Diseño de una arquitectura privada para procesamiento y almacenamiento distribuido en cloud
Design of a private architecture for distributed processing and storage in cloud.

Trabajo de Fin de Grado
Curso 2022–2023

Autor
Daniel Cobos Peñas
Alejandro Moreno Murillo

Director
Yolanda García Ruiz

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Diseño de una arquitectura privada para procesamiento y almacenamiento distribuido en cloud
Design of a private architecture for distributed processing and storage in cloud.

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Daniel Cobos Peñas
Alejandro Moreno Murillo

Director
Yolanda García Ruiz

Convocatoria: *Septiembre 2023*

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

15 de septiembre de 2023

Dedicatoria

A nuestros padres, y hermanas, os lo debemos todo

Agradecimientos

A Jimmy y al equipo de técnicos de informática, por su inestimable ayuda a la gestión de la red y al préstamo de tarjetas de red

Resumen

Diseño de una arquitectura privada para procesamiento y almacenamiento distribuido en cloud

Este trabajo de final de grado consiste en el diseño y la preparación de una arquitectura cloud privada para el procesamiento y almacenamiento distribuido de datos. La principal idea tras este trabajo es crear un cluster compuesto por 5 computadores, mediante el uso de Kubernetes, que actúen en conjunto para formar un sistema distribuido, siguiendo el modelo de maestro/esclavo, para poder procesar grandes conjuntos de datos. De los cinco equipos será el maestro el encargado de repartir las tareas entre todos los nodos, garantizando el equilibrado de carga. El cluster por su parte implementará los servicios que requieran los usuarios pudiendo añadirlos o quitarlos mediante el uso de Docker y Kubernetes. Inicialmente se configurará el cluster para poder usar dos servicios fundamentales. Para el almacenamiento distribuido se instalará MongoDB y para poder realizar el procesamiento distribuido se instalará Spark. Entre medias Jupyter será el encargado de integrar ambos servicios para completar la arquitectura distribuida.

El trabajo pretende mostrar todas las fases que tienen lugar para la correcta implementación del clúster con el objetivo de que pueda ser sencillamente replicado en otras máquinas, por lo que la memoria también pretende ser un manual de instrucciones ante posibles réplicas.

Para la correcta implementación de la arquitectura es imprescindible garantizar la privacidad y consistencia del sistema asegurando una alta tolerancia a fallos, al igual que la correcta elaboración de una red privada funcional en la que solo un único nodo, el maestro, tendrá acceso a internet.

Finalmente también se incluirá un servicio de monitorización que facilite todas las tareas de supervisión al administrador del clúster.

Palabras clave

Clúster, Cloud, Kubernetes, Docker, MongoDB, Spark, red, distribuido, Rook, monitorización

Abstract

Design of a private architecture for distributed processing and storage in cloud.

This final degree project consists of the design and elaboration of a private cloud architecture for distributed data processing and storage. The main idea behind this work is to create a cluster composed of 5 computers, through the use of Kubernetes, that act together to form a distributed system, following the master/slave model, to be able to process large sets of data. Of the five teams, the master will be in charge of distributing the tasks among all the nodes, making sure that load balancing is happening. The cluster, for its part, will have implemented the services that users require, who will be able to add or remove them through the use of Docker and Kubernetes. Initially the cluster will be configured to host two fundamental services. For distributed storage, MongoDB will be installed and to perform distributed processing, Spark will also be installed. In between, Jupyter will be in charge of integrating both services to complete the distributed architecture.

The work aims to show all the phases that take place for the correct implementation of the cluster with the main goal of being easy to replicate on other machines. Due to this the memory is also intended to be a manual of instructions for the possible replicas.

For the correct implementation of the architecture it is essential to guarantee the privacy and consistency of the system ensuring high fault tolerance, just as the right development of a functional private network in which only one node, the master, will have access to the internet.

Finally, a monitoring service will also be included so it facilitates all supervising tasks to the cluster administrator.

Keywords

Cluster, cloud, Kubernetes, Docker, MongoDB, Spark, network, distributed, Rook, monitoring

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este apartado vamos a realizar una breve introducción del trabajo de final de grado realizado, exponiendo distintos apartados introductorios que pretenden dar una visión sencilla del trabajo y la memoria.

1.1. Objetivos

El principal objetivo de este trabajo es poder elaborar con exito una arquitectura privada de procesamiento y almacenamiento distribuido de datos. Para lograr el objetivo es necesario realizar primero una investigación exhaustiva sobre las distintas herramientas que podemos utilizar para posteriormente proceder con la implementación de las mismas. Algunas de estas herramientas son Docker y Kubernetes, de las cuáles profundizaremos en los siguientes capítulos, al igual que de MongoDB y Spark, que serán los dos principales servicios que albergará el sistema. Además, para ofrecer un entorno realmente útil y eficaz otro objetivo es aportar un sistema de monitorización al administrador para poder supervisar correctamente todo el trabajo ya realizado.

Otro objetivo importante que vamos a tener en cuenta a lo largo de la memoria es la posibilidad de réplica. Hemos presentado un trabajo con la finalidad de que cualquier persona con un mínimo de conocimiento sobre informática pueda elaborar su propio sistema siguiendo minuciosamente los distintos apartados de los que consta la memoria.

Consideramos estos dos objetivos los más importantes de todos aunque no son los únicos. Otra finalidad del trabajo es poder aprender y profundizar sobre herramientas menos habituales en la carrera de Ingeniería Informática, para poder tener un abanico de conocimiento más alto fundamentalmente en el campo DevOps. También teníamos como objetivo poder trabajar con servicios punteros y actuales, como es el procesamiento distribuido en cloud.

A medida que avanzabamos con el proyecto y a medida que profundizabamos en los conceptos surgían nuevos objetivos intermedios, de los que explicaremos con

mayor detalle a lo largo de todo el Capítulo 3.

1.2. Motivación

Para enfrentarse a un TFG de estas características es muy importante tener clara cuál es la motivación detrás del mismo. Uno de los aspectos que teníamos en cuenta antes de elegir un trabajo es que nos debería resultar bastante atractivo, y valorábamos positivamente que fuese un trabajo relacionado con algo que no hubiésemos visto en profundidad en la carrera, para poder investigar acerca de ello y adquirir así nuevos conocimientos. Partiendo de esta base, uno de los TFGs que más nos atrajo fue éste. Nos pareció muy interesante ya que nos permitía aprender y mejorar en el uso de tecnologías DevOps, así como profundizar en temas actuales y con mucho futuro como es el Big Data o la tecnología Cloud.

Además de esto, las posibilidades que nos ofrecía también jugaron un punto a su favor. Había mucha libertad para poder investigar y desarrollarlo de la mejor manera posible para cumplir con los objetivos.

1.3. Plan de trabajo

Para llevar a cabo un trabajo como el presentado, en el que se debe acudir regularmente a un punto físico es importante tener una organización clara y eficaz para poder optimizar el tiempo disponible. Uno de los aspectos que tuvimos más claro desde el principio era la urgente necesidad de poder trabajar remotamente con las cinco máquinas, para poder progresar de forma más rápida y continua. Sin embargo, aunque lo conseguimos relativamente pronto, hubo muchos problemas que nos obligaron a acudir presencialmente con mayor frecuencia. Algunos de estos problemas se pueden ver en la Sección 5.1.

Teniendo esto en cuenta nuestro plan de trabajo fue el esperado. Primero empezamos acudiendo al lugar donde residían los cinco ordenadores para poder configurar las máquinas, e instalamos los sistemas operativos y preparamos la arquitectura de red entre ellos, e incluso abrimos los ordenadores para introducir tarjetas de red o modificar componentes que no funcionaban correctamente. También conseguimos dos formas de poder conectarnos a las máquinas de forma remota, por ssh y a través del escritorio remoto. Una vez conseguimos esto último fue cuando pasamos a la instalación y preparación de todo el software necesario, tanto de Kubernetes como Docker como demás herramientas que iremos explicando a lo largo de todo este documento. Primero nos centramos en orquestar las máquinas para poder formar un clúster robusto y eficiente. Conseguido esto nos centramos en la implementación de los servicios requeridos como MongoDB o Spark. También incluímos un sistema de monitorización para poder controlar y vigilar el correcto funcionamiento del clúster y de los servicios.

Una vez se instaló todo el software que consideramos oportuno empezamos con

la realización de pruebas para comprobar que todo funciona como debería y con la elaboración de la memoria. Para las comprobaciones introdujimos bases de datos de prueba y trabajos preparados para Spark.

1.4. Explicaciones sobre la memoria

Para poder completar el trabajo de final de grado es necesario presentar una memoria bien redactada y organizada. Desde el comienzo hemos tenido claro que patrón queríamos que siguiese nuestra memoria para alcanzar nuestros objetivos. Este patrón se podría resumir en los dos capítulos siguientes a éste.

Nuestra idea es basicamente presentar dos capítulos principales con mucho mayor volumen de información que el resto de capítulos. El primero de ellos, el capítulo 2, pretende ser un capítulo puramente teórico, en el que se explican los conceptos y herramientas que se van a utilizar. Al ser un trabajo con mucha carga teórica consideramos fundamental presentar una buena base sólida sobre los términos y conceptos utilizados. El segundo de ellos, el capítulo 3, es un capítulo fundamentalmente práctico. Aquí queremos mostrar todo el trabajo práctico que hemos realizado sobre los conceptos explicados en el anterior capítulo. Este capítulo está formado por distintas secciones en las cuáles cada una de ellas contiene el proceso a seguir para implementar alguna herramienta específica.

El objetivo principal de la memoria es servir como manual de instrucciones para cualquier lector que quiera elaborar su propio clúster de procesamiento distribuido y no conozca las herramientas que vamos a tratar. Por esta razón se ha intentado que la memoria tenga un lenguaje accesible y un patrón de instrucciones muy claro y conciso.

Por otra parte, la memoria ha sido desarrollada usando Overleaf, una herramienta de redacción colectiva que utiliza LaTeX. Para aprender este lenguaje nos servimos de la documentación oficial y de libros como el de Alexander Borbon A. (2017).

1.5. Estructura de la memoria

La memoria consta de 5 capítulos principales:

- **Capítulo 1: Introducción**: Breve introducción del trabajo entregado.
- **Capítulo 2: Contexto tecnológico**: Capítulo teórico que trata todo el contexto tecnológico del trabajo, es decir conceptos teóricos fundamentales para el desarrollo del proyecto.
- **Capítulo 3: Descripción del trabajo**: Como su nombre indica este capítulo contiene todo el trabajo práctico realizado.
- **Capítulo 4: Estado de la cuestión**: En este capítulo comentamos proyectos similares al realizado especificando las cosas que nos han inspirado para el

nuestro.

- **Capítulo 5: Conclusiones y trabajo futuro**: En este capítulo resumimos las conclusiones que hemos obtenido al realizar el trabajo y comentamos las futuras mejoras que nos gustaría implementar.

Además de esto se puede encontrar en el apéndice A una lista con algunos de los comandos más importantes y utilizados en todo el contexto de nuestro proyecto, acompañados de una breve descripción de los mismos.

Capítulo 2

Contexto Tecnológico

En este capítulo vamos a explicar detalladamente todo el contexto tecnológio que envuelve al trabajo realizado. Hablaremos de todas las herramientas que hemos utilizado y de los conceptos teóricos importantes para comprender el desarrollo del proyecto. La idea detrás de este capítulo es poder aportar conocimientos fundamentales que el lector deberá conocer antes de avanzar al siguiente capítulo, en el que se implementarán muchas de las funcionalidades explicadas aquí.

2.1. Computación distribuida en el cloud y su situación actual

A medida que el mundo y la ciencia continúan evolucionando, la informática desde sus orígenes recientes hasta la actualidad ha crecido exponencialmente hasta convertirse en una herramienta esencial y omnipresente. Algunos de nosotros recordaremos cómo se almacenaban los kilobytes en los disquetes hace décadas, los megabytes más tarde en discos duros y, más recientemente, los terabytes en matrices de discos.

Del mismo modo, la necesidad de ejecutar operaciones computacionales cada vez más complejas en un conjunto de datos que no tiene por qué ser necesariamente grande, aunque cada vez es mayor, deriva por igual en la obligación de aumentar los recursos informáticos de las máquinas. Esto se debe a que siempre hay límites para una única CPU, como la limitación en el número de transistores de los que se componen, entre otros factores. De esta evolución y de los requisitos que demandan sistemas de alto rendimiento, nace la computación paralela.

Existen distintos niveles de implementación para este tipo de sistemas. Desde el nivel de bit, donde las operaciones se paralelizan a la escala más básica, pasando por la paralelización de instrucciones e hilos, que permite la ejecución simultánea de múltiples secuencias de comandos, hasta llegar a núcleos y procesadores. A medida que escalamos, los niveles de datos y tareas se enfocan en la distribución y ejecución paralela de conjuntos de datos y operaciones, respectivamente.

En un nivel más alto, la paralelización se extiende a arquitecturas más extensas y complejas, como los clústeres, que son colecciones de computadoras que trabajan juntas, de las que hablaremos más adelante, y el nivel de gran escala, que a menudo involucra múltiples clústeres o centros de datos.

Es aquí donde surge la exigencia de un sistema de almacenamiento distribuido. Dado que las operaciones se distribuyen y escalan en varios procesadores o nodos, el acceso y la gestión de datos deben ser coherentes, rápidos y, lo que es más importante, continuos. Un sistema de almacenamiento centralizado tradicional se convierte en un cuello de botella que limita el rendimiento y la escalabilidad de todo el sistema.

Los sistemas de almacenamiento distribuido, como Ceph, HDFS o Cassandra, como los que implementaremos en este trabajo, dividen y replican datos en varios nodos o dispositivos. Esto no solo proporciona redundancia y resiliencia, lo que garantiza la seguridad de los datos en caso de fallos, sino que también permite operaciones de lectura y escritura en paralelo. Estas operaciones se distribuyen entre nodos, lo que elimina puntos únicos de falla y maximiza el uso de los recursos.

Además, suelen proporcionar mecanismos para equilibrar la carga, garantizar la coherencia de los datos y optimizar el rendimiento en función de la ubicación y la demanda. Al igual que la computación paralela busca maximizar la eficiencia del procesamiento, los sistemas de almacenamiento distribuido buscan hacer lo mismo con el acceso y la gestion de datos, convirtiéndose en un complemento esencial para las arquitecturas distrubuidas modernas de alto rendimiento.

Teniendo en cuenta el contexto actual es donde nace nuestro proyecto. El principal objetivo tras él es poder crear una arquitectura de procesamiento y almacenamiento distribuido que pueda manejar las exigencias actuales de la informática, dónde cada vez se requiere mayor velocidad de procesamiento y mayor capacidad de almacenamiento de datos.

2.2. Sistema operativo Ubuntu. Versión y alternativas

Uno de los aspectos que debemos tener en cuenta a la hora de montar un sistema es el sistema operativo que vamos a instalar en los equipos. Inicialmente no había ningún sistema operativo instalado por lo que tuvimos la libertad de poder analizar diversos SO y elegir uno que fuera compatible con gran parte de las herramentas que ibamos a usar y que fuera ligero para obtener un mayor rendimiento. Teniendo en cuenta que, además, la idea detrás de este trabajo es que se pueda replicar de forma sencilla y que sea compatible con la implementación de algo conocido como contenedores, en lo que profundizamos en la Sección 2.4.

Siguiendo este criterio decidimos optar por Ubuntu, en su última versión, la 22.0.4. Tras investigar vimos que la gran mayoría de trabajos que consistían en la implementación de clústeres trabajaban con Ubuntu, y esto se debe a la gran facilidad que ofrece para integrar herramientas fundamentales para nuestro trabajo como Docker o Kubernetes, de las que hablamos en detalle en las Secciones 2.6 y 2.8

respectivamente. Ubuntu es considerada la plataforma idónea para Docker y Kubernetes, siendo fácil de usar para administrar contenedores y orquestar aplicaciones en el clúster que queremos montar. Además, Ubuntu también proporciona paquetes Docker actualizados en sus repositorios oficiales, lo que facilita la instalación y el mantenimiento, para futuras réplicas. Además de lo mencionado anteriormente, Ubuntu también es conocido por la facilidad de uso, especialmente para usuarios de windows o macOS, por la estabilidad y seguridad que ofrece y por la escabilidad que proporciona.

Algunas de las alternativas que tuvimos en cuenta antes de decantarnos por Ubuntu fueron Debian y PelicanHPC. Este último que hemos mencionado no goza de la popularidad de los dos anteriores pero es bastante útil para la creación de clusters, teniendo en cuenta que fue desarrollada con ese propósito. Básicamente es una distribución derivada de Debian que fue desarrollada únicamente para la creacion de clusters de alto rendimiento basados en computación paralela usando la interfaz de paso de mensajes, según la descripción que se podía encontrar en su web, ahora inaccesible. En la Figura 2.1 podemos ver la pantalla de instalación de este sistema. Entre las razones por las que lo descartamos se encuentra el hecho de que no se siguió desarrollando el sistema y lleva desactualizado muchos años, sin intención de revertir la situación. También preferimos ubuntu por la completitud de este SO frente al anterior y porque pelicanHPC tampoco era compatible con herramientas como Docker o Kubernetes.

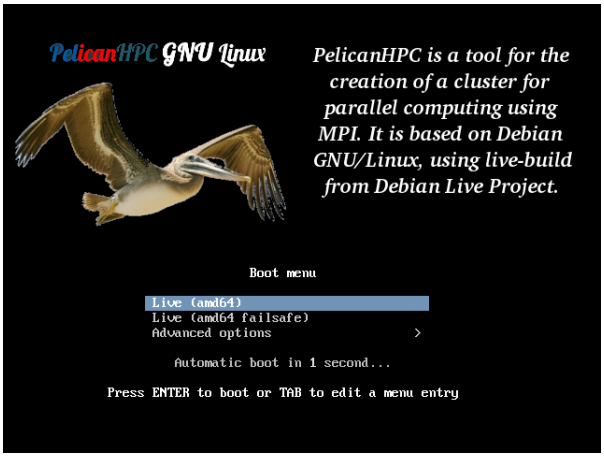


Figura 2.1: Pantalla de instalación de PelicanHPC

2.3. ¿Qué es un clúster? Modelo maestro/esclavo

El término clúster hace referencia al conjunto formado por distintos sistemas informáticos, llamados nodos, que trabajan como si fueran un único sistema y que pueden ser utilizados para una gran multitud de tareas. En nuestro caso nos limitaremos al procesamiento y almcenamiento de datos aunque tambien suelen ser utilizados para problemas en los que se requieran grandes cálculos computacionales.

2.3.1. Clasificación de clústeres

Para realizar la clasificación de los clusters, como en tantas tecnologías, es importante considerar distintos conceptos y atender a distintas características. En este contexto surgen 3 tipos principales de clústeres:

- **Clúster de alto rendimiento (HC)**: Este tipo de sistemas ejecutan tareas que requieren una gran capacidad de cálculo, generalmente muy costoso. Cuando se están realizando este tipo de tareas, los recursos del cluster son utilizados casi en exclusiva durante periodos de tiempo que pueden ser bastante largos.
- **Clúster de alta disponibilidad (HA)**: Con estos clústeres se busca dotar de disponibilidad y confiabilidad a los servicios que ofrecen. Para ello se utiliza hardware duplicado, de modo que al no tener un único punto de fallos se garantiza la constante disponibilidad del sistema. Esto se debe a que aunque se produzca una avería en un componente siempre habrá otro que pueda realizar el mismo trabajo, teniendo en cuenta que realizan la misma función. Por otra parte, también incorporan software de detección y recuperación ante fallos. Un aspecto importante a tener en cuenta en este tipo de clúster es el equilibrado de carga, fundamental para que todos los nodos trabajen por igual.
- **Clúster de alta eficiencia (HT)**: En estos sistemas el objetivo central de diseño es que se puedan ejecutar el mayor número de tareas en el menor tiempo posible, entendiendo que hablamos de tareas individuales cuyos datos no tienen dependencia entre sí.

Siguiendo la anterior clasificación podemos ver que el clúster que queremos implementar estaría incluído dentro de los clústeres de alta disponibilidad. Queremos que si alguno de los cinco equipos falla el clúster pueda seguir funcionando sin problemas. Para ser más específicos, cuando hablamos de fallo puede ser desde la desconexión de un cable o el cese del correcto funcionamiento de cualquiera de los equipos. Es importante que el propio sistema sea capaz de detectar situaciones así y que pueda mantener el servicio sin necesidad de que se produzca ninguna intervención humana.

Obviamente es imposible asegurar al 100% la disponibilidad continua del sistema, por eso también es importante que la recuperación de fallos no solo sea autosuficiente, si no que se garantice también la máxima velocidad posible.

2.3.2. Modelo maestro/esclavo

En realidad tan solo con el nombre ya se puede intuir que significa el modelo maestro/esclavo, y más aún teniendo en cuenta la relevancia o frecuencia que tiene este modelo en muchos ámbitos de la informática, sobre todo en la parte Hardware. Básicamente es un patrón de diseño en el que uno o más sistemas ordenan a otros sistemas las tareas que tienen que llevar a cabo. Los maestros generalmente mantienen una comunicación unidireccional con los esclavos, actuando como un controlador.

En nuestro clúster la idea es que haya una máquina que actúe como maestro mientras que el resto de nodos son esclavos. Ese nodo maestro será aquel que disponga de conexión a la red y que organice las tareas para el resto de nodos. Como veremos en los siguientes apartados este tipo de modelo es el más frecuente en la implementacion de clústeres mediante Kubernetes, la herramienta que más utilizaremos y que explicamos en la Sección 2.8.

2.4. Virtualización y su evolución. Contenerización y Docker

Antes de su adopción generalizada, muchos servidores operaban por debajo de su capacidad máxima. Gracias a la virtualización, es posible consolidar múltiples sistemas y aplicaciones en un solo servidor físico, mejorando en gran medida la eficiencia de los recursos. Esta consolidación no solo reduce los costos asociados con el hardware, la energía con estudios probados (Jin et al. (2012), Kommeri et al. (2012)) y el espacio del centro de datos, sino que también ofrece una flexibilidad sin precedentes. Las máquinas virtuales, aisladas entre sí, brindan una capa adicional de seguridad, y su capacidad para crearse, modificarse, moverse y eliminarse rápidamente se ha vuelto invaluable para el desarrollo y las pruebas, asegurando un entorno consistente y repetible.

Sin embargo, el desarrollo no se detiene ahí. A medida que la virtualización se enfocaba principalmente en emular hardware para ejecutar un sistema operativo completo en una sola máquina, surgió una nueva forma de virtualización: Contenerización. A diferencia de la virtualización tradicional, los contenedores funcionan virtualizando el propio sistema operativo. Esto significa que todos los contenedores en una máquina comparten el mismo sistema operativo, pero se ejecutan en un espacio aislado. Esta arquitectura más liviana permite que los contenedores tengan muchos menos gastos generales que las máquinas virtuales tradicionales, lo que resulta en tiempos de inicio más rápidos y una mejor utilización de los recursos. Además, la naturaleza encapsulada de los contenedores garantiza que las aplicaciones se ejecuten de forma consistente en cualquier entorno que admita un sistema de contenedores, como Docker. Esta portabilidad y consistencia ha llevado a la adopción de una arquitectura de microservicios, donde cada función de una aplicación se aloja en un contenedor separado. Asimismo, los contenedores facilitan los procesos continuos de integración y despliegue, lo que permite un despliegue rápido, consistente y repetible.

En la Figura 2.2 podemos observar cómo los contenedores se ahorran el uso de esa capa de virtualización y cómo operan directamente sobre el sistema operativo.

Este tipo de virtualización ha supuesto un impacto significativo en la computación paralela al facilitar y optimizar la implementación de cargas de trabajo paralelas y distribuidas. La posibilidad de crear instancias virtuales o contenedores de manera rápida y eficiente permite la distribución y escalado dinámico de aplicaciones. Es decir, cuando una aplicación y/o servicio demanda más recursos computacio-

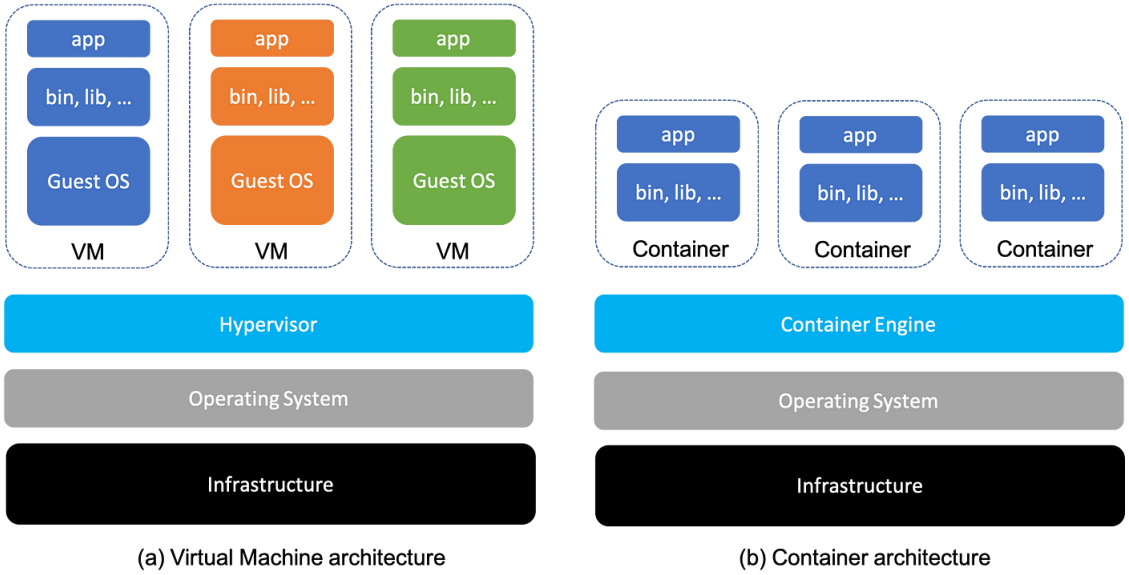


Figura 2.2: Contenerización VS Virtualización

nales, se pueden desplegar instantáneamente nuevos contenedores para manejar la demanda, y posteriormente reducir su número eliminando esos contenedores una vez finalizada la tarea.

Además, los contenedores en particular, por su ligereza y portabilidad, son especialmente útiles para implementar arquitecturas de microservicios. Un microservicio, que puede ser una tarea, proceso o componente individual en un sistema paralelo y distribuido, se puede alojar en un contenedor separado. Esto no solo asegura un aislamiento y una ejecución consistente, sino que también facilita la paralelización al permitir que estos microservicios se ejecuten simultáneamente en distintos nodos de un clúster.

Otra ventaja relevante es la flexibilidad en la selección y combinación de herramientas y frameworks. Los sistemas de computación paralela, como Apache Spark o Hadoop, de los que hablaremos también, pueden ser encapsulados dentro de contenedores, asegurando que todas las dependencias necesarias estén presentes y configuradas adecuadamente. Esto reduce las complicaciones asociadas con la configuración y garantiza que los sistemas paralelos se ejecuten de la misma manera en cualquier entorno que admita contenedores.

Podemos afirmar por tanto que la combinación de virtualización y contenedores con herramientas modernas de orquestación, como Kubernetes, permite la gestión y programación eficiente de cargas de trabajo paralelas en clústeres, optimizando el uso de recursos y garantizando altos niveles de disponibilidad y tolerancia a fallos.

2.5. Conceptos frecuentes en la memoria

Antes de continuar con la explicación de las diversas herramientas que vamos a utilizar como Docker, Kubernetes o MongoDB, vamos a aclarar ciertos términos que van a aparecer con mucha frecuencia a lo largo de la memoria.

Un término que mencionamos mucho es **YAML**. Este es un lenguaje de serialización de datos fácil de comprender y que suele ser utilizado en el diseño de archivos de configuración. Es muy parecido a JSON aunque se suele optar por YAML por su fácil legibilidad. En el contexto de Kubernetes se utiliza para definir los objetos que forman el clúster. En otras palabras, se crean los recursos de Kubernetes, como los pods, los objetivos o las implementaciones con los archivos de configuración que utilizan el lenguaje YAML.

Otro termino que mencionamos frecuente y que es inclasificable dentro de las secciones del capítulo es el **tunneling**. Éste es el proceso que vamos a seguir cada vez que queramos acceder desde fuera a algún servicio alojando dentro de la red del clúster. Desde nuestra máquina creamos un túnel en el que hay que especificar que puertos van a servir de apertura y cierre del mismo y que dirección queremos coger. Los puertos de cierre serán los puertos que habilitemos en el cluster para acceder a los servicios y los de apertura serán aquellos que desde nuestra máquina externa usaremos para conectarnos. Una vez establecemos ese túnel ya podríamos actuar como si nuestra dirección IP fuese la misma que la del clúster. De esta forma podemos acceder vía tunneling a los distintos servicios en nuestro ordenador usando localhost, como mostraremos en las distintas secciones del Capítulo 3 para cada uno de los servicios.

También comentamos mucho el término **proxy**. Un servidor proxy es una tecnología que se utiliza como puente entre cierto origen (un ordenador) y el destino de una solicitud (Internet). En el contexto de nuestro trabajo, se refiere a un componente que actúa como intermediario para redirigir y gestionar el tráfico entre los diferentes servicios y recursos dentro del clúster. El proxy se utiliza para facilitar la comunicación entre los nodos del clúster y garantizar que los servicios puedan comunicarse de manera eficiente y segura.

API Server por su parte es otro término bastante utilizado. Éste se refiere a la componente de software que actúa como intermediario entre las aplicaciones y los servicios, permitiendo que las aplicaciones se comuniquen y realicen operaciones en esos servicios de manera programática. En nuestro caso, actúa como punto de entrada y control para las interacciones entre los usuarios, las herramientas de administración y los componentes internos del clúster. Un API server funciona exponiendo una interfaz, basada en reglas como HTTPS, que define cómo las aplicaciones pueden enviar solicitudes y recibir respuestas.

2.6. Docker. ¿Qué es Docker?

Docker ha sido una de las piezas fundamentales que ha impulsado la revolución de los contenedores en la era moderna de la informática. Naciendo en este contexto de virtualización y evolución hacia la computación paralela, Docker proporciona una plataforma, que permite desarrollar, enviar y ejecutar aplicaciones dentro de contenedores.

El ecosistema de docker se compone de varias herramientas. A continuación explicaremos sus componentes y la terminología que necesitamos entender a la hora de usar Docker.

- **DockerFile**: El dockerfile es el fichero de especifica las insturcciones necesarias para crear un contenedor y que automatiza su proceso de creación. Este es definido por el desarrollador del contenedor. El proceso de construcción de una imagen a partir de un Dockerfile se conoce como **build** y se realiza mediante el comando docker build. Este proceso garantiza que las imágenes de contenedor se creen de manera consistente y reproducible, lo cual es fundamental para asegurar la uniformidad y predictibilidad de los entornos

- **Imágenes de Docker**: La imagen de Docker contiene el código fuente de la aplicación junto con sus herramientas y dependencias necesarias. Cuando ejecutas una imagen, se convierte en un contenedor. Si bien las imágenes se pueden crear desde cero, muchos desarrolladores las obtienen de los diferentes repositorios existentes en internet, como DockerHub. Estas imágenes están compuestas de diferentes capas, representando, cada una, una versión de la imagen. La capa base es el sistema operativo como puede ser el caso de Ubuntu, las capas intermedias son las diferentes fases de la construcción de la imagen definidas en el Dockerfile como instalaciones de paquetes y dependencias y de solo lectura, y la capa de contenedor, a la cual podremos acceder y escribir, servirá para añadir archivos. Esta capa, si el contenedor se elimina y no se ha guardado como una nueva imagen, se pierde y con ella todos los cambios realizados durante su ejecución, ya que los cambios se almacenan sólo durante el periodo de vida del contenedor.

- **Docker Hub**: DockerHub es el repositorio público de imágenes de Docker. Contiene miles de imágenes de todo tipo, tanto oficiales, validadas por Docker, como de publicadores verificados también por Docker como es el caso de Bitnami, del que hemos usado varias en este trabajo. Explicaremos más adelante cómo cualquier persona que se registre con un usuario en Docker puede tener su espacio público o privado en donde puede subir sus imágenes que haya desarrollado o haya modificado de otros usuarios.

- **Docker Daemon**: El Docker Daemon, o también conocido como Docker engine, es un proceso persistente que gestiona los contenedores de Docker en un sistema. El daemon de Docker se comunica con el cliente de Docker mediante la CLI o la API. A su vez, el daemon de Docker se basa en containerd (que es

un runtime de contenedores) para gestionar la ejecución de contenedores. Para comunicarse y gestionar el deamon nosotros usaremos el comando *docker*.

En la Figura 2.3 podemos observar un uso completo de Docker. Las etapas se pueden resumir en las siguientes 3 partes principales:

- **Docker build**: Construimos una imagen especificada en un Dockerfile en la máquina del desarrollador mediante el comando:

  >Docker build -t image DockerfileDir.

  Con la opción -t definimos el tag de la imagen para ponerle una etiqueta y definirla y deberemos especificar también el directorio en el que se encuentra nuestro Dockerfile.

- **Docker push**: Mediante el comando:

  >Docker push image:tag

  lo que haremos es guardar esa imagen en nuestro repositorio. El repositorio será en el que estamos loggeados ya de primeras. Para poder hacer login usaremos el comando:

  >Docker login.

- **Docker pull**: Por último, para hacer uso de la imagen con Docker lo haremos mediante el comando:

  >Docker pull image:tag.

  Sin embargo nosotros a penas usaremos este comando directamente ya que será nuestro orquestador de contenedores el encargado de manejar estos pull de las imagenes especificadas en los archivos de configuración *.yaml* de forma automática.

A continuación explicaremos cómo los contenedores tienen cabida en una arquitectura de procesamiento de datos paralelo y alto rendimiento.

2.7. Clústeres Bare Metal en Arquitecturas Basadas en Contenedores

Los avances tecnológicos en infraestructuras de servidores y computación distribuida han llevado a la adopción de clústeres como una solución estándar para abordar problemas computacionales complejos y demandas de alto rendimiento. Esta organización como hemos destacado en la Sección 2.3, tiene el potencial de proporcionar una mayor capacidad de cálculo, almacenamiento y redundancia que un único servidor.

