UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Orquestación, Despliegue y Escalabilidad de un Servicio de Ejecución de Código Altamente Disponible Basado en Contenedores

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: López Martínez, Miguel Ángel

Tutor/a: Muñoz Escoí, Francisco Daniel

Cotutor/a: Bernabeu Aubán, José Manuel

CURSO ACADÉMICO: 2023/2024

Resum

L'objectiu d'este treball de fi de grau és desplegar un servici d'execució de codi en el núvol mitjançant un clúster altament disponible basat en l'orquestració de contenidors.

Per a això, partint d'un servici prèviament construït, es realitza una anàlisi de la seua arquitectura i es descriuen els passos necessaris per a dissenyar i obtindre, mitjançant tecnologies i estructures formals d'orquestració, virtualització i monitoratge (àmpliament utilitzades en els sectors comercial i investigador), un servici escalable i resilient que permeta proporcionar una estructura de suport al desplegament del servici objectiu.

D'altra banda, s'analitzen les diferents tecnologies utilitzades en la realització del projecte, així com el seu ús en un context actual, descrivint l'arquitectura i funcionament general del servici inicial.

L'arquitectura dels diferents components s'ha adaptat al marc d'un servici d'orquestració més resilient i avançat que l'inicial. Estos canvis permeten exposar i desenvolupar les diferents millores implementades destacant, a més, el potencial d'ús de les mateixes i ajustant-se als diferents criteris del sector tecnològic que han comportat a instituir-les, actualment, com a punts essencials de la indústria.

Paraules clau: Computació en el núvol, Orquestració, Resiliència, Escalabilitat, Clúster, Contenedorización

Resumen

El objetivo de este trabajo de fin de grado es desplegar un servicio de ejecución de código en la nube mediante un clúster altamente disponible basado en la orquestación de contenedores.

Para ello, partiendo de un servicio previamente construido, se realiza un análisis de su arquitectura y se describen los pasos necesarios para diseñar y obtener, mediante tecnologías y estructuras formales de orquestación, virtualización y monitoreo (ampliamente utilizadas en los sectores comercial e investigador), un servicio escalable y resiliente que permita proporcionar una estructura de soporte al despliegue del servicio objetivo.

Por otra parte, se analizan las distintas tecnologías utilizadas en la realización del proyecto, así como su uso en un contexto actual, describiendo la arquitectura y funcionamiento general del servicio inicial.

La arquitectura de los distintos componentes se ha adaptado al marco de un servicio de orquestación más resiliente y avanzado que el inicial. Estos cambios permiten exponer y desarrollar las distintas mejoras implementadas destacando, además, el potencial de uso de las mismas y ajustándose a los diferentes criterios del sector tecnológico que han conllevado a instituirlas, actualmente, como puntos esenciales de la industria.

Palabras clave: Computación en la nube, Orquestación, Resiliencia, Escalabilidad, Clúster, Contenedorización

Abstract

The objective of this final degree project is to deploy a cloud code execution service through a highly available cluster based on container orchestration.

To achieve this, starting from a previously built service, an analysis of its architecture is carried out, implementing the necessary steps to design and obtain a scalable and

resilient service through technologies and formal structures of orchestration, virtualization, and monitoring (widely used in the commercial and research sectors), allowing this service to provide a support infrastructure for the deployment process.

Furthermore, the different technologies used in the project are analyzed, as well as their use in a current context, via the description of the underlying architecture and general operation of the initial service.

The architecture of the different components has been adapted to a more resilient and advanced orchestration service framework than the initial one. These changes allow for the presentation and development of the various improvements implemented, highlighting their potential use while aligning with the different criteria of the technological sector that have led to their establishment as essential industry standards.

Key words: Cloud Computing, Orchestration, Resiliency, Scalability, Cluster, Containerization

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

En el panorama tecnológico actual, la capacidad de desplegar y gestionar servicios en la nube de manera eficiente y resiliente se ha convertido en un pilar fundamental para la mayoría de las empresas y organizaciones. La creciente demanda de soluciones que puedan escalar de forma dinámica y ofrecer alta disponibilidad ha impulsado el desarrollo y adopción de tecnologías como los contenedores y la orquestación de los mismos. En este contexto, la creación de servicios basados en la nube que puedan manejar cargas de trabajo de manera eficiente y con mínima intervención humana es esencial para garantizar la competitividad en el sector.

Este trabajo de fin de grado se centra en el despliegue de un servicio de ejecución de código basado en contenedores en un entorno de nube, asegurando que dicho servicio sea altamente disponible, escalable y resiliente. Para alcanzar este objetivo, se parte de un servicio ya existente, el cual será analizado, optimizado y reestructurado para adaptarse a las exigencias de un entorno de producción moderno utilizando tecnologías avanzadas de orquestación.

A continuación, se detalla la motivación y los objetivos que guían este trabajo, los cuales han sido fundamentales para la elección de la arquitectura, las tecnologías y los métodos implementados.

1.1 Motivación

La motivación principal de este trabajo surge de la necesidad creciente de desarrollar infraestructuras de *TI* que puedan responder de manera efectiva a las demandas fluctuantes de recursos y a las exigencias de alta disponibilidad en un entorno empresarial. Las organizaciones requieren servicios que no solo sean capaces de escalar bajo demanda, sino que también ofrezcan resiliencia ante fallos y eficiencia operativa.

El desarrollo de un servicio de ejecución de código en la nube, basado en el concepto de *Function as a Service (FaaS)*, representa una oportunidad para explorar y aplicar tecnologías modernas que puedan facilitar la gestión y el despliegue de aplicaciones distribuidas. Estas tecnologías no solo mejoran la experiencia del usuario final, sino que también permiten a las organizaciones optimizar el uso de recursos, reducir costos y mejorar la eficiencia operativa.

La transición hacia un entorno de orquestación de contenedores mediante *Kubernetes* es particularmente relevante, dado que se ha convertido en el estándar *de facto* para la gestión de aplicaciones en la nube. Su capacidad para manejar despliegues a gran escala, mantener la disponibilidad del servicio y facilitar el monitoreo y la gestión de los recursos

lo convierte en una herramienta indispensable para la creación de servicios robustos y escalables.

Este trabajo, por tanto, busca no solo mejorar un servicio existente, sino también demostrar el impacto positivo que puede tener la adopción de tecnologías avanzadas en la creación de soluciones modernas y eficientes.

1.2 Objetivos

El objetivo general de este trabajo es diseñar y desplegar un servicio de ejecución de código altamente disponible y escalable en la nube, utilizando tecnologías de orquestación y contenedorización avanzadas. Para lograr este objetivo, se han planteado los siguientes objetivos específicos:

1. **Análisis de la arquitectura actual**: Evaluar la estructura y funcionamiento del servicio existente, identificando sus limitaciones y puntos de mejora en términos de escalabilidad, resiliencia y gestión de recursos.
2. **Diseño de una infraestructura para el soporte del servicio**: Se busca establecer una base sólida que permita la creación de un clúster de *Kubernetes* mediante la configuración de software y hardware en las máquinas virtuales involucradas.
3. **Diseño de una nueva arquitectura**: Proponer una arquitectura basada en *Kubernetes* que permita superar las limitaciones del servicio actual, garantizando alta disponibilidad, escalabilidad y facilidad de mantenimiento.
4. **Implementación de la orquestación con *Kubernetes***: Desplegar los diferentes componentes del servicio como Pods en un clúster de Kubernetes, asegurando su correcta distribución entre múltiples nodos para mejorar la resiliencia y el balanceo de carga.
5. **Documentación y análisis de resultados**: Documentar el proceso de diseño e implementación, así como los resultados obtenidos, destacando las mejoras logradas en comparación con la arquitectura inicial y proponiendo posibles extensiones futuras.

Con la consecución de estos objetivos, se espera proporcionar una solución robusta y moderna que no solo cumpla con los requerimientos actuales, sino que también ofrezca una base sólida para futuras expansiones y mejoras del servicio.

1.3 Estructura de la memoria

El documento se organiza en los siguientes capítulos:

- **Introducción**. Este capítulo presenta la motivación detrás del proyecto, los objetivos que se pretenden alcanzar, la estructura del documento y las convenciones de formato utilizadas.
- **Estado del arte**. En este capítulo se revisan las tecnologías de contenedores, la orquestación de estos y las estrategias para asegurar alta disponibilidad y resiliencia.
- **Análisis**. Se analiza el proyecto desde varias perspectivas: riesgos, planificación, y la arquitectura del servicio inicial, incluyendo sus componentes y limitaciones.

- **Diseño de la nueva arquitectura**. Este capítulo propone el diseño de la nueva infraestructura y la arquitectura de la aplicación para satisfacer los requisitos del proyecto.
- **Implementación de la solución**. Se detalla la configuración de contenedores y *PODS* empleados, así como la orquestación con *Kubernetes*, incluyendo las estructuras necesarias para proporcionar soporte y resiliencia.
- **Conclusión e implementaciones futuras**. Se resumen los hallazgos y conclusiones finales del proyecto, además de describir futuras ampliaciones del servicio.
- **Bibliografía**. Lista de las fuentes y referencias utilizadas en el desarrollo del proyecto.
- **Glosario**. Definiciones y explicaciones de los términos técnicos y conceptos clave utilizados en el documento.
- **Apéndices**. Se incluyen detalles adicionales sobre *Kubernetes* y la relación del proyecto con los Objetivos de Desarrollo Sostenible (*ODS*).

1.4 Convenciones de formato

La memoria, por su naturaleza computacional y desarrollo informático internacional, contiene varias palabras en un idioma extranjero donde, principalmente, se utiliza el inglés para facilitar la comprensión entre los profesionales. En este texto, las palabras en inglés que se han obtenido del léxico técnico informático se escribirán en cursiva. Del mismo modo, los nombres de aplicaciones de *software* y los acrónimos también se escribirán en cursiva en todas sus apariciones.

