero elimina los drivers y componentes menos usados del mismo. Se ejecuta como un solo archivo binario, reduciendo sustancialmente los recursos necesarios para su funcionamiento, lo cual lo convierte en una opción ideal para aquellos que busquen una alternativa sencilla para despliegues de producción poco exigentes y que no requiera de una infraestructura compleja.

- Kubeadm: La alternativa “Oficial”, creado y mantenido por el propio proyecto de Kubernetes. Su objetivo es proveer una forma estándar y correcta de crear clústers válidos para producción, permite el despliegue y configuración de todos los servicios de Kubernetes pero ofreciendo una capa de abstracción que simplifica la configuración de los mismos, dentro de las herramientas existentes, es la más cercana a un despliegue

"manual" de un clúster de Kubernetes.

3.2.2. Creación de la infraestructura

Kubernetes [21] distingue entre 2 tipos de nodos, los control plane y los worker:

- **Control Plane**: Responsable de la gestión global del clúster de Kubernetes, ejecutan los servicios críticos para el funcionamiento del mismo:
  - Kube-apiserver: Actúa como puerta de entrada al Control Plane, permitiendo la configuración del clúster a través de herramientas CLI, bibliotecas de terceros o directamente a través de solicitudes HTTP.
  - etcd: Una base de datos clave-valor consistente y de alta disponibilidad que almacena toda la información relativa a la configuración del clúster, también es usado por los demás elementos del Control Plane para preservar el estado actual del clúster.
  - kube-sheduler: Se encarga de encontrar pods recién creados y de asignarles un nodo en el que ejecutarse teniendo en cuenta los requisitos de recursos, políticas de afinidad, restricciones y otras especificaciones.
  - kube-controller-manager: El componente encargado de ejecutar los procesos de control, si bien cada uno de ellos tiene una lógica distinta, han sido unidos en un solo binarizado para ejecutarse en un único proceso con el fin de reducir complejidad y aumentar la eficiencia.
  - cloud-controller-manager: Este componente es opcional, alberga toda la lógica dedicada a la vinculación del clústerº con la API de proveedores de cloud computing, permitiendo la interacción de forma efectiva entre Kubernetes y la infraestructura de nube elegida.
- **Nodos Worker**: Son los encargados de ejecutar los pods de los despliegues. Estos nodos se suelen etiquetar para permitir políticas de afinidad de los pods.

El clúster estará compuesto por cinco máquinas virtuales Ubuntu Server 22.04, una imagen de ubuntu ligera, ideal para este tipo de despliegues. En este caso, se dedicará una mv

para ejercer de puente entre la red externa y la red interna, otra para que haga de control plane y las otras tres para hacer de nodos worker.

Lo correcto sería que estuviesen funcionando con algún software de virtualización similar a VMware, en este caso se ha usado VirtualBox por mera comodidad.

Lo único relevante de la configuración de las mv para este documento es la configuración de las interfaces de red. El clúster estará ubicado dentro de una red interna, para que tenga acceso al exterior, requerimos de una mv que esté conectada a dicha red externa y a la red inmediatamente superior, para que haga de puente entre ambas y permita políticas de enrutamiento y seguridad.

Para la configuración de red de dicha mv, modificaremos el fichero netplan de la siguiente forma:

```
network:
  ethernets:
    enp0s3:
      dhcp4: true
    enp0s8:
      addresses: [192.168.100.1/24]
      dhcp4: no
  version: 2
```

Listing 3.1: Ejemplo de fichero de configuración de Netplan

Como se puede ver, la red interna será la 192.168.100.0/24, en ella estarán todos los nodos del clúster. Para que puedan acceder a internet, tendrán que hacerlo a través de la máquina puente. Para ello, habrá que editar el fichero /etc/sysctl.conf y modificar el campo net.ipv4.ip_forward de 0 a 1.

Para que el cambio haga efecto, se puede ejecutar un:

```
$ sudo sysctl -p
```

Un fichero de ejemplo del netplan de cada nodo sería el siguiente:

```
network:
  ethernets:
    enp0s3:
      dhcp4: no
      addresses:
        - 192.168.100.20/24
```

```
    routes:
      - to: default
        via: 192.168.100.1
    nameservers:
        addresses: [8.8.8.8, 8.8.4.4]
version: 2
```

Listing 3.2: Ejemplo de fichero de configuración de Netplan

Las direcciones IP del cluster están fijas y corresponen a las siguientes:

- cp1 - 192.168.100.20/24
- k8sworker1 - 192.168.100.31/24
- k8sworker2 - 192.168.100.32/24
- k8sworker3 - 192.168.100.33/24

Para poder acceder a ellos desde ssh de una forma cómoda, se ha usado Termius, un programa que facilita la gestión de terminales ssh y que ofrece características muy útiles.

Desde ella, configuraremos la máquina puente con un nat de iptables para que redirija las solicitudes ssh que reciba desde el exterior a las mv correspondientes:

```
$ sudo iptables -t nat -A PREROUTING -p tcp --dport 2230 -j DNAT
   ↪ --to-destination 192.168.100.30:22
```

Una vez verificado que todas las MV tienen las IP correctas y que tienen acceso a internet, podremos pasar al siguiente punto.

3.2.3. Elección e Instalación de un CRI (Container Runtime Interface

Para que las MV del clúster puedan ejecutar y manejar los contenedores oportunos (recordemos que los servicios de Kubernetes también son contenedores), es necesario instalar

manualmente un **Container Runtime Interface** (**CRI**) [12], si bien no son los únicos, las dos opciones más usadas en este ámbito son containerd y cri-o.

- **Containerd** Fue creado por Docker y evolucionó hasta convertirse en una herramienta independiente, se podría decir que es el CRI más versátil de los disponibles, lo cual puede no ser recomendable para un clúster de kubernetes ya que tiene muchas utilidades que resultan redundantes en la inmensa mayoría de los casos de uso.

- **CRI-O** Diseñado por RedHat para funcionar específicamente con Kubernetes, es mucho más ligero y eficiente que containerd ya que omite todas las funcionalidades innecesarias de containerd de cara a un clúster de k8s.

La elección de estos CRIs es irrelevante de cara al desarrollo y despliegue de aplicativos en el clúster ya que ambos usan **runc** como container runtime. **Runc** es una herramienta de linux que opera a bajo nivel para permitir la ejecución de los contenedores cumpliendo con la OCI (Iniciativa de Contenedores Abiertos).

Los pasos a realizar al principio de la instalación son comunes entre ambos CRIs:

En primer lugar, configuramos el sistema para que cargue automáticamente los módulos del kernel “overlay” y “br_netfilter”

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

Los cambios se harán efectivos en el siguiente reinicio del sistema, para activarlos sin hacerlo, ejecutaremos:

```
sudo modprobe overlay
sudo modprobe br_netfilter
```

Overlay permite el uso de sistemas de archivos overlay, es decir, permite que varios sistemas de archivo se monten simultáneamente y por capas, superponiendo uno sobre otro sin modificar los sistemas de archivos originales. En el contexto de Kubernetes y los contenedores, se utiliza para gestionar la creación de imágenes y montaje de volúmenes de forma eficiente.

br_netfilter permite que las iptables filtren y manipulen los paquetes que pasan a través de un puente en el kernel de Linux. En este contexto, se usa para asegurar el aislamiento entre pods y el control de acceso a los servicios del clúster.

Podemos verificar que se han cargado correctamente con un

```
lsmod | grep br_netfilter
lsmod | grep overlay
```

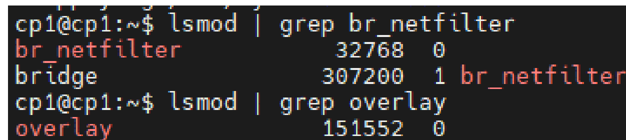


Ilustración 3.1: Verificación de módulos cargados

Añadimos un fichero de configuración en el directorio /etc/sysctl.d, en él, se ubican los archivos relacionados con parámetros de configuración del kernel

```
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
EOF
```

Los cambios implementados son los siguientes:

- **net.bridge.bridge-nf-call-iptables = 1**
  Habilita el filtrado del tráfico de paquetes a través de los bridges por iptables.

- **net.bridge.bridge-nf-call-ip6tables = 1**
  Similar al anterior pero para el tráfico ipv6

- **net.ipv4.ip_forward = 1**
  Activa el reenvío de paquetes IPv4 en el host, esencial para la comunicación entre contenedores ubicados en diferentes nodos.

Para que los cambios surtan efecto inmediatamente, ejecutaremos un

```
sudo sysctl --system
```

Podemos verificarlo usando:

```
sysctl net.bridge.bridge-nf-call-iptables net.bridge.bridge-nf-
  ↪ call-ip6tables net.ipv4.ip_forward
```

```
cp1@cp1:~$ sysctl net.bridge.bridge-nf-call-ipta
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
```

Ilustración 3.2: Verificación de parámetros kernel

Ahora ya podemos instalar el CRI correspondiente, en este caso se ha elegido usar CRI-O [3] debido a su mejor optimización para Kubernetes y rendimiento

Añadimos el repositorio de CRI-O

```
curl -fsSL https://pkgs.k8s.io/addons:/cri-o:/$PROJECT_PATH/deb/
  ↪ Release.key |
    gpg --dearmor -o /etc/apt/keyrings/cri-o-apt-keyring.gpg

echo \"deb [signed-by=/etc/apt/keyrings/cri-o-apt-keyring.gpg]
  ↪ https://pkgs.k8s.io/addons:/cri-o:/$PROJECT_PATH/deb/ /’’ |
    tee /etc/apt/sources.list.d/cri-o.list
```

Instalamos el paquete

```
apt-get update

apt-get install -y cri-o
```

Iniciamos y habilitamos

```
systemctl start crio.service

systemctl enable crio.service
```

3.2.4. Instalación de Kubeadm e Inicialización del clúster

Para la creación y gestión del clúster [13] no es necesario exclusivamente **kubeadm** [15], será necesaria la instalación 2 paquetes adicionales, **kubectl** y **kubelet**.

- **kubelet**: Se encarga de la gestión de todos los contenedores que se ejecutan en cada pod, es el componente de Kubernetes que se comunica con el CRI oportuno, no solo se encarga de arrancarlos y ejecutarlos sino que los monitoriza para asegurarse de que funcionan de forma correcta.