Un clúster **bare metal** se refiere a un sistema donde los servidores operan directamente sobre el hardware físico, en lugar de a través de una capa de virtualización. En este enfoque, el nodo maestro, que coordina y gestiona el clúster, tiene un acceso sin restricciones al hardware subyacente, permitiendo una visión y control precisos sobre los recursos. Los nodos esclavos, que son los ejecutores del clúster, también se

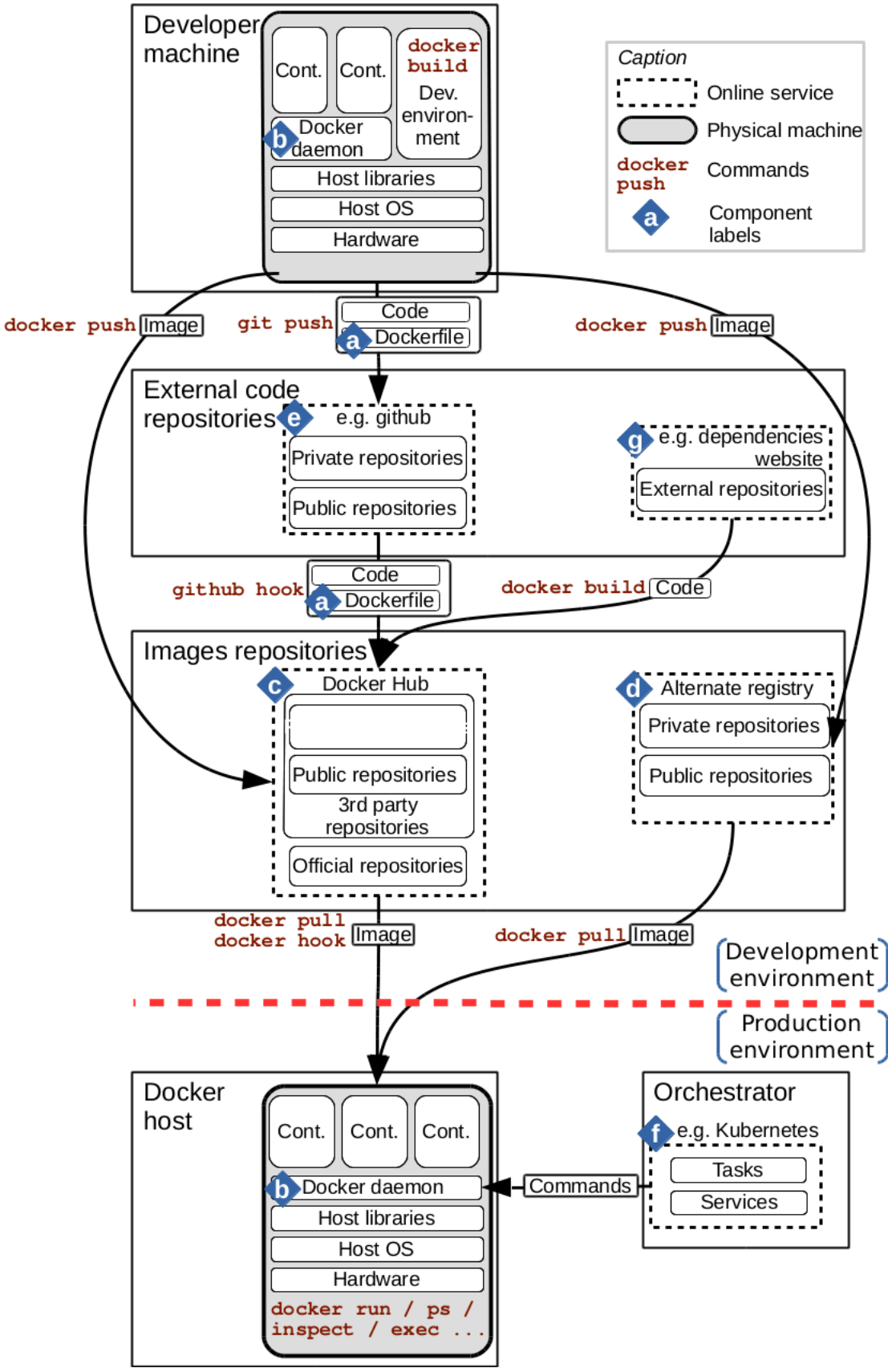


Figura 2.3: Diagrama de uso de Docker

benefician de este acceso directo al hardware, optimizando el rendimiento y minimizando la latencia. Cada tipo de nodo se encargará de una serie de tareas en relación al uso de contenedores.

- **Nodo maestro**: Son los que controlan el uso del hardware y los recursos demandados por los contenedores.
- **Nodo esclavo**: Proveen el entorno en tiempo de ejecucion para los contenedores y los recursos hardware que demandan las aplicaciones como uso de su CPU, memoria RAM y almacenamiento entre otros.

Introducir contenedores en este escenario lleva el concepto de computación distribuida a un nuevo nivel. Los contenedores, como ya hemos explicado, encapsulan un entorno de ejecución, incluyendo las aplicaciones y sus dependencias, en unidades discretas y portables. El objetivo será distribuir esta combinación de contenedores en tódos nuestros nodos del clúster, para que sean capaces de trabajar en conjunto, y además dispongan de una alta disponibilidad.

La integración de contenedores con un clúster bare metal proporciona un conjunto de ventajas:

- **Optimización del Rendimiento:** Los contenedores ya son ligeros por naturaleza. Al ejecutarlos en un entorno bare metal, se eliminan las sobrecargas asociadas con la virtualización, permitiendo que las aplicaciones en contenedores aprovechen al máximo los recursos del sistema.
- **Seguridad y Aislamiento:** Aunque los contenedores ofrecen un alto grado de aislamiento entre aplicaciones, esta característica se potencia en un clúster bare metal. Sin las capas adicionales de virtualización, el aislamiento se vuelve más robusto, reduciendo potencialmente los vectores de ataque.
- **Escalabilidad y Resilencia:** Los clústeres bare metal, combinados con contenedores, ofrecen una escalabilidad sin precedentes. Es posible escalar verticalmente, aprovechando al máximo el hardware de un nodo, o horizontalmente, añadiendo más nodos al clúster.

Al combinar la eficiencia del acceso directo al hardware con la flexibilidad y portabilidad de los contenedores permite la implementación de soluciones de alta disponibilidad y rendimiento. Sin embargo, los contenedores en un entorno como el nuestro necesitan un organizador que los orquestre y los administre en todos los nodos. Esa función de orquestración la cumplirá en nuestro caso Kubernetes.

2.8. Kubernetes

En estos últimos años, Kubernetes (Figura 2.4) se ha convertido en la principal herramieta para orquestar e implementar el uso de contenedores. Según la propia

documentación oficial, Kubernetes es una plataforma portátil, extensible y de código abierto utilizada para administrar cargas de trabajo y servicios en los contenedores, facilitando la automatización así como provisionando a nuestros servicios de una alta disponibilidad y un balanceo de las cargas de trabajo.

Figura 2.4: Logo de Kubernetes

2.8.1. Elementos necesarios para construir un cluster privado con Kubernetes

Uno de los elementos necesarios es contar con un clúster bare metal conectado en una red privada con acceso a internet así como disponer de un instalador de kubernetes como kubeadm, que aunque no es la única opción disponible, sí es la oficial y la más usada. Aunque no es estrictamente necesario, es recomendable que tanto los nodos maestro como los esclavos dispongan de más de 2GB de memoria RAM por máquina y al menos 2 CPUs actuando como maestros para que en caso de que uno falle, el cluster siga siendo operativo. En nuestro caso al contar únicamente con una IP pública solo podemos designar un nodo como maestro. A continuación vamos a definir los recursos básicos en kubernetes relacionados con el ciclo de vida de una aplicación.

2.8.2. Recursos de una aplicación

Los recursos en Kubernetes representan la columna vertebral del despliegue y operación de aplicaciones dentro del ecosistema de este sistema de orquestación. Estos recursos no solo comprenden las cargas de trabajo, como los contenedores que ejecutan el código de la aplicación, sino también configuraciones, políticas, y la infraestructura de red que facilita la comunicación y el descubrimiento de servicios.

Kubernetes opera bajo un modelo declarativo, lo que significa que los usuarios declaran el estado deseado para sus aplicaciones y es trabajo del sistema trabajar para cumplir y mantener ese estado. Esta declaración se realiza a través de objetos o recursos, que son definiciones en archivos YAML que describen las propiedades y comportamientos deseados de las entidades que constituyen una aplicación. Para poder comprender, configurar y aplicar estos estados es necesario comprender los diferentes recursos de una aplicación. Comenzaremos con los que dictan el ciclo de vida de la misma:

- **Pod**: Al contrario que muchos sístemas, Kubernetes no ejecuta los contenedores directamente. En vez de eso envuelve uno o más contenedores en una

estructura de alto nivel denominada pod. Cualquier contenedor incluido en el mismo pod compartirá los mismos recursos y la misma red local. Los contenedores se pueden comunicar de manera sencilla con otros contenedores si están en el mismo pod, como si estuviesen en la misma máquina mientras mantienen un grado de aislamiento respecto a los otros.

Los pods son utilizados como la medida de replicación en Kubernetes. Si tu aplicación se vuelve muy popular y una simple instancia del pod no puede llevar a cabo la carga, Kubernetes puede ser fácilmente configurado para desplegar nuevas réplicas del pod al cluster si es necesario. Incluso cuando la carga no es tan pesada, es típico tener múltiples copias del pod ejecutándose a la vez en un sistema de producción para permitir el balanceado de carga y la tolerancia a fallos, garantizando alto rendimiento y alta disponibilidad.

Los pods también puede mantener múltiples contenedores pero es aconsejable limitarse al menor número posible. Como los pods son escalables como una sola unidad, todos los contenedores incluidos deben escalar juntos tambien, independientemente de sus necesidades individuales. Esto conlleva un desperdicio de recursos y una considerable subida de coste. Para resolver esto los pods se deben mantener lo más pequeños posible, generalmente soportando un único proceso principal y sus contenedores auxiliares con una correlación muy estrecha.

- **ReplicaSet**: Un ReplicaSet en Kubernetes es un recurso que garantiza la ejecución de un número específico de réplicas de un pod en todo momento. En otras palabras, se asegura de que siempre haya un conjunto de pods en ejecución, incluso si algunos pods fallan o se eliminan. El ReplicaSet utiliza selectores de etiquetas para identificar los pods que caen bajo su jurisdicción. Esta relación se define usando etiquetas en los pods y un selector en el ReplicaSet, lo que permite especificar cuántos pods idénticos quieres que estén en ejecución bajo esa etiqueta Si hay demasiados, el ReplicaSet eliminará los adicionales, y si hay muy pocos, el ReplicaSet creará más.

  Un ReplicaSet, en sí, no facilita las actualizaciones a los pods. Si cambias el template del pod en un ReplicaSet, no se afectará a los pods que ya están en ejecución. Por esta razón, generalmente no trabajas directamente con ReplicaSets. En su lugar, usas Deployments.

- **Deployment**: Aunque los pods sean la unidad básica de computación en Kubernetes, no suelen ser directamente lanzados sobre el clúster. En vez de eso los pods son generalmente manejados por una capa más de abstracción: el Deployment o Despliegue.

  Un Deployment en Kubernetes es un recurso de alto nivel que gestiona la implementación y actualización de instancias de aplicaciones contenidas en pods. Es uno de los recursos más comúnmente utilizados en Kubernetes y proporciona características que facilitan el despliegue y escalabilidad de aplicaciones, así como la actualización y el rollback de versiones.

  El principal objetivo de los Deployments es declarar cuantas réplicas de un pod deberían estar ejecutándose a la vez, basándose en el uso de ReplicaSets

internamente. Cuando un despliegue es añadido al clúster, automáticamente modificará el numero de pods requeridos y los monitorizará. Cuando un pod muere el propio despliegue volverá a crearlo de manera automática.

Si bien un Deployment gestiona ReplicaSets y estos a su vez gestionan los pods, no es necesario trabajar directamente con ReplicaSets cuando se usa un Deployment. Los Deployments ofrecen una capa de abstracción que simplifica el proceso de gestión de la disponibilidad y actualización de las aplicaciones.

Supongamos que tenemos una aplicación versión 1.0 en ejecución en un clúster de Kubernetes controlada por un Deployment. Después de algún tiempo, decidimos implementar la versión 2.0 de esa aplicación

Haciendo uso de un Deployment, bastaría con actualizar especificación del pod con la nueva imagen de la versión 2.0. El Deployment creará automáticamente un nuevo ReplicaSet para la versión 2.0 y comienzará a dirigir el tráfico hacia los nuevos pods a medida que se despliegan lo que reduce el tráfico de los pods de la versión 1.0 hasta que todos sean reemplazados.

Si algo va mal con la versión 2.0, podemos hacer uso del el comando de rollback de Kubernetes para volver a la versión 1.0, minimizando así el tiempo de inactividad y el impacto en los usuarios. Comandos como este que hemos mencionado se pueden encontrar en el anéxo A.

- **StatefulSet**: Mientras que muchos recursos en Kubernetes, como Deployments, se diseñaron pensando en aplicaciones sin estado, hay casos en los que se necesitan garantías sobre la identidad y el orden de despliegue y escalado de pods. Aquí es donde entra el StatefulSet. Es una herramienta destinada a aplicaciones con estado, como bases de datos, donde los datos en un pod deben persistir a lo largo de reinicios y reprogramaciones, y donde la identidad y el orden son críticos.

  Un StatefulSet garantiza que los pods se creen con un nombre único y en un orden específico. Además, si un clúster tiene almacenamiento persistente disponible, el StatefulSet puede vincular volúmenes persistentes a pods específicos, garantizando que el almacenamiento también se mantenga a lo largo de reinicios. Por ejemplo, si estamos ejecutando una base de datos en tres nodos, el StatefulSet garantizaría que, incluso si un pod muere y es recreado, el nuevo pod mantendrá los mismos datos y configuración del anterior, permitiendo una recuperación más sencilla y coherente. Este tipo de recurso se usará más adelante en el Deployment de MongoDB.

- **DaemonSet** Mientras que los Deployments y StatefulSets garantizan que un número específico de pods se ejecute en todo el clúster, hay situaciones en las que es necesario asegurarse de que un pod se ejecute en cada nodo del clúster o en un subconjunto específico de nodos. Esto es lo que hace el DaemonSet.

  Imaginamenos que tenemos una herramienta de monitorización o un agente de registro que debe ejecutarse en cada nodo del clúster para recopilar métricas o logs. No sería eficiente ni práctico intentar hacer esto con un Deployment, ya que tendrías que ajustar manualmente el número de réplicas cada vez que

el tamaño del clúster cambie. En su lugar, puedes usar un DaemonSet, que automáticamente desplegará un pod en cada nodo del clúster. Además, cuando agregues nuevos nodos al clúster, el DaemonSet se asegurará de que este pod se despliegue en los nuevos nodos también. Es una forma de garantizar que ciertos pods se ejecuten en todos o en algunos nodos específicos, sin importar el tamaño o la topología del clúster. Este tipo de recurso se usa en gran medida por los operadores de los que hablaremos más adelante.

2.8.3. Componentes de Kubernetes

En esta sección vamos a estudiar todos los componentes que forman parte de la arquitectura que utiliza Kubernetes. Para empezar vamos a ver los distintos componentes que forman un nodo, según si el nodo es el maestro, también llamado Control Plane, o un nodo worker o slave.

Componentetes de un Control Plane (Maestro):

- **Kube-apiserver**: El kube-apiserver es esencialmente la puerta de entrada a la API de Kubernetes. Cada operación en el clúster pasa a través de este componente. Su capacidad para escalar horizontalmente, es vital para grandes clústeres. Además, implementa sofisticados mecanismos de autenticación y autorización, de los que hablaremos más adelante, garantizando que sólo las solicitudes permitidas puedan realizar cambios en el clúster. Antes de que cualquier recurso se escriba en la base de datos, el apiserver procesa *controladores de admisión*, validando y/o modificando los recursos según sea necesario.

- **Etcd**: Es una base de datos de clave-valor distribuida que almacena tanto la configuración como el estado actual del clúster. Uno de sus aspectos más cruciales es su garantía de consistencia. Esta característica asegura que el estado del clúster sea coherente en todos los nodos. La alta disponibilidad es otro de sus puntos fuertes, con la capacidad de replicar datos a través de múltiples nodos. Además, las funciones de snapshot y backup son esenciales para la recuperación en caso de fallos.

- **Kube-scheduler**: Cuando se crea un pod en Kubernetes, no se lanza al azar en cualquier nodo. Aquí es donde entra en juego el kube-scheduler. Este componente observa los pods que aún no están asignados a un nodo y toma una decisión sobre dónde deberían ejecutarse. Utiliza una serie de algoritmos y criterios, como los requisitos del pod, las capacidades del nodo y diversas restricciones. También se puede personalizar y extender su comportamiento, permitiendo a los administradores afinar cómo y dónde se despliegan los recursos.

- **Kube-controller-manager**: Finalmente está el kube-controller-manager. Aunque puede no ser tan visible como algunos de los otros componentes, su papel es fundamental. Ejecuta los controladores, que son esencialmente bucles que garantizan que el clúster se mantenga en el estado deseado. Desde asegurarse

de que haya el número correcto de instancias de un pod, hasta gestionar las conexiones entre servicios y pods, y administrar cuentas y tokens de acceso, este componente es la fuerza motriz detrás de muchas operaciones en Kubernetes.

Pasaremos ahora a enumerar y definir los componentes de un nodo worker:

- **Kubelet**: Un agente que se ejecuta en cada nodo del clúster. Es el encargado de asegurarse que los contenedores están siendo ejectutados en un pod. Este agente recibe un conjunto de especificaciones para los contenedores, que es especificado por el API de control a través de peticiones y que se asegura que los contenedores descritos en esas especificaciones están siendo ejecutados correctamente. Kubelet no actúa sobre contenedores que no hayan sido creado a través de Kubernetes.
- **Kube-proxy**: Es un proxy de red que se ejecuta en cada nodo del clúster, implementado parte del concepto de Servicio Kubernetes. Este componente mantiene las reglas de red sobre los nodos. Dichas reglas permiten la comunicación entre el clúster y los contenedores, a través de una red privada virtual o también llamada VPC (Virtual Private Cloud) en los proveedores cloud.
- **Container runtime**: Se encarga de las operaciones de bajo nivel para ejecutar contenedores, como la gestión de imágenes, la asignación de recursos y la interacción con el sistema operativo subyacente. Inicialmente únicamente era posible usar Docker como runtime. Con el tiempo fueron apareciendo nuevos runtimes de contenedores, incluidos Runc, containerd, Cri-o y otros, más ligeros que docker lo que supuso un gran mejor en el benchmarking del orquestador.

Para comunicarnos con el API server en nuestro clúster operaremos a través del Control Plane mediante el uso del cliente Kubectl.

Una vez definidos los recursos que definen el ciclo de vida de las aplicaciones y las componentes principales de Kubernetes vamos a explicar cómo conseguimos que esas aplicaciones y los datos que almacenan sobrevivan a fallos y reinicios.

2.8.4. Almacenamiento persistente. PersistentVolume y PersistentVolumeClaim

Los pods son efímeros y con ellos toda la información que contienen. Para poder mantener esta información entre distintas ejecuciones del mismo pod así como compartir información entre ellos debemos hacer uso del almacenamiento persistente. Kubernetes nos permite implementar dicho almacenamiento persistente utilizando PersistentVolume y PersistentVolumeClaim.

El primero representa el recurso de almacenamiento creado mediante clases de almacenamiento mientras el segundo hace alusión a las solucitudes de almacenamiento de un usuario, algo así como la capa de abstracción entre el pod y el PersistentVolume. Kubernetes tiene dos formas de provisionar los volúmenes persistentes: de

forma estática, en la que el adminisitrador especifica el número de volúmenes y los crea, y de forma dinámica, en la que ningún Persistent Volume Claim encaja con los volúmenes y entonces el cluster trata de forma dinámica provisionar un volúmen que cumpla con los requisitos de la solicitud, siguiendo su propia clase de almacenamiento. Este modelo de aprovisionamiento dinámico de los volúmenes lo gestionará Rook-Ceph el cual será introducido también en la Sección 2.9.

Ahora imaginemos que este tipo de almacenamiento, el cual viene descrito en el fichero de configuración del deployment de una aplicación, lo quisiéramos usar en más deployments sin necesidad de andar copiando los valores del fichero de un lado a otro. Puede ser también que necesitáramos hacer uso de una misma clave de acceso para varias aplicaciones. Ahí es donde entran en juego los ConfigMaps y los Secrets.

2.8.5. Configuración de las aplicaciones. ConfigMap y Secret

Gestionar configuraciones y datos sensibles es de vital importancia. Kubernetes, en su búsqueda por simplificar y estandarizar la administración de contenedores, introduce dos recursos poderosos y flexibles para preservar y facilitar el uso de estos: ConfigMaps y Secrets.

- **ConfigMap**: La configurabilidad y adaptabilidad son piedras angulares en el desarrollo de aplicaciones actuales. Es habitual que una misma aplicación requiera distintos parámetros dependiendo del entorno en el que se despliegue. Aquí es donde los ConfigMaps entran en juego. Estos recursos permiten a los desarrolladores y operadores separar la configuración del código, una práctica esencial para la integración y la entrega continua (CI/CD). Los ConfigMaps representan una forma eficaz de administrar configuraciones dinámicas sin tener que reconstruir las imágenes de contenedor, adaptando la aplicación a diferentes circunstancias con facilidad y eficiencia.

- **Secret**: En contraposición a la información general que almacenan los ConfigMaps, hay datos que requieren de un cuidado especial debido a su naturaleza sensible. Estamos hablando de credenciales, claves API, certificados y demás información que, si se expone, podría comprometer la seguridad y la integridad de una aplicación o sistema. Secrets son la respuesta de Kubernetes a este dilema. Estos recursos están diseñados para proteger la información crítica, garantizando que solo las entidades autorizadas tengan acceso a ella, y siempre de manera segura. Los Secrets subrayan la importancia de adoptar prácticas de seguridad sólidas en el ecosistema de contenedores.

Ahora una vez que tenemos nuestra aplicación ejecutándose varios los pods, con un cierto grado de replicación y escalabilidad y todos ellos bajo la supervisión de un Deployment ya configurado, de alguna forma necesitaremos exponer este conjunto de pods a los usuarios a traves de algún punto. La forma de exponerlos en Kubernetes será mediante un recurso llamado Service.

2.8.6. Cómo exponer una aplicación. Service

Una de las cosas que nos planteamos al usar Kubernetes era como podíamos exponer nuestros pods ya fuera para que las distintas aplicaciones dentro de nuestro cluster se integraran entre sí como es el caso de la monitorización de aplicaciones, o para permitir el acceso a usuarios desde distintos clientes a esas aplicaciones. Para lograr eso debemos conseguirlo mediante la encapsulación de los pods de una aplicación en un recurso denominado Service.

Un Service en Kubernetes es un recurso que actúa como un punto de acceso y un balanceador de carga para un conjunto de pods. Ofrece una dirección IP estable y un nombre DNS para que otros pods o sistemas puedan comunicarse con los pods subyacentes, sin tener que preocuparse si alguno de esos pods se va, viene o cambia su IP. Básicamente, un Service asegura que el tráfico sea dirigido a un pod funcional, independientemente de los cambios que puedan surgir en el entorno.

En este contexto Kubernetes nos brinda 4 opciones interesantes como son ClusterIP, NodePorts, LoadBalancers e Ingress. Todas estas son formas distintas de obtener tráfico externo en nuestro clúster, y todas lo hacen de diferentes maneras.

- **ClusterIP**: ClusterIP es el servicio por defecto de Kubernetes. Se encarga de proporcionar una IP estable para un grupo de Pods, que no es visible desde el exterior del Cluster. Para solventar este inconveniente necesitamos usar un Proxy. Ejecutando el siguiente comando instanciamos uno:

  ```
  > kubectl proxy --port=8080
  ```

  Cuando el servidor proxy está corriendo ya se podría explorar la API de Kubernetes usando curl, wget o cualquier navegador.

  El archivo de configuración YAML tendría la siguiente estructura:

  ```yaml
  apiVersion: v1
  kind: Service
  metadata:
    name: cluster-ip
  spec:
    type: ClusterIp
    ports:
        name : http
        targetPort: 80
        port: 80
    selector:
        app: ejemplo
  ```

  De este archivo cabe mencionar la diferencia entre targetPort, que es el puerto que está usando la propia aplicación dentro del pod, y port, que es el puerto utilizado para exponer el servicio.

- **NodePort**: Usando NodePort la situación cambia con respecto a ClusterIP. Al crear un servicio NodePort, hacemos que Kubernetes reserve el mismo puerto

en todos sus nodos y dirige todas las conexiones entrantes a los pods que sean parte del servicio.

El archivo de configuración YAML tiene la siguiente estructura:

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport
spec:
  type: NodePort
  ports:
      name : http
      targetPort: 80
      port: 80
      nodeport: 30001
  selector:
      app: ejemplo
```

Algunas de las limitaciones que tiene este servicio es el restrictivo uso de puertos, ya que solo se pueden utilizar los que estén comprendidos en el rango 30000 - 327267. Pese a esto, junto a clusterIP serán las dos maneras principales que usaremos para exponer los servicios en nuestro clúster.

La mayor diferencia con clusterIP reside en el nodePort, que nos permite concretar que puerto del clúster queremos exponer al exterior, algo que no permitía ClusterIP

- **LoadBalancer**: La principal ventaja de este servicio es que automáticamente se aprovisiona un balanceador de carga externo al clúster, que nos proporciona una dirección IP propia por la cuál podremos acceder al resto de servicios que tengamos. Por cada servicio que creamos se aprovisiona un balanceador de carga diferente. Cuando lo eliminamos, el balanceador de carga que tiene asociado también se elimina.

  Es importante que no se use en entornos como Minikube, una distribución reducida de Kubernetes, ya que no dan soporte a LoadBalancer y entonces este servicio actuaría como un NodePort.

  El archivo de configuración YAML es el siguiente:

```
apiVersion: v1
kind: Service
metadata:
  name: loadbalancer
spec:
  type: LoadBalancer
  ports:
      name : http
      targetPort: 80
      port: 80
  selector:
      app: ejemplo
```

- **Ingress**: Este servicio es uno de los más importantes y más utilizados de todo el contexto que abarca Kubernetes. Básicamente se usa para exponer

rutas HTTP y HTTPS desde el exterior hasta el clúster. El tráfico entrante lo controlamos mediante un archivo de configuración que definimos.

En este trabajo el Ingress solo sería útil para el uso uso del Dashboard, ya que para servicios como MongoDB o Spark no es recomendable su uso. MongoDB por ejemplo no suele ser expuesto directamente a través de un Ingress en un entorno de producción. En su lugar, MongoDB se accede normalmente a través de conexiones seguras y autenticadas, y es común limitar el acceso solo a las aplicaciones que necesitan interactuar con la base de datos.

La principal diferencia que presenta respecto a LoadBalancer es que cada balanceador de carga necesita su propia IP mientras que con Ingress solo es necesaria una.

Este servicio funciona de forma diferente ya que con los otros bastaba con crear un objeto Kubernetes de ese tipo de servicio, mientras que con Ingress necesitamos un controlador. Entiéndase como tal un Pod o conjunto de Pods que se ejecutan en nuestro Cluster y cuya función es asegurarse de que el tráfico entrante se administra del modo que nosotros especifiquemos. Kubernetes no tiene un controlador por defecto así que la elección del mismo depende de nosotros. En nuestro caso hemos optado por el controlador de Nginx, que es el controlador del cual hemos encontrado mayor información.

Para el uso correcto de Ingress, además del controlador necesitamos lo que se conoce como Configuración Ingress, que es un objeto de Kubernetes que se usa para describir dónde dirigir el tráfico entrante. Además también se tiene que cifrar este servicio, aportando una conexión segura para que pueda funcionar.

El archivo de Configuración Ingress, que sigue la estructura YAML tiene la siguiente estructura:

```
apiVersion: extensions/vlbetal
kind: Ingress
metadata:
  name: ingress
spec:
  rules: NodePort
    host: ejemplo.com
    http:
      paths : 80
        path : \
        backend :
          serviceName : servicio
          servicePort : 80
```

En las distintas secciones del Capítulo 3 mostramos como hemos implementado los distintos servicios, incluyendo los motivos de su selección. En resumen, Ingress hemos decidido no utilizarlo finalmente debido a la complejidad de su implementación, en vistas a posibles réplicas, y a la necesidad de escalar y de facilitar los accesos a los servicios a los usuarios.

Una vez ya definidos los conjuntos de pod como servicios y estos configurados mediante ConfigMaps y Secretos, llegará un punto en el cual se nos irán acumulando todos estos recursos de manera que será incómodo trabajar con todos ellos en un mismo espacio y será de relativa dificultad administrar el acceso a estos recursos y designar quién tendrá derecho a modificarlos y quién no. Los Namespaces y RBAC serán nuestros aliados en esta tarea.

2.8.7. Organización y Accesos en Kubernetes: Namespaces y RBAC

En una plataforma tan compleja y poderosa como Kubernetes, mantener el orden y asegurar un acceso adecuado a los recursos es esencial. Kubernetes provee múltiples herramientas y prácticas para gestionar la organización y los accesos, entre las más destacadas están los Namespaces y el Control de Acceso Basado en Roles (RBAC).

- **Namespace:** En Kubernetes los namespaces sirven para separar los recursos contenidos en un clúster. Permite crear distintos entornos lógicos dentro de uno para poder gestionar todos los recursos de forma más sencilla y eficiente. Por defecto en Kubernetes se crea un espacio de nombres al que nos referiremos como "Default". Para instalar distintos servicios y recursos sobre el cluster utilizaremos namespaces distintos para intentar encapsular cada recurso y así poder mostrar de forma más ordenada los procesos que hemos seguido. Esta separación podría ser útil en varios escenarios, tales como: entornos múltiples en un mismo clúster con un Namespace para desarrollo, otro para pruebas y uno más para producción; limitarla cantidad de recursos que un Namespace puede usar o si estamos en un caso que requiere diferentes equipos y si varios clientes usan el mismo clúster, cada uno podría tener su propio Namespace.

Control de Acceso Basado en Roles denominado comúnmente con sus siglas; RBAC, permite a los administradores definir políticas de acceso a los recursos del clúster en función de roles. Es una forma eficaz de asignar y limitar lo que los usuarios y procesos pueden hacer en el clúster.

Los componentes principales del RBAC en Kubernetes son:

- **Role y ClusterRole**: Un Role define reglas para cómo se pueden interactuar con ciertos recursos dentro de un Namespace. Por otro lado, un ClusterRole es una versión de un Role que, como su nombre indica, se aplica a nivel de clúster. Esto significa que puede ser utilizado para conceder acceso a recursos en todos los Namespaces o a recursos que son a nivel de clúster (como nodos).

- **RoleBinding y ClusterRoleBinding**: Un RoleBinding otorga las permisiones definidas en un Role a uno o más usuarios. Opera dentro del contexto de un Namespace. Un ClusterRoleBinding, por otro lado, otorga las permisiones definidas en un ClusterRole a uno o más usuarios a nivel de clúster.

Algunos roles son comunes ya predefinidos en muchos clusters de Kubernetes, incluido el nuestro, son:

- **Admin**: Tiene acceso total a todos los recursos dentro de un Namespace.
- **Edit**: Puede crear, modificar y eliminar la mayoría de los recursos en un Namespace, pero no puede modificar el acceso.
- **View**: Puede visualizar la mayoría de los recursos, pero no puede modificarlos.

La combinación de Namespaces y RBAC permite a los administradores de Kubernetes organizar sus recursos de manera efectiva y garantizar que los usuarios y aplicaciones tengan el acceso adecuado a esos recursos. Esto es crucial para mantener la seguridad, la eficiencia y la gestión efectiva de un clúster de Kubernetes.

¿Y si hubiera una forma de empaquetar y automatizar el despliegue y la administración de cada aplicación en un solo recurso? Pues sí, la hay, y se llama operador.

2.8.8. Operadores. Automatización de Recursos

En Kubernetes, un Operador es una metodología diseñada para empaquetar, implementar y administrar aplicaciones Kubernetes de manera nativa. Se trata de una extensión de la lógica de la plataforma, encarnando conocimiento de dominio específico de una aplicación en software. En esencia, un operador es como un controlador personalizado que contiene la lógica operativa específica de una aplicación.

Los Operadores se basan principalmente en el concepto de Custom Resources y Custom Resource Definitions, permitiendo definir y gestionar recursos que extienden la funcionalidad del API de Kubernetes. Pero, ¿Qué son estos recursos?

- **Custom Resource (CR)**: Un Custom Resource (Recurso Personalizado) es una extensión del API de Kubernetes que permite introducir una nueva kind (tipo) de recurso para su clúster, sin necesidad de añadir un servidor API completo. Esencialmente, un CR es una instancia de un Custom Resource Definition (CRD).

  Por ejemplo, si tiene un CRD que define un tipo Database, entonces puede crear varios CRs que describan diferentes bases de datos específicas, basándose en esa definición.

- **Custom Resource Definition (CRD)**: El Custom Resource Definition (CRD) es una configuración que define un nuevo tipo de recurso que se desea añadir al API de Kubernetes. Es una extensión del API existente que permite definir nuevas kinds sin tener que modificar o volver a compilar el servidor API de Kubernetes. Una vez que se ha creado y registrado un CRD en un clúster de Kubernetes, los usuarios pueden crear instancias de este recurso con un comportamiento y especificaciones definidos, similar a cómo crearían un Pod o un Deployment.

- **Custom Controller (CC)**: El Custom Controller (Controlador Personalizado) es una lógica de control que observa y actúa sobre el estado de los recursos personalizados en Kubernetes. En conjunto con los CRDs, el CC forma la base fundamental de la estructura de los Operadores. Mientras que el CRD define el recurso personalizado, el CC define cómo debería comportarse ese recurso en diferentes situaciones o eventos. Por ejemplo, si un CR describe una base de datos y esa base de datos falla, el controlador correspondiente podría manejar la recuperación automáticamente.

En conjunto, el patrón CRD/CC permite a los desarrolladores y administradores de sistemas codificar y automatizar operaciones específicas de la aplicación en el clúster, llevando la *infraestructura como código* al siguiente nivel. Esencialmente, el Operador es la suma de un CRD y su controlador personalizado, proporcionando una gestión automatizada y especializada de recursos específicos.

2.9. Almacenamiento distribuido para Kubernetes. Rook-Ceph Storage

Como hemos mencionado antes, Rook-Ceph es una solución integrada que combina las capacidades de orquestación de Rook con el potente sistema de almacenamiento distribuido de Ceph. Este conjunto proporciona una plataforma de almacenamiento en la nube nativa, altamente escalable y resistente que se integra y gestiona fácilmente dentro de entornos Kubernetes.

Rook es un orquestador de almacenamiento basado en la nube que facilita la integración del robusto sistema de almacenamiento Ceph dentro de entornos orientados a la nube, particularmente en infraestructuras Kubernetes.

Ceph, por su parte, es un sistema de almacenamiento distribuido y escalable que brinda soluciones para almacenamiento de ficheros (CephFS), bloques (RBD) y objetos (RGW), y se despliega para maximizar la redundancia y la disponibilidad. Uno de los prerrequisitos esenciales para el correcto funcionamiento de Ceph es la disponibilidad de particiones libres o discos no formateados, lo que significa que deben estar exentos de cualquier sistema de archivos montado, para ser completamente administrados por Ceph.

La magia de Rook radica en su capacidad para simplificar y automatizar el despliegue, la gestión y el escalado de sistemas Ceph. Utilizando el operador de Rook, se consigue una gestión integrada y cohesiva de Ceph dentro de Kubernetes. Este operador gestiona el ciclo de vida de Ceph, incluyendo la configuración, el despliegue, la provisión, el escalado y la monitorización. De esta manera, Rook transforma la complejidad inherente de Ceph en una experiencia de usuario simplificada y coherente.

Es esencial mencionar el concepto de las clases de almacenamiento en Kubernetes, donde Rook desempeña un papel crucial. Las clases de almacenamiento, o StorageClasses, permiten definir cómo se debe provisionar el almacenamiento. Con

Rook, es posible definir clases específicas que se alineen con los diferentes servicios de almacenamiento que Ceph ofrece, ya sea almacenamiento de bloques, de objetos o de ficheros. Estas clases facilitan el despliegue dinámico de volúmenes persistentes basados en las necesidades específicas de cada aplicación o servicio dentro de la infraestructura Kubernetes.

Cuando un usuario crea un PersistentVolumeClaim que hace referencia a una StorageClass específica de Rook-Ceph, Rook trabajará con Ceph para aprovisionar dinámicamente un volumen que cumpla con las especificaciones de esa clase. Una vez aprovisionado, el volumen se puede montar en un pod y utilizarse como cualquier otro volumen en Kubernetes.

Nosotros hemos considerado hacer uso de estas dos clases de almacenamiento: CephFS, un sistema de archivos distribuido, o Ceph Block Storage, conocido como RBD (Rados Block Device).

CephFS, se distingue por ser un filesystem distribuido, permitiendo su montaje en múltiples pods simultáneamente. Esta capacidad resulta ser una ventaja competitiva para aquellas aplicaciones que requieren un acceso colaborativo a archivos entre diferentes entidades. Además, su estructura jerárquica de directorios y archivos se alinea con aplicaciones que dependen de tal estructura, como es el caso de aplicaciones web con almacenamiento compartido o sistemas de gestión de contenido.

Por otro lado, Ceph Block Storage, o RBD, suele ser preferido en escenarios donde el rendimiento es una prioridad. La posibilidad de un mejor desempeño en comparación con CephFS, particularmente en patrones de acceso específicos, lo hace ideal para aplicaciones de alta demanda de I/O. Es pertinente señalar que los dispositivos de bloque RBD están conceptualizados para ser montados por un único pod en un momento dado, lo que se traduce en un acceso exclusivo al almacenamiento. Esta exclusividad es esencial para aplicaciones como bases de datos, donde la integridad y el rendimiento del dato son cruciales.

Deberemos elegir con criterio la clase de almacenamiento deberá tener así como evaluar los requisitos y los distintos factores antes de tomar una decisión. Los patrones de acceso, ya sean operaciones predominantemente de lectura o escritura, acceso aleatorio versus secuencial, y las necesidades de concurrencia, son todos aspectos fundamentales a considerar. La semántica de los datos también juega un papel crucial; mientras que algunas aplicaciones se benefician de una estructura de directorio, otras podrían necesitar las características de un almacenamiento de bloque puro.

En conclusión, Rook no solo actúa como puente entre Kubernetes y Ceph, sino que hace que Ceph sea más accesible y manejable, incluso para aquellos que no están familiarizados con sus intricados detalles técnicos. La combinación de Rook y Ceph brinda una solución de almacenamiento robusta, escalable y altamente disponible para entornos como el nuestro.

En la Sección 3.3.3 contamos en detalle la implementación de este sistema de almacenamiento persistente y distribuido.

2.10. Gestor de paquetes en Kubernetes. Helm

Con todos estos recursos definidos hay veces que el despliegue de toda esta arquitectura, tanto desde la creación del namespace de una aplicación, hasta la generación de roles, definición de configuraciones, deployments, etc; se puede hacer muy pesado y complejo. Es por ello que la comunidad optó por el desarrollo de un gestor de paquetes de aplicaciones y su posterior despliegue para Kubernetes, llamado Helm.

Helm es una herramienta utilizada para gestionar los paquetes de Kubernetes. La principal función de Helm es definir, instalar y actualizar aplicaciones complejas de Kubernetes. Helm trabaja principalmente con charts, que son una colección de archivos que describen un conjunto relacionado de recursos de Kubernetes.

Para entender verdaderamente que es helm dentro de Kubernetes se podría comparar con apt dentro del sistema operativo Linux.

Los helm charts, que se organizan como una serie de archivos en el interior de un directorio, resultan realmente útil para el administrador de Helm ya que son un recurso con gran facilidad para crear, compartir, publicar y versionar dentro del sistema de Helm. Dentro de las características de un helm chart, la más relevante para nosotros es que contienen la totalidad de las definiciones de recursos que se necesitan para la ejecución de una aplicación, herramienta o servicio determinado en el interior de un clúster del sistema de Kubernetes. Por ejemplo, en nuestro caso usaremos Helm para instalar el controlador de Ingress, aunque finalmente no lo utilizamos, o el Dashboard de grafana, que comentamos en la Sección 2.13. De nuevo, en el apéndice A se pueden encontrar algunos de los comandos fundamentales para utilizar Helm.

2.11. MongoDB

El primer servicio que queremos incluir en nuestro clúster es MongoDB (Figura 2.5). Éste se basa en una arquitectura escalable que se ha vuelto muy popular entre los desarrolladores para el desarrollo de aplicaciones con datos en constante evolución.

Como base de datos no relacional, MongoDB facilita a los desarrolladores poder guardar datos (tanto estructurados como no estructurados), usando un formato similar a JSON para almacenar los distintos documentos.

Figura 2.5: Logo de MongoDB

2.11.1. MongoDB Atlas

En pocas palabras, MongoDB Atlas es un servicio de Cloud Database que nos permite crear y administrar una base de datos de Mondo a través de su plataforma. Uno de los beneficios de este servicio es que permite escalar horizontal y verticalmente las bases de datos de MongoDB de forma sencilla y automatizada.

2.11.2. MongoDB Compass

Compass es una interfaz gráfica de usuario utilizada para interactuar con bases de datos MongoDB de manera intuitiva y eficiente. Esta herramienta está diseñada para facilitar la administración, consulta y visualización de los datos de bases de datos no relacionales de MongoDB.

Algunas de las funcionalidades que ofrece son: permitir a los usuarios explorar y visualizar los datos almacenados en sus bases de datos, facilitar la creación y ejecución de consultas, ayudar en la creación y administración de índices para mejorar el rendimiento de las consultas, o permitir importar y exportar datos desde y hacia bases de datos MongoDB facilitando la migración y la gestión de datos, entre otros.

2.11.3. Sharding. Beneficios y funcionamiento

Sharding, que en español significa fragmentación, es un método para distribuir o partir el almacenamiento de datos entre distintas máquinas. Siguiendo esto podemos conseguir distribuir el almacenamiento entre varios nodos. MongoDB ofrece una arquitectura basada en clúster que sigue este método. Esta arquitectura contiene tres elementos principales:

- **Shards**: Estos son un conjunto de réplicas que contienen un subconjunto de los datos contenidos en el clúster.
- **Mongos**: Mongos actúa como un router de peticiones y consultas para las aplicaciones cliente, realizando tanto operaciones de lectura como de escrituta. Delege las peticiones de los clientes a los shards (fragmentos en español) adecuados y recoge los resultados dentro de una respuesta consistente. Los clientes se conectan a mongos, no a los fragmentos individuales.
- **Config servers**: Estos son la fuente autoritativa de los metadatos de la fragmentación. Estos metadatos reflejan el estado y la organización del almacenamiento fragmentado, conteniendo también una lista de todas las fragmentaciones, información del enrutamiento, entre otros.

El principial beneficio de la fragmentación es la posibilidad que nos brinda de aumentar la escabilidad de nuestra base de datos manejando cargas de datos en constante crecimiento. Un factor importante es la mejora en el rendimiento de operaciones de lectura/escritura. Mediante el paralelismo de datos se obtiene esto. Si

un shard puede albergar 500 operaciones por segundo, cuantos más shards tenga nuestra arquitectura, más operaciones se podrán realizar por segundo. Algo idéntico pasa con el almacenamiento, si un propio fragmento puede albergar 2 TB de datos, cuantos más shards contengan más capacidad de almacenamiento. Otro beneficio que resulta interesante, aunque a nosotros no nos produce ningun beneficio extra, es la fragmentación por zonas. Esto hace alusión a la creación de bases de datos distribuidas sobre aplicaciones geográficamente dispersas. En otras palabras, puedes dividir los fragmentos por zonas (Europa, América del Norte, etc.) y que cada fragmento cumpla las reglas y leyes de la zona en la que se encuentra.

Para saber cómo funciona la distribución de los datos es muy importante conocer lo que son las shard keys. Estas están basadas en los campos dentro de cada documento. Los valores incluidos en dichos campos son los que deciden en qué fragmento va a residir el documento, en acorde a los rangos. Esta información es almacenada en el config server del conjunto de réplicas. MongoDB divide los datos entre chunks (trozos), dividiendo el lapso de los valores de las shard keys entre rangos no superpuestos. Luego MongoDB intenta distribuir esos trozos de forma pareja entre los distintos fragmentos del clúster. Es importante destacar la relevancia que tiene la elección de las shard keys, ya que tienen un impacto directo en el rendimiento del clúster y puede producir que funcione de forma subóptima, debido a la distribución desigual de trozos.

Hay dos principales estrategias en la fragmentación:

- **Por rangos**: Divide los datos entre rangos basados en los valores de la shard key. Cada trozo es asignado un rango basado en esos mismos valores.
- **Por trozos**: Implica calcular un valor de hash de los valores de campo de la shard key. Luego cada trozo es asignado un rango basado en el hash de la shard key.

2.11.4. MongoDB Kubernetes Operator

Para la implementación de MongoDB sobre Kubernetes existe este operador que facilita el despliegue de este servicio sobre un clúster de Kubernetes. Entre las funciones que facilita se encuentran la creación de conjuntos de réplicas y su escabilidad, asegurar conexiones clientes servidor o viceversa con tls, o usar cualquier imagen de Docker de MongoDB, entre otros. Este operador sin embargo en su versión *community* no cuenta con soporte para desplegar un sharded cluster por lo que desestimamos su uso, ya que nuestro principal objetivo es poder proporcionar almacenamiento distribuido dentro del clúster.

2.12. Apache Spark

En Apache Spark o símplemente Spark como nos referimos a él en este trabajo, es un motor de procesamiento distribuido unificado utilizado para procesar un gran

volumen de datos y que integra diversos módulos para SQL, streaming o aprendizaje automático, entre otros. Spark es responsable de orquestar, distribuir y monitorizar aplicaciones que constan de múltiples tareas de procesamiento de datos sobre las máquinas de trabajo que conforman los clústers. Es un framework de código abierto y se encuentra gestionado por la Apache Software Foundation. El gran valor de este framework es su carácter generalista. Spark consta de diferentes APIs y módulos que permiten que sea utilizado con gran variedad en todo el ciclo de vida de los datos.

El ecosistema de Spark contiene 5 componentes fundamentales:

- **Spark Core**: este núcleo constituye la base de los proyectos y facilita el envío de tareas distribuidas, la programación y las funciones básicas de entrada y salida.
- **Spark SQL**: es el módulo de Spark que permite utilizar datos estructurados. Ofrece un método común para acceder a las diversas fuentes de datos. Gracias a este módulo, se pueden consultar datos estructurados de programas de Spark con SQL o con alguna API de DataFrame como puede ser Pyspark, que definimos en la Sección 2.12.1.
- **Spark Streaming**: éste brinda soporte para el procesamiento de datos en tiempo real, mediante un sistema de empaquetamiento de pequeños lotes. Estos lotes son conocidos como RDD y los comentamos un poco más adelante.
- **MLlib**: es la biblioteca escalable de aprendizaje automático de Spark. Contiene herramientas que convierten en sencillas las tareas prácticas de aprendizaje automáticos, además de numerosos algoritmos de aprendizaje de uso habitual, como clasificación o regresión, entre otros.
- **GraphX**: es la API de Spark para grafos y computación en paralelo de grafos.

En la Figura 2.6, obtenida del artículo de Verma (2021) podemos entender claramente todo el proceso de Spark y qué relevancia tienen los elementos explicados anteriormente. En el propio diagrama también se incluyen algunos ejemplos de herramientas de almacenamiento de datos tanto estáticos como de streaming aceptados en el propio proceso de Spark.

Lo que distingue a Spark de otros frameworks es su modelo de programación basado en la abstracción de algo conocido como **Resilient Distributed Dataset**, RDDs. Un RDD, según Spark, se define como una colección de elementos que es tolerante a fallos y que es capaz de operar en paralelo. Es el tipo básico de Spark y se encuentran distribuidos en las distintas máquinas del clúster. Éstos proporcionan una manera eficiente de realizar operaciones de transformación en los datos distribuidos, lo que simplifica la implementación de los análisis de datos a gran escala.

2.12.1. Pyspark

PySpark es una API de Python desarrollada como parte de la colaboración entre Apache Spark y Python. En otras palabras, es una biblioteca escrita en código

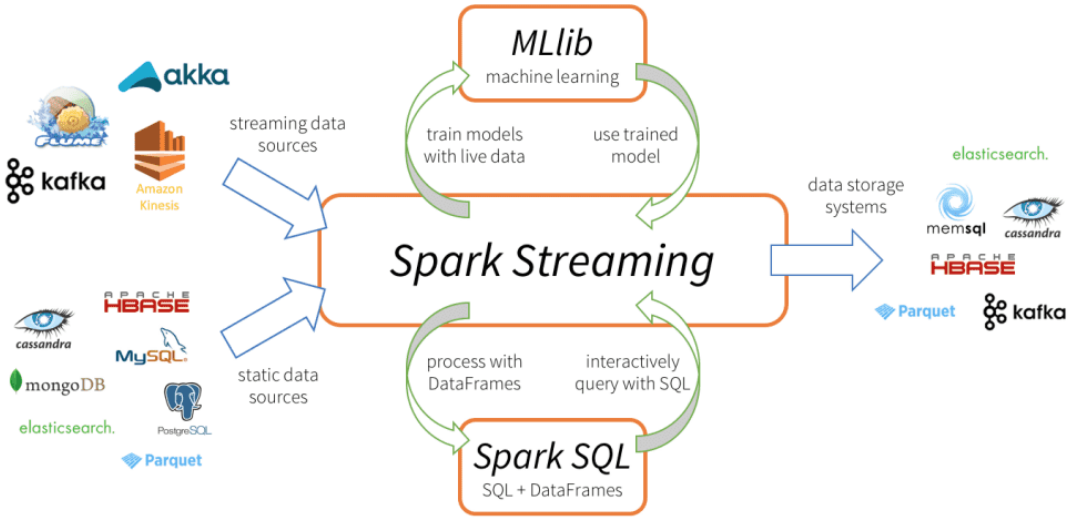


Figura 2.6: Diagrama del procesamiento de datos en Spark

Python que sirve para ejecutar aplicaciones de Python usando las capacidades de Apache Spark. Para trabajar con Pyspark es necesario saber qué es un **SparkContext** y un **SparkSession** y cómo usarlos:

- **SparkContext:** El SparkContext es una entidad fundamental en el funcionamiento de Spark. Representa la conexión a un clúster de Spark y sirve como punto de entrada principal para la funcionalidad de Spark. A través del SparkContext, se pueden realizar diversas operaciones en el clúster, como crear RDDs y establecer configuraciones globales. En esencia, todas las operaciones y tareas que se ejecutan en Spark pasan, de alguna forma, por el SparkContext. Es importante señalar que solo puede haber un SparkContext activo en una instancia de JVM (Java Virtual Machine). Intentar iniciar un segundo SparkContext resultará en errores, por lo que es fundamental gestionarlo adecuadamente.
- **SparkSession:** Introducido en Spark 2.0, SparkSession es un nuevo punto de entrada que encapsula diferentes contextos previamente separados en Spark. Esta entidad combina funcionalidades que antes estaban dispersas en varios contextos, como SQLContext, HiveContext y otros. SparkSession simplifica la interacción con Spark proporcionando un único punto de acceso para diversas operaciones. Esta entidad facilita la ejecución de consultas SQL, la lectura y escritura de datasets en diversos formatos y la configuración de parámetros del clúster, entre otros. Aunque SparkSession es una abstracción de nivel superior, está estrechamente vinculada al SparkContext. De hecho, cuando se crea una SparkSession, se puede acceder y manipular el SparkContext subyacente a través de ella.

¿Cómo trabajan en conjunto contra un clúster de Spark?

Al crear una instancia de SparkContext o SparkSession, se establece una conexión con el gestor del clúster. Esta conexión es esencial para que Spark pueda distribuir y gestionar las tareas entre los diferentes nodos del clúster. Cuando un usuario realiza operaciones, como transformaciones o acciones sobre RDDs o DataFrames, estas operaciones se traducen en tareas que se agrupan en trabajos. Estos trabajos son luego enviados al clúster para ser procesados. Ambas entidades permiten establecer y modificar configuraciones que afectan el comportamiento y rendimiento de Spark. Estas configuraciones pueden incluir aspectos como la asignación de memoria, la cantidad de cores por nodo, entre otros. Al culminar las operaciones en Spark, es recomendable cerrar la instancia de SparkContext o SparkSession. Esto garantiza que se liberen los recursos utilizados y se eviten potenciales problemas de gestión de memoria o de concurrencia en el clúster.

2.12.2. Spark sobre Kubernetes

Una vez hemos visto qué es Spark en líneas generales es importante explicar como actúa en el contexto de Kubernetes. Para entenderlo vamos a utilizar la Figura 2.7 que aparece en la propia documentación de Spark. El proceso que sigue es el siguiente: A través de Spark-submit podemos enviar y ejecutar aplicaciones Spark en nuestro clúster. El mismo Spark manda una petición a la API de Kubernetes para crear un driver de Spark que se ejecuta dentro de un pod de Kubernetes. El driver, que actúa como maestro, crea ejecutores que también se ejecutan dentro de los pods de Kubernetes, se conecta a ellos y ejecuta el código de la aplicación. Cuando la aplicación termina, los pods ejecutores se eliminan pero el pod controlador persiste en un estado de *completo* en el API de Kubernetes hasta que se retira manualmente o hasta que tenga lugar algún proceso automático de limpieza de basura. En cada aplicación de Spark trabajan en conjunto los ejecutores, el controlador y el cluster manager. Este último ayuda al controlador a programar el trabajo entre los nodos. La programación tanto del pod controlador como de los ejecutores es manejada por el API de Kubernetes. Para poder mandar las órdenes de creación intrínsecas en el Spark-submit a la API necesitaremos los privilegios de administrador de una ServiceAccount así como una imagen ya predefinida de Spark para Kubernetes, como las que podemos encontrar en DockerHub, que será la que se ejecute en los pods. Otra característica necesaria para poder usar Spark en Kubernetes es tener los recursos suficientes, como pueden ser 8 GB o más de memoria RAM por nodo o 4 núcleos mínimo por cada CPU disponible.

2.12.3. Formas de ejecutar Spark

Durante el proceso de investigación hemos hallado que hay varias formas de trabajar con Spark en Kubernetes. Principalmente destacaremos dos, una mediante el operador de Kubernetes y otra mediante Spark-submit, que es la que utilizaremos y la que hemos explicado en la sección anterior.

• **Spark on Kubernetes Operator**. Según la definición oficial del grupo desarrollador, El operador de Kubernetes para Spark tiene como objetivo correr apli-

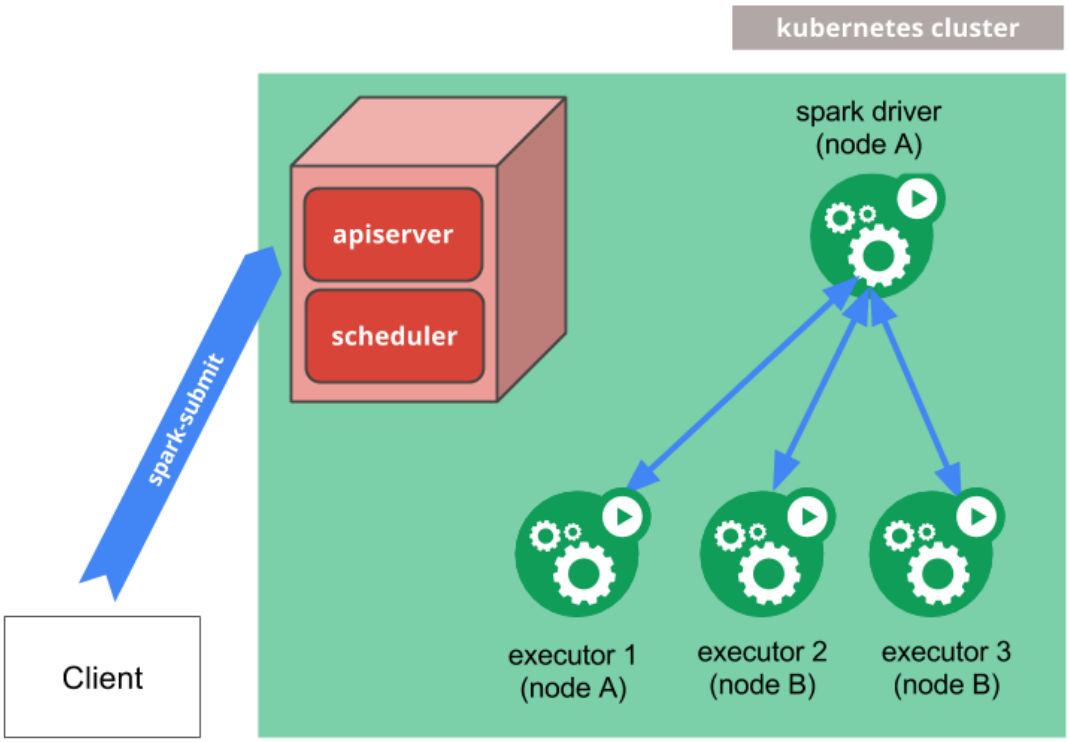


Figura 2.7: Diagrama de ejecución de tareas Spark en Kubernetes

caciones Spark sobre Kubernetes. Para ello utiliza los recursos customizables de Kubernetes para especificar, ejecutar y mostrar el estado de las propias aplicaciones Spark. El motivo por el que hemos descartado esta vía es Jupyter. El operador se utiliza para mandar trabajos que ya tienen un fichero estatico predefinido o para establecer workflows de trabajo. Es recomendable para trabajos que se tienen que planificar para que se ejecuten cada cierto tiempo. No es tan recomendable en entornos de análisis de datos como los cuadernos de Python, ya que no tiene integración directa.

• **Spark-submit**. Spark-submit se utiliza para lanzar una aplicación de Spark directamente en un clúster de Kubernetes. En el Apartado 2.12.2 explicamos brevemente como funciona este método.

Vistas las formas que tiene Spark para integrarse con Kubernetes es hora de mencionar las maneras en las que Spark se puede desplegar en un clúster a secas. En el **modo cliente**, la máquina desde la cual envías la solicitud de ejecución también participa en la ejecución de la aplicación. El cliente envía la solicitud al clúster, donde se inician los nodos ejecutores, y luego el cliente interactúa directamente con esos nodos para monitorizar y recibir los resultados. El cliente debe permanecer activo durante toda la vida útil de la aplicación. En el **modo clúster**, el cliente envía la solicitud al la API de Kubernetes del Control-Plane y este crea los pods en los cuales se ejecutarán los nodos ejecutores de Spark en los Nodos del clúster que el API decida. Una vez que se inician los ejecutores, la interacción del cliente no es necesaria, y el cliente puede cerrarse para ahorrar recursos. Cuando termina

la ejecución del trabajo el en cada pod ejecutor se transmite al pod del driver que tendrá el resultado final del trabajo.

En otras palabras, en el modo cliente, éste está involucrado durante toda la ejecución, pudiendo monitorizar los resultados de cerca, mientras que en el modo de clúster, el cliente solo envía la solicitud y luego el clúster maneja la ejecución de manera autónoma.

2.12.4. Spark History Server

Ésta es una componente de Apache Spark que proporciona una interfaz web para visualizar información detallada sobre las aplicaciones Spark que se han ejecutado en el clúster. Algunas de sus características incluyen la capacidad de ver el historial de aplicaciones, el resumen de tareas y etapas, las estadísticas de uso de recursos y la visualización de la línea de tiempo de ejecución. Esto resulta muy útil para la monitorización, optimización y depuración de aplicaciones Spark, ya que informa sobre cómo se procesaron los datos y cómo se utilizaron los recursos en el clúster.

2.12.5. Jupyter y JupyterHub

Jupyter (Figura 2.8) es un entorno de desarrollo interactivo y una plataforma de código abierto ampliamente utilizada en ciencia de datos, análisis de datos y programación científica. Lo que distingue a Jupyter es su capacidad para crear y compartir documentos llamados notebooks que contienen celdas de código y texto. Estos cuadernos pueden contener código en varios lenguajes de programación, pero son especialmente populares en el contexto de Python y PySpark. Los usuarios pueden escribir y ejecutar código en celdas individuales, ver los resultados de inmediato, y agregar comentarios y visualizaciones para documentar y compartir sus análisis. Esto facilita la experimentación y el análisis de datos, ya que los usuarios pueden iterar rápidamente en su código y ver los efectos de sus cambios en tiempo real.

Figura 2.8: Logo de Jupyter

JupyterHub es una extensión de Jupyter que permite la implementación y el uso compartido de Jupyter Notebooks en un entorno de múltiples usuarios. Al tratarse de un clúster al que van acceder distintos usuarios simultáneamente, JupyterHub es fundamental en nuestro proyecto si queremos ofrecer la oportunidad de usar los cuadernos de Jupyter. Este servicio proporciona una interfaz de inicio de sesión que permite a los usuarios autenticarse y acceder a su propio entorno de Jupyter personalizado. Esto garantiza que cada usuario tenga su espacio de trabajo aislado

para crear y ejecutar cuadernos, lo que ayuda en la colaboración y la administración centralizada en entornos de uso compartido. Para que cada usuario tenga su espacio personalizado tenemos que configurar el almacenamiento persistente de tal manera que sea seguro y privado para cada usuario. En la Sección 3.6.1 comentamos como lo hemos implementado.

2.13. Dashboards: Grafana y Kubernetes

Un dashboard es una interfaz de usuario basada en la web, con la que se puede manejar los distintos recursos del clúster sin tener que acceder a terminal o sin necesidad de aprender los comandos de Kubernetes. Aparte de todo esto también sirve para obtener una vista general de las aplicaciones que se están ejecutando en el clúster, y para crear o modificar recursos individuales de Kubernetes. El aspecto más interesante es poder monitorizar las distintas aplicaciones mediante métricas especificaciones. El uso, que no la configuración, de un dashboard es bastante intuitivo y lo consideramos una buena incorporación al trabajo para poder acceder a la información del clúster de forma sencilla. En la descripcion del trabajo3.7 vemos los pasos necesarios para instalar esta herramienta.

Un dashboard en Grafana (Figura 2.9) es un conjunto de uno o más paneles organizados y juntados en una o más filas. Grafana trabaja con una variedad de paneles que facilitan la construcción de las busquedas de las métricas y que ayudan a personalizar la visualización para crear el dashboard deseado. Cada panel puede interactuar con los datos pertenecientes a cualquier fuente de datos configurada en Grafana. En otras palabras, Grafana es una aplicación que permite configurar paneles para poder ver distintas métricas de distintos servicios para ver el rendimiento de los mismo.

Una posibilidad interesante es que hay dashboards predeterminados creados por miembros de la comunidad aunque es preferible optar por elaborar uno propio, incluyendo las medidas que verdaderamente consideremos oportunas.

Aparte de Grafana, Kubernetes también presenta un dashboard propio. En éste podremos monitorizar todos los recursos del clúster de Kubernetes y así centrar el dashboard de Grafana únicamente para servicios como MongoDB. Además de esto, el propio dashboard de Kubernetes permite modificar de forma gráfica distintas componentes del clúster, permitiendo al administrador hacerlo sin necesidad de usar los comandos de Kubernetes.

Figura 2.9: Logo de Grafana

Capítulo 3

Descripción del Trabajo

En este capítulo vamos a exponer el trabajo práctico que hemos realizado para llevar a cabo este trabajo de final de grado. Seguiremos el orden que consideramos óptimo para facilitar la posible réplica del sistema a quién pueda estar interesado.

Este capítulo lo hemos divido en secciones. La primera comprende toda la preparación del entorno hardware necesario así como la red implementada entre las máquinas disponibles. La segunda por su parte comprende toda el proceso de instalación del clúster de Kubernetes. Como tercer punto tenemos toda la configuración necesaria para organizar el almacenamiento persistente. La cuarta y quinta secciones son la implementación de Mongo y Spark respectivamente mientras que la quinta y última parte contiene el proceso necesario para implementar el sistema de monitorización del clúster. Dentro de la parte de Spark también incluimos el proceso necesario para añadir Jupyter a nuestro clúster. Además de esto se puede encontrar en el apéndice A una guía con algunos de los comandos más útiles para Kubernetes y Docker.

Cabe destacar que a lo largo del trabajo hemos utilizado la misma contraseña para la mayoría de servicios. Por motivos de seguridad no incluiremos la verdadera contraseña y en su lugar escribiremos: *contraseña*. Con los tokens haremos lo mismo solo que escribiendo: *token*.

Por último, queríamos indicar que en algunas secciones de este capítulo se pueden apreciar algunos comandos que hemos utilizado en la terminal. Para poder diferenciarlo claramente de cualquier otra componente aparecerán en negrita y después de este símbolo: '>'

3.1. Preparación del entorno

Uno de los aspectos más importantes a tener en cuenta para realizar un trabajo en el que el hardware tiene un peso relevante es ser consciente del material disponible y del entorno en el que vamos a trabajar. Para poder implementar el clúster primero debemos conocer los 5 equipos que lo van a formar así como las características de los

Nodos	Procesador	Velocidad CPU	Memoria RAM	Capacidad de disco/s
Nodo 1	i7	3.6 Ghz	8 GB	5 (4 + 1) TB
Nodo 2	i7	3.4 Ghz	16 GB	4.5 (4 + 0.5) TB
Nodo 3	i5	2.67 GHz	8 GB	2 TB
Nodo 4	i5	2.67 GHz	8 GB	5 (3.6 + 1.4) TB
Nodo 5	i5	3.2 GHz	8 GB	6 (4 + 2) TB

Tabla 3.1: Características de los equipos

mismos. Una de las razones por las que es importante conocer los distintos aspectos de los computadores es para conocer sus posibilidades y poder determinar qué roles tendrán los equipos dentro de la estructura que vamos a seguir. Las características de los equipos se pueden ver en la Tabla 3.1.

Viendo los distintos equipos podemos apreciar algunos más potentes y con características más interesantes que otros. A simple vista los nodos 1 y 2 parecen los más avanzados ya que son los únicos que contienen un procesador de séptima generación, y ambos son los que presentan una mayor velocidad de CPU. Partiendo de esta distinción el nodo 2 duplica en memoria RAM al primer nodo, sin embargo, no hemos podido contar con el nodo 2 hasta avanzado el trabajo, por lo que optamos por el nodo 1 como el nodo maestro del clúster, que va a seguir el modelo que explicamos en la Sección 2.3.2. De todas formas para la elección de este nodo no es fundamental que el elegido sea aquel con mejores características ya que al final en el clúster todos los computadores van a trabajar como si fueran un único equipo, pero sí es cierto que el nodo maestro es el único que va a acceder a la red y por eso hemos buscado que el elegido sea el que mayor velocidad de CPU tenga.

Una vez tenemos claro el material del que disponemos, es necesario pensar en la estructura de red que se va fijar en el clúster. Teniendo en cuenta la disponibilidad del material con el que trabajamos, la estructura de red que vamos a seguir es la que se puede observar en la Figura 3.1, y los nodos incluidos siguen los valores de la Tabla 3.2. La dirección de red que disponemos para realizar el trabajo es la 147.92.81.119, perteneciente a la Universidad Complutense de Madrid. Para la red privada hemos decidido utilizar la dirección 192.168.1.0/24, asignando direcciones únicas a cada uno de los hosts y en acorde al número de nodo que tienen. Esta estructura de red tiene aspectos muy positivos como la escabilidad, es decir, es muy sencillo poder añadir nuevos nodos a la red privada sin que suponga un trabajo exhausto. Por otro lado, si quisieramos añadir un nodo maestro la situación sería distinta, sabiendo que disponemos de únicamente un cable de red para conectarnos a internet.

3.1.1. Instalación sistemas operativos

Una vez recopilamos toda la información sobre los equipos disponibles tuvimos que instalar el sistema operativo elegido, siguiendo las razones del Apartado 2.2, en todos los equipos. Algunos tenían un sistema operativo distinto ya instalado y otros directamente no tenían nada funcional. Una vez obtuvimos todos los archivos necesarios desde: https://releases.ubuntu.com/jammy/, configuramos todos los as-

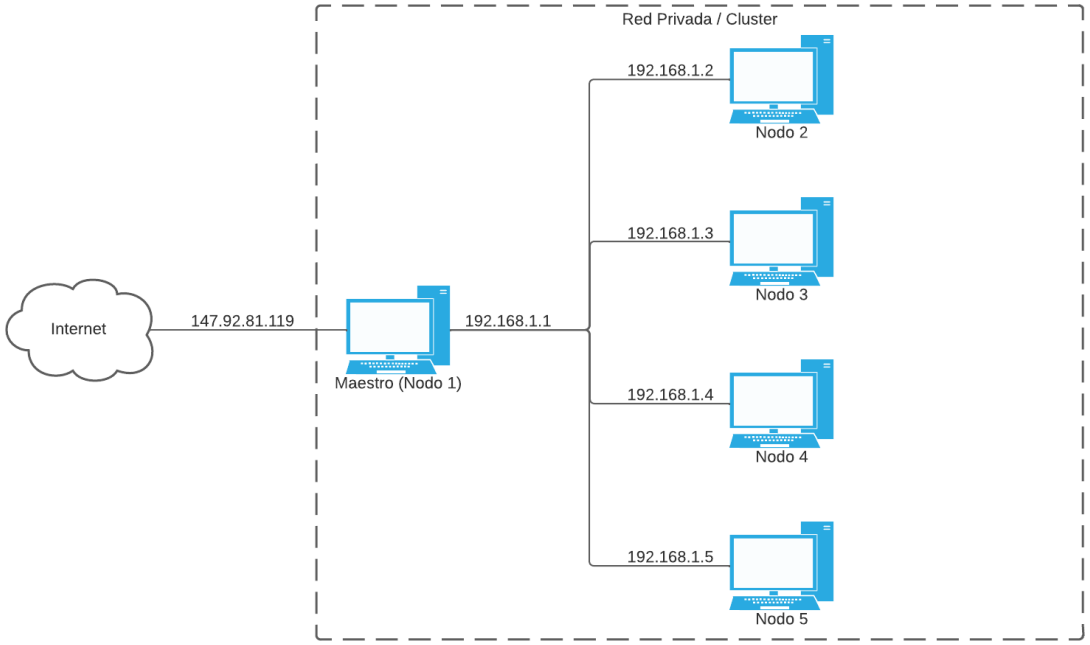


Figura 3.1: Diagrama de red

pectos necesarios del sistema. Consideramos que todo este proceso de instalación es bastante intuitivo por lo que no entraremos en detalles más concretos. Cabe destacar que instalamos una versión reducida de Ubuntu, que no contiene interfaz gráfica.

Nodo	Dir. IP local	Tipo nodo
Nodo 1	196.168.1.1	Nodo Maestro
Nodo 2	196.168.1.2	Nodo Esclavo
Nodo 3	196.168.1.3	Nodo Esclavo
Nodo 4	196.168.1.4	Nodo Esclavo
Nodo 5	196.168.1.5	Nodo Esclavo

Tabla 3.2: Nodos del diagrama

3.1.2. Herramientas auxiliares adicionales

En este apartado vamos a contar qué herramientas adicionales hemos utilizado una vez se configuraron correctamente las máquinas para poder acceder a las mismas desde un sitio remoto. Para poder acceder a los equipos vía SSH utilizamos Termius, un cliente SSH multiplataforma. La instalación de Termius es muy sencilla por lo que tampoco nos detendremos en este aspecto, sin embargo, en la Figura 3.2 podemos ver la configuración que establecimos en la aplicación de escritorio para poder conectarnos al nodo maestro.

Para poder acceder al escritorio directamente de forma remota usamos AnyDesk, que es un programa de software de escritorio remoto. El principal problema que

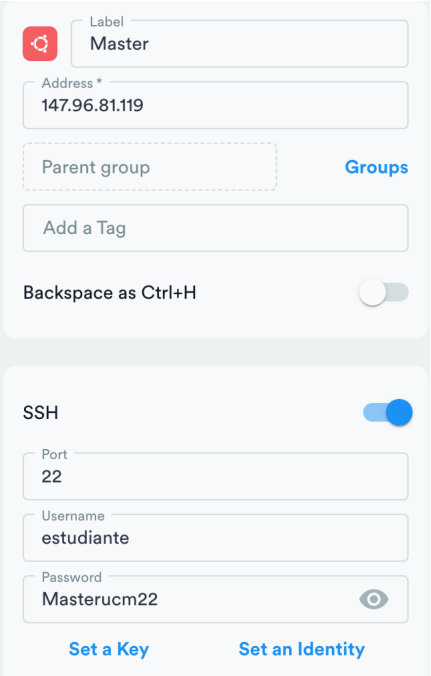


Figura 3.2: Configuración de Host en Termius

encontramos con esto fue el consumo de recursos por parte de la GUI, por lo que decidimos usar la mayoría del tiempo el cliente SSH en vez del escritorio remoto, que funcionaba de forma muy ralentizada.

También, en las étapas finales del desarrollo, utilizamos VS Code para acceder a todo el sistema de archivos de configuración del clúster vía SSH. De esta forma podíamos hacer modificaciones de forma más sencilla que con el propio editor de texto de Ubuntu.

Por último, es importante destacar que para poder acceder a la red es fundamental configurar una VPN, ya que es una dirección perteneciente a la Universidad Complutense de Madrid. Para hacerlo instalamos en nuestros equipos de trabajo GlobalProtect. En este link se puede encontrar toda la información necesaria de forma sencilla: https://www.ucm.es/faq/conexion-vpn/como-configurar-una-conexion-vpn-a-la-red-ucm.

3.2. Primeros intentos

Antes de llegar a la conclusión definitiva de usar Kubernetes primero contemplamos e incluso instalamos otras posibilidades que resultaron ser menos convincentes que la decisión final de utilizar Kubernetes. De hecho, nuestra primera opción no fue Kubernetes, sino Ubuntu MAAS (Metal as a Service). Este software abierto proporcionaba una automatización completa de los servidores físicos para operar como un centro de datos. MAAS basicamente trata los servidores fisicos como máquinas virtuales y convierte un conjunto físico en un recurso elástico similar al cloud, evitando manejar cada servidor de forma individual. El proceso de instalación que seguimos

se puede encontrar bien explicado en la página oficial de Ubuntu (2023)

Sin ser un intento propiamente dicho, al comenzar el trabajo instalamos MongoDB en cada uno de los equipos, sin tener en cuenta la estructura del clúster ni el hecho de que íbamos a usar kubernetes. Esta instalación acabó siendo más perjudicial de lo que pensábamos ya que nos vimos obligados a formatear uno de los equipos cuándo intentamos desinstalarlo y en el proceso se eliminó alguna dependencia que imposibilitaba el uso de apt en el sistema.

3.3. Kubernetes

3.3.1. Instalación kubernetes

Para instalar correctamente el cluster de kubernetes vamos a utilizar kubeadm, que como hemos mencionado anteriormente es el encargado de crear y desplegar clústeres. Además de esto también vamos a instalar kubelet y kubectl. El primero es un agente que se ejecuta en cada nodo trabajador que garantiza que los contenedores se encuentran corriendo en los recursos de pod determinados. El otro en cambio es una interfaz de línea de comandos para poder ejecutar estos sobre despliegues clusterizados de Kubernetes. Esta será la manera estándar de comunicación con el cluster creado. El proceso que llevamos a cabo fue el siguiente:

- Añadimos el repositorio de Kubernetes para Ubuntu 22.04:

```
> sudo apt install curl apt-transport-https -y

> curl -fsSL  https://packages.cloud.google.com
/apt/doc/ apt-key.gpg |sudo gpg --dearmor -o
/etc/apt/trusted.gpg.d/k8s.gpg

> curl -s https://packages.cloud.google.com/apt/doc/
apt-key.gpg | sudo apt-key add -

> echo "deb https://apt.kubernetes.io/
kubernetes-xenial main" | sudo tee
/etc/apt/sources.list.d/kubernetes.list
```

- Instalamos los paquetes mencionados anteriormente (kubeadm, kubelet, kubectl):

```
> sudo apt install wget curl vim git
kubelet kubeadm kubectl -y

> sudo apt-mark hold kubelet kubeadm kubectl
```

- Desactivamos los intercambios. Esto implica configurar el sistema operativo para que no utilice archivos de intercambio (swap) en la memoria virtual. Kubernetes y las aplicaciones en contenedores están diseñados para utilizar la memoria RAM de manera eficiente, y el uso de intercambios puede afectar negativamente el rendimiento y la capacidad de respuesta de los contenedores:

```
> sudo swapoff -a

> sudo vim /etc/fstab
```

```
Comentamos esta linea para desactivar el espacio
de intercambios de Linux
#/swap.img none  swap  sw 0  0
```

• Habilitamos los módulos de kernel y configuramos sysctl

```
> sudo modprobe overlay
```

```
> sudo modprobe br_netfilter
```

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

```
> sudo sysctl --system
```

• Ahora procedemos a instalar el container runtime. Llegados a este punto nos encontramos con diversas opciones. Entre estas se encuentran el Runtime de Docker, de CRI-O y Containerd. Como hemos visto las posibilidades en la Sección 2.8.3, finalmente nos hemos decantado por Containerd, ya que de los tres es el más conocido por su enfoque modular y su facilidad a la hora de intregarse con otras herramientas o sistemas.

– Configuramos la carga persistente de módulos:

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

– Carga en tiempo de ejecución

```
> sudo modprobe overlay
```

```
> sudo modprobe br_netfilter
```

– Recargamos la configuración:

```
> sudo sysctl --system
```

– Instalamos los paquetes requeridos:

```
> sudo apt install -y curl gnupg2
software-properties-common
apt-transport-https ca-certificates
```

– Añadimos el repositorio de Docker:

```
> curl -fsSL https://download.docker.com/
linux/ubuntu/gpg | sudo apt-key add -
```

```
> sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu
\$(lsb_release -cs) stable"
```

– Instalamos containerd y lo configuramos:

```
> sudo apt install -y containerd.io
```

```
> sudo su -
```

```
> mkdir -p /etc/containerd
> containerd config default>/etc/containerd/
config.toml
> sudo systemctl restart containerd
> sudo systemctl enable containerd
> systemctl status containerd
```

• Inicializamos el plano de control (sólo en el nodo maestro). Esto es el encargado de administrar y controlar el clúster de Kubernetes, por eso lo inicializamos únicamente en el nodo maestro:

– Nos aseguramos de que el módulo br_netfilter está cargado:

```
> lsmod | grep br_netfilter
```

– Permitimos el uso de kubelet:

```
> sudo systemctl enable kubelet
```

– Ahora vamos a inicializar la máquina que ejecutará los componentes del control de plano, que incluyen la bd del cluster y el servidor API. Descargamos las imagenes del contenedor:

```
> sudo kubeadm config images pull
```

– Configuramos kubectl usando los comandos que nos mostraba la salida del paso anterior:

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

– Comprobamos el estado del cluster para ver que el maestro se está ejecutando:

```
> kubectl cluster-info
```

• Instalamos el complemento de red de Kubernetes: Para esto usaremos el complemento de Fannel. Este es un complemento de red utilizado para habilitar la conectividad entre los pods del clúster.

– Descargamos el manifesto de instalación

```
> wget https://raw.githubusercontent.com/flannel-
io/flannel/master/Documentation/kube-flannel.yml
```

– Instalamos Flannel, y confirmamos tanto que los pods están ejecutándose y que el master está preparado:

```
> kubectl apply -f kube-flannel.yml
> kubectl get pods -n kube-flannel
> kubectl get nodes -o wide
```

• Añadimos al clúster el resto de nodos: los esclavos.

– Una vez inicializamos el clúster, el propio comando nos devolvió una instruccion para poder utilizar en el resto de nodos que quisieran unirse al clúster. Lo

copiamos y lo ejecutamos en los distintos nodos. El primero de los siguientes comandos nos da el token que vamos a introducir en el siguiente, por lo que los valores del token del segundo comando deben ser modificados:

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

```
> kubeadm join master:6443 --token cs5avf.nhxue27
--discovery-token-ca-cert-hash
sha256:7eba8de5df9f7c6665b5333
```

– Comprobamos en el nodo maestro si el resto de nodos se han unido correctamente al clúster. El resultado de ese comando se debería ver como en la Figura 3.3:

```
> kubectl get nodes
```

De esta forma ya hemos montado el clúster en kubernetes. De momento es un clúster básico de 5 nodos sin ningún servicio ejecutándose. En las siguientes secciones vamos a añadir más servicios para poder convertir este clúster en un sistema funcional de procesamiento y almacenamiento distribuido.

```
estudiante@master1:~$ kubectl get nodes
NAME      STATUS   ROLES           AGE     VERSION
master1   Ready    control-plane   5d22h   v1.26.3
slave2    Ready    worker          5d21h   v1.27.3
slave3    Ready    worker          5d22h   v1.26.3
slave4    Ready    worker          5d21h   v1.26.3
slave5    Ready    worker          5d21h   v1.26.3
estudiante@master1:~$
```

Figura 3.3: Nodos del clúster de Kubernetes

3.3.2. Ingress

Antes de pasar a explicar el proceso para implementar Ingress queremos mencionar que finalmente no lo utilizamos. Dejamos los pasos en la memoria porque es un aspecto interesante a la hora de exponer servicios en Kubernetes y queremos brindar la oportunidad de usarlo.

A la hora de instalar el controlador de Ingress de Nginx, hay diversas formas de hacerlo pero nosotros vamos a continuar usando Helm.

- Para desplegar el controlador de Ingress usamos este comando:

```
> helm upgrade --install ingress-nginx ingress-nginx \
--repo https://kubernetes.github.io/ingress-nginx \
--namespace ingress-nginx --create-namespace
```

- Vamos a comprobar que el controlador se ha desplegado correctamente.

– Algunos pods deberían iniciarse en el espacio de nombres de ingress-nginx. Con el primer comando los vemos y con el segundo esperamos a que el pod del controlador esté preparado. La salida del primer comando debe parecerse a la FIgura 3.4:

```
> kubectl get pods --namespace=ingress-nginx
> kubectl wait --namespace ingress-nginx \
--for=condition=ready pod \
--selector=app.kubernetes.io/component=controller \
--timeout=120s
```

```
estudiante@master:~$ kubectl get pods --namespace=ingress-nginx
NAME                                        READY   STATUS      RESTARTS        AGE
ingress-nginx-admission-patch-qmwm7         0/1     Completed   0               14d
ingress-nginx-controller-5556855b4b-6clzv   1/1     Running     1 (7d11h ago)   10d
estudiante@master:~$
```

Figura 3.4: Pods del controlador Ingress

– Vamos a crear un servidor web sencillo:

```
> kubectl create deployment demo --image=httpd
--port=80
> kubectl expose deployment demo
```

– A continuación creamos un recurso Ingress. Con el siguiente comando usamos un host que mapea a localhost:

```
> kubectl create ingress demo-localhost
--class=nginx \ --rule="demo.localdev.me/*=demo:80"
```

– Reenviamos un puerto local al controlador de entrada:

```
> kubectl port-forward --namespace=ingress-nginx
service/ingress-nginx-controller 8080:80
```

– Finalmente podemos acceder al despliegue usando curl. Para saber si todo ha funcionado correctamente se mostrará por pantalla el contenido de dicha web:

```
> curl --resolve demo.localdev.me:8080:127.0.0.1
http://demo.localdev.me:8080
```

3.3.3. Almacenamiento Persistente con Rook-Ceph

Como paso previo a configurar y lanzar nuestros servicios, necesitaremos un sistema de almacenamiento persistente que nos asegure que la información que estos manejan no se corrompa y se pierda. En el caso de las bases de datos como MongoDB esto es fundamental ya que los datos y la configuración de los pods debe persistir ante reinicios, apagones, etc. Para poder implementarlo necesitaremos discos duros o particiones sin estructura, es decir en formato RAW. Necesitaremos también un mínimo de 3 nodos en nuestro clúster.

• Clonación del proyecto Rook desde Github:

Para comenzar, es necesario obtener el código fuente del proyecto Rook. Para ello, clonamos el repositorio de Rook desde Github en una máquina donde esté configurado y funcionando correctamente kubeconfig.

```
> cd ~/
> git clone --single-branch \
--branch release-1.12 https://github.com/rook/rook.git
```

Este comando clona específicamente la rama release-1.12 del repositorio.

• Despliegue del rook operator: Para facilitar el despliegue del sistema del almacenamiento distribuido como ya hemos explicado anteriormente aprovecharemos en la medida de lo posible el uso de los operadores, en este caso el de rook. Primero desplegaremos los cdrs que permitiran que el operador sea capaz de autogestionar los recursos de los despliegues.

```
> kubectl create -f crds.yaml
```

Ahora aplicaremos el fichero *common.yaml* que creará los RoleBindings, ServiceAccounts, y otros recursos de Kubernetes necesarios para que el operador funcione correctamente. Estos recursos establecen los permisos y contextos adecuados para que el operador interactúe con el clúster de Kubernetes y gestione otros recursos.

```
> kubectl create -f common.yaml
```

Seguidamente procederemos ya al despliegue del operador.Una vez desplegado, el operador comenzará a monitorear y gestionar los recursos definidos por los CRD (en este caso, los recursos definidos en crds.yaml).

```
> kubectl create -f operator.yaml
```

Siempre es importante comprobar que los componentes estén en estado *running* al finalizar el despliegue:

```
> kubectl get all -n rook-ceph
```

Deberíamos ver que el pod del operador está en estado running, que el deployment está listo y disponible y que el replicaset está igualmente disponible y listo como en la Figura 3.5

Si están en ejecución pasaremos ahora a desplegar con el operador el Ceph Cluster.

• Despliegue del Ceph Cluster:

Continuamos ahora con la creación de Ceph Cluster. Antes de empezar estableceremos el namespace por defecto al de rook-ceph que es el namespace que se crea al desplegar el operador.

```
> kubectl config set-context --current --n rook-ceph
```

Teniendo en cuenta una de las principales características de rook, que es el autodescubrimiento de las particiones en RAW que deberán disponer nuestros nodos no será necesario modificar el archivo de configuración del despliegue. Nosotros usaremos el fichero de configuración *cluster-test.yaml*, ya incluido en el repositorio el cual

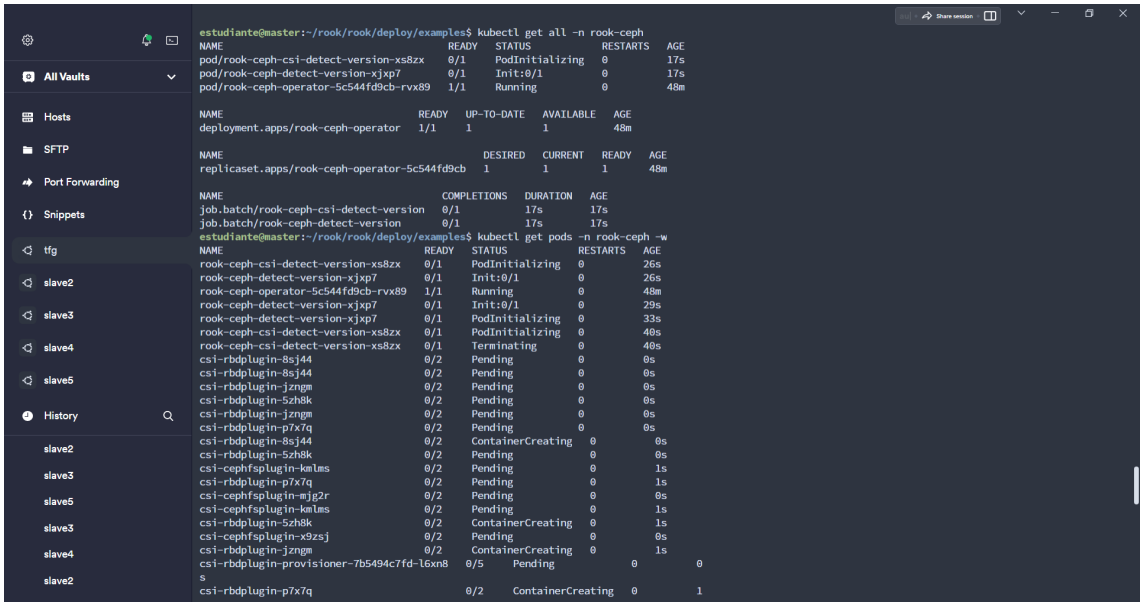


Figura 3.5: Rook Operator desplegado

sirve para entornos de prueba y con recursos limitados debido a que es un despliegue más ligero.

Si por algún motivo no queremos incluir algun nodo en nuestro sistema de almacenamiento distribuido o especificar las particiones concretas que conformarán el sistema, deberemos editar según la documentación de rook, la sección del fichero *cluster-test.yaml* referente al storage:

```
....
 storage:
     useAllNodes: false # indicamos que no use todos los nodos
     useAllDevices: false # indicamos que no use todas particiones
     nodes:
     - name: "master"
       devices:
       - name: "sdb"
     - name: "slave1"
       devices:
       - name: "sdb"
     - name: "slave2"
       devices:
       - name: "sdc"
....
```

No editaremos por ahora esta configuración y procederemos directamente al despliegue.

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

Y ahora supervisaremos durante unos minutos la creación de los recursos:

```
> kubectl get pods -n rook-ceph -w
```

Para verificar Ceph se ha creado correctamente miraremos que tenemos dos OSD ejecutándose, perteneciente cada uno a los dos nodos que ha encontrado con parti-

ciones en RAW disponibles y que concuerda con las características de los equipos. Deberemos ver una salida como la de la figura 3.6

Y finalmente comprobaremos con el siguiente comando que el cluster está listo.

```
> kubectl -n rook-ceph get cephcluster
```

Veremos que la columna *HEALTH* está en *HEALTH WARN* pero esto es debido a que nuestro clúster cuenta con 2 nodos con particiones en RAW cuando por recomendación el mínimo deberían ser tres. No tiene demasiada importancia, debido a que la configuración como ya hemos mencionado antes es la de *cluster-test.yaml* que permite que el Ceph Cluster funcione con 2 nodos.

• Creación de las clases de almacenamiento:

Para hacer uso del sistema de almacenamiento distribuido y persistente que acabamos de montar, deberemos crear StorageClasses o clases de almacenamiento. En ellas se pueden definir algunas características que Rook-Ceph tendrá en cuenta al trabajar con esa clase de almacenamiento, como la *reclaimPolicy* que define qué debe suceder con un volumen físico una vez que se elimina el PersistentVolumeClaim, entre otros. Todos los ficheros necesarios para cada modo están disponibles en el directorio rook/cluster/examples/kubernetes/ceph/

Nos situaremos por tanto en él.

```
> cd rook/deploy/examples
> cd ~/
```

Ahora, empezaremos creando la StorageClass Cephfs:

Con el siguiente comando creamos un cephfilesystem:

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

Comprobaremos que los pools en los osd están creados viendo que el comando devuelve el pool de metadatos y la replicación

```
> ceph osd lspools
1 device_health_metrics
3 replicapool
...
8 <nombre-del-filesystem>-metadata
9 <nombre-del-filesystem>-replicated
```

Ahora una vez creado el filesystem, creamos la StorageClass con el siguiente comando:

```
> kubectl create -f csi/cephfs/storageclass.yaml
```

Comprobaremos con el comando get que la clase de almacenamiento está creada:

```
> kubectl get sc
```

Si nos devuelve una salida con una clase llamada *rook-cephfs* habremos añadido la clase de almacenamiento correctamente.

Una vez configurado Cephfs pasaremos a añadir la Ceph-block.

Con el siguiente comando crearemos la clase:

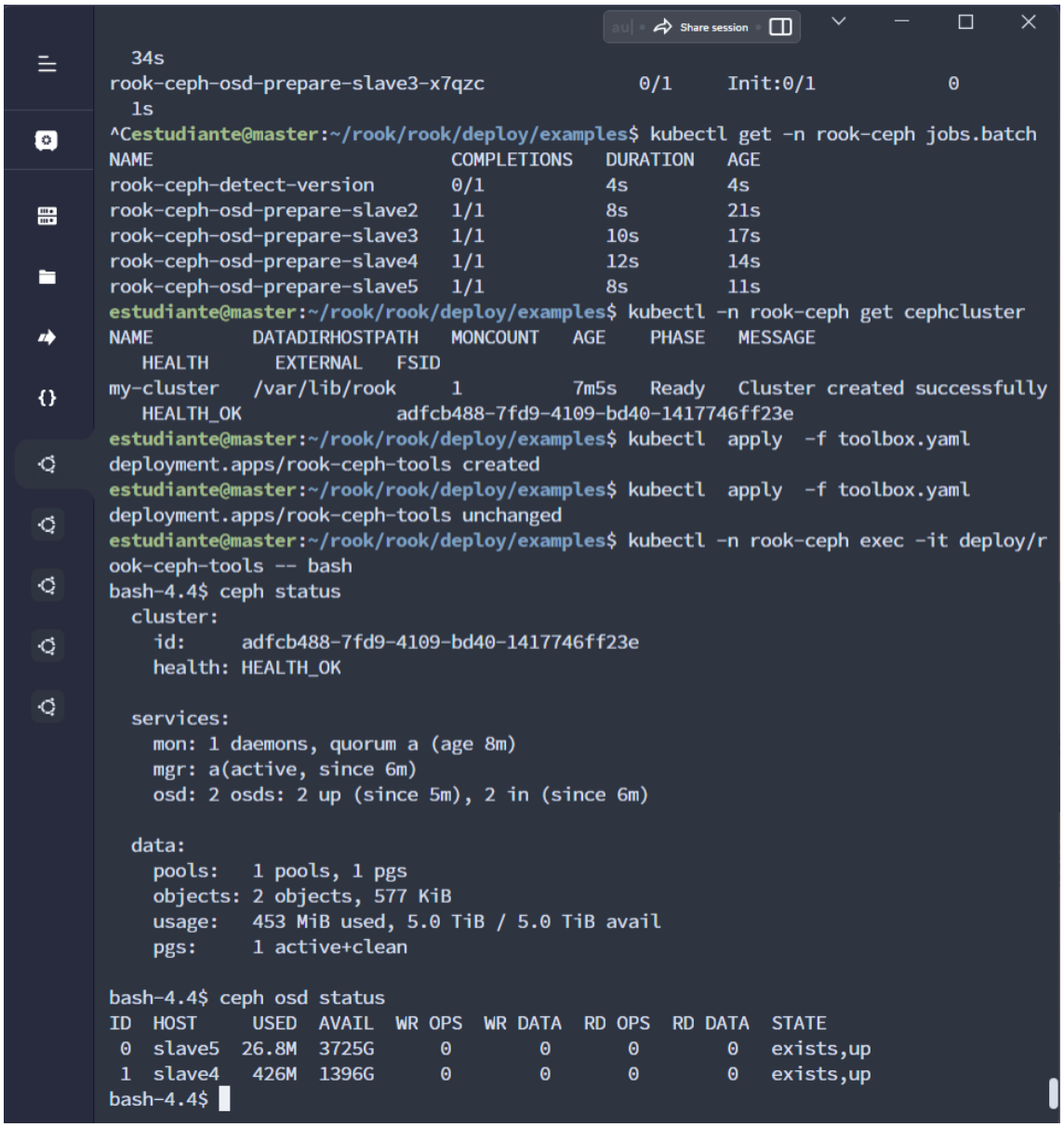


Figura 3.6: Ejemplo de un despliegue de Rook-Ceph exitoso

```
> kubectl  create  -f csi/rbd/storageclass.yaml
```

Seguidamente crearemos el pool:

```
> kubectl -f csi/rbd/pvc.yaml
```

Finalizamos comprobando que efectivamente, ahora tendremos dos StorageClass. *rook-cephfs* y la que acabamos, *rook-ceph-block*.

```
> kubectl get sc
```

Si nos salen esas dos clases, habremos añadido y configurado con éxito un sistema de almacenamiento distribuido, persistente y además escalable en nuestro clúster de Kubernetes.

3.4. MongoDB

En esta sección vamos a describir todo el proceso necesario para instalar correctamente MongoDB en el clúster. Inicialmente configuraremos MondoDB con 4 fragmentos y luego los juntaremos para formar la base de datos distribuida que deseamos añadir a nuestra arquitectura. Nuestra idea principal era usar el operador de MongoDB sobre Kubernetes que mencionamos en la Sección 2.11.4, pero finalmente lo hicimos de manera manual, que permitía mayor libertad de configuración a cambio de una implementación algo más compleja. Tampoco permitía usar el sharding de MongoDB, que es lo verdaderamente imprescindible si queremos proporcionar almacenamiento distribuido.

Antes de abordar todo este laborioso proceso es aconsejable tener algunos consejos en cuenta. En los siguientes puntos vamos a explicar primero los archivos de configuración necesarios y una vez estén todos definidos pasaremos a su aplicación dentro de Kubernetes. Después procederemos a configurar todos los archivos aplicados para poder completar el servicio de MongoDB de forma distribuida. Para cada archivo explicaremos el significado de cada cosa y, en caso de haberlas, las variantes necesarias para equipos distinto del nuestro. Por último, los archivos que vamos a mostrar son las versiones finales de los mismos, por lo que no mostraremos las versiones anteriores de prueba que utilizamos antes de dar con la configuración adecuada y precisa.

3.4.1. Despliegue del clúster de MongoDB

• Antes de nada vamos a crear las credenciales de mongoDB que vamos a necesitar usar en los distintos archivos, con el nombre de usuario y la contraseña que utilizaremos. Estas credenciales las crearemos a través de los secretos de Kubernetes.

```
> kubectl create secret generic mongo-credentials \
    --from-literal=MONGO_INITDB_ROOT_USERNAME=root \
    --from-literal=MONGO_INITDB_ROOT_PASSWORD
    =contrase a
```

• El primer archivo que vamos a crear es **configserver-statefulset.yaml**. Este archivo será fundamental para la instalación de MongoDB dentro del clúster de Kubernetes, ya que desplegará las instancias de servidores de configuración necesarias.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongodb-configserver
spec:
  serviceName: "mongodb-configserver"
  replicas: 3
  selector:
    matchLabels:
      role: mongodb-configserver
  template:
    metadata:
      labels:
        role: mongodb-configserver
    spec:
      containers:
      - name: mongodb-configserver
        image: mongo:4.4
        env:
        - name: MONGO_INITDB_ROOT_USERNAME
          valueFrom:
            secretKeyRef:
              name: mongo-credentials
              key: MONGO_INITDB_ROOT_USERNAME
        - name: MONGO_INITDB_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mongo-credentials
              key: MONGO_INITDB_ROOT_PASSWORD
        command:
          - mongod
          - "--configsvr"
          - "--replSet"
          - configReplSet
          - "--bind_ip"
          - "0.0.0.0"
        ports:
        - containerPort: 27019
        volumeMounts:
          - name: mongodb-configserver-data
            mountPath: /data/db
  volumeClaimTemplates:
  - metadata:
      name: mongodb-configserver-data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "rook-ceph-block"
      resources:
        requests:
          storage: 10Gi
```

En el anterior archivo de configuración especificamos las credenciales creadas

anteriormente. También cogemos la versión 4.4 de la imagen de MongoDB de Docker y especificamos el bloque de almacenamiento persistente que hemos configurado en el paso anterior.

- A continuación creamos el siguiente archivo: **configserver-service.yaml**. Con este expondremos como servicio el servidor principal de MongoDB.

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb-configserver
  namespace: mongo-sharded
  labels:
    role: mongodb-configserver
spec:
  ports:
  - port: 27019
  clusterIP: None
  selector:
    role: mongodb-configserver
```

- Después de estos archivos vamos a crear los dos necesarios para configurar *mongos*, que actuará como un router dentro del sistema de mongo, como hemos visto en la Sección 2.11.3. El primer archivo, **mongos-deployment.yaml**, tiene el siguiente contenido:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongodb-mongos
  namespace: mongo-sharded
spec:
  replicas: 2
  selector:
    matchLabels:
      role: mongodb-mongos
  template:
    metadata:
      labels:
        role: mongodb-mongos
    spec:
      containers:
      - name: mongodb-mongos
        image: mongo:4.4
        env:
        - name: MONGO_INITDB_ROOT_USERNAME
          valueFrom:
            secretKeyRef:
              name: mongo-credentials
              key: MONGO_INITDB_ROOT_USERNAME
        - name: MONGO_INITDB_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mongo-credentials
              key: MONGO_INITDB_ROOT_PASSWORD
        command:
```

```
      - mongos
      - "--configdb"
      - configReplSet/mongodb-configserver-
      -0.mongodb-configserver:27019,mongodb-
      -configserver-1.mongodb-
      -configserver:27019,mongodb-
      -configserver-2.mongodb-
      -configserver:27019
      - "--bind_ip"
      - "0.0.0.0"
    ports:
    - containerPort: 27017
```

Del anterior archivo es importante entender el contenido del tag de *command*, que será el comando que se ejecutará dentro del contenedor cuando se cree. *Mongos* es el enrutador de MongoDB, *–configdb* indica al enrutador dónde encontrar la configuración del clúster, la cadena que sigue a esta opción es la lista de nodos de servidor de configuración y sus puertos a los que el enrutador debe conectarse para obtener información de configuración. *–bind_ip*, especifica la dirección IP a la que el enrutador de MongoDB estará vinculado. 0.0.0.0 significa que el enrutador aceptará conexiones entrantes desde cualquier dirección IP.

- A continuación creamos el archivo **mongos-service.yaml**:

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb-mongos
  namespace: mongo-sharded
  labels:
    role: mongodb-mongos
spec:
  ports:
  - port: 27017
  selector:
    role: mongodb-mongos
```

- Una vez configurado el servidor y mongos procedemos a crear los archivos necesarios para los shards de la base de datos MongoDB. Nosotros vamos a utilizar 4, y para no repetir en la memoria 4 veces los dos mismos archivos de configuración vamos a usar el término *X* como número. El primer archivo de configuración que debemos crear es: **shardX-statefulset.yaml**. En realidad deberíamos crear cuatro archivos iguales cambiando la x para poner los 4 primeros números (0,1,2 y 3).

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongodb-shardX
  namespace: mongo-sharded
spec:
  serviceName: "mongodb-shardX"
  replicas: 3
  selector:
    matchLabels:
      role: mongodb-shardX
```

```
  template:
    metadata:
      labels:
        role: mongodb-shardX
    spec:
      containers:
      - name: mongodb-shardX
        image: mongo:4.4
        env:
        - name: MONGO_INITDB_ROOT_USERNAME
          valueFrom:
            secretKeyRef:
              name: mongo-credentials
              key: MONGO_INITDB_ROOT_USERNAME
        - name: MONGO_INITDB_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mongo-credentials
              key: MONGO_INITDB_ROOT_PASSWORD
        command:
          - mongod
          - "--shardsvr"
          - "--replSet"
          - shardXReplSet
          - "--bind_ip"
          - "0.0.0.0"
        ports:
        - containerPort: 27018
        volumeMounts:
          - name: mongodb-shard0-data
            mountPath: /data/db
  volumeClaimTemplates:
  - metadata:
      name: mongodb-shardX-data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "rook-ceph-block"
      resources:
        requests:
          storage: 10Gi
```

De este archivo es importante entender que estamos reservando 10 gigabytes de almacenamiento en Rook para cada shard, lo que hará que el clúster en total ocupe unos 40gb aproximadamente y pudiendo extender esta cifra mediante el tag de *storage*. También se vuelven a necesitar las credenciales que hemos creado al principio.

• Para terminar con los archivos creamos a continuación el siguiente: **shardX-service.yaml**. De nuevo tenemos que crear cuatro versiones del mismo archivo.

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb-shardX
  namespace: mongo-sharded
  labels:
```

```
    role: mongodb-shardX
spec:
  ports:
  - port: 27018
  clusterIP: None
  selector:
    role: mongodb-shardX
```

Si todos los archivos se han creado correctamente deberíamos ver los archivos no tachados de la Figura 3.7.

```
estudiante@master:~/mongo$ ls
comanditos.txt                  mongodb-sharded                 shard0-service.yaml     shard2-service.yaml
configserver-service.yaml       mongo-exporter-service.yaml     shard0-statefulset.yaml shard2-statefulset.yaml
configserver-statefulset.yaml   mongos-deployment.yaml          shard1-service.yaml     shard3-service.yaml
mongodb-exporter-deployment.yaml mongos-service.yaml            shard1-statefulset.yaml shard3-statefulset.yaml
```

Figura 3.7: Archivos creados para la configuración de MongoDB en Kubernetes

- Una vez creados los ocho archivos vamos a introducirlos en el entorno de Kubernetes. En vez de aplicar uno a uno cada uno de ellos vamos a crear un fichero que incluya los ocho archivos anteriores para solo tener que ejecutar el comando una vez. Para esto creamos un archivo identico al de la Figura 3.8, y ejecutamos el siguiente comando en la consola:

```
> kubectl apply -k -n mongo-sharded
```

Es importante que el archivo tenga el mismo nombre, ya que el archivo funciona dentro del contexto de Kustomize. Esta es una herramienta en Kubernetes que permite personalizar y modificar recursos de Kubernetes de manera declarativa.

```
estudiante@master:~/mongo$ cat kustomization.yaml
resources:
- configserver-statefulset.yaml
- configserver-service.yaml
- mongos.deployment.yaml
- mongos.service.yaml
- shard0-statefulset.yaml
- shard0-service.yaml
- shard1-statefulset.yaml
- shard1-service.yaml
- shard2-statefulset.yaml
- shard2-service.yaml
- shard3-statefulset.yaml
- shard3-service.yaml
estudiante@master:~/mongo$
```

Figura 3.8: Archivo de Kustomization de Kubernetes para MongoDB

3.4.2. Configuración del clúster de MongoDB

Una vez tenemos todos los archivos correctamente situados en el clúster debemos indicarle al configServer qué shards conformarán la base de datos distribuida. Actualmente tenemos 4 instancias de MongoDB pero cada una de ellas actúa de forma individual. Necesitamos que actúen como Sharding. Lo primero que tenemos que hacer es iniciar los conjuntos de replicación.

```
> kubectl exec -it -n mongo-sharded
mongodb-configserver-0 -- mongo --port 27019

> rs.initiate({
 _id: "configReplSet",
configsvr: true,
members: [
    { _id: 0, host: "mongodb-configserver-0.mongodb-
    configserver:27019" },
    { _id: 1, host: "mongodb-configserver-1.mongodb-
    configserver:27019" },
    { _id: 2, host: "mongodb-configserver-2.mongodb-
    configserver:27019" }
]
});
```

• A continuación vamos a ejecutar los siguientes comandos **para cada shard**. De nuevo debemos cambiar la X por los números correspondientes de cada shard.

```
> kubectl exec -it -n mongo-sharded
mongodb-shardX-0 -- mongo --port 2018

> rs.initiate({
    _id: "mongodb-shardX",
    members: [
    { _id: 0, host: "mongodb-shardX-0.mongodb
    -shard0:27018" },
    { _id: 1, host: "mongodb-shardX-1.mongodb
    -shard0:27018" },
    { _id: 2, host: "mongodb-shardX-2.mongodb
    -shard0:27018" }
]
});

> rs.stepDown(0);
```

Con el último comando de estos forzamos a cada shard del conjunto de replicación a dejar de ser el nodo primario.

• A continuación vamos a configurar el enrutamiento de Sharding. Tenemos que decirle a mongos la forma en la que va a gestionar las direcciones de las replicas, es decir, tenemos que decirle la forma en la que va a enrutar.

–Primero debemos obtener el nombre del Pod asociado a Mongos. Para eso ejecutamos el siguiente comando y obtenemos un resultado similar al de la imagen 3.9. Nos quedamos con el conjunto alfanumérico presente en el pod de mongos.