Adicionalmente y como se comprobará a continuación, los objetos propios de la tecnología *Kubernetes* seguirán las directrices indicadas en su hoja de estilo nativa [1].

[1] https://kubernetes.io/docs/contribute/style/style-guide/

CAPÍTULO 2
Estado del arte

Comprender el panorama actual de las tecnologías de uso de contenedores, los sistemas de orquestación y las estrategias para garantizar alta disponibilidad y resiliencia es esencial para diseñar y desplegar arquitecturas de servicios en la nube que sean eficientes, escalables y confiables.

La creciente demanda de aplicaciones y servicios que puedan responder de manera inmediata y continua a las necesidades de los usuarios ha impulsado la adopción de tecnologías que permitan desplegar y gestionar infraestructuras de forma más ágil y flexible. En este contexto, los contenedores han emergido como una solución eficaz para empaquetar y distribuir aplicaciones con todas sus dependencias, asegurando consistencia entre diferentes entornos y facilitando la escalabilidad horizontal.

No obstante, la gestión eficiente de múltiples contenedores en entornos de producción requiere de herramientas avanzadas que automaticen tareas como el despliegue, escalado y monitoreo de aplicaciones. Es aquí donde entran en juego los sistemas de orquestación de contenedores, siendo *Kubernetes* el más destacado entre ellos debido a su robustez, extensibilidad y amplia adopción en la industria. Esta tecnología proporciona un conjunto de funcionalidades que simplifican la administración de aplicaciones que emplean contenedores, permitiendo mantener altos niveles de disponibilidad y resiliencia incluso bajo condiciones de carga variable o ante fallos inesperados.

El diseño de un servicio de ejecución de código que sea altamente disponible implica también la implementación de estrategias de resiliencia que aseguren la continuidad del servicio frente a posibles interrupciones. Esto incluye la utilización de patrones de diseño y prácticas como la replicación de servicios, balanceo de carga, despliegues continuos y mecanismos de recuperación automática. La combinación de estas estrategias con las capacidades proporcionadas por los contenedores y sistemas de orquestación resulta en una arquitectura capaz de adaptarse dinámicamente a las demandas operativas y garantizar una experiencia de usuario consistente y fiable.

A lo largo de este capítulo, se presentan los conceptos y tecnologías fundamentales sobre los cuales se basa el proyecto ya que son elementos cruciales para la arquitectura de servicios en la nube.

2.1 Tecnologías de contenedores

Para entender la relevancia de la tecnología y los cambios implementados en el proyecto actual, vamos a observar, en su justa medida, los diversos desarrollos previos que han contribuido a la creación y evolución de los contenedores, así como las tecnologías

de despliegue subsecuentes que han conllevado a alcanzar la situación actual, todo ello junto con sus respectivas limitaciones e impedimentos.

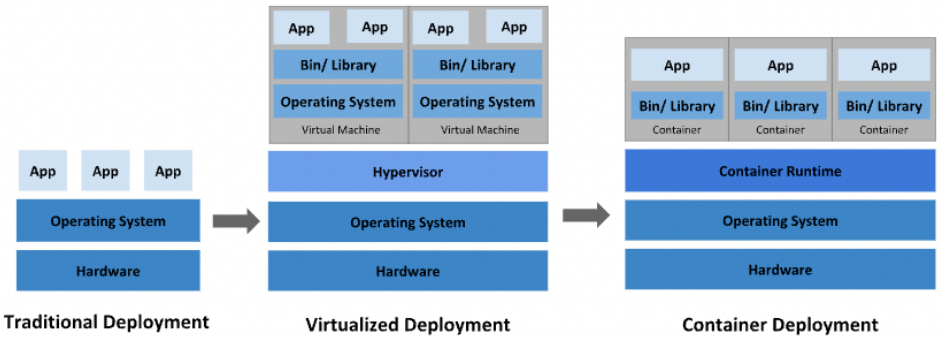

Figura 2.1: Evolución de las Tecnologías de Despliegue

Como podemos observar en la figura 2.1, al comienzo de la generación del software, las organizaciones ejecutaban aplicaciones en servidores físicos. Por la naturaleza de esta aproximación resultaba difícil (y requería de revisiones constantes con cada actualización) la definición de límites y mantenimiento de dependencias. Por una parte, en todos los sistemas Unix, han existido siempre diferentes órdenes para limitar los recursos accesibles para cada proceso, como `ulimit` se pueden establecer topes a la memoria asignada, con `nice` se puede rebajar la prioridad de acceso al procesador, etc. La problemática surgía cada vez que, al generar versiones nuevas del servicio que se brindaba, esos límites era altamente probable que debieran cambiar. Además, estos dependían de los recursos disponibles en el sistema anfitrión.

Como solución se introdujo la **virtualización**, que permite ejecutar múltiples Máquinas Virtuales (VMs) en el hardware de un único nodo físico. Esta permite una mejor utilización de los recursos y una mejor escalabilidad, ya que una aplicación puede añadirse o actualizarse de forma más sencilla, reduciendo los costos de hardware, tiempos de despliegue, etc. Por otra parte, la virtualización seguía representando una forma relativamente pesada de desplegar aplicaciones. Si bien aumentaba el aislamiento, trabajar con máquinas virtuales implica tener que virtualizar los sistemas operativos, con ayuda de un hipervisor, en sus diferentes versiones y esquemas de arquitectura, la virtualización presenta una serie de desventajas que podrían establecerse como esenciales en el contexto actual, como puede ser la disminución de la velocidad de ejecución del hardware emulado o la velocidad de escalado de máquinas virtuales[1]. Además, si nos centramos únicamente en la lógica de negocio, el uso de máquinas virtuales para ejecutar el software elegido implicaría la necesidad de instalar todas las dependencias necesarias dentro del sistema virtualizado mediante imágenes de sistema operativo, servidores de arranque o *scripts* que permitan realizar los pasos necesarios para obtener las configuraciones deseadas. Eso, en un sistema tradicional sin virtualización ni contenedores, resultaba muy complejo, aunque no imposible, ya que desde hace algunos años es posible modificar la ubicación de las bibliotecas a utilizar mediante la asignación de valores a variables de entorno.

Estas desventajas anteriormente mencionadas son, en su mayoría, solventadas con la creación de los **contenedores**, que siguen una lógica similar a las VMs, pero tienen propiedades de aislamiento relajadas para compartir el sistema operativo (SO) entre las aplicaciones, considerándose livianos. Al igual que una VM, un contenedor tiene su propio sistema de archivos, una parte de la CPU, memoria, espacio de procesos, y más. Al

estar desacoplados de la infraestructura subyacente, adquieren propiedades como la portabilidad a través de diferentes nubes, equipos y distribuciones de SO.

Son una buena forma de empaquetar y ejecutar aplicaciones, ya que, en un entorno con servidores en producción[1], se necesita gestionar los contenedores que ejecutan las aplicaciones y asegurar que no existe tiempo de inactividad. Es en este caso donde *Kubernetes* proporciona un marco para ejecutar sistemas distribuidos de manera resiliente, encargándose de la escalabilidad y recuperación ante errores, proporcionando patrones de despliegue, etc.

2.2 Alta disponibilidad y resiliencia

La alta disponibilidad y la resiliencia son aspectos esenciales para cualquier sistema que debe operar de manera continua y sin interrupciones significativas. Estos conceptos son fundamentales en el contexto de aplicaciones en la nube, donde la capacidad de mantener los servicios accesibles y operativos, incluso ante fallos de hardware, interrupciones de red o actualizaciones de software, es crucial para garantizar una experiencia de usuario consistente y fiable.

Por una parte, la alta disponibilidad (conocida también por sus siglas en inglés *HA: High Availability*) se refiere a la capacidad de un sistema para asegurar un funcionamiento ininterrumpido y minimizar el tiempo de inactividad [3]. Para lograr alta disponibilidad, se emplean varias estrategias, entre las cuales destaca la **replicación**, que implica la creación de múltiples instancias de un componente del sistema. Al tener réplicas en diferentes nodos o ubicaciones, el sistema puede continuar operando incluso si una instancia falla o un nodo se vuelve inactivo. Esta técnica asegura que siempre haya suficientes instancias activas para manejar la carga y mantener el servicio disponible.

Por otra parte, otra estrategia clave para mantener alta disponibilidad es el empleo de balanceo de carga. Consiste en distribuir el tráfico de red entre varias instancias de un servicio o aplicación [4]. Esto evita que una sola instancia se sobrecargue y asegura que el tráfico se dirija a instancias saludables y operativas, ayudando a equilibrar la demanda y mejorando la eficiencia y disponibilidad del sistema.

La resiliencia, por otro lado, se refiere a la capacidad de un sistema para recuperarse rápidamente de fallos y adaptarse a cambios o perturbaciones sin comprometer su funcionamiento. Según el Diccionario de la lengua española se define resiliencia como la capacidad de un material, mecanismo o sistema para volver a su estado original una vez que la perturbación a la que estaba sometido ha cesado. En especial, el término resiliencia, enmarcado en el campo de los sistemas de computación, se ha empleado como un sinónimo de tolerancia a fallos [5], es decir, como la habilidad de un sistema para ofrecer y mantener un nivel de servicio adecuado frente a fallos y dificultades durante su operación normal [6].

Es decir, en los sistemas altamente disponibles, la alta disponibilidad y resiliencia son pilares fundamentales para el funcionamiento continuo y fiable de sistemas y aplicaciones. Implementar estas estrategias asegura que los servicios se mantengan operativos y eficientes, incluso en situaciones adversas, y que puedan recuperarse rápidamente de fallos o interrupciones.