- **kubectl**: Es la herramienta de línea de comandos para interactuar con los nodos del clúster, todas las acciones de despliegue de aplicativos, visualización de datos, configuraciones, todo se realiza mediante kubectl.

- **kubeadm**: Como mencionamos anteriormente, es la herramienta oficial de Kubernetes para el despliegue rápido de clústeres, su propósito principal es el arranque y configuración de los elementos necesarios para el funcionamiento de los nodos control plane y worker. También se encarga del ciclo de vida del clúster, permitiendo la modificación de todos los elementos y la actualización de los mismos a sus versiones más recientes.

Los pasos para instalarlo son los mismos que para CRI-O.

Añadimos el repositorio

```
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.29/deb/Release.key
   ↪  | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-
   ↪ keyring.gpg

echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
   ↪  https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /' | sudo tee /
   ↪ etc/apt/sources.list.d/kubernetes.list
```

Instalamos el paquete:

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

Una buena práctica podría ser marcar los paquetes instalados para evitar que se actualicen

y generen problemas de incompatibilidad.

```
sudo apt-mark hold cri-o kubelet kubeadm kubectl
```

Kubernetes requiere que el swap del sistema esté desactivado, ya que puede afectar al rendimiento de los contenedores y complicar el manejo de ellos. Así pues, debemos desactivarlo y evitar que se vuelva a activar después de un reinicio modificando el fichero **/etc/fstab**

```
swapoff -a

nano /etc/fstab
```

Ahora ya podemos proceder con la inicialización del clúster, para ello, vamos a la máquina que hará de control plane y ejecutamos

```
sudo kubeadm init --pod-network-cidr=172.16.0.0/16
```

El parámetro **–pod-network-cidr** se usa para especificar el rango de direcciones IP que utilizarán los pods de los aplicativos dentro del clúster, el rango debe de ser único y no puede solaparse con el rango de IPs que utilizan los nodos, importante tenerlo en cuenta a la hora de elegir el pod network.

Al ejecutarlo, kubeadm iniciará una serie de comprobaciones para verificar que el nodo está correctamente configurado, una vez realizados con éxito, se creará el clúster y el nodo en el que se ha ejecutado pasará a ser el control plane del mismo. El output del mismo debería de ser similar al siguiente.

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

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

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.100.20:6443 --token dmjy95.vnhfwwik07q2lsbt \
        --discovery-token-ca-cert-hash sha256:e407f8ddb1d8280436ff55558311614196a0a677a5d68d4a30f66de84fd2bace
```

Ilustración 3.3: Output kubeadm init

Como podemos ver, nos especifica que para usar kubectl como un usuario normal, deberemos ejecutar los siguientes comandos:

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

En nuestro caso, seguiremos configurándolo desde el usuario root, por tanto, deberemos ejecutar

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

Ahora ya podremos ejecutar los comandos de kubectl en el clúster, como podemos ver, nos ha proporcionado el comando necesario para que los nodos worker se unan al clúster junto al token de validación correspondiente. Estos tokens tienen una validez por defecto de 24h, una vez expirados, podemos obtener otro con un:

```
kubeadm token create
```

Este comando configura la variable de entorno **KUBECONFIG** para que apunte al archivo **admin.conf**, el cual contiene la configuración necesaria para que kubectl u otras herramientas puedan conectarse y autentificarse en el clúster.

Ahora procedemos a ejecutar el comando de join en los nodos worker, al hacerlo, kubeadm hará los cambios correspondientes en el nodo y mandará una solicitud al control plane, una vez hecho, podremos comprobar que se han unido correctamente ejecutando el siguiente comando en el control plane.

```
kubectl get nodes
```

A lo que debería respondernos con un output similar al siguiente:

```
root@cp1:/home/cp1# kubectl get nodes
NAME          STATUS   ROLES           AGE     VERSION
cp1           Ready    control-plane   18d     v1.29.3
k8sworker1    Ready    <none>          18d     v1.29.3
k8sworker2    Ready    <none>          18d     v1.29.3
k8sworker3    Ready    <none>          6d22h   v1.29.3
```

Ilustración 3.4: Nodos disponibles en el clúster

El campo de **VERSION** corresponde a la versión de kubeadm instalada en el nodo, todo rol que no sea control-plane, incluido el **none** implica que se trata de nodos worker, recordemos que los roles sirven únicamente para políticas de afinidad a la hora de desplegar pods y servicios.

De cara a mantener la configuración iptables, es recomendable guardarlas para que se mantengan en los próximos reinicios del sistema.

```
netfilter-persistent save
```

3.2.5. Instalación de un Pod Network add-on

Para que los pods puedan comunicarse entre ellos, es obligatorio el despliegue de un Container Network Interface (CNI) [17] [10] a través de un pod network add-on [18]

Si bien kubeadm simplifica la gestión del clúster, no maneja por sí misma la red de contenedores, dicha función la relega en proyectos de terceros de confianza en forma de add-ons (o pluggins) desplegables en el clúster.

El modelo de Kubernetes especifica que los pods deben de poder comunicarse entre sí a través de una red propia, en la que cada pod debe de tener una identificación única. Para hacerlo, se requiere de algún tipo de sistema que se encargue de crear una red interna dentro del clúster y asignar las direcciones IPs de los pods y coordinarse con **coreDNS** para poder enrutar las comunicaciones entre ellos. Recordemos que **coreDNS** es el servicio de Kubernetes que se encarga de la resoluciones de dominio de todos los servicios y pods dentro del clúster.

Existe una gran cantidad de pod network add-ons disponibles para instalar en un clúster de kubernetes, los más usados son los siguientes:

- **Calico**: Es uno de los que mejor relación complejidad-rendimiento tienen, asigna IPs únicas dentro de la red de pods para enrutar las comunicaciones entre ellos, aporta un rogusto conjunto de políticas de seguridad en red y tiene una fácil instalación e implementación.

- **Cilium**: El más complejo de las 3 opciones barajadas y probablemente el más completo, al contrario que la mayoría de pod network add-ons, no emplea enrutamiento IP para

la comunicación entre los pods, en su lugar, usa la tecnología eBPF (Berkeley Packet Filter), una tecnología de manipulación de paquetes que permite injectar identidades directamente en el header del paquete y permite un nivel de seguridad muy por encima que las alternativas. Sin embargo, esta metología limita su escalabilidad a un total de 255 clúster y 65.000 identidades, más que suficiente para la inmensa mayoría de casos de uso.

- **Flannel**: El más simple de los tres anteriores, se limita a proporcionar un medio de comunicación entre los pods y ofrece muy poca persolacización y no tiene una configuración de seguridad integrada. Es ideal para el despliegue en clústers de forma rápida y sencilla y en los que el escalado o la política de seguridad no es prioridad.

Para este trabajo se ha decidido usar Calico ya que me parecía la opción más versátil, ofrece unas características suficientes para la mayoría de casos de usos y la instalación básica es muy sencilla.

Primero, instalaremos el operador de tigrera [30]. En kubernetes, un **Operador** es un software diseñado para gestionar de forma automática el ciclo de vida de las aplicaciones, en este caso, del pluggin de Calico, es la pieza que se encarga de elegir como, cuando y donde desplegar los pods necesarios para que funcione caliclo.

```
kubectl create -f https://raw.githubusercontent.com/projectcalico/
   ↪ calico/v3.28.0/manifests/tigera-operator.yaml
```

Ahora, instalaremos el **Custom Resource** necesario, un **CRD** es un elemento vital en Kubernetes, se usan para comunicarse con la api de Kubernetes y definir nuevos tipos de recursos que funcionan de forma similar a los nativos de Kubernetes, es decir, al contrario que un despliegue normal de pods, estos se convierten en una extensión del clúster, como es, en este caso, la pod network.

```
curl https://raw.githubusercontent.com/projectcalico/calico/v3
   ↪ .28.0/manifests/custom-resources.yaml -O
```

Este fichero contiene la configuración básica de Calico. Por defecto, intenta instalar la pod-network en la red 192.168.0.0/16, en nuestro caso, al iniciar el clúster, especificamos un **–pod-network-cidr = 172.16.0.0/16**, por tanto, deberemos modificar el parámetro **cidr:** para que coincida con el especificado en la creación del clúster.

```
 spec:
  # Configures Calico networking.
```

```
  calicoNetwork:
    ipPools:
    - name: default-ipv4-ippool
      blockSize: 26
      cidr: 172.16.0.0/16
      encapsulation: VXLANCrossSubnet
      natOutgoing: Enabled
      nodeSelector: all()
```

Ahora, podemos instalarlo instalarlo:

```
kubectl create -f custom-resources.yaml
```

Al hacerlo, nos empezará a crear todos los servicios necesarios ya arrancará el demonset de calico. Una vez finalizado, podremos comprobar el correcto despliegue ejecutando:

```
kubectl get pods -n calico-system -o wide
```

```
cp1@cp1:~$ kubectl get pods -n calico-system -o wide
NAME                                       READY   STATUS    RESTARTS   AGE   IP                NODE
calico-kube-controllers-74c5f848fb-jbh4z   1/1     Running   10         33d   172.16.176.57     cp1
calico-node-5qp4k                          1/1     Running   11         33d   192.168.100.20    cp1
calico-node-6v8rt                          1/1     Running   8          33d   192.168.100.32    k8sworker2
calico-node-bznxl                          1/1     Running   9          33d   192.168.100.33    k8sworker3
calico-node-jz78z                          1/1     Running   9          33d   192.168.100.31    k8sworker1
calico-typha-685547c6bb-5trtj              1/1     Running   11         33d   192.168.100.20    cp1
calico-typha-685547c6bb-9svcg              1/1     Running   8          33d   192.168.100.31    k8sworker1
csi-node-driver-cprn5                      2/2     Running   16         33d   172.16.8.47       k8sworker2
csi-node-driver-k2hgw                      2/2     Running   18         33d   172.16.230.248    k8sworker1
csi-node-driver-pbkfz                      2/2     Running   16         33d   172.16.137.4      k8sworker3
csi-node-driver-tltb9                      2/2     Running   20         33d   172.16.176.61     cp1
```

Ilustración 3.5: Output pods de Calico

Podemos ver que sale la información más relevante de los pods, si nos fijamos en las direcciones IP vemos que los que brindan servicio directo a los pods se encuentran dentro de la red correspondiente y los que se encargan de la gestión de Calico operan una capa por encima, es decir, tienen la dirección IP del nodo en el que se encuentran.

Los restarts pueden ser por fallos del pod o por un reinicio del nodo en el que se encuentran (como es el caso)

3.2.6. Configuración de Almacenamiento

La unidad básica de almacenamiento en Kubernetes es el Persistent Volume (PV) [19], el cual es un recurso que puede ser provisto de forma manual y estática por parte de un administrador o de forma dinámica usando un pluggin. Para Kubernetes, los PV son un recurso como cualquier otro, como por ejemplo un Pod, y se pueden aplicar las mismas operaciones sobre él.

Para administrarlo de forma dinámica es necesario un Storage Class [20], el cual es una forma que tienen los administradores de especificar los tipos de almacenamiento que ofrece el clúster y son totalmente configurables, y responden a diferentes necesidades, tales como rendimiento o redundancia de datos.

El recurso de almacenamiento final para las aplicaciones son los Persistent Volume Claim (PVC), el cual es una solicitud de almacenamiento por parte del desarrollador.

La opción más flexible y sencilla para suplir estas necesidades pasa por la instalación de un add-on de orquestación de almacenamiento, de forma similar a la instalación del pod-network. Algunas de las opciones más conocidas son las siguientes:

- **OpenEBS**: Es una solución de almacenamiento flexible y sencilla de instalar y usar, ofrece almacenamiento local y distribuido, de alta disponibilidad y con opción de realizar snapshots y clonación de volúmenes. Su arquitectura se basa en el uso de contenedores y la usa para escalar en base a las necesidades de almacenamiento. Ideal para implantaciones rápidas pero sin perder en rendimiento ni características.

- **Rook**: Más complejo que OpenEBS usa Ceph como backend de almacenamiento. Ofrece una mayor cantidad de recursos y configuraciones.

- **Longhorn**: Al igual que las anteriores, cuenta con utilidades que permiten la toma de snapshots de forma incremental y backups automáticos, así como recuperación frente a desastres.

Todos ellos son de código abierto y ofrecen características muy similares, reduciendo al elección de alguno de ellos a las características únicas de cada clúster y a preferencias personales del administrador.

A continuación se describirá la instalación y uso tanto de OpenEBS usando **Helm**, como

Rook de forma convencional

3.2.6.1. Instalación de Helm:

Helm es un gestor de paquetes para Kubernetes, muy similar a un **apt**, es la alternativa al empleo de manifiestos. Descargamos [7] el script de instalación y lo ejecutamos

```
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/
    ↪ helm/main/scripts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
```

Una vez instalado podemos pasar a descargar el repositorio de OpenEBS

3.2.6.2. OpenEBS

Añadimos el repositorio de OpenEBs a la lista de paquetes disponible de Helm [23]

```
helm repo add openebs https://openebs.github.io/charts
helm repo update
```

Ahora lo instalamos con la configuración deseada:

```
helm install openebs openebs/openebs --namespace openebs --create-
    ↪ namespace \
--set jiva.enabled=true \
--set localprovisioner.enabled=true \
--set ndm.enabled=true \
--set cstor.enabled=false
```

Esta configuración instala **OpenEBS** con la siguiente configuración, si bien la instalación de los diferentes módulos no fuerza su uso, se suele instalar solo lo necesario de cara a la eficiencia y ligereza del mismo.

- **jiva.enabled=true**
  Activa el motor de almacenamiento de Jiva, proporcionado por OpenEBS. Se encarga de proporcionar una forma de contar con replicas de almacenamiento de los volúmenes para qué, en caso de perdida de alguno, el funcionamiento no se vea afectado.

- **localprovisioner.enabled=true**
  Permite que se emplee el almacenamiento local del nodo en el que se está ejecutando el pod que solicita el almacenamiento.

- **ndm.enabled=true**
  Activa el Node Disk manager (NDM), el componente que se encarga de descubrir bloques de almacenamiento disponibles en los nodos, facilitando la instalación de nuevas unidades de almacenamiento externas dentro de los nodos del clúster.

- **cstor.enabled=false**
  cSTOR es el módulo de OpenEBS encargado de la clonación de volúmenes y generación de snapshots, en este caso ha sido desactivado debido a que el clúster cuenta con pocos recursos de almacenamiento, se modificará en implantaciones futuras.

```
NAME: openebs
LAST DEPLOYED: Thu Apr  4 20:40:49 2024
NAMESPACE: openebs
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Successfully installed OpenEBS.
```

Ilustración 3.6: Output Instalación OpenEBS

Podemos verificar que funciona correctamente mirando el estado de los pods de OpenEBS

```
kubectl get pods -n openebs -o wide
```

El output esperado debería de ser algo similar al siguiente:

```
root@cp1:/home/cp1# kubectl get pods -n openebs -o wide
NAME                                          READY   STATUS    RESTARTS        AGE     IP               NODE
openebs-jiva-csi-controller-0                 5/5     Running   0               9m47s   172.16.230.194   k8sworker1
openebs-jiva-csi-node-bnftb                   3/3     Running   1 (9m13s ago)   9m47s   192.168.100.33   k8sworker3
openebs-jiva-csi-node-lzcbc                   3/3     Running   0               9m47s   192.168.100.32   k8sworker2
openebs-jiva-csi-node-xvr2g                   3/3     Running   1 (9m13s ago)   9m48s   192.168.100.31   k8sworker1
openebs-jiva-operator-69bd68fccc-gkrv9        1/1     Running   0               9m47s   172.16.8.2       k8sworker2
openebs-localpv-provisioner-56d6489bbc-nnxxc  1/1     Running   0               9m47s   172.16.8.1       k8sworker2
openebs-ndm-9dvjr                             1/1     Running   0               9m47s   192.168.100.32   k8sworker2
openebs-ndm-b6wxz                             1/1     Running   0               9m48s   192.168.100.33   k8sworker3
openebs-ndm-n8l8z                             1/1     Running   0               9m47s   192.168.100.31   k8sworker1
openebs-ndm-operator-5d7944c94d-v87t2         1/1     Running   0               9m47s   172.16.137.5     k8sworker3
```

Ilustración 3.7: Output Pods OpenEBS

Ahora podemos proceder a crear un Storage Class usando **Jiva**, para ello generamos un fichero **.yaml** de la siguiente forma:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: openebs-jiva-csi-replicas
  annotations:
    openebs.io/cas-type: "jiva"
    cas.openebs.io/config: |
      - name: ReplicaCount
        value: "3"
provisioner: jiva.csi.openebs.io
allowVolumeExpansion: false
```

De esta forma definimos un Storage Class llamado **openebs-jiva-csi-replicas**, con las **annotations** configuramos las características de dicha clase, al especificar un **ReplicaCount** al valor de 3, indicamos que queremos 3 réplicas del almacenamiento. Con **provisioner** espeficiamos el provisionador que creará los volúmenes cuando se solicite uno de esta clase, en este caso es **jiva.csi.openebs.io**, que utilizará el Container Storage Interface de Jiva.

Para activarlo, lo desplegamos como cualquier otro deployment:

```
kubectl apply -f storageClassReplica.yaml
```

Podemos comprobar su estado ejecutando

```
kubectl get sc
```

```
root@cp1:/home/cp1# kubectl get sc
NAME                        PROVISIONER                    RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION
openebs-device              openebs.io/local               Delete          WaitForFirstConsumer   false
openebs-hostpath            openebs.io/local               Delete          WaitForFirstConsumer   false
openebs-jiva-csi-default    jiva.csi.openebs.io            Delete          Immediate              true
openebs-jiva-csi-replicas   jiva.csi.openebs.io            Delete          Immediate              false
openebs-jiva-default        openebs.io/provisioner-iscsi   Delete          Immediate              false
```

Ilustración 3.8: Output Get SC

Ahora podemos crear algún **PVC** que requiera de dicho **StorageClass** para verificar que se asigna correctamente, volvemos a realizar un deploy de un fichero .yaml en el que se especifique el recurso.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bookreview-bdd-pvc
```

```
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: openebs-jiva-csi-replicas
```

Como se puede ver, es necesario especificar el nombre del **SC** creado usando el campo **openebs-jiva-csi-replicas**.

```
kubectl apply -f pvcTest.yaml
```

Para comprobar el estado del pvc haremos un

```
kubectl get pvc
```

```
root@cp1:/home/cp1/k8s_deployments# kubectl get pvc
NAME                          STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS
bookreview-bdd-pvc-replicas   Bound    pvc-cf52f96a-64e9-4901-ba18-78b49ab61c0c   1Gi        RWO            openebs-jiva-csi-replicas
```

Ilustración 3.9: Output Get PVC

Vemos que el **status** es el correcto, **bound** indica que se le ha asignado un **PV** y qué, por tanto, cuenta con un espacio de almacenamiento asignado. El número de réplicas del recurso no se puede ver desde los pv o pvc ya que es un tipo de recurso diferente, para poder verlo, debemos realizar un:

```
kubectl get jivavolumes -n openebs
```

```
root@cp1:/home/cp1/k8s_deployments# kubectl get jivavolumes -n openebs
NAME                                       REPLICACOUNT   PHASE   STATUS
pvc-cf52f96a-64e9-4901-ba18-78b49ab61c0c   3              Ready   RW
```

Ilustración 3.10: Output Get JivaVolumes

Podemos apreciar como, efectivamente, hay 3 réplicas del recurso.

3.2.6.3. Rook-Ceph

Como su propio nombre indica, el pluggin de Rook-Ceph [26] está compuesto por dos sistemas diferentes,

- **Ceph** Es un sistema de almacenamiento distribuido conocido por su integridad, escalabilidad y alto grado de configuración.

- **Rook** Es plataforma plataforma de orquestación de Ceph para Kubernetes, la cual simplifica la implementación y gestión del sistema de almacenamiento de Ceph actuando como un operador de Kubernetes.

Así pues, Rook-Ceph permite funciones más avanzadas en relación con el almacenamiento y gestión de datos de los que no dispone OpenEBS, el hecho de usar Ceph permite alcanzar una escalabilidad horizontal con unas garantías y consistencias difíciles de alcanzar con OpenEBS. Un ejemplo puede ser la posibilidad de utilizar **Erasure Coding**.

El **Erasure Coding** [32] es una técnica de protección de datos bastante interesante en comparación con la replicación convencional, en el caso de la replicación convencional, por línea general se requiere siempre de un mínimo de 3 réplicas de información, esto es debido a que los sistemas de almacenamiento realizan checksum checks para verificar la integridad de los objetos de almacenamiento. En el caso de que algún checksum sea diferente del de otra réplica, deberá decidir cual es el que sigue integro y cual es el que está dañado, si solo se tuvieran 2 réplicas, no sería capaz de garantizar que el objeto que elige como íntegro sea el correcto y podría asumir que el que en realidad está dañado es el que está ”healthy”. Por ese motivo siempre se recomienda un mínimo de 3 réplicas, así, en caso de que 1 se dañe, podrá compararla con las otras 2 y podrá distinguirlo correctamente.

No obstante, sigue existiendo la misma çapa”de fallo, en el caso de que 2 de esas réplicas se corrompan, interpretará como ünhealthy.ª la que en realidad está ”healthy”, si bien es un escenario poco realista, fomenta la creación de un mayor número de réplicas, lo cual tiene un coste de gestión y almacenamiento muy elevado.

El **Erasure Coding** es una solución a este problema, en vez de copiar la totalidad del objeto de almacenamiento, lo divide en múltiples fragmentos a los que se les añade fragmentos de paridad calculados mediante algoritmos, luego distribuyen esos fragmentos a través de los diferentes medios de almacenamiento disponibles, como pueden ser discos o nodos completos,

entre otros, de esta forma, es posible reconstruir los datos originales a partir de cualquier subconjunto de fragmentos lo suficientemente grande.

Las ventajas de este método son evidentes, al requerir menos espacio de almacenamiento que la replicación tradicional, aumenta su escalabilidad y supone un gran ahorro en el coste del almacenamiento de datos.

3.2.6.4. Instalación Rook-Ceph

Si bien es posible la utilización de 2 orquestadores de almacenamiento diferentes en un mismo clúster, se procederá al la desinstalación de OpenEBS para evitar posibles conflictos.

En esta ocasión descargaré e instalaré Rook-Ceph de una forma convencional en vez de usando Helm, ya que ofrece un mayor nivel de personalización. Comenzaremos clonando su **Rook Repository** en el directorio correspondiente.

```
git clone --single-branch --branch v1.14.4 https://github.com/rook
    ↪ /rook.git
```

En el momento de la creación de este documento, la última versión estable es la 1.14.4

A continuación nos ubicaremos en el directorio **deploy/examples**, ahí se encuentran las definiciones necesarias para implementar Rook-Ceph [24], ahora ejecutaremos el siguiente comando:

```
kubectl create -f crds.yaml -f common.yaml -f operator.yaml
```

Recordemos que el **create** se distingue del **apply** en la forma en la que Kubernetes gestiona los recursos, al usar **create** se creará un objeto siguiendo las instrucciones especificadas en el yaml y solo se puede hacer una vez. Por contra, al usar **apply**, se especificará a Kubernetes el estado deseado y Kubernetes hará lo necesario para llegar a él en función de los objetos ya existentes.

- **crds.yaml** Fichero en el que se encuentran los Custom Resource Definitions (CRDs) necesarios.

- **common.yaml** Fichero que define los objetos necesarios para el funcionamiento de Rook-Ceph, la cantidad de definiciones es considerablemente extensa y consiste en la creación del namespace, los roles de seguridad y permisos, la creación de cuentas de servicio para interactuar con la API de Kubernetes y su asociación con los permisos.

- **operator.yaml** Despliega la configuración del operador que gestionará el ciclo de vida de Rook-Ceph, en él se configuran los parámetros generales, la configuración CSI, políticas de actualización, prioridades de pod, etc...

IMPORTANTE: A menos que se configure de forma diferente a la predeterminada, Rook-Ceph intentará buscar sistemas de almacenamiento que no estén en uso, por tanto, la mejor opción consiste en acoplar discos sin formatear en los nodos del clúster, Rook-Ceph los reconocerá y empezará a usar sin ningún tipo de acción extra.

Ahora crearemos el objeto final de Rook-Ceph, en el que se definen los parámetros del funcionamiento a más alto nivel del orquestador de almacenamiento:

```
kubectl create -f cluster.yaml
```

En este archivo se especifíca la cantidad de monitores, administradores, las imágenes de contenedor de Ceph que usará, el directorio de datos, entre otros.

Dejaremos pasar unos minutos y comprobaremos el correcto funcionamiento con:

```
kubectl -n rook-ceph get pod
```

Como se puede ver, todos los pods están en ready a excepción de los de los **rook-ceph-osd-prepare-NOMBRENODO-XXXXX**, lo cual es perfectamente normal ya que su propósito consiste en preparar los dispositivos de almacenamiento de los nodos del clúster, que salga como **completed** indica que la última tarea se realizó correctamente y que salga en 0 implica que no está realizando ninguna tarea nueva, ya que no tienen ningún nuevo dispositivo que configurar.

Para estar más seguros, podemos hacer uso del **Ceph-Toolbox**, para ello, ejecutaremos

```
kubectl create -f deploy/examples/toolbox.yaml
```

Esperaremos a que esté ready y accederemos al pod mediante:

```
cp1@cp1:~/pluggins/rook-ceph$ kubectl -n rook-ceph get pod
NAME                                                READY   STATUS      RESTARTS        AGE
csi-cephfsplugin-56pxf                              2/2     Running     9               2d4h
csi-cephfsplugin-lpnx7                              2/2     Running     7               2d4h
csi-cephfsplugin-ng6sq                              2/2     Running     4               2d4h
csi-cephfsplugin-provisioner-7f4d578c49-9hkkf       5/5     Running     0               66m
csi-cephfsplugin-provisioner-7f4d578c49-l7mc5       5/5     Running     19 (51m ago)    2d4h
csi-rbdplugin-fz7jx                                 2/2     Running     7               2d4h
csi-rbdplugin-lws2f                                 2/2     Running     4               2d4h
csi-rbdplugin-nssc6                                 2/2     Running     11 (73m ago)    2d4h
csi-rbdplugin-provisioner-5b69756d9d-nf8rm          5/5     Running     15 (51m ago)    2d4h
csi-rbdplugin-provisioner-5b69756d9d-pmpd6          5/5     Running     0               66m
rook-ceph-crashcollector-k8sworker1-574b78669c-55h8n 1/1    Running     0               50m
rook-ceph-crashcollector-k8sworker2-5d69466dc4-8hw5h 1/1    Running     0               37m
rook-ceph-crashcollector-k8sworker3-85c8bf9fbc-b6hpw 1/1    Running     0               29m
rook-ceph-exporter-k8sworker1-f94f4cc7b-kf77p       1/1     Running     0               50m
rook-ceph-exporter-k8sworker2-d5846645f-g9qrl       1/1     Running     0               37m
rook-ceph-exporter-k8sworker3-7d77bb49c8-8w62q      1/1     Running     0               29m
rook-ceph-mds-myfs-a-84d4785b97-clmtn               2/2     Running     2 (31m ago)     37m
rook-ceph-mds-myfs-b-7495b89679-srn97               2/2     Running     2 (31m ago)     37m
rook-ceph-mgr-a-74cf485f9f-wgl7m                    3/3     Running     0               54m
rook-ceph-mgr-b-84c885d8bb-z2mcw                    3/3     Running     0               37m
rook-ceph-mon-a-76465445b9-l4vr5                    2/2     Running     5 (50m ago)     2d4h
rook-ceph-mon-b-7f96c55688-dnm6k                    2/2     Running     0               32m
rook-ceph-mon-c-7bd49b56b6-jcjqn                    2/2     Running     5 (50m ago)     2d3h
rook-ceph-operator-f6f8c9d56-zr6mt                  1/1     Running     0               37m
rook-ceph-osd-0-5d64fbdc65-2bmws                    2/2     Running     3 (50m ago)     2d3h
rook-ceph-osd-1-7465b797bd-cp4qc                    2/2     Running     3 (50m ago)     2d3h
rook-ceph-osd-2-77fcd6c5ff-qxjpf                    2/2     Running     0               32m
rook-ceph-osd-prepare-k8sworker1-vm625              0/1     Completed   0               28m
rook-ceph-osd-prepare-k8sworker2-snlxz              0/1     Completed   0               28m
rook-ceph-osd-prepare-k8sworker3-hmkbl              0/1     Completed   0               28m
```

Ilustración 3.11: Output Get Rook-Ceph Pods

```
kubectl -n rook-ceph exec -it deploy/rook-ceph-tools -- bash
```

Una vez dentro, ejecutaremos

```
ceph status
```

A lo que debería de responder con lo siguiente:

Evidentemente, el valor más importante es el **HEALTH_OK**, que indica que el cluster de Rook-Ceph está Healthy, también muestra información básica de los recursos del mismo.

El resto de parámetros indican lo siguiente:

```
root@cp1:/home/cp1/pluggins/rook-ceph/rook# kubectl -n rook-ceph exec -it deploy/rook-ceph-tools -- bash
bash-4.4$ ceph status
  cluster:
    id:     2f941ccc-c88b-4a55-b508-20bf3434d375
    health: HEALTH_OK

  services:
    mon: 3 daemons, quorum b,a,c (age 7m)
    mgr: a(active, since 6m), standbys: b
    osd: 3 osds: 3 up (since 6m), 3 in (since 7m)

  data:
    pools:   1 pools, 1 pgs
    objects: 2 objects, 577 KiB
    usage:   81 MiB used, 30 GiB / 30 GiB avail
    pgs:     1 active+clean
```

Ilustración 3.12: Output Get Rook-Ceph Pods

- **mon**: Número de monitores, éstos mantienen un registro del estado actual del clúster de almacenamiento, también manejan la autentificación de las peticiones de almacenamiento y son críticos para el funcionamiento del mismo, por ello siempre se recomienda tener más de 1. Cuando una mayoría definida de monitores están activos y se pueden comunicar entre sí, se dice que están en **"Quorum"**, como es el caso.

- **mgr**: Estado del Ceph Manager Deamon, este es responsable de proporcionar servicios adicionales que no estén directamente relacionados con el almacenamiento de datos, como pueden ser el reporte de rendimiento, interfaz de usuario (a través del dashboard, por ejemplo) o la integración con herramientas de monitorización externas.

- **OSDs**: los object storage daemons, cada uno es responsable de manejar el almacenamiento de datos, replicación, recuperación y rebalanceo en cada uno de los dispositibos de almacenamiento del clúster.

3.2.6.5. Creación de Storage Class Convencional

Recordemos que para que los pods puedan solicitar el acceso a un PVC, es necesario especificar los SC disponibles a Rook-Ceph, para ello, podemos crear uno a partir de sus ficheros ya ofrecidos en su github oficial [25].