```
> kubectl get pods -n mongo-sharded
```

–Una vez lo tenemos entonces podemos ejecutar el siguiente conjunto de comandos:

```
> kubectl exec -it -n mongo-sharded
mongodb-mongos-77cb5d8765-mdgqc -- mongo

> sh.addShard("shard0ReplSet/mongodb-shard0-0.mongodb-
shard0:27018,
mongodb-shard0-1.mongodb-shard0:27018,mongodb-
shard0-2.mongodb-shard0:27018");

> sh.addShard("shard1ReplSet/mongodb-shard1-0.mongodb-
```

```
shard1:27018,
mongodb-shard1-1.mongodb-shard1:27018,mongodb-
shard1-2.mongodb-shard1:27018");

> sh.addShard("shard2ReplSet/mongodb-shard2-0.mongodb-
shard2:27018,
mongodb-shard2-1.mongodb-shard2:27018,mongodb-
shard2-2.mongodb-shard2:27018");

> sh.addShard("shard3ReplSet/mongodb-shard3-0.mongodb-
shard3:27018,
mongodb-shard3-1.mongodb-shard3:27018,mongodb-
shard3-2.mongodb-shard3:27018");
```

```
estudiante@master:~/mongo$ kubectl get pods -n mongo-sharded
NAME                                    READY   STATUS    RESTARTS   AGE
grafana-7fb95bcfb9-jmc8x                1/1     Running   0          6d9h
mongodb-configserver-0                  1/1     Running   0          25d
mongodb-configserver-1                  1/1     Running   0          18d
mongodb-configserver-2                  1/1     Running   0          10d
mongodb-exporter-deployment-74dcb4f9bf-64cqq   1/1     Running   0          17d
mongodb-mongos-77cb5d8765-mdgqc         1/1     Running   0          25d
```

Figura 3.9: Pods del namespace de mongo-sharded

Si se han seguido todos los pasos ya estaría MongoDB Sharding correctamente configurado dentro del clúster de Kubernetes. Para poder acceder al servicio vía tunneling desde una máquina externa al clúster solo es necesario ejecutar el siguiente comando:

```
> ssh -L 30159:localhost:30159
estudiante@147.96.81.119
```

3.4.3. Conexión a través de MongoDB Compass y creación de una Sharded Collection

Para acceder al clúster de mongo lo podemos hacer de diversas formas. Nosotros hemos optado por usar MongoDB Compass aunque también se podrá acceder con cualquier otro cliente como puede ser monosh.

Para continuar deberemos tenerlo instalado en nuestro equipo, ejecutarlo y configurar nuestra URI de conexión como mostramos en la figura 3.10 Es importante tener nuestro script .bat ejecutándose con el port-forwading activo al nodo maestro de nuestro clúster ya que sino será imposible acceder a la base de datos como localhost.

Una vez dentro deberemos ver una consola de mongo en la parte de abajo. Allí ejecutaremos los comandos:

```
> use mydb

> sh.enableSharding("mydb")

> db.createCollection("myCollection")
```

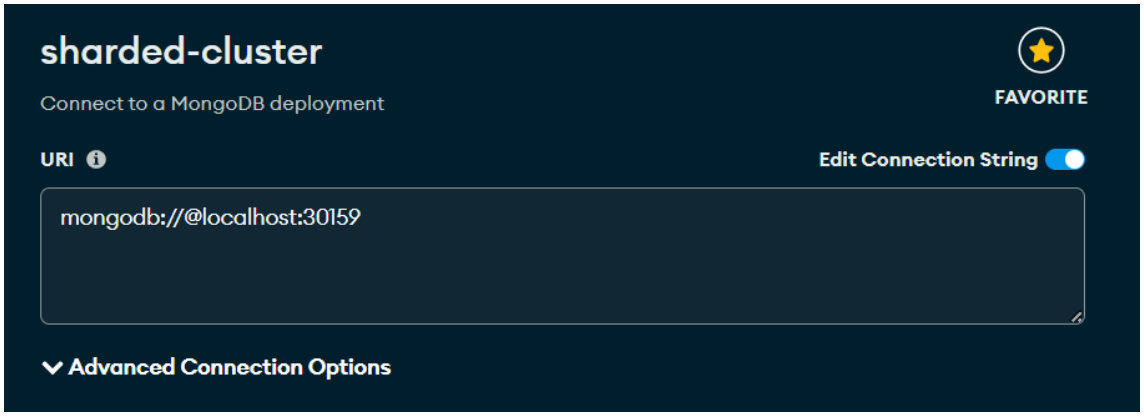


Figura 3.10: Ejemplo conexión al clúster mediante MongoDB Compass

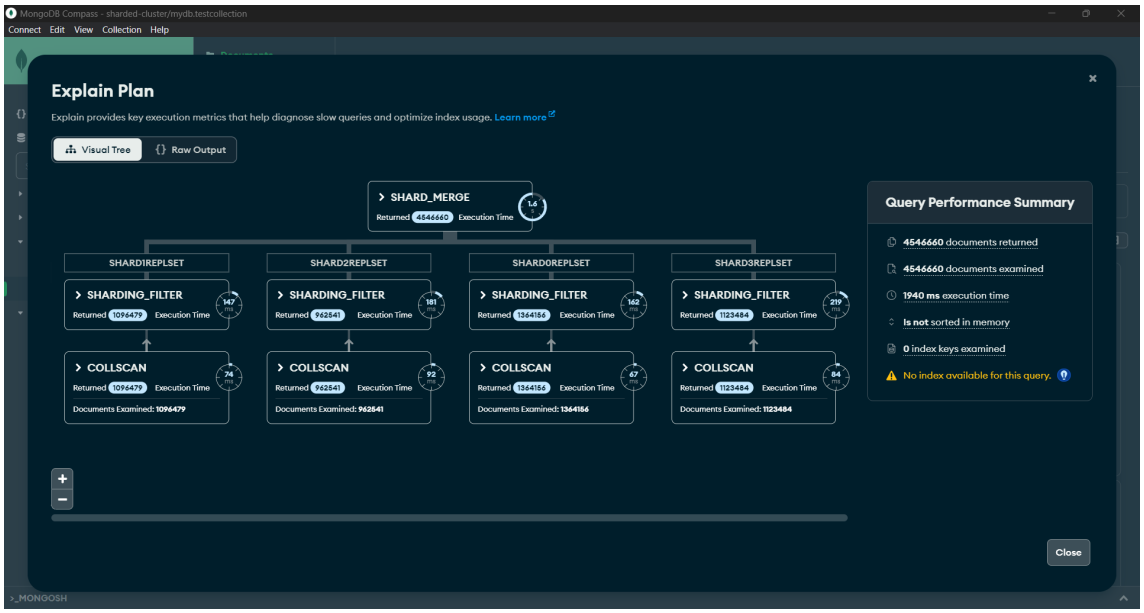


Figura 3.11: Explain Plan de un MongoDB Sharded Cluster

```
> sh.shardCollection("mydb.mycollection",
{ "shardKeyField": 1 })
```

Con esto habremos creado una base de datos llamada *mydb*, habilitado el sharding en la base de datos, creado una colección llamada *mycollection* y habilitado el sharding para la colección por el primer campo que generalmente es el id para fragmentar la colleción. Podremos realizar además un explain plan en el cual comprobaremos efectivamente que el sharding funciona correctamente. Nosotros insertamos varios datasets en la colección de un total de 8gb y 4 millones de documentos para realizar el explain plan y ver que efectivamente como muestra la Figura 3.11 se ejecuta con rapidez.

3.5. Spark on Kubernetes

Una vez tenemos nuestro cluster de MongoDB desplegado y accesible como servicio, procederemos a montar Spark que será, como ya hemos explicado en 2.12, el framework de computación del clúster.

Diferenciamos en la Sección 2.12.3 varias formas de trabajar con Apache Spark en Kubernetes. Entre ellas hemos optado por implementar dos: Spark en modo Cluster Nativo de Kubernetes y Spark usando el Spark Operator para Kubernetes.

Comenzamos con la ayuda de la documentación oficial de Spark on Kubernetes la cual nos especifica que para trabajar con Spark en nuestro clúster deberemos instalarlo de forma local. Nosotros nos serviremos de momento de imágenes ya preconfiguradas disponibles en DockerHub y en Google Cloud proporcionadas por el propio Apache.

3.5.1. Despliegue de spark-on-kubernetes

• Instalación de Spark:

–Para instalar Spark necesitaremos ejecutar los siguientes comandos:

```
> wget https://.../spark-3.4.1-bin-hadoop3.tgz
> tar xvf spark-3.4.1-bin-hadoop3.tgz
> sudo mv spark-3.2.1-bin-hadoop3.2/ /opt/spark
```

–Con estos comandos tendremos los binarios de Spark en local. Ahora le diremos a bash para poder lanzar trabajos con spark-submit en qué directorio estan sus binarios.

```
> nano vim ~/.bashrc
> export SPARK_HOME=/opt/spark
> export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
```

Una vez hecho esto ya podremos lanzar trabajos de Spark tanto en modo cluster como en modo standalone. Es muy importante crear de antemano un espacio donde se crearán los pods y las credenciales necesarias para ello.

• Configuración de RBAC y Namespace: Crearemos un Namespace llamado spark-jobs, así como una ServiceAccount, con un rol de administrador del clúster, ya que deberá poder comunicarse con la API para que esta pueda crear y eliminar los pods a petición.

```
> kubectl create namespace spark-jobs
> kubectl create serviceaccount spark -n spark-jobs
> kubectl create clusterrolebinding
spark-role --clusterrole=cluster- admin --
serviceaccount=default:spark -n spark-jobs
```

• Con estas credenciales ya estaremos autorizados y podremos lanzar el comando spark-submit:

```
> ./spark-submit \
 --master k8s://https://master:6443 \
```

```
--deploy-mode cluster \
--name spark-pi \
--class org.apache.spark.examples.SparkPi \
--conf spark.executor.instances=2 \
--conf spark.kubernetes.container.image=apache/
spark-py:
latest \
--conf spark.kubernetes.container.image.pullPolicy
=IfNotPresent \
--conf spark.kubernetes.authenticate.driver.
serviceAccountName=spark \
--conf spark.kubernetes.authenticate.executor.
serviceAccountName=spark \
--conf spark.kubernetes.namespace=spark-jobs \
local:///opt/spark/examples/jars/
spark-examples_2.12-3.4.0.jar 100
```

Deberemos especificar mediante flags primero el API endpoint de Kubernetes como maestro. Este endpoint si se olvida se puede consultar en cualquier momento con el comando *kubectl get config*

Luego especificamos el modo de despliegue a cluster con –deploy-mode, con –class el nombre de la clase del trabajo y luego con –conf configuraremos las opciones de spark. Es importante especificar el namespace mediante:
spark.kubernetes.authenticate.driver.serviceAccountName y
spark.kubernetes.namespace los nombres del ServiceAccount y el Namespace respectivamente. Con *.container.image* especificaremos el nombre de la imagen de spark con la que se ejecutarán los trabajos de Spark y con *.container.image.pull* le diremos que si la imagen no está presente en local en los Nodos haga el pull de ella.

Con esto ya habremos lanzado nuestro primer job con spark-submit.

Ahora procederemos con el despliegue del operador de spark-on-K8s que nos permitirá lanzar sparkApplications mediante la descripción de un archivo .yaml con un abanico de configuraciones más amplio. Las tanto instrucciones de instalación como los ejemplos se pueden obtener del repositorio oficial del operador Platform y Contributors (2023) que además es propiedad de la plataforma de Google Cloud.

Esta instalación la haremos mediante Helm por lo que necesitaremos de antemano añadir el repositorio del operador, luego actualizar la lista de repositorios para finalmente podes desplegarlo:

```
> helm repo add spark-operator
https://googlecloudplatform.github.io/
spark-on-k8s-operator

> helm repo update

> helm install spark-operator spark-operator/
spark-operator
--namespace spark-operator
--create-namespace
--set sparkJobNamespace=spark-jobs
--set enableBatchScheduler=true
--set enableWebhook=true
--set enableShuffle=true
```

El despliegue tardará alrededor de uno o dos minutos. Un vez desplegado ya podremos ejecutar sparkApplications. Estos son una serie de CRDs que se crean junto con el operador. Como ya explicamos en la Sección 2.8, estos CDRs son unas recursos personalizadas que en nuestro caso definen el tipo de trabajo de Spark que va a ejecutar el operador al hacer el apply del .yaml. Distinguiremos scheduledSparkApplications que son trabajos que se pueden planificar para que se lanzen cada cierto tiempo y sparkApplications que son trabajos de spark que se ejecutan únicamente cuando se aplica el .yaml.

A continuación mostraremos dos ejemplos de .yaml de estos dos recursos:

```yaml
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: pyspark-pi
  namespace: spark-jobs
spec:
  type: Python
  pythonVersion: "3"
  mode: cluster
  image: "gcr.io/spark-operator/spark-py:v3.1.1"
  imagePullPolicy: Always
  mainApplicationFile: local:///opt/spark/exa/src/main/python/pi.py
  sparkVersion: "3.1.1"
  restartPolicy:
    type: OnFailure
    onFailureRetries: 3
    onFailureRetryInterval: 10
    onSubmissionFailureRetries: 5
    onSubmissionFailureRetryInterval: 20
  driver:
    cores: 1
    coreLimit: "1200m"
    memory: "512m"
    labels:
      version: 3.1.1
    serviceAccount: spark
  executor:
    cores: 1
    instances: 1
    memory: "512m"
    labels:
      version: 3.1.1
```

```yaml
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: ScheduledSparkApplication
metadata:
  name: spark-pi-scheduled
  namespace: spark-jobs
spec:
  schedule: "@every 5m"
  concurrencyPolicy: Allow
  template:
    type: Scala
```

```
mode: cluster
image: "gcr.io/spark-operator/spark:v3.1.1"
imagePullPolicy: Always
mainClass: org.apache.spark.examples.SparkPi
mainApplicationFile: "local:///opt/spark/examples/jars/spark
-examples_2.12-3.1.1.jar"
sparkVersion: "3.1.1"
restartPolicy:
  type: Never
driver:
  cores: 1
  coreLimit: "1200m"
  memory: "512m"
  labels:
    version: 3.1.1
  serviceAccount: spark
executor:
  cores: 1
  instances: 1
  memory: "512m"
  labels:
    version: 3.1.1
```

Estos se podrán desplegar con el comando:

```
> kubectl apply -f spark-pi-scheduled.yaml
-n spark-jobs
```

Cabe mencionar que la imagen de Spark que usan los pods que despliega el operador proviene también del Artifact Registry de Google Cloud, su propio repositorio de contenedores el cual también es público.

3.5.2. Configuración de spark-on-kubernetes operator

Una vez ya tenemos el operador de spark-on-kubernetes funcionando nos interesaría aprovechar las ventajas que nos ofrece con respecto al comando spark-submit. De ellas nos han llamado la atención sobre todo tres: la monitorización de los trabajos con prometheus, el cual desplegaremos y configuraremos más adelante en la Sección 3.7, acceder a los logs de los trabajos en un History Server de Spark y aprovechar el shuffling con la posibilidad de autoescalar el número de pods ejecutores con DynamicAllocation. Procederemos a ir configurándolas en orden.

• Spark History Server: Monitorizar y acceder a los logs de los trabajos de Spark así como saber el tiempo que tardan en ejecutarse las diferentes fases es una posibilidad muy interesante que nos permite este operador. Los pods como ya hablamos en el Apartado 2.8, si no se les especifica un almacenamiento persistente, tienen un almacenamiento efímero. Debido a esto los logs que se generan al ejecutarse los trabajos se pierden una vez finalizados, ya sea por que se completan exitosa u ocurre algún error inesperado. Acceder al clásico puerto 4040 del history server de los pods se convierte entonces en todo un reto. Habría que obtener las ips del pod del driver de spark y de sus ejecutores y acceder en tan sólo un intervalo de 30 segundos a sus

logs para poder saber el resultado del trabajo antes de que se eliminen. Es por ello que investigando en internet y gracias a ejemplos para almacenamiento en la nube como Amazon S3 de Escura (2019) decidimos aprobechar nuestro sistema de almacenamiento persistente y distribuido para implentar esta funcionalidad. La clave es desplegar un Spark History Server con un PVC de la clase CephFS y aprovechar el tipo de acceso de escritura/lectura ReadWriteMany o RWX para que los pods puedan escribir en mismo fichero sus logs al mismo tiempo y que el History Server los pueda leer.

Vamos a desplegarlo:

• Creación del PVC de Cephfs: Crearemos un Persistent Volume Claim de la clase Cephfs especificando el modo ReadWriteMany.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: spark-eventlogs-pvc
  namespace: spark-jobs
spec:
  storageClassName: rook-cephfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

```
> kubectl apply -f spark-eventlogs-pvc.yaml
-n spark-jobs
```

• Despliegue del History Server: Desplegamos el .yaml con el PVC anterior montado como volumen en el pod.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spark-history-server
  namespace: spark-jobs
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spark-history-server
  template:
    metadata:
      labels:
        app: spark-history-server
    spec:
      containers:
        - name: spark-history-server
          image: docker.io/apache/spark:v3.3.0
          command: ["/opt/spark/sbin/start-history-server.sh"]
          env:
```

```
        - name: SPARK_NO_DAEMONIZE
          value: "false"
        - name: SPARK_HISTORY_OPTS
          value: "-Dspark.history.fs.logDirectory=file:/mnt/events"
      ports:
        - containerPort: 18080
      volumeMounts:
        - name: spark-eventlog-volume
          mountPath: "/mnt/events"
      securityContext:
        runAsUser: 0
  volumes:
    - name: spark-eventlog-volume
      persistentVolumeClaim:
        claimName: spark-eventlogs-pvc
```

```
> kubectl apply -f spark-history-server.yaml
-n spark-jobs
```

• Exposición del Pod como Service: Deberemos exponer el Spark History Server como un servicio de tipo Nodeport para que sea accesible desde el exterior.

```
apiVersion: v1
kind: Service
metadata:
  name: spark-history-server-nodeport
  namespace: spark-jobs
spec:
  type: NodePort
  selector:
    app: spark-history-server
  ports:
    - protocol: TCP
      port: 18080
      targetPort: 18080
       nodePort: 30458
```

```
> kubectl apply -f spark-history-server-nodeport.yaml
-n spark-jobs
```

• Deberemos configurar los SparkApplications para que tanto el pod del driver como el pod de los executors escriban sus logs en el mismo PV que deberán tener montados. Esto lo haremos mediante las label *driver.volumeMounts* y *executor.volumeMounts* así como decirle a los pods que se ejecuten con usuario root para poder acceder al volumen con *driver.securityContext.runAsUser=0* y *executor.securityContext.runAsUser=0*. Al aplicar el archivo y lanzar la SparkApplication podremos acceder al History Server con el port-forwarding activado a la dirección localhost:30458 para ver el detalle del Spark-Job.

```
spec:
  ...
  template:
```

```
...
sparkConf:
  "spark.eventLog.enabled": "true"
  "spark.eventLog.dir": "/mnt/events"
volumes:
  - name: spark-eventlog-volume
    persistentVolumeClaim:
      claimName: spark-eventlogs-pvc
driver:
  ...
  volumeMounts:
    - name: spark-eventlog-volume
      mountPath: "/mnt/events"
  securityContext:
    runAsUser: 0
executor:
 ...
  volumeMounts:
    - name: spark-eventlog-volume
      mountPath: "/mnt/events"
  securityContext:
    runAsUser: 0
```

Ya tenemos nuestro History Server en ejecución para monitorizar los logs de forma accesible y persistente que se verá como el de la Figura 3.12

Podremos ver además detalles de una Stage y de los Executors como muestran las Figuras 3.13 y 3.14 respectivamente

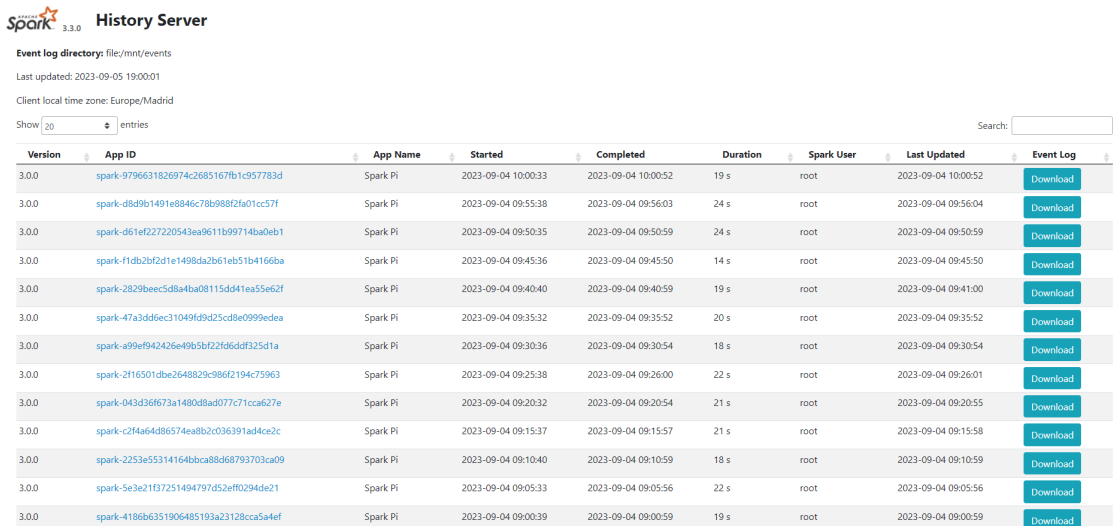


Figura 3.12: Pantalla principal del History Server

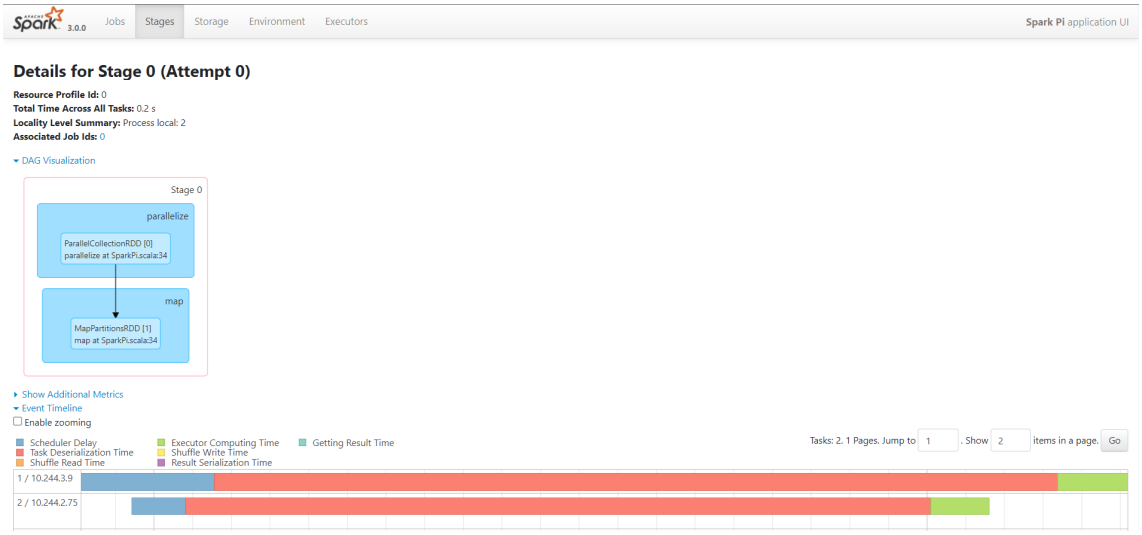


Figura 3.13: Ejemplo del detalle de una Stage en History Server

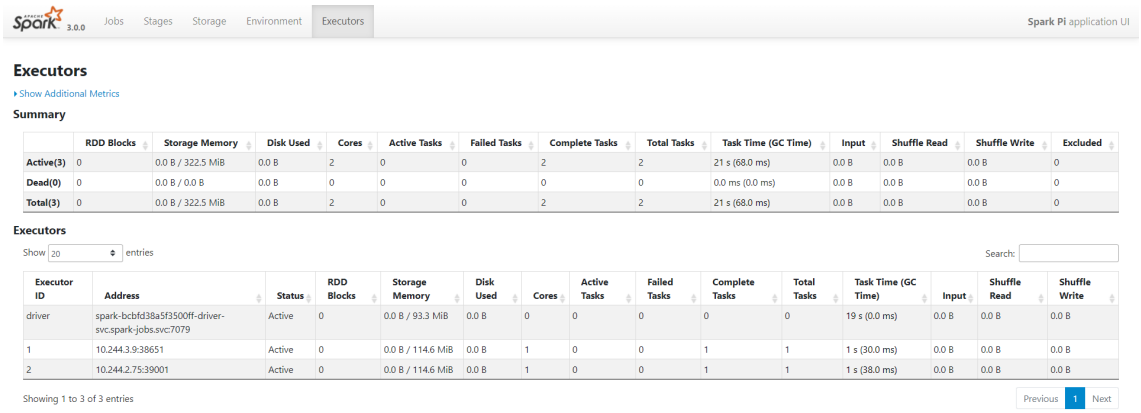


Figura 3.14: Ejemplo de las métricas de los Executors en History Server

Ahora vamos a configurar nuestros SparkApplications para que usen dynamicAllocation. Al activar esta funcionalidad se podrá autoescalar el número de executors del Spark Job en función de si hay mucha o poca demanda para así garantizar un mejor rendimiento en caso de que se requieran más executors, o si son ligeros ahorrar recursos así como mejorar también el tiempo de reagrupación de datos.

Para habilitarlo es necesario que al desplegar el chart estén los flags *-set enableWebhook=true* y *–set enableShuffle=true* de antemano como hicimos anteriormente. Ahora únicamentemos especificar en la especificación del SparkApplication que se va a usar DynamicAllocation así como un número mínimo y un número máximo de executors que pueden llegar a tener los jobs:

```
spec:
    ...
    template:
        ...
        dynamicAllocation:
          enabled: true
          initialExecutors: 2
          minExecutors: 2
          maxExecutors: 10
```

Con esto ya habremos terminado la configuración de Spark-on-Kubernetes y podemos seguir con la instalación de la siguiente herramienta para el procesamiento y el análisis de dato: Jupyter.

3.6. JupyterHub

Jupyter, como hemos explicado en la Sección 2.12.5, y en concreto Jupyter Notebook, es una aplicación web que facilita la creación y el intercambio de documentos de programas literarios. En estos documentos se incluyen numerosas celdas en las que cada una puede contener texto real o código. Para poder integrar Spark y Mongo ejecutar los trabajos de Spark lo más recomendable es usar este entorno interactivo. En esta sección vamos a explicar como implementar un servidor de JupyterHub en nuestro clúster de Kubernetes que permitirá a los usuarios trabajar en Jupyter de forma concurrente.

3.6.1. Despliegue de JupyterHub

• Lo primero que debemos hacer es configurar el archivo necesario para solicitar al clúster el almacenamiento persistente y reservar un espacio al entorno de Jupyter. El objetivo es que los usuarios puedan trabajar de forma aislada unos de otros así como conservar sus notebooks entre las diferentes sesiones. Comenzaremos creando el siguiente archivo de configuración yaml al que vamos a llamar *Jupyter-pvc-claim.yaml* y posteriormente lo aplicamos usando el comando *kubectl -f apply jupyter-pvc-claim.yaml*:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: jupyterhub-pvc
    namespace: spark-jobs
spec:
    accessModes:
        - readWriteOnce
    resources:
        requests:
            storage: 5Gi
    storageClassName: rook-ceph-block
```

Es importante especificar en el archivo anterior el nombre de la clase de almacenamiento que queremos utilizar, que en este caso es rook-ceph-block. Esta elección del sistma de almacenamiento en bloque la realizamos porque asegura que ningun otro servicio puede acceder a la información almacenada aquí. En los siguientes apartados veremos como para cada usuario que acceda a jupyter utiliza un pod distinto y de esta manera nos aseguramos que un usuario no pueda acceder a los datos de otro.

• Después de crearlo y aplicalo procedemos a instalar el propio Jupyter en nuestro clúster.

–Primero añadimos el repositorio. Para esto volvemos a utilizar Helm:

```
> helm repo add jupyterhub
https://hub.jupyter.org/helm-chart/
> helm repo update
```

–A continuación creamos el archivo de configuración (config.yaml) necesario para poder instalar jupyter. Es necesario introducir el token secreto del proxy[1] para poder acceder a los pods:

```
proxy:
  secretToken: "{introducir aqui token del proxy}"
singleuser:
  serviceAccountName: spark
  storage:
    type: "static"
    static:
      pvcName: "jupyterhub-pvc"
      subPath: 'jupyter-{username}-{servername}'
    extraVolumes:
    - name: sparkmagic-config-volume
      configMap:
        name: sparkmagic-config
    extraVolumeMounts:
    - name: sparkmagic-config-volume
      mountPath: /home/jovyan/.sparkmagic/
  image:
    name: jupyter/pyspark-notebook
    tag: python-3.10.11
```

[1]Se puede conseguir introduciendo el comando: kubectl exec -n spark-jobs <nombre-del-pod-proxy>- jupyterhub token <username>

```
  networkPolicy:
    allowedIngressPorts:
      - 8889
      - 43221
      - 27017
hub:
  db:
    pvc:
      storageClassName: rook-ceph-block

  extraConfig:
    00-first-config: |
        c.KubeSpawner.service_account = 'spark'
proxy:
  service:
    type: NodePort
    nodePorts:
      http: 30080
```

–Finalmente instalamos Jupyter en el clúster con el siguiente comando:

```
> helm install jupy-release jupyterhub/jupyterhub
--name-space spark-jobs -f config.yaml
```

• Para terminar con el proceso de configuración e instalación tenemos que asegurar que Spark se utilice dentro del cluster y no en local, es decir para que el pod del usuario pueda interactuar con el cluster tiene que ser expuesto como servicio. Con el siguiente archivo de configuración conseguimos esto aunque únicamente con un usuario, que en este caso será *estudiante*. Para cada usuario sería necesario crear este archivo con su información correspondiente. En la Sección 5.2.2 comentamos como podría automatizarse este proceso para no resultar tan tedioso. De nuevo aplicamos el archivo utilizando *kubectl -f apply jupyter-pod-service.yaml*:

```
apiVersion: v1
kind: Service
metadata:
  name: jupyter-estudiante-service
  namespace: spark-jobs
spec:
  selector:
    app: jupyterhub
    hub.jupyter.org/username: estudiante
  ports:
    - name: sparkmaster
      protocol: TCP
      port: 8889
      targetPort: 8889

    - name: https
      protocol: TCP
      port: 8888
      targetPort: 8888

    - name: sparkpartition
      protocol:
      port: 29414
```

```
    targetPort: 29414
  type: ClusterIP
```

Del anterior archivo es importante entender que se especifica el nombre de usuario y que se introducen 3 puertos. Estos son los puertos que necesita tener el pod abiertos para poder comunicarse con los pods ejecutores de Spark. El primero de ellos, el 8889, es con el que manda los trabajos a los ejecutores. El segundo, el 8888, es el que permite la conexión http vía web y el tercero, el 29414, es necesario para particionar los datos de los trabajos, al querer aprovechar el procesamiento paralelizado.

• Una vez todo está instalado y configurado podemos acceder vía tunneling desde nuestro equipo externo ejecutando el siguiente comando en la consola de comandos:

```
> ssh -L 30080:localhost:30080
estudiante@147.96.81.119
```

Para comprobar que todo ha ido correctamente, podemos buscar en la barra de busqueda del navegador la siguiente dirección: "localhost:30080". Una vez veremos una imagen identica a la Figura 3.15 en la que nos pedirá unas credenciales de inicio de sesion. Si el usuario no se ha creado entonces el propio clúster creará un pod con la información de ese usuario y su bloque de almacenamiento correspondiente, del que tan solo ese usuario tendrá acceso. Si el usuario ya había sido creado previamente tan solo accederá a él con su información almacenada.

La Figura 3.16 muestra el proceso de carga del propio servidor jupyter del usuario. Una vez termina accederíamos a la pantalla principal de Jupyter que luce como la Figura 3.17. En la parte de la izquierda tenemos los archivos que tenemos guardados. En este caso hemos introducido algunos cuadernos de Spark, que hemos ejecutado con éxito, para mostrar cómo se vería la pantalla principal siendo verdaremente utilizada.

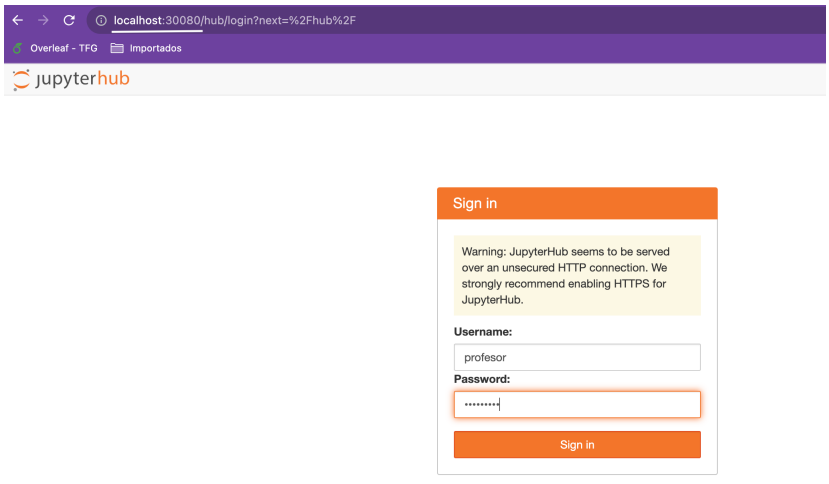


Figura 3.15: Inicio de sesión en Jupyter

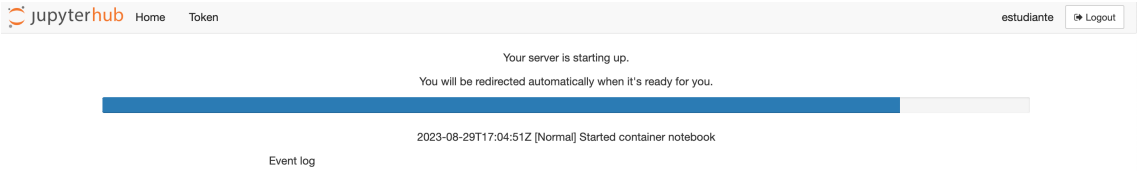


Figura 3.16: Pantalla de carga en Jupyter

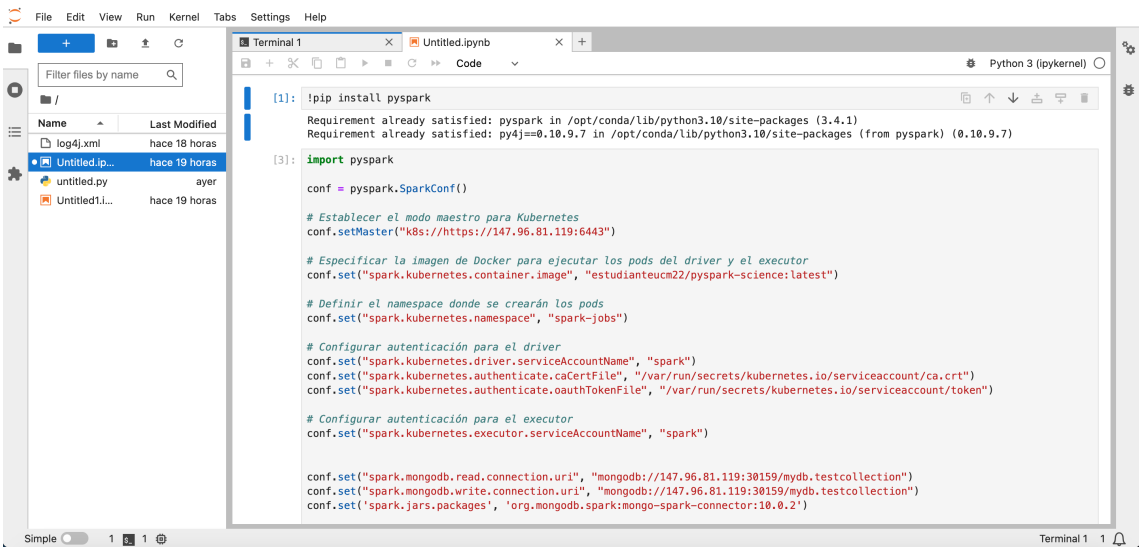


Figura 3.17: Pantalla principal de Jupyter

3.6.2. Configuración e integración con MongoDB y Spark

Con el servidor de Jupyter ya desplegado el siguente paso sería aprovechar la flexibilidad que nos ofrece su compatibilidad con Python para hacer análisis de datos en un notebook. La idea es usar como origen de datos nuestro sharded cluster de MongoDB con un framework de la potencia de Apache Spark; extraer esos datos de manera paralela, analizarlos y guardar los resultados de vuelta en MongoDB o símplemente mostrarlos en el notebook. Para configurar nuestro entorno nos inspiramos en trabajos similares como el de Oakes (2019)

Para ello usaremos el API de Python para Spark, Pyspark, que se definió en la Sección 2.12.1. Por tanto antes de empezar a extraer datos de MongoDB deberemos configurar y crear una SparkSession de Pyspark la cual servirá como punto de entrada a la funcionalidad de SQL Spark y se utiliza para crear DataFrames y Datasets. Si queremos usar un SparkContext para poder crear RDD y poder paralelizarlo podremos obtenerlo igualmente de la SparkSession.

Vamos a ver con exactitud y detalle cómo configurar esta SparkSession:

Instalación de librerías: Deberemos instalar la librería de Pyspark para poder lanzar trabajos desde Jupyter. La instalaremos si se lo en una celda del cuaderno con:

```
> !pip install pyspark
```

• Preparación de la imagen de Spark: Para sacar partido a las posibilidades que nos ofrece Kubernetes vamos a preparar una imagen de spark que contenga algunas librerías útiles para el análisis de datos y el conector de MongoDB el cual es necesario para los executors se puedan conectar a un clúster de MongoDB como el nuestro. Vamos a partir de una imagen base de spark-py. Es necesario que esta imagen tenga la misma versión de Python que el del pod de Jupyter ya que sino Spark se verá incapaz de devolver el resultado de los trabajos a Jupyter. En concreto nosotros tenemos un cuaderno de Jupyter que usa la versión 3.10 de Python al igual que la imagen base de spark-py.

El siguiente Dockerfile será sobre el que haremos el build de nuestra imagen que se llamará pyspark-science:latest

```
FROM apache/spark-py:latest

USER root

RUN apt-get update && \
    apt-get install -y python3-pip && \
    pip3 install numpy pandas scipy scikit-learn matplotlib
       seaborn

ARG MONGO_CONNECTOR_VERSION=3.0.1
ADD https://repo1.maven.org/maven2/org/mongodb/spark/mongo-
   spark-connector_2.12/${MONGO_CONNECTOR_VERSION}/mongo-spark-
   connector_2.12-${MONGO_CONNECTOR_VERSION}.jar /usr/local/
   spark/jars/
```

```
ENV PYSPARK_SUBMIT_ARGS= \
"--jars /usr/local/spark/jars/mongo-spark-connector_2.12- \
${MONGO_CONNECTOR_VERSION}.jar pyspark-shell"

CMD ["/bin/bash"]
```

Esta imagen podemos construir y subir al repositorio con los comandos:

```
> docker build ./Dockerfile -t estudianteucm22/
pyspark-science:latest
> docker push -t estudianteucm22/
pyspark-science:latest
```

• Configuración del Cluster de Kubernetes: Utilizamos nuestro control-plane igual que hicimos en el Apartado 2.12.2 como maestro de spark especificándolo con la opción conf.setMaster. Hemos especificado nuestra imagen personalizada *estudianteucm22/ pyspark-science:latest* con *spark.kubernetes.container.image* y con *spark.kubernetes. namespace* le indicaremos el namespace, en concreto spark-jobs, sobre el que se lanzarán los executors.

• Autenticación:

Establecemos la autenticación indicándo a Spark que tanto para el driver como para el executor deben lanzarse con una ServiceAccount con rol de administrador en Kubernetes. Esto garantiza que los pods de Spark tienen los permisos necesarios para operar dentro de nuestro cluster de Kubernetes, utilizando cuentas de servicio y tokens. Nosotros ya tenemos esa ServiceAccount configurada y creada en el namespace spark-jobs.

• Integración con MongoDB:

URI de Conexión: Configuramos las URI para interactuar con MongoDB, especificando la colección testcollection en la base de datos mydb. Conector de MongoDB: Añadimos el conector para permitir que Spark se comunique e interactúe con MongoDB. Esta URI puede ser tanto localhost:30159 como 147.96.81.119:30159 ya que la base de datos está sobre un servicio nodeport como se indicó en la Sección 3.4.2.

• Configuración del Driver:

Establecemos spark.driver.host a *jupyter-estudiante-service.spark-jobs.svc.cluster.local*, lo que indica que el proceso del driver de Spark se encuentra en un servicio específico dentro de Kubernetes. Con spark.driver.port y spark.blockManager.port, definimos los puertos 8889 y 29414 respectivamente, para las conexiones entrantes al driver y para la gestión de bloques de datos. Es importante que estos puertos esten abiertos en el pod como indicamos en la Sección 3.6.1 Además, al configurar *spark.driver. bindAddress* a "0.0.0.0", permitimos que el driver escuche en todas las interfaces de red, facilitando su accesibilidad en cualquier entorno.

• Creación de SparkSession:

Después de todas las configuraciones previas, instanciamos una SparkSession. Esta sesión, denominada "MongoDBIntegration", se convierte en nuestro principal punto de acceso para todas las operaciones de Spark, permitiéndonos trabajar con DataFrames e interactuar con bases de datos y otras fuentes.

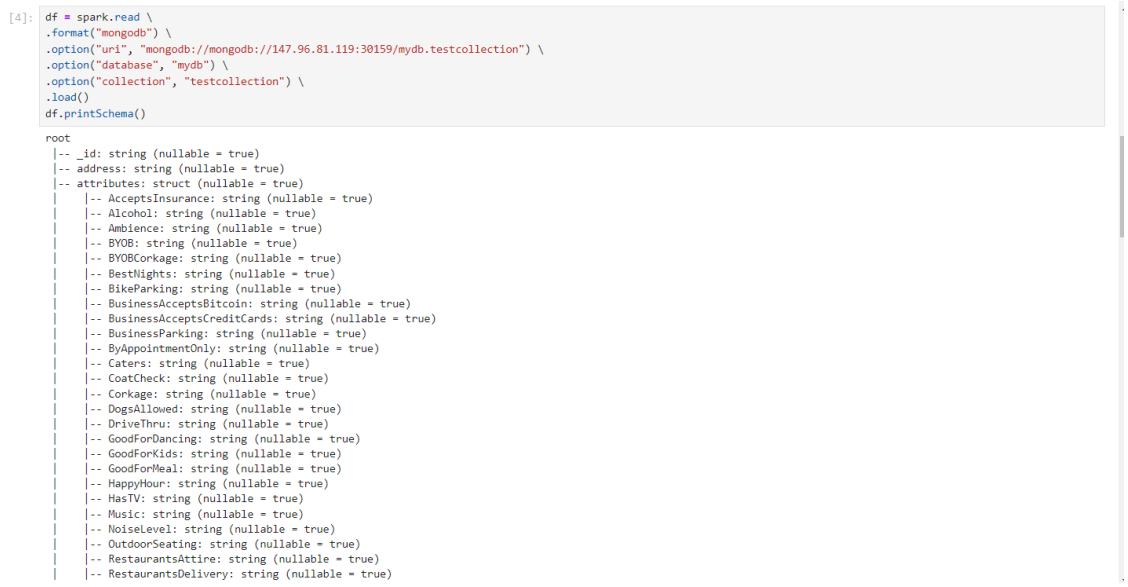
```
import pyspark

conf = pyspark.SparkConf()

# Establecemos el modo maestro para Kubernetes
conf.setMaster("k8s://https://147.96.81.119:6443")

# Especificamos la imagen de Docker para ejecutar los pods del driver y el executor
conf.set("spark.kubernetes.container.image", "estudianteucm22/pyspark-science:latest")

# Definimos el namespace donde se crearán los pods
conf.set("spark.kubernetes.namespace", "spark-jobs")

# Configuramos autenticación para el driver
conf.set("spark.kubernetes.driver.serviceAccountName", "spark")
conf.set("spark.kubernetes.authenticate.caCertFile", "/var/run/secrets/kubernetes.io/serviceaccount/ca.crt")
conf.set("spark.kubernetes.authenticate.oauthTokenFile", "/var/run/secrets/kubernetes.io/serviceaccount/token")

# Configuramos autenticación para el executor
conf.set("spark.kubernetes.executor.serviceAccountName", "spark")

conf.set("spark.mongodb.read.connection.uri", "mongodb://147.96.81.119:30159/mydb.testcollection")
conf.set("spark.mongodb.write.connection.uri", "mongodb://147.96.81.119:30159/mydb.testcollection")
conf.set('spark.jars.packages', 'org.mongodb.spark:mongo-spark-connector:10.0.2')

# Configuramos el hostname y el puerto del driver
conf.set("spark.driver.host", "jupyter-estudiante-service.spark-jobs.svc.cluster.local")
conf.set("spark.driver.port", "8889")
conf.set("spark.driver.bindAddress", "0.0.0.0")
conf.set("spark.blockManager.port","29414")

from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .appName("MongoDBIntegration") \
    .config(conf=conf) \
    .getOrCreate()
```

Figura 3.18: Construcción de una SparkSession

Nos quedará un notebook como el de la Figura 3.18

Ahora ya podemos probar la conexión con MongoDB como muestra la Figura 3.19, la creación de RDDs y algunas operaciones como las de la Figura 3.20.

Por último queremos remarcar que el acceso y uso de este servidor de JupyterHub se puede hacer también desde el IDE del ordenador de un cliente como puede ser DataSpell de JetBrains como muestra la figura 3.21 que ofrece más funcionalidades que la versión de navegador de Jupyter, como sincronización con repositorios, administración de bases de datos como nuestro clúster de MongoDB, gestión de paquetes, entre otras muchas.



```
[4]: df = spark.read \
     .format("mongodb") \
     .option("uri", "mongodb://mongodb://147.96.81.119:30159/mydb.testcollection") \
     .option("database", "mydb") \
     .option("collection", "testcollection") \
     .load()
     df.printSchema()

root
 |-- _id: string (nullable = true)
 |-- address: string (nullable = true)
 |-- attributes: struct (nullable = true)
 |    |-- AcceptsInsurance: string (nullable = true)
 |    |-- Alcohol: string (nullable = true)
 |    |-- Ambience: string (nullable = true)
 |    |-- BYOB: string (nullable = true)
 |    |-- BYOBCorkage: string (nullable = true)
 |    |-- BestNights: string (nullable = true)
 |    |-- BikeParking: string (nullable = true)
 |    |-- BusinessAcceptsBitcoin: string (nullable = true)
 |    |-- BusinessAcceptsCreditCards: string (nullable = true)
 |    |-- BusinessParking: string (nullable = true)
 |    |-- ByAppointmentOnly: string (nullable = true)
 |    |-- Caters: string (nullable = true)
 |    |-- CoatCheck: string (nullable = true)
 |    |-- Corkage: string (nullable = true)
 |    |-- DogsAllowed: string (nullable = true)
 |    |-- DriveThru: string (nullable = true)
 |    |-- GoodForDancing: string (nullable = true)
 |    |-- GoodForKids: string (nullable = true)
 |    |-- GoodForMeal: string (nullable = true)
 |    |-- HappyHour: string (nullable = true)
 |    |-- HasTV: string (nullable = true)
 |    |-- Music: string (nullable = true)
 |    |-- NoiseLevel: string (nullable = true)
 |    |-- OutdoorSeating: string (nullable = true)
 |    |-- RestaurantsAttire: string (nullable = true)
 |    |-- RestaurantsDelivery: string (nullable = true)
```

Figura 3.19: Ejemplo de conexión y creación de un df con datos de MongoDB

```
[8]: sc = spark.sparkContext
     rdd = sc.parallelize(range(1000), 10)
     r = sum(range(1000))
     abs(rdd.sumApprox(1000) - r) / r < 0.05

[8]: True

[9]: print(abs(rdd.sumApprox(1000) - r) / r < 0.05)

     True

[11]: # Create a distributed data set to test the session.
      t = sc.parallelize(range(10))

      # Calculate the approximate sum of values in the dataset
      r = t.sumApprox(3)
      print('Approximate sum: %s' % r)

      Approximate sum: 45.0

[13]: import numpy as np
      lst=np.random.randint(0,10,20)
      A=sc.parallelize(lst)

[14]: type(A)

[14]: pyspark.rdd.RDD

[15]: A.collect()

[15]: [1, 5, 2, 0, 7, 9, 3, 1, 9, 0, 3, 0, 7, 5, 3, 2, 6, 5, 3, 9]

[16]: A.glom().collect()

[16]: [[1, 5, 2, 0, 7, 9, 3, 1, 9, 0], [3, 0, 7, 5, 3, 2, 6, 5, 3, 9]]
```

Figura 3.20: Ejemplo de uso de un SparkContext

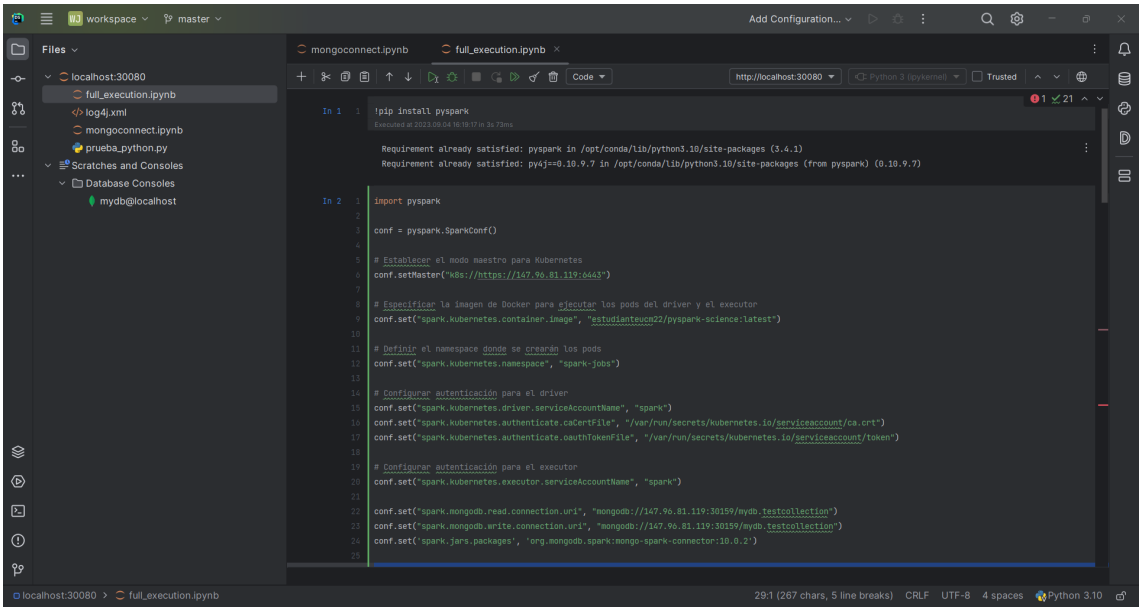


Figura 3.21: Ejemplo de un IDE externo conectado al servidor de Jupyter

3.7. Dashboards. Grafana y Prometheus

Llegados a este punto, con el cluster operativo y la base de MongoBD funcionando y aceptando consultas e inserciones, un aspecto muy interesante para el administrador es la posibilidad de poder monitorizarlo todo. Como usuarios no es relevante ver cómo está funcionando el cluster siempre y cuando funcione, pero si eres el administrador tanto de la base de datos como del propio cluster es requisito casi imprescindible poder observar como se está comportando siguiendo distintas métricas. Por poner un ejemplo, al estar usando sharding mongodb una situación fundamental a monitorizar es que todos los shards estén funcionando correctamente o que el equilibrado de carga está funcionando como debería.

Para poder observar todos estos comportamientos vamos a utilizar Grafana, como hemos mencionado en la Sección 2.13. En este apartado vamos a mencionar como lo hemos instalado, configurado y el proceso que hemos seguido para poder exportar la información de mongo en un formato reconocible por grafana, que en este caso es prometheus.

- El primer paso que hay que seguir es instalar grafana en nuestro cluster.
  - Añadimos el repositorio de helm de grafana:

```
> helm repo add grafana
https://grafana.github.io/helm-charts
> helm repo update
```

  - Instalamos grafana en el clúster. Es muy importante ver la salida del siguiente comando, ya que ofrece otro comando necesario para ver la contraseña que luego tendremos que utilizar al iniciar sesión en grafana. La salida deja claramente diferenciado ese comando.

```
estudiante@master:~$ kubectl get services -n mongo-sharded
NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)           AGE
grafana               NodePort    10.107.38.89     <none>        80:31363/TCP      5d13h
```

Figura 3.22: Servicio grafana en Kubernetes y su puerto asociado

Para instalar Grafana hemos utilizado nodeport, por lo que luego para acceder al servicio de grafana deberemos usar tunneling usando ese puerto.

```
> helm install grafana grafana/grafana
--set service.type=NodePort
```

– Como no tenemos preferencia por ningún puerto dentro de los disponibles en nodeport, tenemos que buscar el puerto asignado. Aplicasmos el siguiente comando (especificando el namespace en el que se ha desplegado el servicio, si no es el mismo para todos. En nuestro caso es mongo-sharded).

En la Figura 3.22 podemos ver que el puerto asociado es el 31363

```
> kubectl get services -n mongo-sharded
```

• A continuación mediante tunneling vamos a acceder al servicio desde fuera del cluster. En nuestra consola de comandos ejecutamos el siguiente comando:

```
> ssh -L 5000:147.96.81.119:31363
estudiante@147.96.81.119
```

• Una vez dentro nos solicita usuario y contraseña. El usuario será admin y la contraseña la obtenida por el comando mencionado antes. Una vez se han introducido las credenciales puedes cambiar la contraseña y crear nuevos usuarios con distintos permisos. Dentro de grafana podemos ver en el lateral izquierdo un menú desplegable que muestre las distintas opciones que podemos elegir. La que nos interesa es la de crear un dashboard.

Antes de continuar con grafana debemos hacer un inciso fundamental, y es que grafana no permite coger información directamente de mongo, no de forma gratuita al menos, pero existe prometheus. La clave es poder exportar la información del servicio de Mongo a Prometheus, y una vez exportada en Prometheus poder acceder a ella a través de grafana.

3.7.1. MongoDB exporter y Prometheus

Como hemos dicho, para poder recopilar metricas desde grafana es necesario exportar los datos de Mongo. En este contexto aparece el exporter de Percona, que será el que utilicemos para poder realizar la exportación.

• Para desplegar el exporter primero debemos descargarlo del repositorio oficial. Nosotros hemos instalado la versión 37.

```
> wget https://github.com/percona/mongodb_exporter/
releases/download
/v0.37.0/mongodb_exporter-0.37.0.linux-amd64.tar.gz
```

```
> tar xvzf mongodb_exporter-0.37.0.linux-amd64.tar.gz
```

```
> cd mongodb_exporter-0.37.0.linux-amd64.tar.gz
```

• Para poder utilizar el exporter, es necesario definir un usuario con permisos de administrador sobre la base de datos que queremos exportar. Con el siguiente comando creamos un usuario con el rol de monitor del cluster, y sobre la base "mydb", que fue la que utilizamos de prueba, se permite el rol de lector. Antes de crearlo debemos acceder al propio servicio de mongo del clúster para crear el usuario.

```
> kubectl exec -it -n mongo-sharded
mongodb-mongos-77cb5d8765-mdgqc -- mongo

> db.createUser({user: "admin",pwd: "password",roles:
[{ role: "clusterMonitor", db: "admin" },
{ role: "read", db: "mydb" }]})
```

• A continuación vamos a lanzar el exporter. En esta parte contemplamos e implementamos múltiples posibilidades, como lanzarlo como Pod usando distintos métodos como Docker o Podman, que es otra herramienta de gestión de contenedores, pero uno de los problemas que tenían es que expiraba y los pods no se mantenían ejecutandose. Debido a esto finalmente nos decantamos por lanzarlo como servicio. Para ello lo lanzamos como Deployment, que es el método más eficaz si tan solo lo necesitamos para recopilar métricas. El siguiente archivo de configuración es el que hemos utilizado nosotros pero según los puertos que se utilicen o el nombre de la base de datos que se quiera exportar puede cambiar, por lo que es importante saber que estamos haciendo en cada momento. La última linea por ejemplo incluye nuestro nombre de la base "mydb", pero, en casos de réplicas, puede tener otro nombre. El puerto 9216 sin embargo se va a mantener siempre igual ya que es el propio del exporter de Mongo.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: mongodb-exporter-deployment
    labels:
        app: mongodb-exporter
spec:
    replicas: 1
    selector:
        matchLabels:
            app: mongodb-exporter
    template:
        metadata:
            labels:
                app: mongodb-exporter
        spec:
            containers:
            - name: mongodb-exporter
              image: docker.io/percona/mongodb_exporter:0.39
              ports:
              - containerPort: 9216
              args:
              - "--mongodb.uri=mongodb://147.96.81.119:30159"
              - "--discovering-mode"
              - "--web.listen-address=:9216"
```

```
- "--web.telemetry-path=/metrics"
- "--mongodb.direct-connect"
- "--compatible-mode"
- "--collect-all"
- "--mongodb.collstats-colls=mydb"
```

• Aplicamos el anterior archivo de configuración con el siguiente comando:

```
> kubectl apply -f mongodb-exporter-deployment.yaml
```

• Ahora tan solo nos queda definir el servicio que va a exponer al deployment anterior. Para realizarlo definimos un archivo de configuración yaml, al que hemos llamado "mongo-exporter-service.yaml", de la siguiente manera:

```
apiVersion: v1
kind: Service
metadata:
    name: mongodb-exporter-service
    namespace: mongo-sharded
spec:
    selector:
        app: mongodb-exporter
    ports:
        protocol: TCP
        port: 9216
        targetPort: 9216
```

• De nuevo y por último, tenemos que aplicar el anterior archivo de configuración en Kubernetes con el siguiente comando:

```
> kubectl apply -f mongodb-exporter-service.yaml
```

Si todo ha ido correctamente deberíamos ver una línea similar a la última que aparece en la Figura 3.23 si ejecutamos el comando *kubectl get services -n mongo-sharded*.

```
estudiante@master:~$ kubectl get services -n mongo-sharded
NAME                       TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)           AGE
grafana                    NodePort    10.107.38.89     <none>        80:31363/TCP      10d
mongodb-configserver       ClusterIP   None             <none>        27019/TCP         12d
mongodb-exporter-service   ClusterIP   10.102.146.179   <none>        9216/TCP          4d23h
```

Figura 3.23: Algunos de los servicios de Kubernetes en el namespace mongo-sharded

Una vez tenemos el exporter bien configurado tenemos que preparar el entorno de prometheus para poder recopilar las métricas extraidas del exporter a un formato valido en grafana.

• Primero creamos el archivo config de Prometheus, **rometheus-config.yaml**. Dentro de los trabajos incluimos al exporter de mongo, con su clusterIP asociado (se puede ver cuál es con el comando kubectl get services -n mongo-sharded). También incluimos Spark.

```
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: prometheus-config
  namespace: monitoring
data:
  prometheus.yml: |
    global:
      scrape_interval:     15s
      evaluation_interval: 15s
    alerting:
      alertmanagers:
      - static_configs:
        - targets:
    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
        - targets: ['localhost:9090']
      - job_name: 'mongodb-exporter'
        static_configs:
        - targets: ['10.102.146.179:9216']
      - job_name: 'spark'
        static_configs:
        - targets: ['spark-pi-metrics.spark-jobs:8090']
```

Lo aplicamos dentro del contexto de Kubernetes con el siguiente comando, especificando un namespace identificativo, que en nuestro caso es monitoring.

• A continuación configuramos el archivo de configuración para lanzarlo como deployment, siendo **prometheus-deployment.yaml** el nombre de dicho fichero.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus
  namespace: monitoring
  labels:
    app: prometheus
spec:
  replicas: 1
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  selector:
    matchLabels:
      app: prometheus
  template:
    metadata:
      labels:
        app: prometheus
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "9090"
    spec:
      securityContext:
        runAsUser: 0
      containers:
```

```
    - name: prometheus
      image: prom/prometheus
      args:
        - '--storage.tsdb.retention=6h'
        - '--storage.tsdb.path=/prometheus'
        - '--config.file=/etc/prometheus/prometheus.yml'
      ports:
      - name: web
        containerPort: 9090
      volumeMounts:
      - name: prometheus-config-volume
        mountPath: /etc/prometheus
      - name: prometheus-storage-volume
        mountPath: /prometheus
    restartPolicy: Always

    volumes:
    - name: prometheus-config-volume
      configMap:
          defaultMode: 420
          name: prometheus-config

    - name: prometheus-storage-volume
      persistentVolumeClaim:
          claimName: pvc-prometheus-data
```

Del anterior archivo cabe destacar algunos de los tags que utilizamos. El tag de *strategy* indica la estrategia de actualización durante las implementaciones. Al establecer *maxSurge* a 1 definimos que solo se pueda crear un pod adicional durante la instalación mientras que con *maxUnavailable* nos aseguramos la constante disponibilidad del servicio ya que establece el número máximo de pods que pueden desactivarse durante la actualización. El tag de *annotation* indica cómo debe Prometheus recopilar las métricas. Se indica el puerto en el que el pod expone las metricas, 9090, y se pone a true el valor de *scrape* para indicar que sí debe recolectarlas. En la sección de los contenedores que se ejecutarán en los pods creados por este deployment se especifica la imagen de Docker utilizada y se pasan distintos argumentos. El primero establece la retención de datos de Prometheus en 6 horas, el segundo especifica la ubicación en el sistema de archivos donde Prometheus almacenará sus datos de métricas y el tercero indica la ubicación del archivo de configuración de Prometheus. Por último, se definen dos volúmenes de almacenamiento. El segundo de ellos será el que almacene la información extraída de Prometheus.

• Por último solo nos queda definir un archivo de configuración, que será **prometheus -service.yaml**. De nuevo volvemos a usar NodePort para exponer nuestro servicio de Prometheus, usando el puerto 30000.

```
apiVersion: v1
kind: Service
metadata:
    name: prometheus-service
    namespace: monitoring
    annotations:
        prometheus.io/scrape: 'true'
```

```
        prometheus.io/port:    '9090'
spec:
    selector:
        app: prometheus
    type: NodePort
    ports:
    - port: 8080
      targetPort: 9090
      nodePort: 30000


  ...

    global:
  scrape_interval: 15s
  evaluation_interval: 15s

alerting:
  alertmanagers:
    - static_configs:
        - targets:
scrape_configs:
  - job_name: "prometheus"
    static_configs:
      - targets: ["localhost:9090"]

  - job_name: 'mongodb-exporter'
    static_configs:
      - targets: ['10.102.146.179:9216']
```

Como a lo largo de toda la memoria hace falta aplicar dichos archivos en el sistema de Kubernetes. Para esto aplicamos los siguientes comandos:

```
> kubectl apply -f prometheus-config.yaml
> kubectl apply -f prometheus-deployment.yaml
> kubectl apply -f prometheus-service.yaml
```

3.7.2. Configuración del dashboard de Grafana

A continuación vamos a ver la configuración final de grafana para poder tener un dashboard atractivo y útil, para finalizar con la monitorización del clúster y en específico de la base de datos. Para poder realizarlo introducimos en el clúster una base de datos de mongo de aproximadamente 3GB, para poder mostrar capturas del dashboard que hemos configurado.

Primero de todo debemos configurar una fuente de datos, que será la que usará el dashboard como recolector de métricas. Como se puede ver en la Figura 3.24 Prometheus espera una dirección a la que poder acudir para recopilar la información. Usamos el puerto 30000 ya que hemos asignado ese puerto al servicio de Prometheus. Se pueden configurar más variables como el periodo de refresco para la recopilación de métricas pero hemos dejado los valores por defecto.

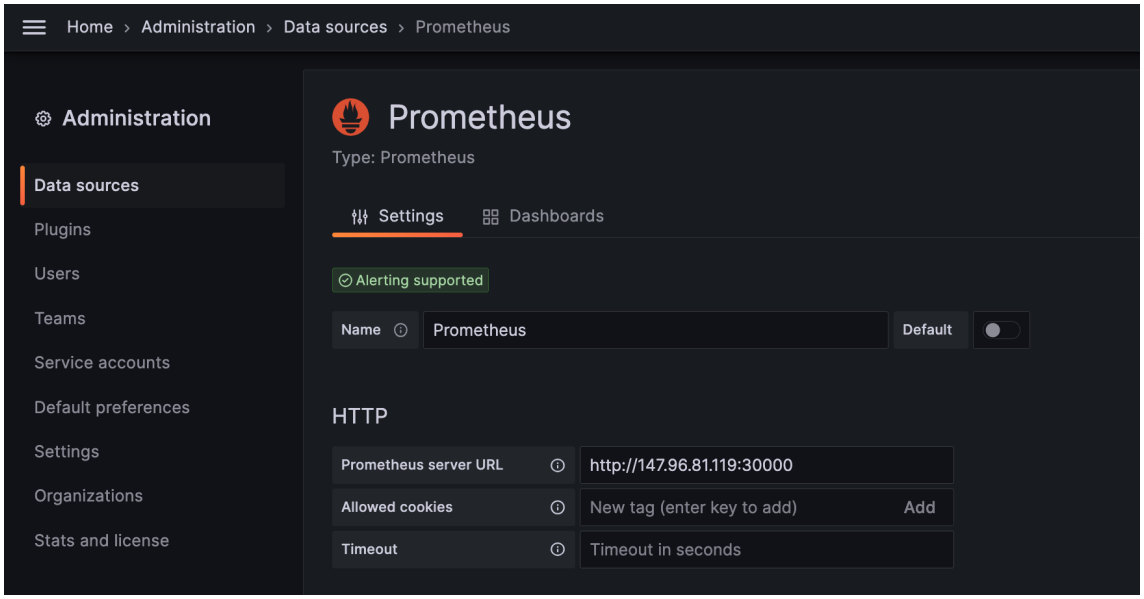


Figura 3.24: Configuración del data source de Prometheus en Grafana

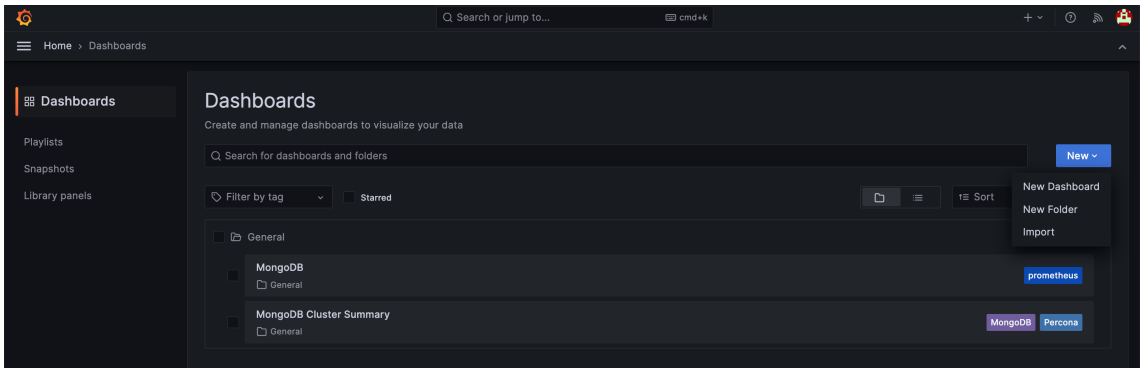


Figura 3.25: Ventana inicio de dashboards en Grafana

Una vez el data source de Prometheus está configurado es hora de trabajar en el propio dashboard. En el mismo menú desplegable de la izquierda aparece la opción de dashboards. Dentro podemos ver los dashboard que ya tenemos o podemos crear uno nuevo, como aparece en la parte izquierda de la Figura 3.25.

Lo creamos vacío aunque se pueden importar dashboards predeterminados creados por la comunidad, aunque presentan un inconveniente y es que muchas veces el formato de las métricas no tiene por qué ser el mismo y es probable que no aparezca ninguna en los propios paneles, por lo que es aconsejable crear uno de nuevo.

En esta pestaña aparece en el centro la opción de añadir visualización y si lo pulsamos nos llevará a la Figura 3.26. Aquí podemos ver todas las opciones que grafana ofrece para poder visualizar las distintas métricas de múltiples formas posibles.

En la imagen, en el primer circulo podemos encontrar la sección de queries. Aquí se introduce la consulta que queremos realizar para calcular métricas. Como se puede ver abajo a la izquierda aparece un menú desplegable con todas las distintas posibilidades, cómo el número de operaciones de escritura, de lectura, el tamaño libre

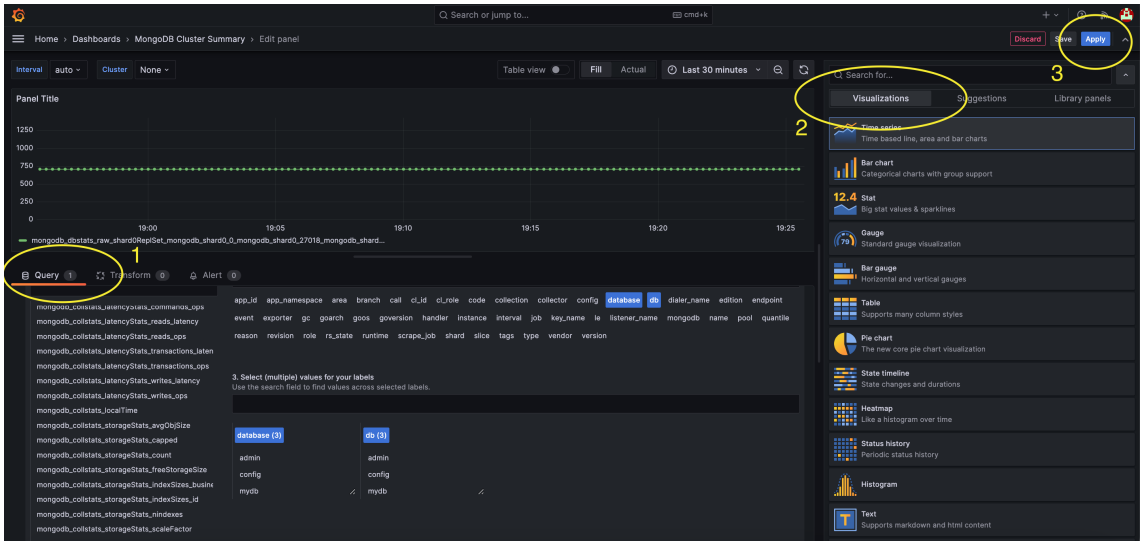


Figura 3.26: Ventana inicio de dashboards en Grafana

disponible o la media de tamaño de los objetos, entre muchísimas otras métricas. También se puede seleccionar las bases de datos sobre las que quieres calcular la información, ya que puede haber más de una base de datos. En esta misma imagen, en el segundo círculo vemos las distintas opciones de visualización que dispone grafana. Se pueden mostrar los datos usando un histograma, un diagrama de barras o un gráfico de líneas entre muchos otros. Para poder guardar el panel tan solo hay que darle al botón presente en el tercer círculo de la imagen.

Después de analizar las distintas métricas y configurar los paneles para intentar formar el dashboard más atractivo e interesante posible hemos elaborado el dashboard de la Figura 3.27. Como se puede apreciar hemos incluido algunos paneles que pueden ser de mucha ayuda para los administradores. Los paneles que hemos preparado incluyen los siguientes:

- **Operaciones**: Hemos incluido dos gráficos que muestran el número de operaciones de lectura y de escritura que han tenido lugar en la base de datos.
- **Información del sharding de Mongo**: Los números iniciales muestrán información relevante como el número de shards, o el tamaño de la base de datos o si los distintos chunks están equlibrados.
- **Chunks en cada shard**: Una gráfico que muestra los chunks que hay en cada shard, al igual que el mínimo, el máximo y la media de chunks que ha habido.
- **Datos de almacenamiento**: En la esquina inferior izquierda mostramos en número total de objectos, el tamaño del Index y la cantidad de espacio libre disponible

También se puede configurar el intervalo de refresco necesario al igual que el intervalo de tiempo que se muestra en las gráficas.

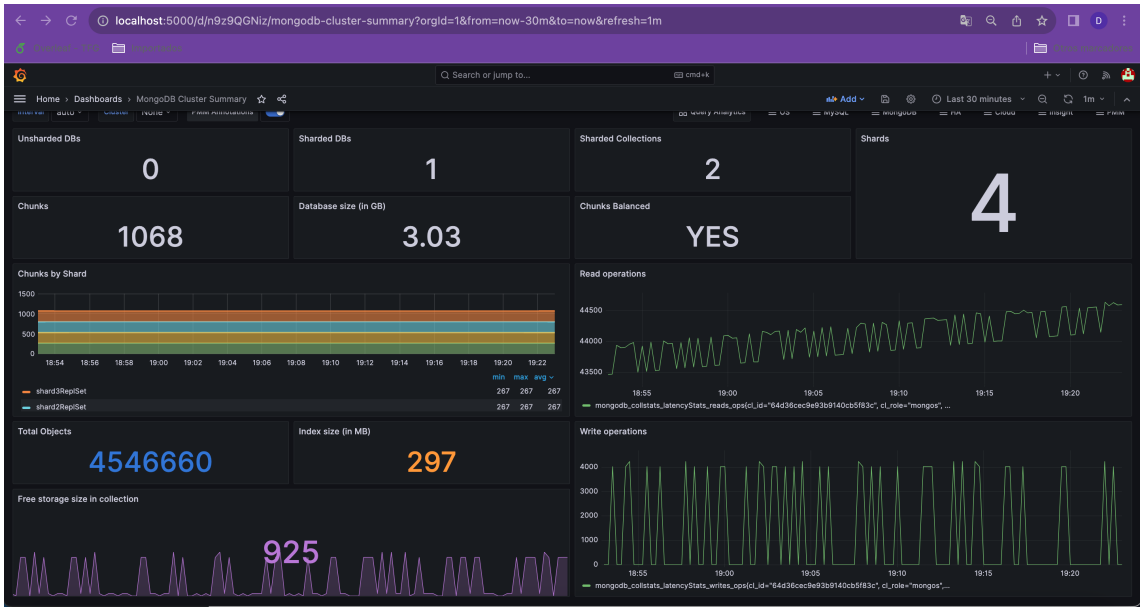


Figura 3.27: Dashboard final de Grafana

3.7.3. Configuración de alertas

Otra posibilidad que ofrece Grafana y que puede resultar de especial interés es la configuración de alertas. Grafana permite al administrador poder configurar distintos avisos para que le notifiquen si cierta métrica ha cambiado de valor o si no llega a cierto parámetro deseado. Esto puede ser muy útil para recibir alertas si un nodo del clúster ha dejado de funcionar, aunque en nuestro caso no hemos usado Grafana para monitorizar el clúster como tal, o si algún shard de la base de datos de MongoDB está trabajando mucho más que el resto o si está albergando mucha mayor cantidad de los datos de almacenamiento que los demás.

En la barra de la izquierda aparece la sección de alertas. Una vez ahí podemos configurar las reglas de alertas que deseemos. Un ejemplo de configuración de alertas se puede ver en la Figura 3.28. Se puede apreciar que la forma de seleccionar las métricas es idéntica a cuándo configurabamos el propio dashboard. Lo que cambia es la necesidad de introducir un umbral para saber cuando Grafana tiene que enviar la notificación o no. En la imagen hemos establecido que cuando el número de shards baje de 4 notifique al administrador.

Lo interesante de esta subsección es saber que existe esta posibilidad y facilitar al administrador la tarea de monitorizar la base de datos MongoDB.

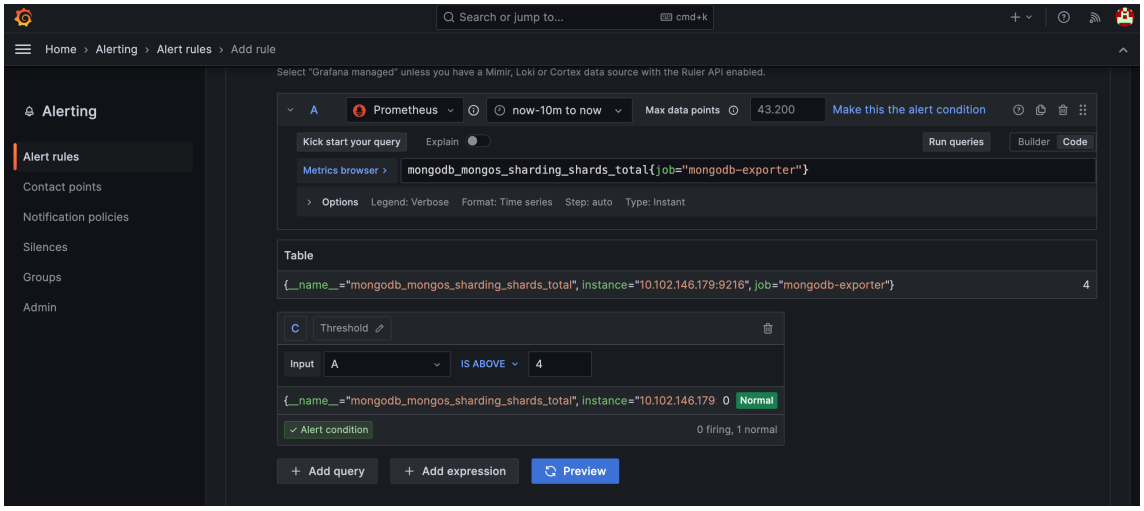


Figura 3.28: Ejemplo de configuración de alertas en Grafana

3.7.4. Kubernetes dashboard

Si bien en el apartado anterior hemos explicado como configurar grafana para poder monitorizar la base de datos mongo ahora vamos a explicar otro método de monitorización que lo vamos a utilizar para poder controlar el propio clúster de Kubernetes en sí. Para esto vamos a instalar el propio dashboard de kubernetes, en vez de utilizar grafana. La decisión de no utilizar el mismo recurso, aparte de para otorgar mayor variedad al proyecto, es que el propio dashboard de kubernetes permite al administrador poder modificar ciertos aspectos del clúster directamente desde ahí, sin necesidad de acudir a comandos de bajo nivel. En esta sección vamos a explicar como se ha instalado el servicio y algunos de los aspectos de monitorización que hemos configurado.

Para instalarlo hay que seguir los siguientes pasos:

• Primero, agregamos el repositorio de Helm del dashboard de Kubernetes. El proceso se puede hacer sin usar helm pero debido a la facilidad de uso y al hecho de que se ha utilizado en la mayoria de instalaciones posibles seguimos utilizando helm.

```
> helm repo add kubernetes-dashboard
https://kubernetes.github.io/dashboard/
> helm repo update
```

• A continuación instalamos el dashboard, especificando que queremos instalarlo como un servicio NodePort. Después de ejecutarlo deberia salir el servicio implementado como en la Figura 3.29, aunque no tiene que ser con el mismo nodeport (ya que selecciona uno aleatoriamente de entre 30000 y 32768):

```
> helm install kubernetes-dashboard
kubernetes-dashboard/kubernetes-dashboard
--set service.type=NodePort
```

• Una vez instalado ya podremos acceder al servicio mediante tunneling, pero antes de ello es necesario crear un usuario con permisos especiales y obtener un

```
estudiante@master:~$ kubectl get service kubernetes-dashboard -n monitoring
NAME                   TYPE       CLUSTER-IP    EXTERNAL-IP   PORT(S)         AGE
kubernetes-dashboard   NodePort   10.99.19.76   <none>        443:30084/TCP   52s
```

Figura 3.29: Servicio del dashboard de Kubernetes

token para poder utilizar el dashboard.

–Lo primero que tenemos que hacer es crear un archivo de configuración, que vamos a llamar *dashboard-adminuser.yaml*. En él tenemos que escribir lo siguiente, y después aplicarlo utilizando "kubectl -f apply dashboard-adminuser.yaml":

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: monitoring
```

–Después tenemos que enlazar este usuario al clúster, para esto creamos un nuevo archivo de configuración al que llamaremos *dashboard-cluster-bind.yaml*. Al igual que antes debemos escribir lo siguiente y volver a aplicarlo usando "kubectl -f apply dashboard-cluster-bind.yaml":

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

–Finalmente para conseguir el token tan solo tenemos que ejecutar el siguiente comando. Es importante guardar el token para poder utilizarlo en el dashboard, ya que pasado cierto tiempo el servicio te volverá a solicitar el token. Hemos configurado el comando para que el token no expire hasta pasados 10 años.

```
> kubectl -n monitoring create token admin-user
--duration=8760h
```

• Finalizada toda la configuración pertinente ahora tan solo tenemos que acceder al servicio mediante tunneling. Para lograrlo ejecutamos el siguiente comando en la terminal de nuestro equipo:

```
> ssh -L 30084:localhost:30084
estudiante@147.96.81.119
```

Después de ejecutarlo escribimos en cualquier navegador la siguiente direccion: *https://localhost:30084*.

• Una vez dentro introducimos el token previamente guardado e iniciamos sesión. Si todo ha ido correctamente en la pantalla se debería ver una imagen como la de

la Figura 3.30. Como se puede apreciar en la barra lateral de la izquierda aparecen multitud de componentes que se pueden monitorizar como los pods o los persistent volume claims. En la pantalla de inicio se pueden ver distintos gráficos circulares incluyendo el número de despliegues o de pods, indicando cuales han sido fallidos o cuales están siendo ejecutados en el mismo momento. También se puede seleccionar el namespace del que queremos obtener la distinta información, lo que puede ser realmente útil y más para un caso como el nuestro en el que para cada contexto (MongoDB, Spark, monitorización...) hemos usado un namespace distinto.

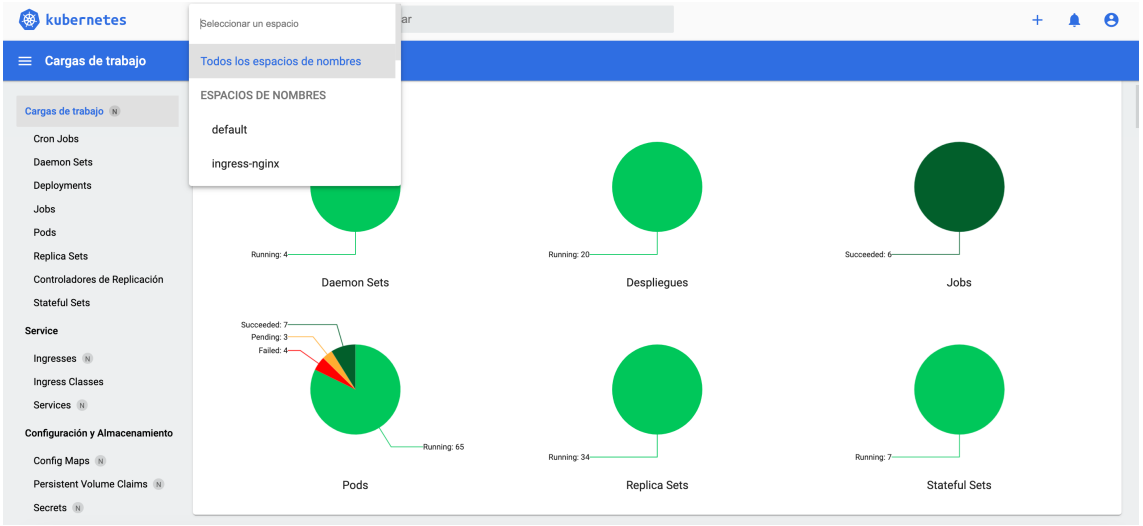


Figura 3.30: Pantalla de Inicio por defecto del dashboard de Kubernetes

Algunas ventanas que podemos ver en el dashboard son las de las Figuras 3.31 y 3.32. En la primera de estas podemos ver los pods que se han creado en el clúster, incluyendo su estado actual, el namespace en el que fueron creados, el nodo en el que están siendo ejecutados, la fecha en la que fueron creados o la imagen que están utilizando. La segunda imagen muestra el estado de los nodos del clúster de kubernetes. Aquí podemos ver los límites de CPU de cada uno de ellos así como la capacidad de la CPU y de la memoria o los límites de ésta última. También se puede ver cuantos pods está albergando cada uno de ellos y el estado de los nodos. En caso de que alguno esté fallando se podría ver aquí.

Otro aspecto interesante que ofrece este dashboard y que Grafana no implementa es la capacidad de poder modificar los recursos del clúster desde su propia aplicación. Como vemos en la Figura 3.33, desde este dashboard tenemos la opción de editar un recurso sin la necesidad de usar los comandos de Kubernetes.

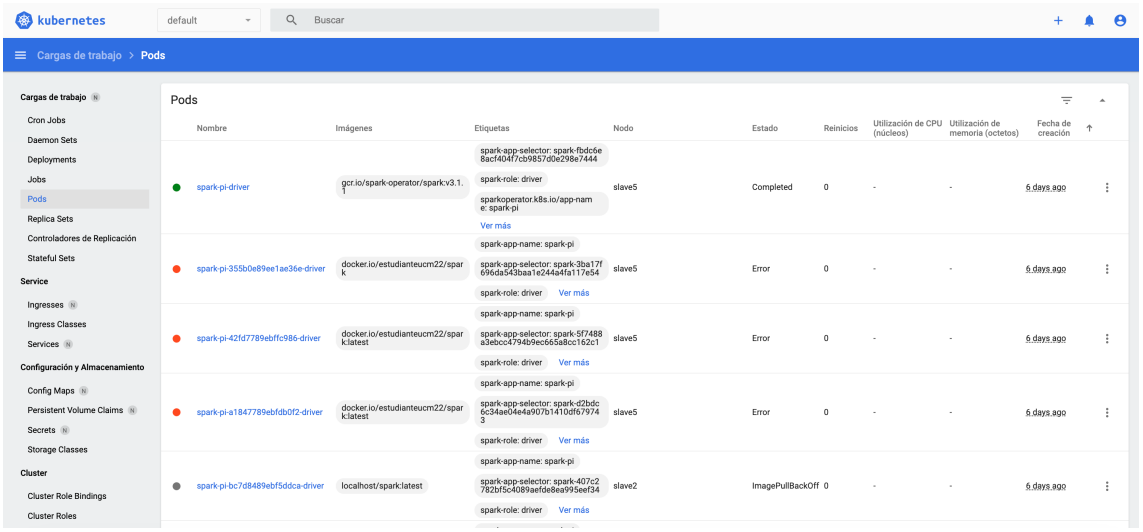


Figura 3.31: Pantalla de pods del dashboard de Kubernetes

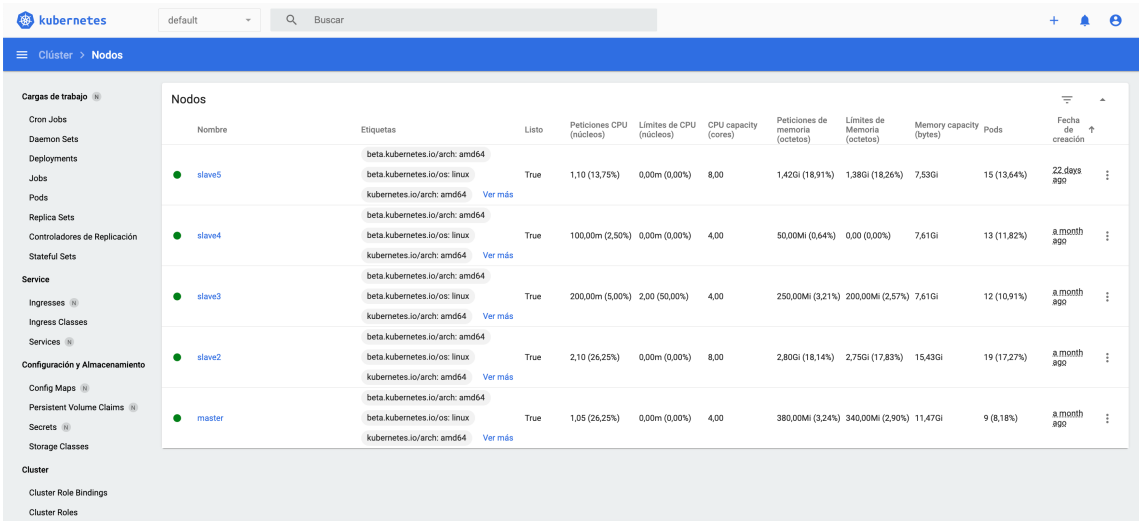


Figura 3.32: Pantalla de nodos del clúster del dashboard de Kubernetes

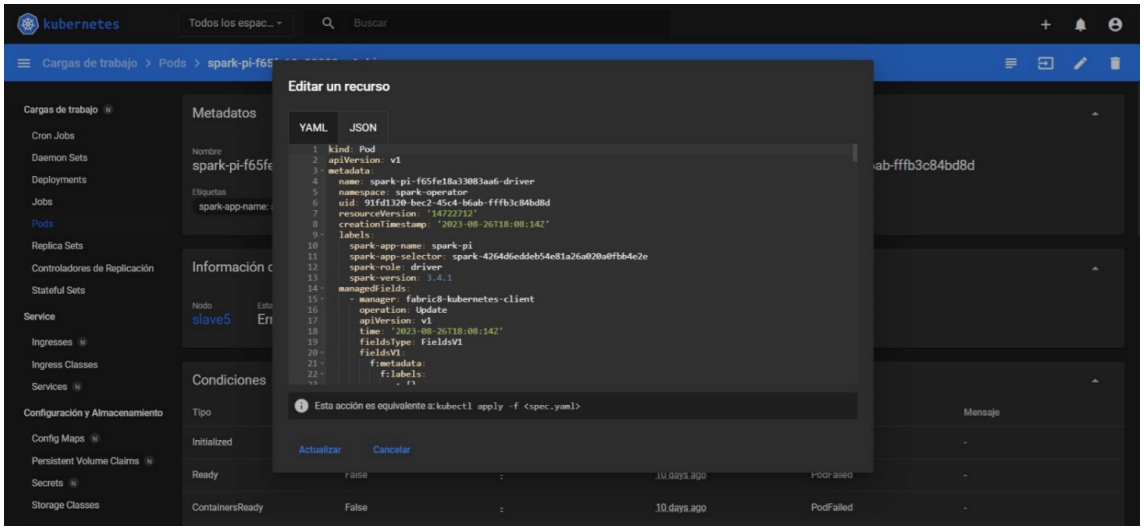


Figura 3.33: Ventana de edición de un archivo de configuración

3.8. Arquitectura completa

Llegados a este punto, si se ha seguido ordenadamente los pasos mencionados, la arquitectura de procesamiento y almacenamiento distribuido basada en un clúster de Kubernetes ya estaría implementada y funcionando correctamente. Para poder acceder a todos los servicios directamente, lo más recomendable es crear un fichero con extensión .bat para poder ejecutarlo y poder utilizar en localhost todos los puertos necesarios para acceder a todos los servicios a la vez, sin necesidad de tener que hacer un tunneling distinto cada vez que queremos acceder a un servicio. Para hacerlo tan solo tendriamos que crearlo como se ve en la Figura 3.34. Es importante tener en cuenta que en el proceso de implementación de los servicios, algunos puertos no serán los mismos que hemos usado nosotros.

En la Tabla 3.3 resumimos los puertos utilizados en nuestro clúster y los servicios que tienen asociados.

Nº de puerto	Servicio
30458	Spark History Server
30159	MongoDB
30084	Kubernetes Dashboard
30000	Prometheus
31363	Grafana
30080	Jupyter
8080	Apache

Tabla 3.3: Nº de puertos y servicios asociados

Todos los archivos de los que hemos usado están subidos a un repositorio de github (https://github.com/Mormur22/KubernetesArchitectureFiles) ordenados por carpetas obtenidos del espacio de la carpeta */home/estudiante* del nodo Master

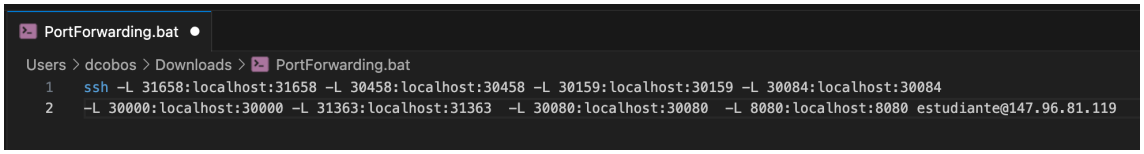


Figura 3.34: Ejemplo de ejecutable para acceder a los servicios del clúster

3.9. Rendimiento ante diversas situaciones

En esta sección vamos a comentar cómo se comporta el clúster ante diversas situaciones que consideramos interesantes. En concreto, vamos a comentar algunos escenarios de fallo y cómo el sistema puede reponerse si es posible.

3.9.1. Caso 1: fallo en un nodo

Al tratarse de un entorno hardware real, la posibilidad de que alguno de los nodos falle, ya sea por un fallo en alguna componente o por equivocación humana, es alta. Ante esta situación cabe destacar dos posibilidades muy distintas y es que el clúster no actuará igual si el nodo que falla es un nodo esclavo o si el que falla es el maestro.

En caso de que falle un nodo esclavo, gracias a Kubernetes y en concreto a la replicación de mongoDB, el resto de nodos tienen una replica activa del fragmento de la base de datos de dicho nodo por lo que no se perdería información y Kubernetes auto regularía el clúster transformándolo en uno con 4 nodos momentáneamente. De todas formas esto está más ligado a que falle el nodo dentro del contexto de Kubernetes que a que falle la propia máquina física. En caso de que fuese el propio computador que se apagase por fallo en la red eléctrica o cualquier acción que simplemente lo desconectase y no lo estropease, modificamos la Bios de las máquinas físicas para que si se apagasen de repente se encienderan de nuevo sin necesidad de interacción humana. En caso de fallo en algún componente entonces será necesario sustituirlo por uno nuevo que funcione correctamente.

En caso de que falle un nodo maestro la situación sería totalmente distinta. Al tratarse de ser el único nodo conectado a internet, si éste falla el clúster dejará de funcionar, ya que no podrá acceder a ningún servicio del exterior del clúster. Si se apaga la máquina, al igual que el resto de nodos, se encenderá de nuevo gracias a la configuración de la Bios.

En cualquier caso, Kubernetes tiene herramientas para poder regular el clúster y adaptarlo a las situación. Si todas las máquinas tuviesen tarjeta de red propia y estuvieran conectadas a internet directamente sin necesidad del maestro la situación sería diferente.

3.9.2. Caso 2: fallo dentro de Kubernetes

Otra situación que podría tener lugar es que alguna componente de Kubernetes deje de funcionar correctamente. La posibilidad de que algún pod o algún nodo del propio clúster de Kubernetes deje de funcionar no es nula. Ante esta situación no hay ningún método automático que asegure su constante funcionamiento pero gracias al dashboard de Kubernetes podemos ver de forma fácil los ficheros de Log, como podemos ver en la Figura 3.35. De esta forma podemos ver qué ha fallado en la ejecución de un pod.

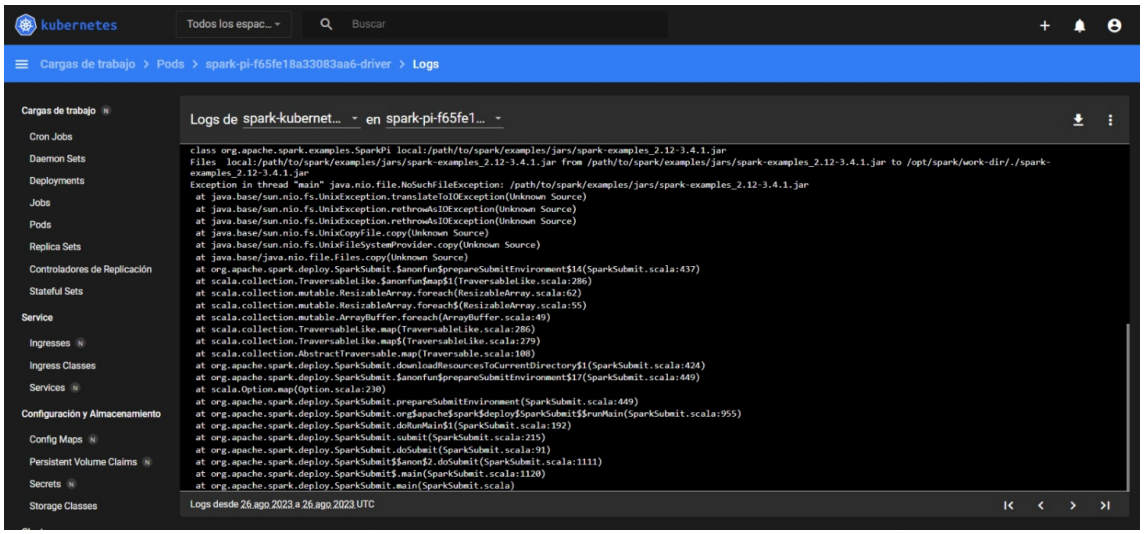


Figura 3.35: Ventana de visualización de log del dashboard de Kubernetes

Capítulo 4

Estado de la Cuestión

En esta capítulo vamos a exponer la investigación que hemos realizado sobre trabajos similares. Al no ser un trabajo puro de investigación en el que debamos aportar algo nuevo de un gran valor científico, vamos a centrar este capítulo en exponer toda la información que hayamos encontrado sobre trabajos similares al nuestro. La idea es presentar dichos trabajos explicando qué es lo que diferencian del nuestro y argumentar qué hemos seleccionado de ellos para nuestro propio trabajo y qué hemos descartado.

4.1. Trabajos universitarios similares

Parte del proceso de investigación conllevó la busqueda de trabajos universitarios similares al nuestro. Aunque encontramos documentos interesantes como el de Lluís Baró Cayetano (2021) o el de Ernesto Gaspar Aparicio (2022), que se centraban en Kubernetes principalmente, finalmente hemos decidido comentar dos que fueron verdaderamente relevantes para nuestro proyecto, independientemente de si finalmente incluyesemos más o menos sobre ellos en nuestro propio trabajo. En realidad, para nuestro trabajo no nos hemos centrado en otros trabajos porque generalmente había diferencia insalvables entre ambos que no nos facilitaban el trabajo.

4.1.1. TFG - 2018, Marta Frías Zapater

Este es un trabajo que utilizamos, ofrecido por nuestra tutora, y que reunía bastantes características con el trabajo que teníamos que realizar. Éste era el realizado en 2018 por Marta Frías Zapater (2018), que consistía en el despliegue de una arquitectura tolerante a fallos para el aprovisamiento automático de máquinas físicas. En este trabajo se utilizó Ubuntu MAAS, para realizar el aprovisamiento de forma dinámica y flexible de recursos hardware. Nuestra primera idea era utilizar este servicio de forma similar para poder agrupar las distintas máquinas que disponíamos. De hecho instalamos MAAS y estuvimos trabajando durante unos días con esa herramienta, hasta que nos encontramos un problema que nos obligaba a

reconsiderarlo todo, y es que MAAS era incompatible con los recursos hardware que nosotros disponíamos, en concreto con la placa base y la fuente de alimentación. Por esta razón descartamos este recurso.

Otro aspecto incluido en su trabajo y que nos resultó atractivo fue la parte de evaluación del sistema. Aquí la autora presenta de forma breve distintos escenarios en los que se puede ver envuelto el sistema. Ante cada escenario comenta el comportamiento del propio sistema y la posible solución en caso de que haya. Como se puede apreciar en la Sección 3.9

4.1.2. TFG - 2022, Luis Piña Cubas

El trabajo presentado por Luis Piña Cubas (2022) está centrado principalmente en el aspecto de monitorización de clústeres en Kubernetes. El autor usa Grafana como principal aplicación para supervisar el rendimiento de cierta aplicación que él mismo ha desplegado sobre un cluster de Kubernetes, montado sobre máquinas virtuales. Uno de los aspectos más interesantes de este trabajo es ver las distintas posibilidades que ofrece Grafana y cómo se puede usar Prometheus también como sistema de monitorización. Sin embargo, en nuestro trabajo solo nos ha servido para conocer la existencia de estos servicios, ya que no hemos seguido la misma forma de instalación y despliegue que el autor del trabajo.

Un aspecto que sí hemos seguido es el de configuración de alertas. Grafana ofrece un método al administrador para poder gestionar avisos si ciertas métricas no cumplen con ciertos parámetros. Esta novedad podía resultar muy interesante en nuestro caso para configurar avisos si algún nodo ha dejado de funcionar o si alguna partición de la base de datos de MongoDB está trabajando muy por encima del resto de shards. En nuestro trabajo hemos incluido la configuración de alertas y la configuración del despliegue para incluir el envío de correos mediante SMTP, el protocolo estándar para dicha acción en internet. Debido al tiempo y a un número bajo de pruebas hemos decidido incluir unicamente un ejemplo de configuración de alertas en el Capítulo 3, para mostrar esta posibilidad. La parte de configuración del servidor STMP hemos decidido no incluirla, ya que al editar un despliegue se reinicia el servicio y hay que volver a configurar el cambio de contraseña y los propios dashboards.

4.2. Proyectos similares

Además de fijarnos en otros trabajos universitario, también buscamos proyectos similares al nuestro. Aunque no encontramos ninguno que reuniese todas las características, encontramos algunos proyectos que podían cubrir algunas partes del nuestro.

4.2.1. Videotutoriales de Anton Putra

Uno de los proyectos que encontramos fueron los videos tutoriales de Putra (2023), en los que cubre distintos aspectos relevantes para nosotros como el proceso de instalación de MongoDB sobre Kubernetes o la instalación de Prometheus o incluso el uso de los secretos en el entorno de Kubernetes.

Sus videos nos ayudaron a orientarnos un poco al respecto de esos temas, aunque no seguimos su guía al pie de la letra ya que estaba trabajando en un entorno virtual con una configuración muy distinta a la nuestra.

Este mismo canal tambien contiene muchos videos teóricos sobre Kubernetes que nos ayudaron a enteder mejor muchos de los conceptos que explicamos en la Sección 2.8.

4.2.2. Guía de ComputingForGeeks

Sin duda de todos los proyectos encontrados, el de Mutai (2023) se lleva la corona. En esta web se incluye todo un proyecto de Kubernetes entero, que va desde la instalación, hasta la implementación de los sistemas de monitorización o la configuración de los sistemas de almacenamiento persistente, entre otros. Pese a que trabaje en un entorno distinto al nuestro sigue siendo útil para aprender a cómo instalar kubernetes desde cero y para incluso conocer algunas herramientas intermedias necesarias. De hecho, incluye distintas variantes según el Container Runtime que queramos utilizar o incluso la manera en la que queramos exponer nuestro Dashboard, ya sea usando Ingress o NodePort.

Aunque vengan todos los comandos necesarios, si que consideramos que no se involucra tanto en aspectos teóricos y da la sensación de no saber muy bien que se está instalando a cada paso. Esto hace que pueda resultar díficil para alguien que no haya investigado previamente las distintas componentes que forman Kubernetes y sus posibilidades.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Dificultades encontradas

A la hora de abordar un trabajo del que se dispone tan poca información desde un comienzo, siempre se presupone que no va a ser una tarea sencilla, y más teniendo en consideración el hecho de que a lo largo de toda la carrera no hemos trabajado con las herramientas que hemos tenido que utilizar para realizar este trabajo de final de grado. Aún sabiendo todo esto, decidimos decantarnos por este proyecto para profundizar en temas interesantes como puede ser Cloud, siendo conscientes de la dificultad que supondría todo lo mencionado anteriormente.

Abordaremos este capítulo mencionando las principales dificultades que nos hemos encontrado, priorizando aquellas que mayor esfuerzo nos llevó solventar o aquellas dificultades que fuesen más determinantes y que sin una solución imposibilitasen acabar el trabajo.

5.1.1. Gestión de la red

Uno de los problemas más recurrentes que tuvimos fue todo aquel relacionado con la red. Desde un comienzo no tuvimos asignada una dirección IP específica y tuvimos que compartir la red con la impresora que se encontraba en el mismo despacho que las cinco máquinas. Esto suponía que en ocasiones el uso de la impresora impedía que pudiésemos acceder a la red, de forma que muchas veces tuvimos que acceder presencialmente al despacho para volver a conectar la red. En estos tiempos entre que se desconectaba y volvíamos para conectarla no podíamos trabajar remotamente en el clúster.

Necesitamos también del permiso por parte de los Servicios Informáticos de la Universidad Complutense de Madrid para evitar el posible filtrado del punto de red ya que en la red de la universidad está prohibido tener varias direcciones MAC, es decir varios ordenadores, con una dirección IP pública.

Además, el propio filtrado de puertos que no fueran el 22, 8080 y el 443 dentro

de la universidad, provocó que a la hora de exponer los servicios del clúster, tuvieramos que exponerlos mediante nodeport y acceder a ellos con port-forwarding y no mediante una VPN lo que ralentiza a veces el tráfico y puede provocar cuellos de botella en el nodo por el que se accede.

5.1.2. Falta de trabajos similares

Otro aspecto que descubrimos al iniciar el trabajo era la falta de informacion y de trabajos similares que había al respecto. La mayoria de trabajos relacionados con clústeres y sistemas distribuidos se hacían sobre máquinas virtuales y no sobre equipos como los nuestros.

Los trabajos que se muestran en el estado de la cuestión difieren en bastantes cosas a nuestro trabajo por lo que no han servido de mucha inspiración.

Trabajar sobre máquinas virtuales es mucho más sencillo gracias a la fácil configuración que se puede hacer sobre las características de dichas máquinas, ajustándolas a los requisitos favorables de las herramientas que hemos utilizado. Además tampoco habría problemas como el mencionado anteriormente de la impresora.