[1] Entendemos como servidores de producción aquellos entornos "donde finalmente se ejecuta la aplicación donde acceden los usuarios finales y donde se trabaja con los datos de negocio en sí mismos.[2]"

CAPÍTULO 3
Análisis del problema

3.1 Análisis de la arquitectura del servicio inicial

3.1.1. Descripción del servicio actual

El objetivo que cumple este servicio se centra en la publicación de un ejecutor en la nube basado en contenedores. Se ha seleccionado como ejemplo de servicio una arquitectura que permite gestionar un modelo *Function as a Service (FaaS)*. El concepto de *FaaS* se desarrollará más adelante.

El presente servicio tuvo su origen en el marco de la asignatura *Cloud Computing*, perteneciente al Máster Universitario en Computación en la Nube y de Altas Prestaciones. Su diseño inicial se concibió con el propósito de proporcionar una comprensión práctica de las arquitecturas subyacentes a una plataforma *Function as a Service*. En esta fase preliminar, el enfoque se centraba en la construcción de una solución sencilla y funcional que facilitara el despliegue de funciones sobre una única máquina anfitriona, empleando la tecnología de contenedores *Docker*. Esta decisión fue clave para garantizar que la implementación fuera manejable dentro del ámbito académico y permitiera aplicar conceptos fundamentales de manera práctica.

El objetivo actual es extender ese prototipo inicial a un entorno más realista, donde se emplee una arquitectura distribuida y escalable que pueda desplegarse y gestionarse sobre múltiples anfitriones utilizando *Kubernetes*. Este paso ha facilitado la introducción de mejores prácticas en cuanto a gestión de contenedores, escalabilidad automática y resiliencia, características esenciales en la operativa de sistemas en la nube de alta disponibilidad.

El servicio, actualmente, basa su funcionamiento en el empleo de contenedores, la tecnología *NATS* y en el uso del lenguaje de programación *Go* [7].

Por una parte, la tecnología *NATS* (según sus siglas en inglés “Neural Autonomic Transport System")[8] es un sistema de mensajería de código abierto que se destaca por su diseño ligero y eficiente. Está desarrollado para facilitar la transferencia de mensajes entre aplicaciones distribuidas, proporcionando una comunicación rápida y fiable en entornos escalables. Su arquitectura permite la conexión de diversos servicios de manera ágil para conseguir una baja latencia, lo que, en un entorno distribuido basado en contenedores, se considera esencial.

Adicionalmente, y para seguir proporcionando características como la escalabilidad y baja latencia, el servicio implementa patrones internos de comunicación de tipo publicación/suscripción (pub/sub) complementados mediante colas de trabajadores para gestionar la interacción entre los distintos componentes de la red.

FaaS: Function as a Service

La Función como Servicio (en inglés *Function as a Service*), conocida por sus siglas *FaaS*, es un modelo de computación en la nube que, según Roberts, M. [9], permite a los desarrolladores ejecutar código en respuesta a eventos sin la complejidad de construir y mantener la infraestructura subyacente.

En otras palabras, este tipo de funciones permiten ejecutar código de la capa de negocio sin tener que gestionar la infraestructura informática y las aplicaciones de servidor de larga duración[1] subyacentes que conlleva la creación y disposición del servicio.

El concepto de *FaaS* viene fuertemente ligado al contexto de la novedosa arquitectura de sistemas tipo *serverless*. La computación *serverless* es una implementación parcial de un ideal basado en la computación impulsada por eventos, en el cual las aplicaciones son definidas por acciones y los eventos que las desencadenan.

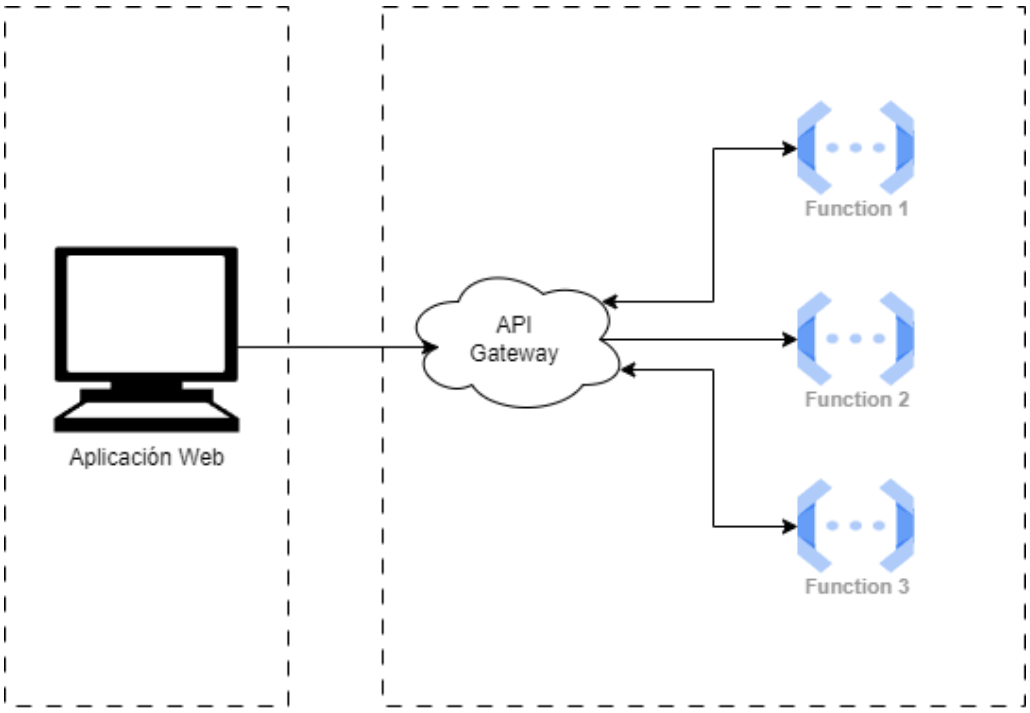

Figura 3.1: Arquitectura Serverless

Este tipo de computación ha demostrado ser una buena opción en algunos campos, como por ejemplo en las aplicaciones de *IoT*. Por ejemplo, grandes empresas como *Amazon Web Services* están centrando sus esfuerzos actualmente para integrar computación *serverless* dentro de la "jerarquía de los centros de datos" para potenciar así la proliferación prevista de dispositivos *IoT*.

La computación *serverless* permite a los desarrolladores de aplicaciones descomponer grandes aplicaciones en pequeñas funciones, como podemos observar en la figura 3.1, lo que permite escalar individualmente cada componente de la aplicación. Sin embargo esto presenta un nuevo problema: dejamos de lado la gestión general de infraestructura, orquestación, resiliencia, etc; y necesitamos centrar nuestros esfuerzos en la gestión coherente de una gran cantidad de funciones.[10]

[1] Cabe destacar que, el término de infraestructura de larga duración, es una diferencia clave al compararlo con otras tendencias arquitectónicas como, por ejemplo, los contenedores.

3.1.2. Componentes del servicio

Para comprender el objetivo y requerimientos del servicio actual podemos observar su arquitectura.

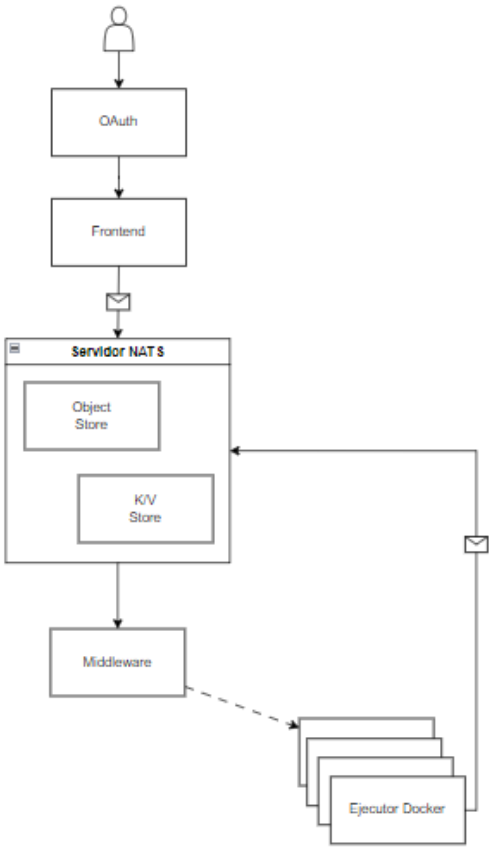

Figura 3.2: Arquitectura del proyecto

Como se puede observar en la figura 3.2, el punto de acceso inicial al sistema es el contenedor encargado de la autenticación, implementado mediante la tecnología *OAuth*. Tras una autenticación exitosa a través de esta capa de *OAuth*, el usuario obtiene acceso a la capa de presentación final, desde donde podrá interactuar con el servicio *FaaS* (Función como Servicio) diseñado. Este enfoque de diseño se adopta con el propósito de aislar los componentes internos del sistema, impidiendo el acceso directo del usuario final a elementos críticos de la infraestructura, lo que refuerza la seguridad y garantiza la autenticidad de los mensajes mediante el uso de una red privada y aislada. En este contexto, el servidor *NATS* desempeña un papel fundamental, gestionando la comunicación entre los diferentes componentes de la arquitectura, concretamente entre la capa de presentación y el *middleware*.

El *middleware*, desarrollado en el lenguaje de programación *Go*, es uno de los componentes clave del sistema, encargado de procesar los eventos provenientes de la cola de mensajes y de iniciar las correspondientes imágenes *Docker* para la ejecución de las tareas solicitadas. Al inicio de su operación, el *middleware* se conecta a la cola gestionada por *NATS* y se suscribe a los eventos que coincidan con los patrones “cola.go”, “cola.javascript” y “cola.c”. Para cada trabajo recibido, se genera un identificador único que es almacenado en el *Key-Value Store* proporcionado por *NATS*, etiquetando el esta-

do de dicho trabajo como "Pendiente". Posteriormente, se lanza el ejecutor adecuado, en función del lenguaje de programación requerido en la capa de presentación.