```
kubectl create -f deploy/examples/csi/rbd/storageclass.yaml
```

Esto nos creará un SC básico con 3 réplicas, que funcinarán en un sentido muy similar al de OpenEBS, el inconveniente de estos tipos de SC es que solo podrán ser accedidos por 1

solo pod al mismo tiempo. De cara a un gran aplicativo con escalado, resulta más interesante la opción de un sistema de almacenamiento que admita acceso simultaneo.

Para ello, primero deberemos de crear un **Filesystem Storage**, el cual consiste en un sistema de archivos de Ceph dentro del clúster y especifica como debe de gestionar el almacenamiento de archivos a bajo nivel [27].

```
apiVersion: ceph.rook.io/v1
kind: CephFilesystem
metadata:
  name: myfs
  namespace: rook-ceph
spec:
  metadataPool:
    replicated:
      size: 3
  dataPools:
    - name: replicated
      replicated:
        size: 3
  preserveFilesystemOnDelete: false
  metadataServer:
    activeCount: 1
    activeStandby: true
```

Vemos que especificamos un nivel de replicación total de 3 unidades, la opción de **preserveFilesystemOnDelete** resulta muy interesante ya que evita borrados totales por error, como estamos en un escenario de prueba, será puesto a **false**.

Crearemos el objeto usando

```
kubectl create -f filesystem.yaml
```

El operador de Rook-Ceph se encargará de preparar todos los recursos necesarios para su funcionamiento.

Ahora, podremos crear el Storage Class correspondiente, indicando el uso del **Filesystem** creado previamente.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
```

```
metadata:
  name: rook-cephfs
# Change "rook-ceph" provisioner prefix to match the operator
   ↪ namespace if needed
provisioner: rook-ceph.cephfs.csi.ceph.com
parameters:
  # clusterID is the namespace where the rook cluster is running
  # If you change this namespace, also change the namespace below
     ↪ where the secret namespaces are defined
  clusterID: rook-ceph

  # CephFS filesystem name into which the volume shall be created
  fsName: myfs

  # Ceph pool into which the volume shall be created
  # Required for provisionVolume: "true"
  pool: myfs-replicated

  # The secrets contain Ceph admin credentials. These are
     ↪ generated automatically by the operator
  # in the same namespace as the cluster.
  csi.storage.k8s.io/provisioner-secret-name: rook-csi-cephfs-
     ↪ provisioner
  csi.storage.k8s.io/provisioner-secret-namespace: rook-ceph
  csi.storage.k8s.io/controller-expand-secret-name: rook-csi-
     ↪ cephfs-provisioner
  csi.storage.k8s.io/controller-expand-secret-namespace: rook-ceph
  csi.storage.k8s.io/node-stage-secret-name: rook-csi-cephfs-node
  csi.storage.k8s.io/node-stage-secret-namespace: rook-ceph

reclaimPolicy: Delete
```

Esta última definición también se encuentra en el github oficial, lo aplicaremos usando

```
kubectl create -f deploy/examples/csi/cephfs/storageclass.yaml
```

```
cp1@cp1:~/storageClasses$ kubectl get sc
NAME              PROVISIONER                     RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
rook-ceph-block   rook-ceph.rbd.csi.ceph.com      Delete          Immediate           true                   2d2h
rook-cephfs       rook-ceph.cephfs.csi.ceph.com   Delete          Immediate           false                  2d1h
```

Ilustración 3.13: Output Get Rook-Ceph SC

Ahora ya podrán ser empleados mediante una solicitud PVC

3.3. Conversión de aplicativo en Docker a Kubernetes

Al igual que con un docker-compose, los recursos necesarios para el despliegue de un aplicativo en Kubernetes se especifican en formato yaml, una práctica común es dividir las partes del aplicativo en ficheros diferentes, en los que se especifican todos los recursos necesarios para el correcto funcionamiento del mismo.

Por ejemplo, todos los recursos necesarios para el de la base de datos estarán especificados en el mismo fichero, lo mismo con la api, el front, etc...

A continuación se expone un caso práctico, partiendo del docker-compose de la aplicación de fichaje de Inerza:

```
version: '3'

services:
  control-horario-postgres:
    image: "postgres:12.10"
    environment:
      - POSTGRES_PASSWORD=Hemstitch9-Dipped-Moonbeam
      - POSTGRES_USER=iz_controlhorario
      - POSTGRES_DB=iz_controlhorario
    ports:
      - "5432:5432"
    volumes:
      - controlHorario_postgresql_vol:/var/lib/postgresql/data

  control-horario-api:
    image: docker-registry.inerza.com/izcontrolhorario-api:latest-↩
       dev
    ports:
      - "9881:8080"
    environment:
      - POSTGRES_HOSTNAME=control-horario-postgres
      - POSTGRES_PORT=5432
      - POSTGRES_USERNAME=iz_controlhorario
      - POSTGRES_PASSWORD=Hemstitch9-Dipped-Moonbeam
      - POSTGRES_DB_NAME=iz_controlhorario
      - ODOO_URI=https://odoo12-pre.inerza.com
      - ODOO_DB=odoo_v11_pro
    depends_on:
      - control-horario-postgres

  control-horario-pwa:
    image: docker-registry.inerza.com/izcontrolhorario-pwa:latest-↩
```

```
      dev
    container_name: control-horario-pwa
    ports:
      - "9880:80"
    environment:
      - API_URL=http://docker-dev.inerza.loc:9881/
    depends_on:
      - control-horario-api

volumes:
  controlHorario_postgresql_vol:
```

Vemos que está dividido en 3 servicios, a continuación especificaremos el despliegue del servicio de control-horario-postgress en Kubernetes, si bien no es necesario, todo estará en el mismo fichero:

Comenzaremos especificando el pvc, como ya ha sido explicado con anterioridad, es el equivalente a una solicitud de volumen de Docker. Puede ser provisto de forma automática mediante un orquestador de almacenamiento como OpenEBS ó Root o puede ser creado con anterioridad por un administrador de forma manual.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: controlhorario-postgresql-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Donde:

- **kind** Corresponde al tipo de recursos que se desea crear, existe una gran cantidad de ellos, desde la unidad más básica como el Pod, hasta reglas ingress, CronJob ó HorizontalPodAutoscale. Como se puede ver, los nombres son bastante intuitivos.

- **name** El nombre del recurso, será usado por otros recursos para acceder a él.

- **spec** Aquí se definen las especificaciones del recurso, cada recurso tiene una serie de campos específicos

- **accessModes** Se usa en la configuración de pvcs para especificar su política de acceso, existen 3 opciones
  - **ReadWriteOnce** Hace que el pvc pueda ser montado como lectura-escritura por un solo nodo, es útil para sistemas de archivo que no permiten la lectura y escritura simultanea.
  - **ReadOnlyMany** Permite que el pvc sea montado como solo lectura en múltiples nodos, para cuando se va a trabajar con estructuras de datos que no necesitan ser modificados, pero que deben ser accesibles desde múltiples nodos.
  - **ReadWriteMany** Permite que el pvc sea montado como lectura-escritura por múltiples nodos.
- **storage** Especifica la cantidad de almacenamiento deseada.

Ahora, pasaremos a la explicación del deployment en sí, al encontrarse dentro del mismo documento, y ser un recurso diferente, deberemos de separarlo usando 3 guiones: ' — '

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: control-horario-postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: control-horario-postgres
  template:
    metadata:
      labels:
        app: control-horario-postgres
    spec:
      containers:
        - name: postgres
          image: postgres:12.10
          env:
            - name: POSTGRES_PASSWORD
              value: Hemstitch9-Dipped-Moonbeam
            - name: POSTGRES_USER
              value: iz_controlhorario
            - name: POSTGRES_DB
              value: iz_controlhorario
          ports:
```

```
        - containerPort: 5432
      volumeMounts:
        - mountPath: /var/lib/postgresql/data
          name: controlhorario-postgres-storage
  volumes:
    - name: controlhorario-postgres-storage
      persistentVolumeClaim:
        claimName: controlhorario-postgresql-pvc
  imagePullSecrets:
    - name: inerza-docker
```

- **kind: Deployment** Al especificar un recurso de tipo deployment, al hacerlo, le indicamos a kubernetes que es un recurso que puede ser actualizado ó modificado y que debe de asegurarse de mantener el número especificado de réplicas en ejecución.

- **replicas** Indica el número total de réplicas del Pod deseadas.

- **selector** Se utiliza para especificar cómo el controlador identifica los pods quie debe gestionar.

  - **matchLabels app** Es el más común y el que se emplea en esta ocasión, indica que debe controlar los pods con la etiqueta definida, en este caso, control-horario-postgress.

  - **matchExpressions** También se puede usar esa variante, se usa para definiciones más complejas, empleando operadores como 'In', 'NotIn', 'Exist' y 'DoesNotExist' para definir qué pods deben ser gestionados. Por ejemplo, se podría usar para gestionar todos los pods cuya etiqueta tenga el valor "frontend"

- **template** Este paco describe el pod a crear y gestionar por parte del deployment

  - **name** El nombre del contenedor

  - **image** La imagen de Docker que se utilizará para el contenedor

  - **env** Variables de entorno de la configuración del contenedor

  - **ports** El puerto interno del contenedor que será expuesto, campo oblicatorio.

- **volumeMounts** En sus opciones se especifica el mountPath del volumen y el nombre del volumen en sí, que será especificado más abajo basándose en el PVC solicitado previamente.

- **volumes** Especificamos el nombre del volumen que va a solicitar el deployment, este es un campo a parte debido a que, como ya hemos mencionado, el pvc puede haber sido creado previamente por un administrador.

- **imagePullSecrets** Este campo no es necesario debido a que se hace uso de una imagen pública, no obstante, está puesto para que sirva de ejemplo.

En caso de usar una imagen de un repositorio privado, deberemos especificar las credenciales en el deployment, lo cual se puede hacer de dos formas, la primera y menos recomendable por motivos obvios es escribiéndolas en forma de texto plano de la siguiente forma:

```
users:
- name: docker-registry.inerza.com
  user:
    username: correoInerza
    password: passwordInerza
```

La forma más correcta sería mediante el uso de un ”secreto.en kubernetes, primero deberá ser creado mediante línea de comandos

```
kubectl create secret docker-registry inerza-docker \
  --docker-server=docker-registry.inerza.com \
  --docker-username=correoInerza \
  --docker-password=passwordInerzaInerza
```

Una vez creado el secreto, se podrá emplear de la forma que ha sido indicada previamente.

Ahora procederemos a explicar la definición del servicio, un recurso que se usa para exponer las aplicaciones que se ejecutan en pods a otros pods o al exterior del clúster:

```
apiVersion: v1
kind: Service
metadata:
  name: control-horario-postgres
spec:
  ports:
    - port: 5432
  selector:
    app: control-horario-postgres
```

Vemos que el simplemente es necesario especificar el puerto por el que se expondrá el servicio, al no especificar el campo **type**, se pondrá, por defecto, **ClusterIP**, las opciones totales posibles son las siguientes:

- **ClusterIP** Expone un servicio a una dirección IP interna del clúster, esto hace que el servicio solo sea accesible desde dentro, ideal para backend interno o bases de datos

- **NodePort** Expone un servicio en un puerto estático en todos los nodos del clúster, kubernetes enrutará todo el tráfico que llegue a ese puerto, en cualquier nodo, a dicho servicio.

- **LoadBalancer** Similar a un NodePort pero solicitando un LoadBalancer al proveedor en la nube en el que esté alojado el clúster.

- **ExternalName** En lugar de una dirección IP interna o externa, expone un alias DNS especificado en otro subcampo, **externalName**

- **Headless** Más que un tipo en si, es un comportamiento que se configura no asignando una dirección IP al servicio, en lugar de balancear el tráfico a través de una dirección IP, el clúster resolverá directamente las direcciones IP de los pods.

3.4. Configuración de Escalado Automático

3.4.1. HPA y Metrics Server

Una de las mayores ventajas de Kubernetes es escalado automático, éste permite crear hacia arriba o hacia abajo los pods de los deployment o replicasets en función del uso de los recursos que se les ha sido asignados o en función de determinadas directivas.

La forma en la que Kubernetes gestiona el escalado de los pods es mediante los **HPA** (Horizontal Pod Autoscaler). Estos son una herramienta que gestiona el escalado de forma automática de los pods desplegados en el cluster en base a las definiciones hechas por el administrador [14].

La diferencia entre un **HPA** y un **Operador** consiste en que estos últimos suponen una especie de extensión de una API o servicio específico y por tanto requiere de un conocimiento más profundo y encapsulado del funcionamiento de éste, además, pueden llevar a cabo cambios más profundos en el comportamiento del clúster. Los HPA sin embargo, como su propio nombre indica, solo escalan "horizontalmente", es decir, solo modifican la cantidad de pods desplegados, por tanto, son menos complejos y más fáciles de implementar.

Las definiciones que pueden ser tenidas en cuenta por un HPA pueden ser creadas en base los siguientes parámetros:

- **Recursos**, como pueden ser uso de cpu, de memoria, número de solicitudes por segundo, latencia en las respuestas, etc...
- **Métricas Externas**, datos de una API o de uso de un servicio en la nube
- **Métricas de Objetos**, una cola de trabajo de algún CRD como vimos en el uso de Rook-Ceph
- **Eventos**, es decir, cambios en la configuración, o hechos específicos de la lógica de negocio.

En esta ocasión solo trabajaremos con uso de recursos, para lograrlo, el HPA necesita una forma de "medir" el uso de recursos de los pods y de los nodos, para ello, emplea el **Metrics**

Server, el cual es un recurso que debe de ser instalado a parte y sirve para proveer a los HPA una fuente de datos fiable sobre el uso de recursos, sin éste, los HPA no podrían funcionar ya que no tendrían forma de medirlo.

3.4.2. Instalación Metrics Server

Los pasos para instalar **Metrics Server** son muy simples, primero, descargaremos el fichero de definición:

```
curl -L -o components.yaml https://github.com/kubernetes-sigs/
  ↪ metrics-server/releases/latest/download/components.yaml
```

Y lo aplicaremos con un

```
kubectl create -f components.yaml
```

Si se ha seguido este documento al pie de la letra, lo más probable es que este paso falle debido a la falta de un certificado TLS con un **Subject Alternative Name** que incluya las direcciones IP de los nodos, por tanto, se debería de crear uno usando **openssl** y ubicarlo en la carpeta correspondiente, en nuestro caso sería **/var/lib/(kubelet/pki/kubelet.crt** acto seguido, se deberá de reiniciar el servicio de de **kubelet**.

No obstante, este último paso también fallará debido a que el certificado debe de ser firmado por una CA reconocida. Al ser un escenario de prueba, podemos saltarnos esta verificación editando el fichero de deployment de metrics server para que acepte conexiones inseguras dentro del clúster:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    k8s-app: metrics-server
  name: metrics-server
  namespace: kube-system
spec:
  selector:
    matchLabels:
      k8s-app: metrics-server
  strategy:
```

```
    rollingUpdate:
      maxUnavailable: 0
  template:
    metadata:
      labels:
        k8s-app: metrics-server
    spec:
      containers:
      - args:
        - --cert-dir=/tmp
        - --secure-port=10250
        - --kubelet-preferred-address-types=InternalIP,ExternalIP,↩
           Hostname
        - --kubelet-use-node-status-port
        - --metric-resolution=15s
        command:
        - /metrics-server
        - --kubelet-insecure-tls
        - --kubelet-preferred-address-types=InternalIP
```

Una vez realizado el cambio, podemos comprobar su funcionamiento ejecutando un

```
kubectl get deployment metrics-server -n kube-system
```

A lo que debería de devolver algo similar a lo siguiente:

```
cp1@cp1:~/pluggins$ kubectl get deployment metrics-server -n kube-system
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
metrics-server   1/1     1            1           10d
```

Ilustración 3.14: Output Metrics Server Deployment

También podemos ejecutar un

```
kubectl top nodes
```

Para ver el uso de recursos de los nodos:

En este caso los valores mostrados corresponden a los siguientes:

- **NAME** Nombre del nodo

```
cp1@cp1:~/pluggins$ kubectl top nodes
NAME          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
cp1           156m         7%     1387Mi          36%
k8sworker1    74m          3%     1097Mi          58%
k8sworker2    109m         5%     1390Mi          74%
k8sworker3    121m         6%     1397Mi          74%
```

Ilustración 3.15: Output Top Nodes

- **CPU(cores)** Cantidad de CPU utilizada por los procesos del nodo medida en milicores
- **CPU %** Porcentaje de uso de la CPU total asignada al nodo
- **MEMORY(Bytes)** Cantidad de memoria RAM utilizada por el nodo medido en MiB
- **MEMORY %** Porcentaje de uso de la memoria RAM total asignada al nodo

Ahora ya podemos asegurar que **Metrics Server** instalado correctamente y disponible para su consumo por parte de cualquier **HPA**

3.4.3. Creación de un HPA

Los HPA se pueden asociar a un deployment, replicaset o statefulset y son dependientes de un deployment en concreto. Si bien ya hemos convertido el aplicativo de fichaje de Inerza, al depender de recursos externos de la empresa, resultará imposible su uso y testeo, por tanto, para este ejemplo se hará uso de un aplicativo de prueba también hecho por Inerza para uno de sus cursos de DevOps impartidos en la ULPGC.

3.4.3.1. Aplicativo de Prueba

El aplicativo es muy simple pero sirve como ejemplo práctico, está compuesto por un servicio frontend con Angular, uno backend con Springwood y una base de datos de Postgresql, los ficheros de deployment para K8s se pueden encontrar en el siguiente **fork** de su github.

3.4.3.2. Liveness y Readiness Probes

El único añadido relevante que no ha sido tratado hasta ahora consiste en las liveness y readiness probes [11]:

- **Liveness Probe** Determina si un contenedor está en funcionamiento, si la sonda falla, se asumirá que el contenedor ha colapsado y lo reiniciará.

- **Readiness Probe** Utilizada para determinar si el contenedor está listo para recibir peticiones, en caso de que falle, Kubernetes evitará enviar peticiones a ésta, muy útil para determinar cuando el pod ya ha terminado de .ªrrancarüna vez iniciado.

Las opciones disponibles para realizarlas son las siguientes:

- **HTTP Get** Kubernetes enviará una petición HTTP al contenedor, si éste responde con un código de estado dentro del rango 200-399, se interpretará como exitoso.

- **TCP Socket** Kubernetes intentará abrir un socket TCP con el contenedor, si es posible, se considerará exitoso.

- **Exec** Ejecutará el comando especificado dentro del contenedor, si devuelve un 0, se considera exitoso.

Dichas sondas deben de ser implementadas en la sección de configuración del contenedor del pod y han sido introducidas en todos los deployment, si bien las respuestas a las mismas se pueden personalizar, la mayoría de las APIs o Base de datos vienen con una implementación de las mismas, las cuales se detallarán a continuación:

En el caso de Postgres, las sondas se definen de la siguiente forma:

```
livenessProbe:
    exec:
        command:
        - pg_isready
        - "-U"
        - "bookreview"
        - "-d"
        - "bookreview"
        - "-h"
        - "localhost"
```

```
    initialDelaySeconds: 30
    periodSeconds: 10
    timeoutSeconds: 5
readinessProbe:
    exec:
      command:
      - pg_isready
      - "-U"
      - "bookreview"
      - "-d"
      - "bookreview"
      - "-h"
      - "localhost"
    initialDelaySeconds: 5
    periodSeconds: 10
    timeoutSeconds: 5
```

Como podemos ver, el comando para comprobarlo es el mismo en ambos casos, la diferencia consiste en la forma en la que la API de Kubernetes las gestiona y los márgenes de tiempo configurados.