El principal problema que hubiera supuesto trabajar con máquinas virtuales es el posible coste que hubiera tenido mantenerlas activas. En el caso de nuestros equipos físicos el único motivo por el que aumentó el coste fue porque muchos de ellos no tenían tarjetas de red, aunque gracias al departamento de sistemas informáticos, dónde uno de nosotros estaba trabajando por aquel entonces, pudimos obtenerlas.

5.1.3. Reinicios de equipos

Al pie de lo visto en el anterior apartado, trabajar con equipos físicos puede tener incovenientes como el malfuncionamiento de algún componente, entre otros. En nuestro caso, al trabajar con herramientas nuevas y tras probar numerosas opciones para encontrar la mejor solución posible, muchas veces nos hemos encontrado que en el proceso de instalacion o en la prueba de las diversas herramientas hemos eliminado alguna dependencia que imposibilitaba la instalación de otras, o que por lo menos no encontramos ninguna forma de revertir la situación. Ante esta situación hemos tenido que formatear el ordenador en el que ocurrió y volver a instalar el sistema operativo de nuevo y todo lo demás.

5.2. Posibles y futuras mejoras

5.2.1. StreamProcessing

Apache Spark, tradicionalmente reconocido por sus capacidades de procesamiento en lote, ofrece un módulo robusto para el procesamiento de streams denominado Spark Streaming. Al incorporar Spark Streaming en nuestro cluster Kubernetes, no

solo potenciamos el cluster, sino que también nos preparamos para poder procesar conjuntos de datos que cambian en tiempo real, como pueden ser casos de uso de Internet of Things.

Para realizar esta incorporación es necesitaríamos una conexión sólida con fuentes de datos en tiempo real como Kafka, Flume o Kinesis. Esta integración garantizaría una transmisión eficiente entre las fuentes y Spark. Una vez establecida la conexión, podríamos desarrollar aplicaciones que utilicen las APIs de Spark Streaming, definiendo así las operaciones específicas de procesamiento que se deben llevar a cabo en los datos entrantes.

5.2.2. Automatización con Ansible

Ansible es una herramienta de automatización y administración de configuraciones ampliamente utilizada en el mundo de la administración de sistemas. Su principal objetivo es automatizar y agilizar tareas repetitivas de configuración, implementación y administración de sistemas. Ansible tiene unos elementos llamados "playbooks". Estos son archivos de configuración escritos en YAML que describen una serie de tareas y acciones que Ansible debe realizar sobre los servidores en los que se ejecuten.

En nuestro trabajo teníamos pensado implementar distintos playbooks de Ansible que permitiesen automatizar ciertas tareas como el incremento de réplicas dentro de la base de datos distribuida o cambiar la capacidad de almacenamiento reservada a cierto servicio. Básicamente estos archivos buscaban ayudar al administrador a escalar rápidamente la arquitectura y a modificar ciertos valores sin tener la necesidad de modificar los archivos originales.

5.2.3. Implementación de un sistema de entrega contínua con ArgoCD

ArgoCD es una herramienta declarativa, basada en Git, para la entrega continua (CD) de aplicaciones en Kubernetes. Permite que los usuarios definan y mantengan aplicaciones directamente desde repositorios Git, y asegura que el estado real en el clúster coincida con el estado deseado especificado en Git. En pocas palabras, permite *git-ops*.

Puedes versionar y mantener tus configuraciones de Spark y Jupyterhub usando Git y usar ArgoCD para implementar o actualizar esas configuraciones en tu clúster de Kubernetes de manera automática. Esto se podría conseguir mediante la creación de GitHub Actions de tal forma que si añadimos un script de python estático en nuestro repositorio que por ejemplo realiza un análisis de datos que requiera de spark, se cree un recurso SparkApplication en nuestro clúster con su respectivo ConfigMap que monte ese fichero en un volumen y cuyo MainApplicationFile apunte al fichero en cuestión. Esto lo que garantiza es que cualquier commit que hagamos en el repositorio lanzará el respectivo SparkApplication que será lanzado de manera automática con kubectl y que el Operador de Spark on Kubernetes gestionará para

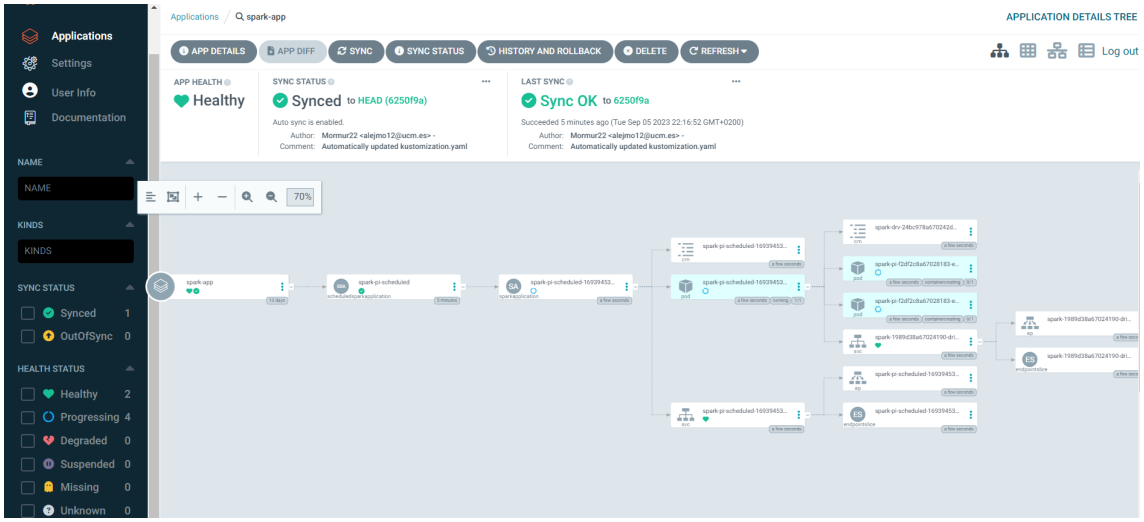


Figura 5.1: Ejemplo de una aplicación con entrega contínua en ArgoCD

crear los pods.

Si alguna configuración no funciona como se esperaba, ArgoCD ofrece apacidades de rollback para volver a una configuración anterior.

Ya tenemos la base del servidor de Argo sincronizado con un repositorio con sus GitHub Actions y una SparkApplication como se pueden ver en las Figuras 5.1, 5.2 y 5.3 por lo que terminar de configurarlo para conseguir un despliegue contínuo no llevaría demasiado tiempo.

5.2.4. Programar y monitorizar flujos de trabajo complejos con Apache Airflow

Esta es una plataforma robusta diseñada para la programación y monitorización de flujos de trabajo. Airflow tiene la particularidad de permitir a los usuarios definir estos flujos usando Python. Estos flujos pueden abarcar tareas que se despliegan en Kubernetes haciendo uso al igual que ArgoCD del operador de Spark on Kubernetes. Podríamos establecer un flujo de trabajo con Datapipelines, que fueran desde la ingesta de datos almacenados en MongoDB, limpieza y transformación mediante recursos como una SparkApplication que puede ser lanzada con ArgoCD, análisis de datos con los cuadernos de Jupyter y visualización de los datos en un dashboard como PowerBi.

Este tipo de arquitectura ya está estandarizada y se aplica en entornos de Data Operations empresariales a gran escala con un rendimiento eficiente y probado.

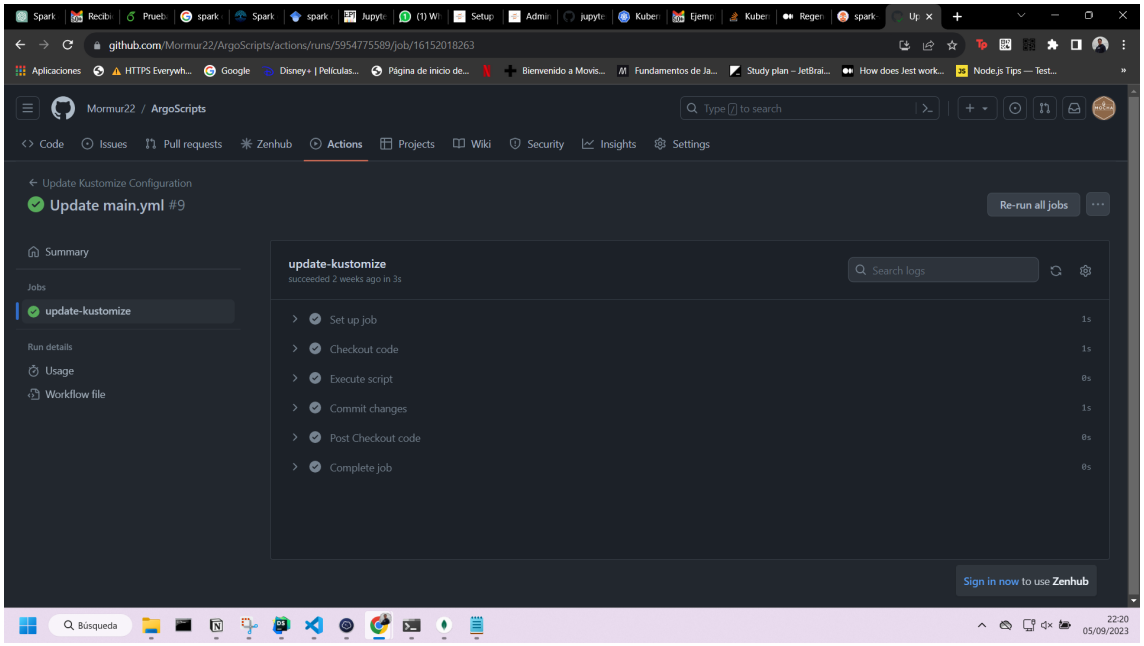


Figura 5.2: Ejemplo de una GitHub Action

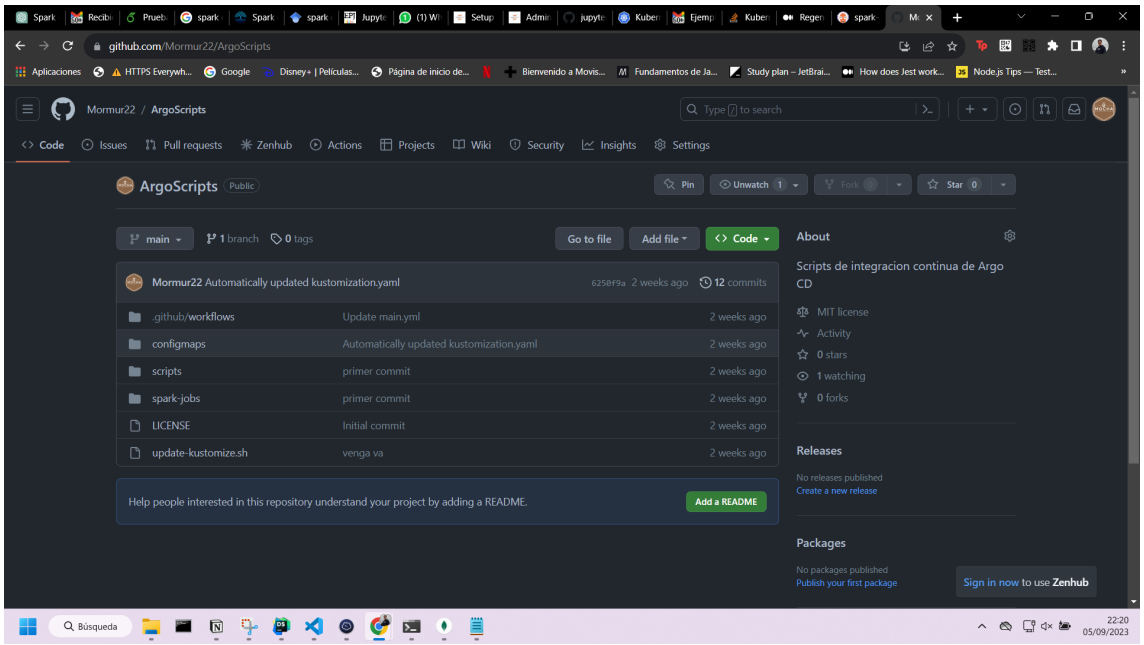


Figura 5.3: Ejemplo de repositorio en GitHub para entrega contínua

Introduction

In this chapter we are going to make a brief introduction of our final degree project, showing different introductory sections which aim to offer a simple view about the memory and the final project.

5.3. Objectives

The main objective in this project is to elaborate successfully a private architecture to do distributed processing and storage of data. To make it work is necessary to firstly investigate the different tools we can use to later proceed to the implementation of them. Some of this tools are Docker and Kubernetes, of which we will get deep in the following chapters, just as MongoDB and Spark, which will be the two main services that the system hosts. Apart from this, to offer a fully useful and efficient system we also have as an objective to implement a method for the administrator to monitor and supervise the whole work done.

Another main objective that we'll try to aim throughput the memory is the replication possibility. We have presented a project making sure that anybody with a minimum computer knowledge can elaborate their own system following thoroughly the different sections of which the memory consists.

We consider this two goals as the most important ones even though they are not the only ones. Another purpose is to be able to learn and dig into less common tools in the Computer Engineering degree, to have a wider range of knowledge mainly in DevOps field. We also had as an objective to work with leading and actual services like distributed processing in Cloud can be.

As we progressed with the project and as we were getting deep into the concepts, new intermediate goals surfaced, of which we will explain in greater detail throughout the entire chapter 3

5.4. Motive

To face a final degree project of these characteristics is really import to be clear about the motivation behind of it. One of the aspects we took into account before choosing a project was that it had to be attractive and we valued positively the fact that it was related with tools that we had not seen in depth in the whole degree, to investigate about it and thus acquire new and wider knowledge. Starting from this point, one of the final degree projects that most attracted us was this one. We found it very interesting since it allowed us to learn and improve in the use of DevOps technologies, as well as delve into current and future topics such as Big Data or Cloud technology.

In addition to this, the possibilities it offered us also played a huge point in its favor. There was a lot of freedom to be able to investigate and develop it in the best possible way to meet the objectives.

5.5. Work plan

To carry out a project like the one presented, in which you must go regularly to a physical point, it is important to have a clear and efficient organization in order to optimize the available time. One of the aspects that was clearest to us from the beginning was the urgent need to be able to work remotely with the five machines, in order to progress quicker and continuously. However, although we achieved it relatively soon, there were many problems that forced us to go in person more frequently. Some of these problems can be seen in the section 5.1

Bearing this in mind, our work plan was as expected. First we started by going to the place where the five computers resided to configure the machines. Later we installed the operating systems and built the network architecture between them, and we even opened the computers to insert network cards and modify components that were not working correctly. We also got two ways to be able to connect to the machines remotely, by ssh and via remote desktop.

Once we achieved the latter, it was when we proceeded to the installation of all the necessary software like Kubernetes and Docker as well as other tools that we will explain throughout this document. First we focused on orchestrating the machines in order to form a robust and efficient cluster. Once this is achieved, we focused on the implementation of the required services such as MongoDB or Spark. We also include a monitoring system to be able to control and monitor the correct functioning of the cluster and the services.

Once all the software we considered appropiate was installed we began with the testing phase to make sure all was working as it should and with the writing of the memory. For testing we introduced test databases in MongoDB and jobs prepared for Spark.

5.6. Explanations about the memory

To finish the final degree project is essential to present a correctly redacted and organized memory. Since the beginning we had in mind the path we wanted our memory to follow to achieve all our purposes. This path could be summed up in the two chapters following this one.

Our idea is basically to present two main chapters with much greater volume of information than the rest of them. The first one, chapter 2, pretends to be a chapter purely theoretical, in which the main concepts and tools that are going to be used are explained. Being this work with a lot of theoretical load, we consider it essential to present a good solid foundation on the terms and concepts used. The second one, chapter 3, it is a fundamentally practical chapter. Here we want to show all the practical work we have done in context to the concepts explained in the previous chapter. This one is made up of different sections in which each of them contains the process we have to follow to implement a specific tool.

The main goal of the memory is to be used as a manual to every lector who wants to create their own distributed processing and storage cluster and who doesn't know nothing about the implemented tools. This is why we've pretended to include a really accessible language and to follow a clear and direct instruction path in the memory.

Besides, the memory has been redacted using Overleaf, a collective redaction tool that uses LaTeX. To learn about this language we've used the official LaTeX documentation and some books like the one from Alexander Borbon A. (2017).

5.7. Memory structure

Memory consists of 5 main chapters:

- **Chapter 1: Introduction**: Brief introduction about the presented project.
- **Chapter 2: Technological context**: Theoretical chapter that contains all the technological context of the project. In other words, all the essential theoretical concepts needed in the development of the project.
- **Chapter 3: Work done description**: As the name indicates this chapter includes all the practical work that has been done.
- **Chapter 4: State of the question**: In this chapter we comment projects similar to ours, specifying the things that has inspired us for our project.
- **Chapter 5: conclusions and future work**: In this one we sum up the conclusions we've obtained elaborating the project and we comment the future improvements we would like to implement.

Apart from this, we can find in appendix A a list with some of the most used and important commands in the whole context of out project, accompanied by a brief description of them.

Conclusions and Future Work

5.8. Difficulties we've faced

While getting ready to start a work of which so little information is available from the beginning, its always assumed that its not going to be an easy task, taking also into account the fact that throughout the whole degree we haven't work with any of the tools that we had to use for this final degree work. Even though we knew all of this we still decide to face this work and getting deep in such an interesting themes like Cloud can be, knowing the difficulties we would face with all the things previously mentioned

We will organize this chapter mentioning the main difficulties we've face, prioritizing those who bigger effort was required to solve o those who were more determining and which if we didn't get a solution the work wouldn't have been completed.

5.8.1. Network's management

Some of the most recurring problems we faced were related to the network. Since the beginning we hadn't available an specific IP direction so we had to share the internet access with the printer located in the same place as the five machines. This led to make it impossible for us to access the network if the printer was used. In those cases we had to access in person to the lab to reconnect the network cable. During this cases we couldn't work remotely on the cluster. We also needed permission from the IT Services of the Complutense University of Madrid to avoid the possible filtering of the network point, since in the university network it is forbidden to have several MAC addresses, that is, several computers with a public ip.

In addition, the filtering of ports other than 22, 8080 and 443 within the university, caused that when exposing the cluster services, we had to expose them through Nodeport and access them with port-forwarding and not through a VPN which sometimes slows down traffic and can cause bottlenecks in the node through which it is accessed.

5.8.2. Lack of Similar Works

Another aspect that we discovered at the beginning of the work was the lack of information and similar work that existed on the subject. Most of the work related to clusters and distributed systems was done on virtual machines and not on computers like ours.

The work shown in the state of the art differs from our work in many ways, so it was not very inspiring.

Working on virtual machines is much easier thanks to the easy configuration that can be done on the characteristics of these machines, adjusting them to the favorable requirements of the tools we used. There would also be no problems like the printer problem mentioned above.

The main problem that would have been associated with working with virtual machines is the possible cost of keeping them active. In the case of our physical machines, the only reason for increased costs was that many of them did not have network cards, although we were able to obtain them thanks to the IT department where one of us worked at the time.

5.9. Future Work

Conclusions of the work and lines of future work.

5.9.1. Flow processing

Apache Spark, traditionally recognized for its batch processing capabilities, offers a robust stream processing module called Spark Streaming. By incorporating Spark Streaming into our Kubernetes cluster, we not only power the cluster, but also prepare ourselves to be able to process data sets that change in real-time such as Internet of Things use cases.

To realize this addition we would need a robust connection to real-time data sources such as Kafka, Flume or Kinesis. This integration would ensure efficient transmission between the sources and Spark. Once the connection is established, we could develop applications that use Spark Streaming APIs, thus defining the specific processing operations to be performed on the incoming data.

5.9.2. Implementation of a continuous delivery system with ArgoCD

ArgoCD is a declarative, Git-based tool for continuous delivery (CD) of applications on Kubernetes. It allows users to define and maintain applications directly from Git repositories, and ensures that the actual state in the cluster matches the desired state specified in Git. Simply put, it enables Òtextitgit-ops.

You can version and maintain your Spark and Jupyterhub configurations using Git and use ArgoCD to deploy or update those configurations to your Kubernetes cluster automatically. This could be achieved by creating GitHub Actions so that if we add a static python script in our repository that for example performs a data analysis that requires Spark, a SparkApplication resource is created in our cluster with its respective ConfigMap that mounts that file on a volume and whose MainApplicationFile points to the file in question. What this guarantees is that any commit we make in the repository will launch the respective SparkApplication that will be launched automatically with kubectl and that the Spark on Kubernetes Operator will manage to create the pods.

If any configuration does not work as expected, ArgoCD offers rollback capabilities to revert to a previous configuration.

We already have the Argo server base synchronized with a repository with its GitHub Actions and a SparkApplication as you can see in the figures 5.1, 5.2 and 5.3 so finishing the configuration to achieve a continuous deployment would not take too much time.

5.9.3. Scheduling and monitoring complex workflows with Apache Airflow

This is a robust platform designed for scheduling and monitoring workflows. Airflow has the particularity of allowing users to define these flows using Python. These flows can encompass tasks that are deployed in Kubernetes making use like ArgoCD of the Spark on Kubernetes operator. We could set up a workflow with Datapipelines, ranging from ingesting data stored in MongoDB, cleansing and transformation using resources such as a SparkApplication that can be launched with ArgoCD, data analysis with Jupyter notebooks and visualization of the data in a dashboard such as PowerBi.

This type of architecture is already standardized and applied in large-scale enterprise Data Operations environments with efficient and proven performance.

Contribuciones Personales

En esta sección vamos a desgranar el trabajo realizado en tres partes. La primera de ellas es el trabajo que se ha realizado en conjunto y las dos siguientes son las partes que cada miembro ha realizado de forma personal.

Trabajo conjunto

En este apartado detallamos qué partes del trabajo la hemos realizado los dos miembros indicando las funciones de cada uno.

Empezamos por la gestión de red. Esto incluye el acceso al despacho para configurar los equipos, tanto a nivel de hardware, como a nivel de software. En todo momento ambos estuvimos a la vez en el lugar donde residen las cinco máquinas. Ambos diseñamos la arquitectura de red y ambos instalamos los cinco sistemas operativos. Quizá por horario alguno de nosotros llegaba antes al lugar pero siempre estábamos los dos en todas las sesiones. La investigación previa de los sistemas operativos y del esquema de red óptimo también se realizó a la par.

Continuamos con toda la investigación necesaria para iniciar con las implementaciones. Ambos leímos otros trabajos similares y rápidamente llegamos a la idea de usar Ubuntu MAAS. Toda esta investigación no nos llevó a buen puerto ya que luego oímos hablar de Kubernetes y esta fue la herramienta que íbamos a utilizar. Para buscar información sobre Kubernetes ambos leímos numerosas páginas web que íbamos compartiendo con el otro para que en cada momento tuvieramos los dos la información. Sobre la investigación de MongoDB en el entorno de Kubernetes actúamos igual, mientras que sobre otros aspectos sí decidimos separarnos, como especificaremos en las dos siguientes secciones.

Para implementar todas las herramientas siempre estaba el handicap de que éste no es un trabajo de desarrollo de software al uso, donde un miembro puede programar un aspecto mientras otro se centra en otro. Cuando implementamos Kubernetes establecimos horarios y reuniones para que los dos miembros siempre estuviésemos presentes. Alternando, uno de nosotros ejecutaba los comandos y compartía su pantalla mientras que el otro hacía seguimiento. Ambos diseñamos la estrategia a seguir y ambos veíamos que comandos ibamos a utilizar en cada momento.

Para la instalación de MongoDB actúamos de la misma forma. Toda la investigación sobre el operador de Kubernetes que finalmente no utilizamos fue conjunta. Cabe destacar que en esta parte del trabajo Alejandro fue la persona que siempre estuvo escribiendo los distintos archivos de configuración mientras Daniel permaneció activo en llamada apoyando para la definición de los mismos.

Sobre Spark y la monitorización profundizamos más adelante ya que estos fueron los puntos donde ha habido gran diferencia entre ambos.

Por parte de la memoria ambos hemos contribuido por igual, en cuanto a porcentaje de texto escrito en cada capitulo. El capitulo introductorio que consta de cuatro secciones fueron escritas dos y dos. El capitulo de contexto tecnológico fue escrito conjuntamente, independientemente del enfásis de cada uno en Spark o en la monitorización.

Daniel Cobos Peñas

Aquí vamos a añadir las contribuciones exclusivas de Daniel. Cómo hemos mencionado anteriormente la principal diferencia ha residido en Spark y en el sistema de monitorización.

Daniel se ha encargado de todo el sistema de monitorización. Todo lo relacionado con Grafana y el dashboard de Kubernetes fue contribución suya, al igual que la forma de exponer la información de la base de datos de MongoDB a Prometheus.

Alejandro Moreno Murillo

Aquí vamos a añadir las contribuciones exclusivas de Alejandro. Al igual que antes la diferencia reside sobre todo en Spark y en el sistema de monitorización.

Alejandro se ha encargado de toda la implementación de Spark. Desde la investigación sobre cómo instalar Spark en Kubernetes como la forma de integrar Jupyter en Spark para poder acceder a la base de datos MongoDB.

Bibliografía

Sharding in mongodb. Disponible en https://www.mongodb.com/basics/sharding.

Spark official documentation: running on kubernetes. Disponible en https://spark.apache.org/docs/latest/running-on-kubernetes.html.

Aaron Culich, C. H. E. S. R. L. Y. P. L. G., Carol Willing. Zero to jupyterhub with kubernetes - gihtub repo. Disponible en https://github.com/jupyterhub/zero-to-jupyterhub-k8s.

Alexander Borbon A., W. M. F. *LaTeX 2017*. 2017.

Authors, G. Data source management - grafana. Disponible en https://grafana.com/docs/grafana/latest/administration/data-source-management/?utm_source=grafana_gettingstarted.

authors, T. K. Kubernetes official documentation. Disponible en https://kubernetes.io/docs/.

Contributors, P. J. Jupyterhub official documentation. Disponible en https://z2jh.jupyter.org/en/stable/jupyterhub/index.html.

Ernesto Gaspar Aparicio. Despliegue de un cluster Kubernetes altamente disponible en Google Cloud Platform. 2022. Trabajo Fin de Grado.

Escura, C. Run spark history server on kubernetes using helm. https://medium.com/@carlosescura/run-spark-history-server-on-kubernetes-using-helm-7b03bfed20f6, 2019.

Haines, S. *Modern Data Engineering with Apache Spark: A Hands-On Guide for Building Mission-Critical Streaming Applications*. Apress, 2022. ISBN 978-1-4842-7451-4. Available at https://doi.org/10.1007/978-1-4842-7452-1.

infosegur. Seguridad informática - clusters de servidores. Disponible en https://infosegur.wordpress.com/unidad-2/clusters-de-servidores/.

Jin, Y., Wen, Y. y Chen, Q. Energy efficiency and server virtualization in data centers: An empirical investigation. 2012.

Kommeri, J., Niemi, T. y Helin, O. Energy efficiency of server virtualization. 2012.

Lluís Baró Cayetano. Creation of a Kubernetes Infrastructure. 2021. Trabajo Fin de Grado.

Luis Piña Cubas. Estudio de sistemas clúster de Kubernetes y su monitorización e implantación en entorno productivo. 2022. Trabajo Fin de Grado.

Marta Frías Zapater. Diseño y despliegue de una arquitectura tolerante a fallos para el. 2018. Trabajo Fin de Grado.

Mormur22. Kubernetesarchitecturefiles. `https://github.com/Mormur22/KubernetesArchitectureFiles`, 2023. GitHub repository.

Mutai, J. Install kubernetes cluster on ubuntu 20.04 with kubeadm. Disponible en `https://computingforgeeks.com/deploy-kubernetes-cluster-on-ubuntu-with-kubeadm/`.

Nerd, P. Docker swarm vs kubernetes - ¿cuál es mejor? Disponible en `https://www.youtube.com/watch?v=UPetPzsiu-w`.

Oakes, R. Spark on kubernetes: Jupyter and beyond. `https://www.oak-tree.tech/blog/spark-kubernetes-jupyter`, 2019.

Platform, G. C. y Contributors. spark-on-k8s-operator: Kubernetes operator for managing the lifecycle of apache spark applications on kubernetes. `https://github.com/GoogleCloudPlatform/spark-on-k8s-operator`, 2023.

Project, A. Argocd documentation. `https://argo-cd.readthedocs.io/en/stable/`, 2023a.

Project, A. A. Apache airflow documentation. `https://airflow.apache.org/docs/apache-airflow/stable/index.html`, 2023b.

Putra, A. Kubernetes tutorial. Disponible en `https://www.youtube.com/playlist?list=PLiMWaCMwGJXmoKAmRh38U1-QEeh2dGEOX`.

Sephinreji. Understanding spark deployment modes: Client vs cluster vs local. Disponible en `https://medium.com/@sephinreji98/understanding-spark-cluster-modes-client-vs-cluster-vs-local-d3c41ea96073`.

Siddhant Prateek. Helm Explained. Disponible en `https://medium.com/@siddhantprateek/helm-explained-d950d90c6136`.

Smyth, D. G. Qué es un clúster de servidores y qué tipos de clusters hay. Disponible en `https://www.smythsys.es/14171/que-es-un-cluster-de-servidores-y-que-tipos-de-clusters-hay/`.

Sundaray, H. Clusterip vs nodeport vs loadbalancers. key differences. Disponible en https://kodekloud.com/blog/clusterip-nodeport-loadbalancer/#.

Ubuntu, A. How to do a fresh install of maas. Disponible en https://maas.io/docs/how-to-do-a-fresh-install-of-maas.

Verma, V. Real time data streaming using apache spark. Disponible en https://www.analyticsvidhya.com/blog/2021/06/real-time-data-streaming-using-apache-spark/.

Wan, Z., Zhang, Z., Yin, R. y Yu, G. Kfiml: Kubernetes-based fog computing iot platform for online machine learning. *IEEE Internet of Things Journal*, vol. 9(19), páginas 19463–19476, 2022.

Zhu, C., Han, B. y Zhao, Y. A comparative study of spark on the bare metal and kubernetes. En *2020 6th International Conference on Big Data and Information Analytics (BigDIA)*, páginas 117–124. 2020.

Álvarez, R. Running apache spark on kubernetes. Disponible en https://medium.com/empathyco/running-apache-spark-on-kubernetes-2e64c73d0bb2.

Apéndice A

Resumen de comandos útiles

A lo largo de sus capítulos la memoria ha tenido siempre como objetivo ser un manual de instrucciones para poder implementar una arquitectura distribuida propia. Para esto se han usado herramientas como Docker o Kubernetes, que actúan bajo comandos. En este apéndice vamos a comentar algunos de los comandos más importantes para que cualquier persona que quiera replicarlo tenga un resumen de los comandos que deberá utilizar para sacar el máximo provecho.

A.1. Kubernetes

–Ver información del clúster.

```
> kubectl cluster-info
```

–Ver lista de los nodos del clúster.

```
> kubectl get nodes
```

–Ver lista de los servicios.

```
> kubectl get service
```

–Ver lista de los pods del clúster.

```
> kubectl get pods
```

–Ver lista de los volúmenes persistentes.

```
> kubectl get pv
```

–Ver lista de los pods del clúster con mayor nivel de detalle.

```
> kubectl get pods -o wide
```

–Ver lista de los deployments.

```
> kubectl get deployments
```

–Ver lista de los namespaces

```
> kubectl get namespaces
```

–Ver los pods dentro de un namespace llamado ejemplo

```
> kubectl get pods -n ejemplo
```

–Exponer un deployment utilizando NodePort

```
> kubectl expose deployment ejemplo-deployment
--port=80 --type=NodePort
```

–Obtener informacion detallado de un pod llamado ejemplo

```
> kubectl describe pod ejemplo
```

–Eliminar un servicio llamado ejemplo

```
> kubectl delete service ejemplo
```

–Eliminar un deployment llamado ejemplo

```
> kubectl delete deployment ejemplo
```

–Acceder al pod llamado ejemplo en el namespace ejemplo

```
> kubectl --namespace=ejemplo exec -it ejemplo bash
```

–Crear un secret en Kubernetes

```
> kubectl create secret generic passws
--from-literal=password=contrase a
```

–Aplicar el contenido de un fichero de configuración

```
> kubectl apply -f ejemplo.yaml
```

–Aplicar el contenido de varios ficheros de configuración

```
> kubectl apply -f ejemplo1.yaml -f ejemplo2.yaml
```

–Mostrar la configuración actual de kubectl

```
> kubectl config view
```

–Añadir un nuevo usuario (user) a la configuración de Kubernetes que soporte autorización básica, con contraseña (pwd)

```
> kubectl config set-credentials kubeuser
/foo.kubernetes.com --username=user --password=pwd
```

–Listar los tokens

```
> kubeadm token list
```

–Eliminar a un usuario llamado ejemplo

```
> kubectl config unset users.ejemplo
```

–Obtener la documentación para los manifiestos de los pods

```
> kubectl explain pods
```

–Obtener los pods que estén siendo ejecutados.

```
> kubectl get pods
--field-selector=status.phase=Running
```

–Compara el estado actual del clúster contra el estado en el que estaría si se aplica el archivo de configuración llamado nuevo.

```
> kubectl diff -f nuevo.yaml
```

–Ver la información de log de un pod llamado ejemplo.

```
> kubectl logs ejemplo
```

–Ver la información de log de un pod llamado ejemplo en un flujo continuo.

```
> kubectl logs -f ejemplo
```

–Ejecutar comando (ejemplo ls) en un pod existente.

```
> kubectl exec pod-ejemplo -- ls /
```

–Ver la información de log de un deployment.

```
> kubectl logs deploy/my-deployment
```

–Escuchar en el puerto local 5000 y reenviar al puerto 5000 en el backend del servicio my-service.

```
> kubectl port-forward svc/my-service 5000
```

–Crear un token con permisos de administrador en el namespace de ejemplo

```
> kubectl -n ns_ejemplo create token admin-user
--duration=8760 h
```

A.1.1. Helm

–Encontrar charts disponibles públicamente.

```
> helm search hub chart-que-deseas-buscar
```

–Instala una instancia de un chart.

```
> helm install my-release stable/ejemplo
```

–Actualiza una instancia de un chart.

```
> helm update my-release stable/ejemplo
```

–Revierte una release a la versión anterior.

```
> helm rollback my-release 1
```

–Elimina un release y sus recursos asociados.

```
> helm uninstall my-release
```

–Muestran todos los releases del clúster.

```
> helm list
```

–Muestra información sobre un chart.

```
> helm show chart stable/mysql
```

–Muestran los valores predeterminados que utiliza un chart.

```
> helm show values stable/mysql
```

–Agrega un repositorio de charts para buscar charts y sus actualizaciones.

```
> helm repo add my-repo https://example.com/charts
```

–Actualiza todos los repositorios de Helm.

```
> helm repo update
```

A.2. Docker

–Ver versión de Docker.

```
> docker --version
```

–Mostrar información de Docker

```
> docker info
```

–Mostrar imágenes de Docker en el host.

```
> docker images
```

–Muestra los contenedores que se ejecutan en el host (en Docker, no en Kubernetes).

```
> docker ps
```

–Construir una imagen especificada en un Dockerfile en la maquina del desarrollador. Con -t se especifica image como tag

```
> docker build -t image DockerfileDir
```

–Guardar una imagen en nuestro repositorio.

```
> docker push image:tag
```

–Hacer login en Docker.

```
> docker login
```

–Hacer uso de una imagen de Docker.

```
> docker pull image:tag
```