Para comprender las mejoras derivadas de la integración de un orquestador de contenedores en la arquitectura del sistema, es fundamental analizar cómo se gestionan los flujos de trabajo. En este caso, la estrategia seleccionada para la ejecución de tareas en el *middleware* se basa en el paradigma conocido como **Docker-in-Docker (DinD)**. Esta técnica permite ejecutar contenedores *Docker* dentro de otros contenedores *Docker*, lo que otorga al contenedor externo acceso al *daemon* de *Docker* del sistema anfitrión. Gracias a este acceso, el contenedor externo puede realizar operaciones como la construcción y ejecución de otros contenedores. Esta operación es posible mediante el uso del *socket* de *Unix*, ubicado en `/var/run/docker.sock`, que sirve como punto de entrada principal para comunicarse con el *daemon* de *Docker* en el sistema *host*. En la infraestructura diseñada, este *socket* se expone al *middleware* mediante el montaje de un volumen *Docker*, permitiendo así una interacción directa con el sistema anfitrión para la creación y gestión de contenedores adicionales.

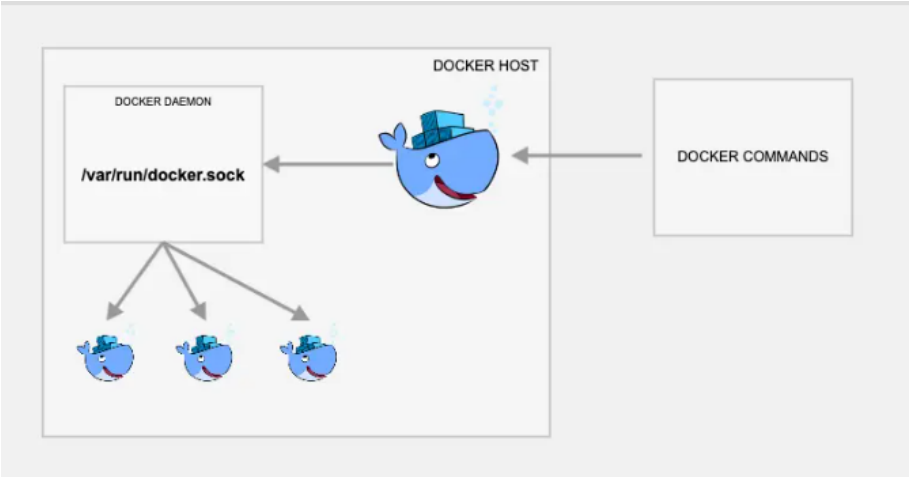

Figura 3.3: Funcionamiento DinD

Networking

El sistema completo emplea tres subredes virtualizadas con el fin de evitar que los usuarios finales accedan a componentes de la capa de negocio, lo cual representaría un grave riesgo para la integridad y seguridad del sistema. Para mitigar estos riesgos, se han definido tres *subnets*, agrupadas de la siguiente manera:

- **Red *Frontend***: Esta red, de tipo puente (*bridge*), conecta los contenedores que componen el servicio de *frontend* y autenticación, como se puede observar en la figura 3.4. Su función principal es rechazar todo tráfico externo que no haya sido identificado y autorizado por el servicio de *OAuth*.
- **Red Común**: También de tipo puente, esta red es responsable de interconectar los servicios principales (*core*) de la aplicación, como el servicio *Frontend*, el sistema de colas *NATS*, y el *middleware*. El uso del controlador de red tipo *bridge* permite que los contenedores se descubran mutuamente mediante sus nombres de servicio, facilitando la comunicación interna.
- **Red de Ejecutores**: Esta red conecta los contenedores ejecutores y el sistema de colas *NATS*. Dado que los contenedores ejecutores se despliegan momentáneamente

para realizar tareas específicas y luego se destruyen, es fundamental que estén aislados de la red principal. Esta red se utiliza para aliviar el tráfico de la red general, permitiendo una modularización más eficiente de los ejecutores. Solo se conecta al clúster de colas *NATS* para enviar los resultados de las tareas una vez completadas.

Como se puede observar, la arquitectura no incluye balanceadores de carga. Durante el diseño, se decidió no implementarlos por las siguientes razones:

- Al emplear redes de *Docker* tipo *bridge* para conectar servicios en la misma máquina *host*, la comunicación entre los servicios es directa, sin necesidad de pasar por un balanceador de carga. En este contexto, un balanceador de carga podría generar cuellos de botella en escenarios de tráfico elevado.[2]

- La gestión de colas *NATS* ya distribuye eficientemente los mensajes entre los servicios suscritos a temas específicos. Durante las pruebas, el uso de balanceadores de carga con múltiples instancias de *NATS* no mostró mejoras significativas, y solo incrementó la complejidad del sistema.

- La infraestructura actual es relativamente sencilla, con servicios directamente conectados entre sí. En entornos más complejos, donde los servicios son distribuidos dinámicamente a través de múltiples nodos, el uso de balanceadores de carga se vuelve más importante, pero no es necesario en esta implementación.

[2] El diseño ideal en una infraestructura más compleja sería similar al utilizado en clústeres de *Kubernetes*, donde un servicio de balanceo gestiona la distribución del tráfico entre réplicas. Sin embargo, en nuestra opinión, agregar este nivel de complejidad en el contexto de *docker-compose* sería innecesario.

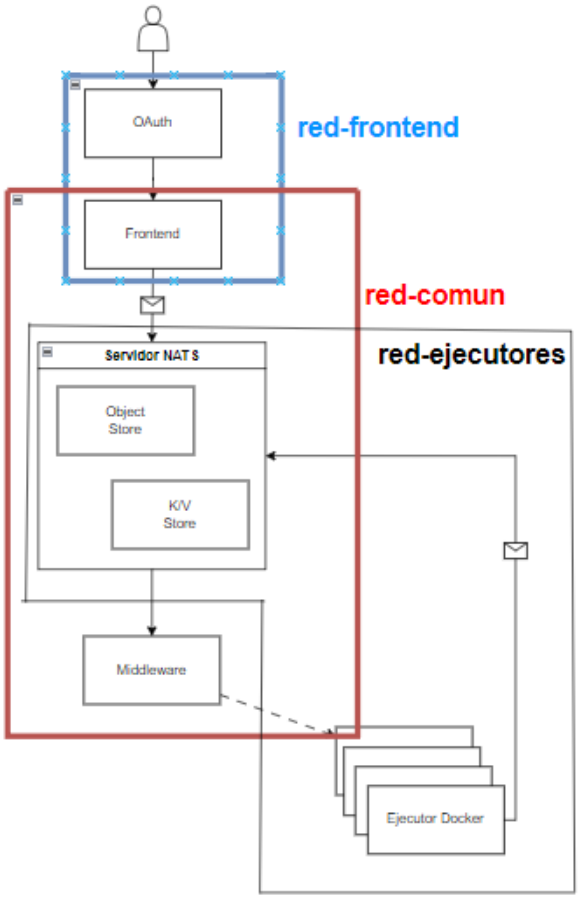

Figura 3.4: Estructura de red

Este tipo de diseño nos permite aliviar carga del *middleware*, ya que simplemente se encargará de procesar los mensajes de la cola de entrada y lanzar el contenedor correspondiente. No se encarga de monitorizarlos, ya que el contenedor ejecutor generado se encargará de procesar y compilar la solicitud, empaquetarla, y enviarla a la cola de salida. Esta lógica permite que el *middleware* procese muchas más solicitudes sin necesidad de ningún tipo de escalado, ya que su límite dependerá de la capacidad de la máquina anfitrión.

Ejecutores

Los ejecutores son pequeños contenedores generados dinámicamente por el *middleware*, cuya función principal es descargar, compilar y procesar el código proporcionado por el usuario. Además, estos ejecutores se encargan de empaquetar el resultado y enviarlo a las colas de salida, desde donde será recogido por el servicio de la capa de presentación. En nuestra aplicación, como se ha mencionado anteriormente, se admite el procesamiento de tres tipos de lenguajes distintos. Para ello, se crean tres imágenes *Docker* independientes, configuradas para recibir las dependencias de los archivos proporcionados por el usuario y establecer entornos óptimos para la ejecución y procesamiento del código.

Dado que estas instancias están fuera del ámbito de la orquestación inicial, se inician e integran en el resto del sistema mediante un enfoque declarativo, utilizando la lógica definida en el archivo `docker-compose`. En términos prácticos, esto se lleva a cabo a través de una librería de *Golang*, que ejecuta el comando `docker run` con los parámetros

adecuados (como identificadores de usuario o redes *Docker* a las que deben conectarse) para que los contenedores puedan cumplir su función correctamente.

Las tareas de ejecución, compilación y recolección de datos se gestionan mediante un *script* `bash` que se define como `ENTRYPOINT` durante la compilación de la imagen, utilizando el parámetro `CMD` para especificar el valor por defecto que permite `docker run` al momento de la ejecución.

Despliegue

El despliegue del servicio actual se realiza mediante la tecnología *Docker Compose*. *Docker Compose* es una herramienta que permite definir y ejecutar aplicaciones *Docker* de múltiples contenedores. Mediante *Docker Compose*, se puede utilizar un archivo de configuración, normalmente llamado `docker-compose.yml`, para describir la configuración de los servicios, redes y volúmenes que necesita el servicio a desplegar. Tras el proceso de creación del archivo de configuración, podemos arrancar toda la infraestructura mediante la aplicación de un único comando [11].