En el caso de la API de Springwood, sería de la siguiente forma:

```
livenessProbe:
    httpGet:
      path: /actuator/health/liveness
      port: 8080
    initialDelaySeconds: 120
    periodSeconds: 20
readinessProbe:
    httpGet:
      path: /actuator/health/readiness
      port: 8080
    initialDelaySeconds: 60
    periodSeconds: 5
```

En cuando a los pods del front, bastaría con realizar un http get a cualquier path

```
livenessProbe:
  httpGet:
    path: /
    port: 80
  initialDelaySeconds: 10
  periodSeconds:
readinessProbe:
```

```
  httpGet:
    path: /
    port: 80
  initialDelaySeconds: 15
  periodSeconds: 5
```

Podremos desplegar los deployment con un

```
kubectl apply -f nombre-deployment
```

3.4.3.3. Creación de los HPA

A continuación se explicará el HPA del deployment del servicio de front del aplicativo:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: bookreview-front-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: bookreview-front
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 25
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Pods
        value: 1
        periodSeconds: 60
    scaleUp:
      stabilizationWindowSeconds: 0
      policies:
      - type: Percent
```

```
        value: 100
        periodSeconds: 15
```

Los aspectos fundamentales del mismo son los siguientes:

- **scaleTargetRef** La referencia hacia el objetivo de escalado, en este caso, el deployment de bookreview-front

- **min-maxReplicas** El rango posible de réplicas de los pods, mínimo 1 y máximo 5.

- **metrics** El tipo de métricas a tener en cuenta de cara al escalado, en este caso, se especifica que es de tipo **resource** y que corresponde al porcentaje medio de uso de CPU asignado al POD

- **behavior** El comportamiento del escalado, hay dos tipos:

  - **scaleDown** Escalado hacia abajo, con el parámetro **stabilizationWindowSeconds** se especifica que el sistema deberá de esperar 300 segundos antes de comenzar a escalar hacia abajo, una vez comenzado, podrá eliminar 1 pod cada 60 segundos. De esta forma, evitamos cambios bruscos que podrían empeorar la calidad del servicio

  - **scaleUp** Escalado hacia arriba, en este caso, al establecer el **stabilizationWindowSeconds** a 0 indicamos que no espere en caso de necesitar aumentar la cantidad de pods, por motivos evidentes. También especificamos que puede escalar hacia arriba hasta el 100 % de los PODS permitidos y que cada operación se realizará en un intervalo mínimo de 15 segundos. Con el escalado hacia arriba es mejor ser menos conservadores ya que en caso contrario empeoraría el servicio en situaciones de mucha carga.

Una vez desplegados, podremos ver que están correctos y se comunican bien con el metrics server comprobando los porcentajes de uso:

```
cp1@cp1:~/libraryApp/hpa$ kubectl get hpa
NAME                  REFERENCE                      TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
bookreview-back-hpa   Deployment/bookreview-back     2%/25%    1         5         1          2d22h
bookreview-bdd-hpa    Deployment/bookreview-bdd      2%/25%    1         3         1          2d22h
bookreview-front-hpa  Deployment/bookreview-front    0%/25%    1         5         1          10d
```

Ilustración 3.16: Output Get HPA

3.4.3.4. Prueba de Estrés

Para comprobar el escalado, podemos realizar una prueba de estrés simulando tráfico, para ello, haremos uso de la herramienta de código abierto **Locust**. Ésta permite realizar simulaciones de tráfico en la que usuarios simulados realizan tareas escritas en Python por el developer, es flexible y muy fácil de usar, además, una vez ejecutado el archivo, genera una interfaz web que facilita su uso. Para emplearlo, primero deberemos de instalar la librería oficial en el host desde el que se quieren realizar las pruebas [22].

En este caso realizaré las pruebas en un host de windows, así pues, desde una terminal de powershell, ejecutaremos:

```
pip install locust
```

Ahora deberemos de definir el fichero de configuración de Locust, simplemente definiremos una **@task**, que consistirá en acceder a la página principal, no obstante, gracias a la flexibilidad de Python, Locust también permite testear APIs y una gran cantidad de recursos diferentes.

```
from locust import HttpUser, task, between

class WebsiteUser(HttpUser):
    wait_time = between(1, 5)

    @task
    def load_page(self):
        self.client.get("/", name="Homepage")
```

Luego nos ubicaremos en la carpeta en la que se encuentra el fichero y ejecutaremos