El archivo *Docker Compose* proporciona una representación estructurada de la infraestructura y servicios que conforman el servicio general. El documento está organizado en secciones que representan diferentes servicios. Cada servicio, como *oauth*, *frontend*, *middleware*, y *nats-1*, tiene su propia configuración específica que incluye detalles como la imagen a utilizar, dependencias con otros servicios, configuración de redes, y, en algunos casos, *healthchecks*.

Para evitar levantar servicios que incurran en errores de arranque se realiza un proceso de orquestación mediante el establecimiento de **dependencias entre servicios**, mediante la sección ***depends_on***. Es decir, el orden de arranque de los servicios se gestiona automáticamente por *Docker Compose*, en función de las dependencias declaradas. El orden es el siguiente:

1. `NATS-1`: Este servicio es el primero en iniciarse, ya que no depende de ningún otro servicio.
2. *Frontend* y *Middleware*: Ambos servicios se inician tras el arranque de `NATS-1`. Se ha configurado que esperen hasta que el contenedor de *NATS* alcance el estado `service_started`, lo que significa que no comenzarán sus procesos hasta que `NATS-1` haya iniciado. Este estado indica únicamente que el servicio ha comenzado a ejecutarse, sin garantizar que esté completamente funcional o saludable.
3. *OAuth*: Este es el último servicio en iniciar y depende de que tanto *Frontend* como *Middleware* se encuentren en estado saludable. Es decir, el servicio *OAuth* espera a que ambos servicios anteriores lleguen al estado de `service_healthy`. La salud de un servicio se verifica a través de *healthchecks*, que son comprobaciones realizadas por *Docker* para asegurar que el servicio está en un estado funcional. En el caso de *Frontend*, el *healthcheck* valida que el servicio web sea accesible localmente, mientras que para *Middleware*, verifica que el proceso del programa *Go* esté escuchando el tráfico SSL entrante.

3.2 Análisis de riesgos, limitaciones y áreas de mejora

En el proceso de modernización de la infraestructura de nuestra aplicación, es fundamental seleccionar herramientas y tecnologías que no solo soporten el desarrollo y pruebas, sino que también ofrezcan robustez, escalabilidad y alta disponibilidad en entornos

de producción. Hasta ahora, la arquitectura planteada ha utilizado la herramienta *Docker Compose*. Si bien *Docker Compose* ha sido valiosa para el desarrollo rápido y la gestión de aplicaciones multicontenedor en entornos controlados, sus limitaciones se hacen evidentes cuando la aplicación debe ser desplegada a gran escala en un entorno de producción.

3.2.1. Limitaciones de *Docker Compose*

Docker Compose permite la definición y ejecución de aplicaciones multicontenedor mediante archivos *YAML*, lo que simplifica la orquestación en entornos de desarrollo. Sin embargo, tiene ciertas limitaciones clave que la hacen menos adecuada para entornos de producción:

- **Escalabilidad limitada:** Aunque *Docker Compose* permite escalar servicios utilizando la opción `scale`, este proceso es manual y no se adapta dinámicamente a las necesidades de la aplicación basadas en la carga de trabajo, donde la demanda puede variar considerablemente y la falta de procesos de auto-escalado basado en métricas (como el uso de CPU o memoria) puede llevar a un rendimiento ineficiente o a la sobrecarga de los servicios.
- **Gestión básica de fallos:** *Docker Compose* tiene la capacidad de reiniciar contenedores si fallan, pero no proporciona un mecanismo para reprogramar automáticamente estos contenedores en otros nodos dentro de un clúster. Esto significa que en caso de un fallo en el *host*, los servicios pueden quedar inaccesibles hasta que se realice una intervención manual, lo que no es ideal para entornos de alta disponibilidad.
- **Limitaciones de orquestación en clústeres distribuidos:** *Docker Compose* está diseñado para operar principalmente en un solo *host*. Esto limita su capacidad para gestionar aplicaciones distribuidas que necesitan ejecutarse en múltiples nodos para garantizar escalabilidad horizontal y resiliencia. La falta de capacidades nativas para balancear la carga, gestionar redes distribuidas o coordinar contenedores a través de diferentes nodos hace que *Docker Compose* no sea adecuado para infraestructuras de producción modernas y distribuidas.
- **Falta de soporte para almacenamiento persistente distribuido:** Aunque *Docker Compose* permite el uso de volúmenes para el almacenamiento persistente, no ofrece soluciones avanzadas para gestionar el almacenamiento en un clúster distribuido, donde los datos deben ser accesibles desde múltiples nodos y deben ser resilientes a fallos.
- **Ausencia de despliegues continuos y actualizaciones sin interrupciones:** *Docker Compose* no ofrece soporte nativo para estrategias de despliegue avanzadas como las actualizaciones continuas (Rolling Updates), lo que es crucial para minimizar el tiempo de inactividad durante las actualizaciones de software en producción.

Aunque *Docker Compose* ha demostrado ser una herramienta valiosa para el desarrollo y la gestión de aplicaciones multicontenedor en entornos controlados, sus limitaciones lo hacen menos adecuado para el despliegue de aplicaciones a gran escala. La falta de capacidades avanzadas en áreas clave como la escalabilidad automática, la gestión de fallos en clústeres distribuidos, el almacenamiento persistente y los despliegues continuos sin interrupciones, resalta la necesidad de adoptar una solución más robusta y flexible. Es por esto que se hace necesario migrar hacia plataformas de orquestación más avanzadas y diseñadas para superar estas limitaciones y satisfacer las demandas de aplicaciones modernas en producción.

3.3 Planificación del Proyecto

La planificación del proyecto se estructura en cinco fases principales: **Análisis**, **Diseño**, **Implementación**, **Test** y **Documentación**. Estas fases se definen utilizando una **Estructura de Desglose del Trabajo** (Figura 3.5). La estructura de desglose del trabajo, en adelante *EDT*, permite descomponer el trabajo en componentes más manejables, facilitando la asignación de recursos y la estimación de tiempos.

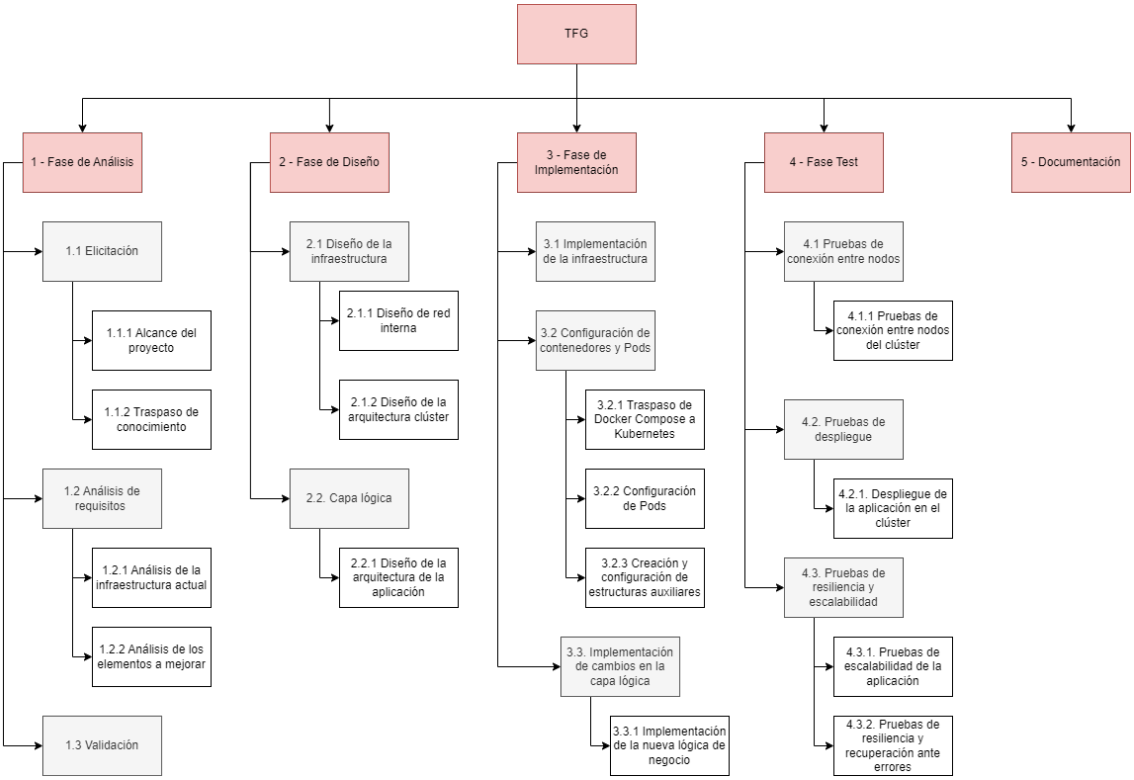

Figura 3.5: Estructura de Desglose del Trabajo (*EDT*)

Para una visualización más clara y detallada de la planificación de cada fase y sus respectivas tareas, se ha utilizado un **Diagrama de Gantt**. Este diagrama, mostrado en la Figura 3.7, proporciona una representación gráfica del cronograma del proyecto. El Diagrama de Gantt permite observar de manera intuitiva la duración de las tareas, sus fechas de inicio y finalización, y las dependencias entre ellas.

Además, en nuestro caso, emplea un código de colores para facilitar la identificación de las diferentes fases y tareas dentro del esquema de planificación.

TASK	PROGRESS	START	END
Fase de análisis			
Alcance del proyecto	100%	7/1/24	7/4/24
Traspaso de conocimiento	100%	7/3/24	7/5/24
Análisis de la infraestructura actual	100%	7/5/24	7/9/24
Análisis de los elementos a mejorar	100%	7/6/24	7/11/24
Validación	100%	7/3/24	7/29/24
Fase de diseño			
Diseño de la red interna	100%	7/30/24	8/3/24
Diseño de la arquitectura de clúster	100%	8/3/24	8/8/24
Diseño de la arquitectura de la aplicación	100%	8/8/24	8/15/24
Fase de implementación			
Implementación de la infraestructura	100%	8/15/24	8/25/24
Configuración de contenedores y Pods	100%	8/19/24	8/26/24
Orquestación: Implementación de estructuras de soporte	100%	8/22/24	8/28/24
Implementación de cambios en la capa lógica	100%	8/28/24	9/2/24
Fase de test			
Pruebas de conexión entre nodos	100%	8/22/24	8/26/24
Pruebas de despliegue	100%	8/29/24	9/5/24
Pruebas de resiliencia y escalabilidad	100%	8/30/24	9/5/24
Documentación			
Documentación	100%	7/1/24	9/5/24

Figura 3.6: Tabla de Tareas del Diagrama de Gantt

Como observamos, de nuevo, en la figura 3.7 el proyecto comienza con su **fase de análisis**, el 1 de julio de 2024. Esta fase es crucial para comprender las necesidades del proyecto. Comienza con la tarea de conocer el **alcance** del mismo, donde se define claramente lo que se espera lograr y asegura que todos los involucrados estén alineados respecto a los objetivos y limitaciones.

Posteriormente, se lleva a cabo el **traspaso de conocimiento**, donde se recopila y transfiere el conocimiento necesario del sistema actual a todas las partes involucradas, asegurando que todas entienden las tecnologías y arquitecturas existentes.

La siguiente tarea es el **análisis de la infraestructura actual**, que implica examinar la infraestructura ya implementada, identificando sus fortalezas y debilidades. Esto se complementa con el **análisis de los elementos a mejorar**, donde se identifican las áreas específicas que requieren mejoras para cumplir con los nuevos objetivos.

Finalmente, se realiza la **validación** de los resultados del análisis, asegurando que los hallazgos y las recomendaciones estén alineados con las metas del proyecto y que las soluciones propuestas sean factibles.

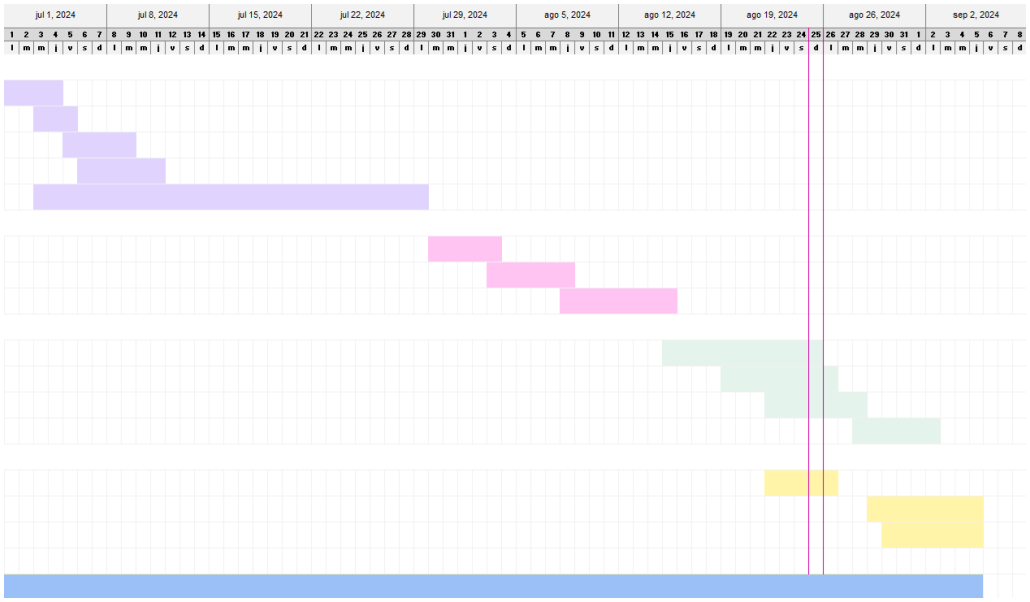

Figura 3.7: Diagrama de Gantt

Después de la fase de análisis, y aproximadamente a principios de agosto, se inicia la siguiente etapa crucial en el desarrollo de la nueva arquitectura: la **fase de diseño**.

Este periodo comienza con el **diseño de la red interna**, en el que se planifica cuidadosamente la arquitectura de red que sustentará el clúster de contenedores. El objetivo es garantizar una conectividad robusta y segura entre los diferentes nodos, fundamental para el rendimiento y la seguridad del sistema.

Posteriormente, se procede al **diseño de la arquitectura del clúster**, donde se define la distribución y gestión de los contenedores a lo largo de los nodos. Aquí, se optimizan aspectos clave como la alta disponibilidad y el escalado automático, asegurando que el sistema pueda manejar incrementos en la carga de trabajo sin comprometer la estabilidad.

Finalmente, se lleva a cabo el **diseño de la arquitectura de la aplicación**. En esta etapa, se adapta la aplicación existente a la nueva infraestructura del clúster, integrando las modificaciones necesarias para su orquestación. El diseño se enfoca en asegurar que la aplicación funcione de manera eficiente y sin contratiempos en el nuevo entorno.

Concluida la fase de diseño, se avanza a la **fase de implementación**, donde el diseño conceptual se convierte en una realidad funcional.

Esta fase comienza con la **implementación de la infraestructura**, que abarca la configuración de servidores, redes y almacenamiento necesarios para soportar el clúster. Una vez completado este paso, se procede a la **configuración de contenedores y *Pods***, donde los contenedores se implementan en el clúster y los *Pods* se configuran para asegurar la correcta ejecución de las aplicaciones.

El siguiente paso es la **orquestación e implementación de estructuras de soporte**, en el cual se configuran los aspectos específicos de la orquestación proporcionados por *Kubernetes*, garantizando una administración eficiente y automatizada del clúster.

Finalmente, se realiza la **implementación de cambios en la capa lógica**, donde se ajusta la lógica de la aplicación para que funcione de manera óptima en el nuevo entorno de clúster, asegurando su compatibilidad y rendimiento.

Una vez completada la implementación, se procede a la **fase de pruebas** para verificar que todo funcione según lo planeado.

Esta fase comienza con las **pruebas de conexión entre nodos**, que aseguran que todos los nodos del clúster se comuniquen de manera efectiva y segura. Luego, se realizan las **pruebas de despliegue**, donde se valida que las aplicaciones se desplieguen correctamente en el clúster y que los procesos de orquestación operen como se espera.

Finalmente, se ejecutan las **pruebas de resiliencia y escalabilidad**, que incluyen pruebas de estrés para verificar que el sistema pueda escalar según la demanda y que sea capaz de recuperarse de fallos sin pérdida de datos ni funcionalidad.

Por último, es interesante remarcar el caso especial de la **fase de documentación**. Esta fase implica la creación de documentación detallada que cubra todos los aspectos del proyecto, desde el análisis inicial hasta la implementación y pruebas finales y, como se ha podido observar en la figura 3.7, se desarrolla a lo largo de todo el proyecto. Esto es debido a su impacto en la calidad y eficiencia del trabajo.

Documentar de manera continua asegura que toda la información relevante, decisiones y cambios se registren de manera precisa y oportuna, lo que facilita la comunicación entre los miembros del equipo y evita la pérdida de datos importantes. Esta práctica permite una mejor gestión del conocimiento, reduce la posibilidad de errores por omisión y proporciona una base sólida para la revisión y el mantenimiento futuro del proyecto.

Además, mediante la generación constante de documentación, cumplimos con las características ideales de la documentación técnica de proyectos [12], como la evolución constante de la misma, incorporación de navegabilidad documental mediante el empleo de índices y desgloses actualizados, etc.

Este enfoque estructurado y meticuloso garantiza que la nueva arquitectura cumpla con los más altos estándares de rendimiento, seguridad y escalabilidad.

CAPÍTULO 4
Diseño de la nueva arquitectura

Dado que la aplicación necesita evolucionar para cumplir con los requisitos de un entorno de producción moderno y distribuido, se ha decidido migrar la infraestructura a *Kubernetes*.

> **Nota:** En este capítulo se introducen varios términos y conceptos técnicos relacionados con la tecnología *Kubernetes*. Si fuera necesario profundizar en la comprensión de estas estructuras o si se desea obtener una explicación más detallada de la terminología utilizada, se recomienda consultar el *Anexo* A, donde se amplía toda la información relevante sobre la tecnología y se referencia la documentación disponible sobre la misma.

Kubernetes es una plataforma de orquestación de contenedores que fue diseñada específicamente para resolver las limitaciones mencionadas y ofrecer una serie de beneficios clave:

- **Autoescalado:** Con *Kubernetes*, podemos definir políticas de auto-escalado que permiten que los *Pods* (conjuntos de contenedores) se escalen automáticamente en respuesta a la demanda real de la aplicación, basándose en métricas como el uso de CPU o memoria. Esto asegura que los recursos se utilicen de manera eficiente y que la aplicación pueda manejar picos de carga sin intervención manual.
- **Gestión avanzada de fallos:** *Kubernetes* proporciona un sistema de auto-recuperación donde los contenedores fallidos son reiniciados automáticamente. Si un nodo completo falla, *Kubernetes* reprograma los *Pods* afectados en otros nodos del clúster, garantizando que los servicios sigan disponibles sin intervención manual.
- **Orquestación en clúster distribuido:** A diferencia de *Docker Compose*, *Kubernetes* fue diseñado para gestionar aplicaciones distribuidas a lo largo de múltiples nodos. Su capacidad para coordinar la distribución de *Pods* entre diferentes nodos permite un escalamiento horizontal efectivo y una alta disponibilidad.
- **Almacenamiento persistente distribuido:** Esta tecnología permite definir volúmenes persistentes que pueden ser montados por los *Pods* en cualquier nodo, lo que asegura la disponibilidad y persistencia de los datos incluso en entornos distribuidos.
- **Despliegues continuos y actualizaciones sin interrupciones:** *Kubernetes* soporta estrategias de despliegue avanzadas como *Rolling Updates* y *Blue-Green Deployments*, que permiten actualizar la aplicación sin interrumpir el servicio. Además, ofrece la

Nombre	Sistema Operativo	CPU	Memoria	Almacenamiento
master01	Ubuntu-server-24.04	3	4096 MB	25.53 GB
worker01	Ubuntu-server-24.04	3	5119 MB	20,14 GB
worker02	Ubuntu-server-24.04	3	4096 MB	20,14 GB

Tabla 4.1: Especificaciones de las máquinas virtuales

posibilidad de realizar rollbacks en caso de que una actualización falle, lo que mejora la confiabilidad de las actualizaciones.