```
locust
```

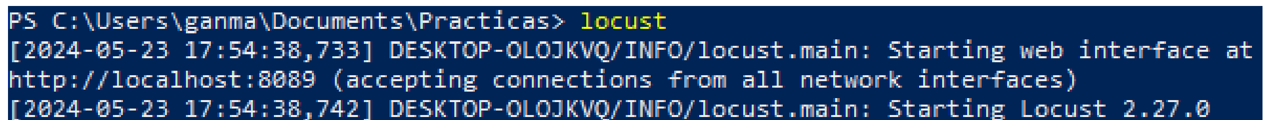


Ilustración 3.17: Inicialización de Locust

Como podemos ver, nos indicará que se puede acceder a la interfaz gráfica a través de la dirección **localhost:8089** accederemos desde un navegador web y nos saldrá la siguiente

pantalla de configuración:

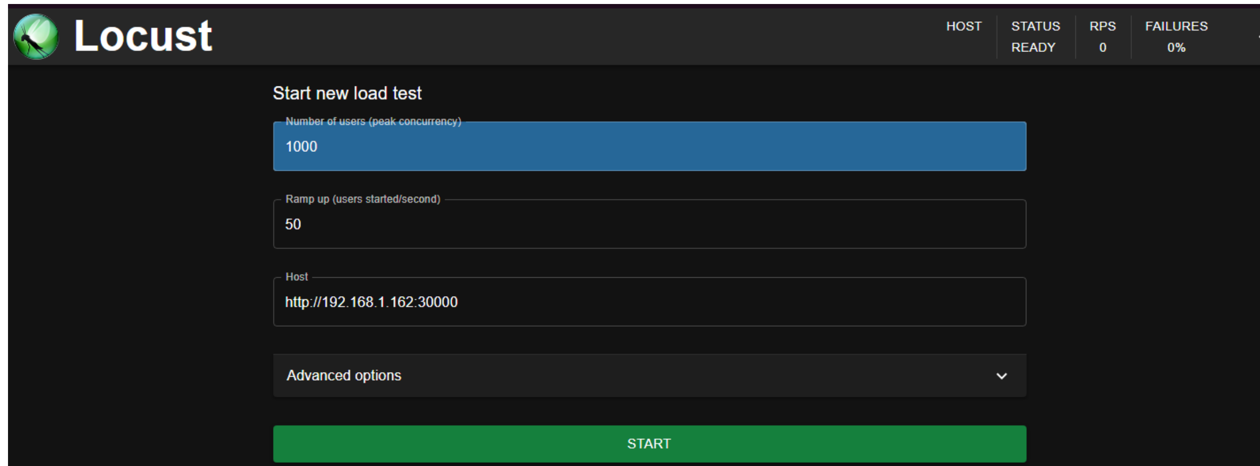
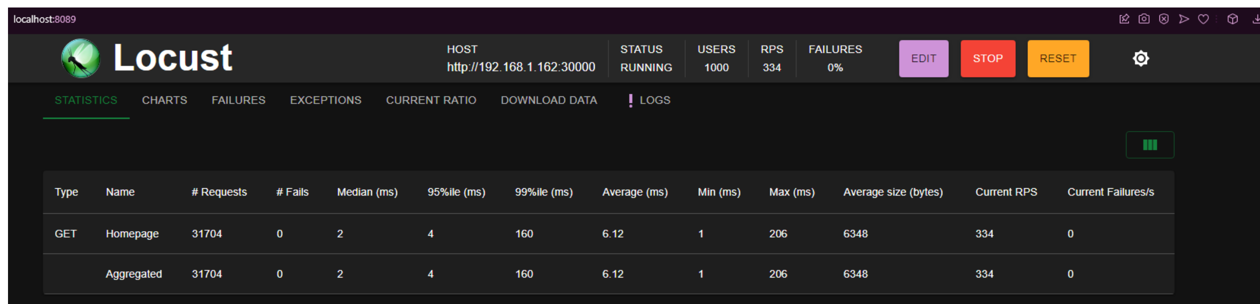


Ilustración 3.18: Interfaz Gráfica Locust

Indicaremos el número total de usuarios simulados y el rango en el que podrán aumentar en cada segundo, acto seguido iniciaremos apretando con el botón start. Comenzará a simular el tráfico, podremos ver el tiempo medio de respuesta y el índice de éxito de las peticiones.



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	Homepage	31704	0	2	4	160	6.12	1	206	6348	334	0
	Aggregated	31704	0	2	4	160	6.12	1	206	6348	334	0

Ilustración 3.19: Locust Ongoing

Si realizamos un watch de los hpa podremos ver como a medida que aumentan las peticiones, el uso de recursos de los PODs aumenta y, por ende, el hpa creará nuevas réplicas del mismo:

Ilustración 3.20: Watch HPA

Capítulo 4

Conclusiones y trabajo futuro

4.1. Conclusiones

Llegados a este punto, podemos asegurar que se han visto suficiente contenido de Kubernetes como para hacerse una idea general de sus capacidades y de las implicaciones que conlleva la implementación de un clúster del mismo.

Se ha demostrado que si bien la creación de un clúster de Kubernetes es un proceso mucho más largo y complejo que uno de Docker Swarm, esta dificultad no supone una barrera que imposibilite su despliegue.

Como ya ha sido mencionado en este informe, en el sector existe una imagen de dificultad en relación a Kubernetes que no se corresponde con la realidad actual, años antes, solo se podía crear un clúster de K8s a "mano", es decir, configurando manualmente todos y cada uno de los módulos y servicios necesarios, como hemos visto, esto ya no es necesario gracias a herramientas como **Kubeadm**, que automatiza la configuración de estas pero sigue permitiendo su configuración al más bajo nivel, esto convierte Kubernetes en una opción mucho más atractiva que otras alternativas como puede ser Docker Swarm.

Kubernetes es, indiscutiblemente, el estándar actual del sector, tiene la comunidad de desarrolladores más grande y los mayores proveedores de servicios en la nube se centran en K8s, no en Docker Swarm. Esto fomenta un grado de actualización e incorporación de ca-

racterísticas nueva con el que no puede competir Docker Swarm. Como hemos podido ver, una de las mayores ventajas de Kubernetes es su alto grado de modularidad, no solo permite alterar el funcionamiento de sus componentes más básicos en función de unas necesidades específicas, sino que además ofrece una larga cantidad de opciones a elegir para poder implementar sus funciones de alto nivel, como puede ser la **Pod Network** o el **Orquestador de almacenamiento**, el cual supone una centralización de la gestión de almacenamiento muy práctica. No olvidemos que el funcionamiento básico de estos componentes también puede ser modificado para suplir nuestras necesidades.

También permite elegir del **container runtime**, un componente básico para el funcionamiento del mismo, si bien la opción de **CRI-O** suele considerarse la mejor ya que está optimizado para Kubernetes y es mucho más eficiente que **containerd**. Todas estas opciones de modularización son imposibles de realizar en Docker Swarm, la cual sigue una arquitectura mucho más "**monolítica**" en comparación a Kubernetes, y ofrece mucho menor rango de configuración. Esto hace no resulte descabellado decir decir que no hay nada que haga Docker Swarm que no pueda hacer Kubernetes y **mejor**, es más, muchas de las capacidades de Kubernetes solo pueden ser emuladas por Docker Swarm a través de herramientas externas que no pueden compaginarse con éste al nivel que lo puede hacer Kubernetes a través del uso de los **CRDs**.

Otro de los atractivos más grandes de Kubernetes es implantación con los servicios de proveedores en la nube, los más grandes ofrecen directamente un **control plane** totalmente funcional, lo cual libra a las empresas de la parte más tediosa de usar Kubernetes. Este servicio se ofrece por un precio en torno a los 73$ mensuales, si bien es cierto que este precio se **sextuplica** en caso de no actualizarlo a la versión más reciente de Kubernetes dentro del margen de tiempo especificado por el proveedor. Además, el escalado de un clúster de Kubernetes en la nube no tiene por qué limitarse a nivel de deployments, los proveedores en la nube permiten **solicitar y liberar nodos** automáticamente en función de las reglas establecidas. Este aspecto positivo puede no ser relevante para empresas cuyos servicios se ofrecen directamente desde un CPD **on-premise**, como es el caso de **Inerza**

Resulta imprescindible resaltar que, al contrario que la creencia popular del sector, **Kubernetes no funciona con Docker**, al menos, no al mismo nivel que al que se piensa. Como hemos podido ver, el único componente de Docker que ha sido empleado han sido las imágenes de contenedores subidas a Docker Hub, el resto de componentes, incluído el **CRI**, son totalmente ajenos a Docker.

No todo son aspectos positivos, si bien es cierto que Kubernetes tiene una mejor gestión

de recursos a nivel global debido a que prescinde de aspectos innecesarios de Docker, el coste computacional para alcanzar el provisionamiento básico de servicios del clúster es más alto que en Docker Swarm. Recordemos también que la forma de proveer estos servicios es mediante PODs, y que cada nodo de K8s tiene un límite lógico de PODs límitado, por lo cual, a mayor cantidad de servicios y complejidad de estos, menos espacio para el despliegue de aplicativos convencionales queda en el nodo, lo cual implica el aumento de nodos y el coste que esto supone.

La popularidad de Kubernetes en el sector implica que a menudo se vea como la única opción posible para la orquestación de contenedores, no obstante, si bien es cierto que es superior a Docker Swarm en todos los aspectos, en muchos casos la elección de K8s puede ser excesiva, ya que supone una alternativa innecesariamente compleja para despligar aplicativos simples y con poca fluctuación en el tráfico de peticiones. Muchas veces basta con un **docker-compose** o un pequeño clúster de Docker Swarm, no porque sea "mejor" que Kubernetes, sino porque es más simple y barato, aspectos que la mayoría de desarrolladores no tienen en cuenta pero que son esenciales en la toma de decisiones de este tipo.

Sí es cierto que existen determinados aplicativos que solo pueden ser desplegados en producción mediante Kubernetes, o que su despliegue en Docker Swarm implicaría un sobreesfuerzo lo suficientemente grande como para justificar la creación de un clúster de Kubernetes. Un ejemplo práctico podría ser un servicio de **streaming en vivo** mediante **rtmp** nativo, en el que cada vez que una persona decide hacer un streaming, es necesario lanzar un contenedor específico para esa persona con una configuración que cambia en función de varios parámetros y que nunca es estática, además, en función del número de personas que estén viendo esa retransmisión, podría ser necesario aumentar el número de contenedores con esa configuración y desplegar servicios de loadbalancer bajo demanda. Estos requisitos se podría suplir con un **operador** de Kubernetes, pero sería sumamente tedioso de realizar con Docker Swarm.

En cuanto a la sustitución de un clúster de Docker Swarm por uno de Kubernetes, la decisión solo estaría justificada en caso de ser parte de un esfuerzo de modernización general de la infraestructura o por el despliegue de un aplicativo con un grado de complejidad y requisitos que pueda hacer que Docker Swarm se quede corto de capacidad. También es posible que la versatilidad de Kubernetes y los pluggins que permite incoorporar supongan una reducción de costes con respecto a la infraestructura y servicios ya implementados.

4.2. Trabajo Futuro

Si bien el resultado final del clúster se podría definir como uno totalmente apto para producción, no podemos pasar por alto el aspecto de la seguridad, algo que no ha sido tenido en cuenta lo suficiente en este TFT. Para empezar, en un clúster real, es necesaria la creación de varios perfiles de administrador con sus correspondientes limitaciones, tanto a nivel de permisos como de recursos disponibles para que el clúster de correcto servicio a la empresa, ya que en escenarios reales, en muy pocas ocasiones el clúster está mantenido por una sola persona, y menos, con permisos de root.

Además de las consideraciones de seguridad internas del clúster, sería necesario también reforzar la seguridad en la máquina que hace de puente entre la red interna y la red externa, así como valorar modificar la infraestructura general de este tipo de “enlace” entre el exterior y el interior. También sería muy recomendable añadir herramientas de monitorización de acceso y tráfico.

Si bien el desarrollo no debería de ser muy diferente, podría valorarse desplegar el clúster en algún tipo de proveedor de servicios en la nube, como podría ser AWS, Azure o DigitalOcean.

Bibliografía

[1] Atlassian. Kanban boards — atlassian. https://www.atlassian.com/es/agile/kanban/boards.

[2] Microsoft Azure. Azure container instances — microsoft azure. https://azure.microsoft.com/es-es/products/container-instances/.

[3] CRI-O. Cri-o — lightweight container runtime for kubernetes. https://cri-o.io.

[4] Docker. Key concepts of docker swarm — docker documentation. https://docs.docker.com/engine/swarm/key-concepts/.

[5] Docker. What is a container? — docker. https://www.docker.com/resources/what-container/.

[6] Red Hat. What is container orchestration? — red hat. https://www.redhat.com/en/topics/containers/what-is-container-orchestration.

[7] Helm. Installing helm — helm documentation. https://helm.sh/docs/intro/install/.

[8] IBM. Container orchestration — ibm. https://www.ibm.com/topics/container-orchestration.

[9] Inerza. Qué hacemos — inerza. https://www.inerza.com/que-hacemos/.

[10] Container Networking Interface. Cni: Container networking interface — github repository. https://github.com/containernetworking/cni.

[11] Kubernetes. Configure liveness, readiness and startup probes - kubernetes documentation. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/.

[12] Kubernetes. Container runtimes — kubernetes documentation. https://v1-29.docs.kubernetes.io/docs/setup/production-environment/container-runtimes/.

[13] Kubernetes. Creating a cluster with kubeadm — kubernetes documentation. https://v1-29.docs.kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/.

[14] Kubernetes. Horizontal pod autoscaler - kubernetes documentation. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[15] Kubernetes. Install kubeadm — kubernetes documentation. https://v1-29.docs.kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/.

[16] Kubernetes. Kubernetes — production-grade container orchestration. https://kubernetes.io.

[17] Kubernetes. Network plugins — kubernetes documentation. https://v1-29.docs.kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.

[18] Kubernetes. Networking and network policy — kubernetes documentation. https://v1-29.docs.kubernetes.io/docs/concepts/cluster-administration/addons/#networking-and-network-policy.

[19] Kubernetes. Persistent volumes — kubernetes documentation. https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[20] Kubernetes. Storage classes — kubernetes documentation. https://kubernetes.io/docs/concepts/storage/storage-classes/.

[21] Kubernetes. Components — kubernetes documentation. https://kubernetes.io/docs/concepts/overview/components/, 2024.

[22] Locust. Locust - scalable user load testing tool. https://locust.io.

[23] OpenEBS. Installation guide — openebs documentation. https://openebs.io/docs/2.12.x/user-guides/installation.

[24] Rook. Example configurations - getting started with rook. https://rook.io/docs/rook/latest-release/Getting-Started/example-configurations/.

[25] Rook. Rook — github repository. https://github.com/rook/rook.

[26] Rook. Rook — open-source storage for kubernetes. https://rook.io.

[27] Rook. Shared filesystem cephfs - storage configuration guide. https://rook.io/docs/rook/latest-release/Storage-Configuration/Shared-Filesystem-CephFS/filesystem-storage/.

[28] Amazon Web Services. Aws fargate — amazon web services. https://aws.amazon.com/es/fargate/.

[29] Terraform. Terraform by hashicorp. https://www.terraform.io.

[30] Tigera. Calico quickstart for kubernetes - tigera documentation. https://docs.tigera.io/calico/latest/getting-started/kubernetes/quickstart.

[31] Wikipedia. Docker (software) — wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Docker_(software).

[32] Wikipedia. Erasure code — wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Erasure_code.

[33] Wikipedia. Taiichi ohno — wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Taiichi_Ohno.

[34] Wikipedia. Virtualización a nivel de sistema operativo — wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/VirtualizaciÃşn_a_nivel_de_sistema_operativo.