- **Extensibilidad y Ecosistema:** *Kubernetes* cuenta con un ecosistema amplio y extensible que permite integrar herramientas y servicios adicionales como *Prometheus* para monitoreo, *Helm* para gestión de paquetes, y muchas otras. Esta extensibilidad es vital para adaptar la plataforma a las necesidades específicas de la lógica de negocio.

La nueva arquitectura estará centrada en resolver las limitaciones actuales y proporcionar una base sólida para la futura expansión y mejora de la aplicación. En esta arquitectura, los diferentes servicios de la aplicación se desplegarán como *Pods* en un clúster de *Kubernetes*, distribuidos a lo largo de múltiples nodos para garantizar alta disponibilidad y resiliencia.

4.1 Diseño de la infraestructura

Para implementar un entorno de *Kubernetes* similar a un entorno de producción en el que realizar las pruebas necesarias necesitamos crear una serie de máquinas virtuales que actúen como *hosts* físicos.

Para ello se ha escogido emplear la tecnología *VirtualBox*. *VirtualBox* es un software de virtualización de código abierto que permite a los usuarios ejecutar múltiples máquinas virtuales, con su propio sistema operativo, simultáneamente en un solo equipo físico.

El laboratorio en cuestión donde se implementará el clúster se compone de un total de 3 máquinas virtuales. Una de ellas, de nombre `master01`, actuará como *Plano de Control*, mientras que las dos restantes, `worker01` y `worker02`, actuarán como nodos *worker*. La configuración de estas máquinas se ha basado en las aportaciones de [13, 14, 15, 16]. Adicionalmente, las especificaciones *hardware* que han sido empleadas en cada máquina virtual se pueden observar en la tabla 4.1.

Por otra parte en cada máquina virtual estarán activas dos interfaces de red. Esto se debe a que emplearemos, para obtener una comunicación efectiva entre todos los nodos, dos redes virtuales.

Como podemos observar en la figura 4.1, los puertos *Ethernet* de número idéntico de cada máquina, en adelante referidos con las siglas `eth`, estarán conectados a la misma red. En el caso de los puertos `eth0`, estarán vinculados a una red virtualizada de tipo *Host-Only*.

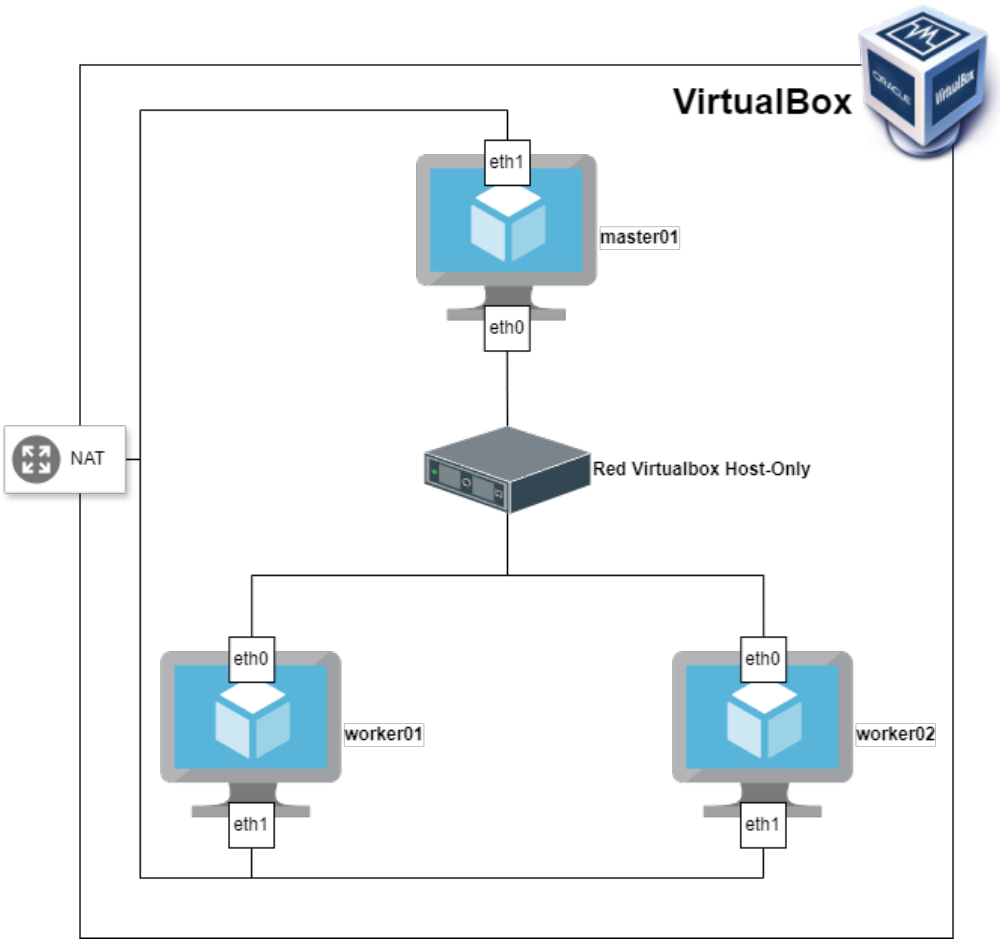

Figura 4.1: Arquitectura del clúster

Las redes *Host-Only* en *VirtualBox* permiten que las máquinas virtuales se comuniquen entre sí y con el host, pero sin acceso a Internet o a redes externas. Se puede considerar un híbrido entre las redes tipo *bridge* y las redes internas. Al igual que en la red tipo *bridge*, las máquinas virtuales pueden comunicarse entre sí y con el *host* como si estuvieran conectadas a través de un *switch Ethernet* físico.

Por otra parte, asemejando el modo de funcionamiento de una red interna, no es necesario que exista una interfaz de red física ya que las máquinas virtuales no pueden comunicarse con el mundo exterior porque no están conectadas a una interfaz de red física [17].

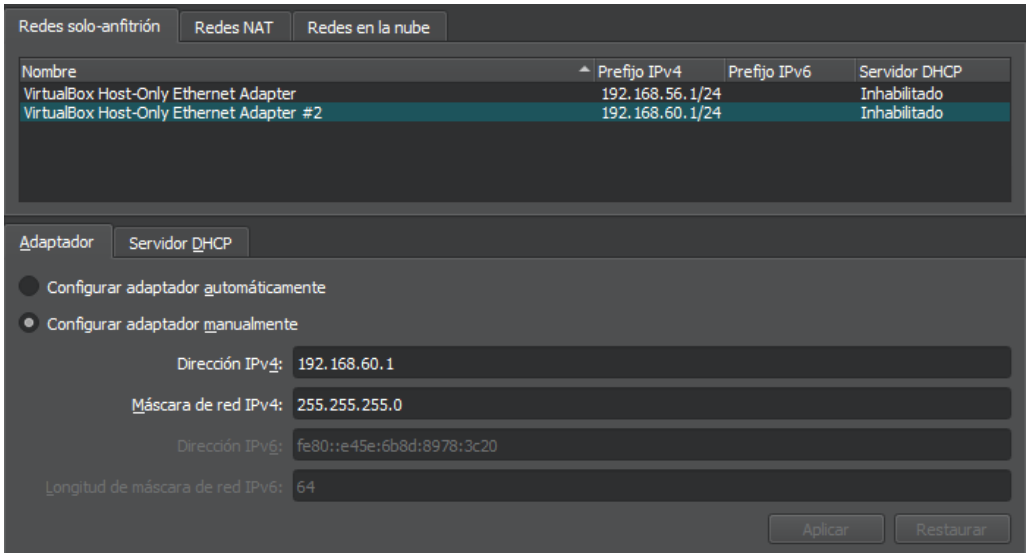

Figura 4.2: Configuración del adaptador de red *Host-Only*

Adicionalmente, para realizar una comunicación sin conflictos dentro de esta red debemos definir en la configuración del adaptador red la *IP* y máscara de red que se empleará para definir el rango de direcciones *IP* que se asignan a las máquinas virtuales y el *host*. Como se puede observar en la figura 4.2, la dirección establecida como puerta de enlace será `192.168.60.1`, además de indicar una máscara de red de 24 bits, es decir, damos a la red capacidad para asignar 253 direcciones, ya que tendríamos que descontar las siguientes direcciones *IP* reservadas:

- **Dirección de red**, `192.168.60.0`: Esta es la dirección de red, que se utiliza para identificar la red en sí. No se puede asignar a un dispositivo.
- **Dirección de *broadcast***, `192.168.60.255`: Esta dirección se utiliza para enviar mensajes a todos los dispositivos en la red simultáneamente y tampoco se puede asignar a un dispositivo.
- **Dirección del adaptador**, `192.168.60.1`: Asignada en la configuración

Si, por otra parte, nos referimos a la asignación de *IPs* a los diferentes *hosts* implicados es notable destacar que, en esta red de tipo *Host-Only*, asignaremos las *IPs* de forma **estática**. Este tipo de asignación asegura que un servidor siempre tenga la misma dirección *IP*, lo que es crucial para servicios que necesitan ser accesibles de manera constante, como es el caso de nuestros nodos del clúster. La asignación establecida para cada máquina puede ser consultada en la tabla 4.2.

Como podemos observar, existe una diferencia numérica notable entre las asignaciones de *IPs* para nodos tipo `master` y nodos tipo `worker`. Esta diferencia se ha establecido para evitar colisiones entre direcciones *IPs* que pudieran surgir si el clúster aumentara de tamaño, especialmente en situaciones de alta demanda computacional donde se requiriera la existencia de varios nodos `master`.

Por otra parte observamos que en la figura 4.1 que las interfaces red `eth1` están conectadas a otra red, en este caso una red *NAT*. La red *NAT* es la forma más simple de acceder a una red externa desde una máquina virtual. En este tipo de redes, el motor de red de *VirtualBox* actúa como *router*, mapeando el tráfico hacia y desde la máquina virtual de manera transparente. Este *router* virtual se sitúa entre cada *MV* y el *host*, proporcionando una capa adicional de seguridad, ya que traduce las direcciones *IP* de la máquina virtual,

	Nombre	IP
Masters	master01	192.168.60.11
Workers	worker01	192.168.60.21
	worker02	192.168.60.22

Tabla 4.2: Asignación de IPs en cada nodo

permitiendo conexiones salientes pero impidiendo que dispositivos externos accedan a la *VM* sin configuraciones adicionales [17].

En nuestro caso, estas interfaces reciben su dirección de red y configuración en la red privada de un servidor *DHCP* integrado en *VirtualBox*. Dado que se puede configurar más de una tarjeta de una máquina virtual para usar *NAT*, la primera tarjeta se conecta a la red privada `10.0.2.0`, la segunda tarjeta a la red `10.0.3.0`, y así sucesivamente [17]. En nuestro caso las máquinas reciben la *IP* `10.0.2.15/24`.

Una vez desarrolladas todas las redes implicadas en la comunicación del clúster podemos reflejar la configuración final en el esquema de red general, como se puede observar en la figura 4.3.

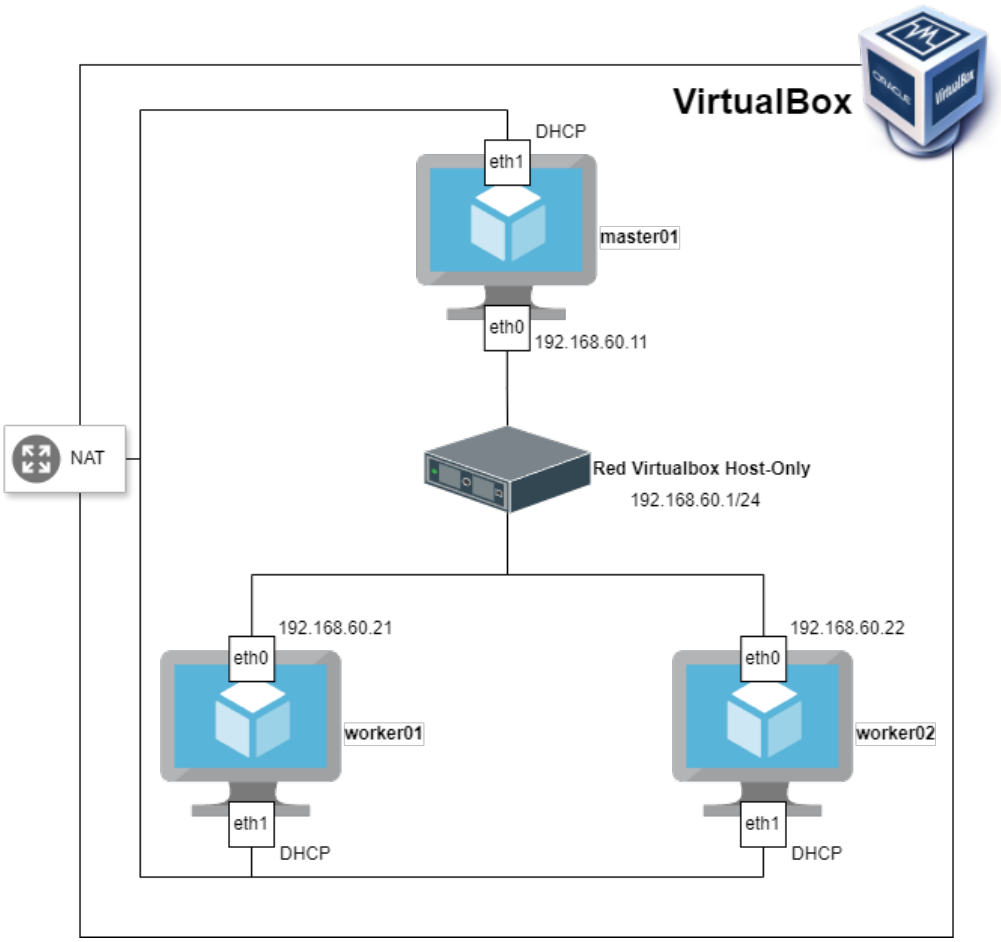

Figura 4.3: Arquitectura con asignación de IPs

4.2 Configuración de las máquinas virtuales

Para establecer una base que soporte la creación de un clúster de *Kubernetes* es necesario la configuración de *software* adicional para que los distintos nodos involucrados

en el proceso de funcionamiento del clúster tengan la capacidad de ejecutar todas las funcionalidades de la tecnología sin causar errores.

Para ello dividiremos el proceso de instalación y configuración de los distintos nodos en cuatro partes: **Instalación de software común**, **instalación de software específico para el soporte de *Kubernetes***, **instalación de herramientas específicas de *Kubernetes*** y **coordinación del clúster**.

4.2.1. Instalación común (Nodos Primarios y Secundarios)

La paquete de instalación de software será común para todos los nodos del clúster, independientemente del rol que desempeñen. Es por eso que, para ahorrar trabajo, configuraremos únicamente una máquina virtual y, una vez configuradas todas las herramientas comunes, pasaremos al proceso de duplicado y personalización del resto de nodos.

Tras configurar el *hardware* de la *MV*, es momento de instalar el sistema operativo. No hay complicaciones en este paso, excepto que debemos asignar manualmente la dirección *IP* deseada dentro del rango proporcionado por el adaptador *Host-Only*, como ha sido mencionado anteriormente.

Para ello tendremos que iniciar la máquina virtual e iniciar el proceso de instalación de `Ubuntu-server-24.04`. Avanzamos por las diferentes pantallas de configuración, seleccionando idioma e imagen base (Imagen completa de Ubuntu Server), hasta llegar a la sección de configuración de red.

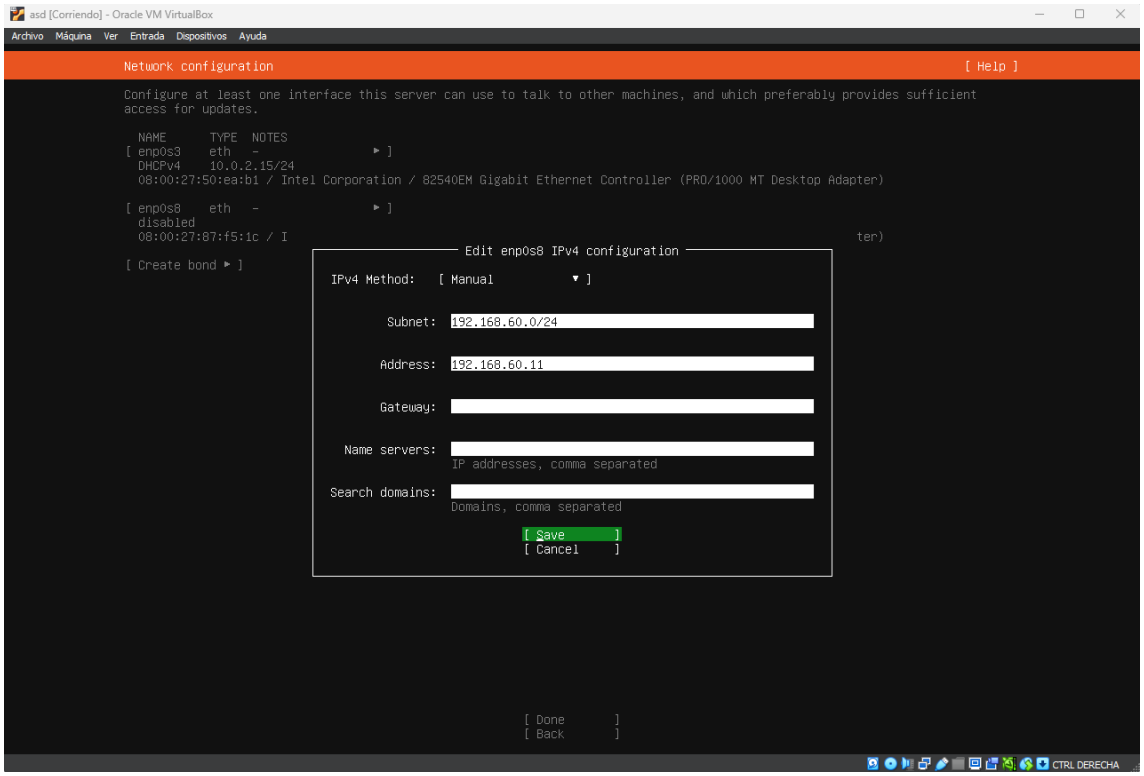

Figura 4.4: Configuración de red del nodo `master`

Como en nuestro caso configuramos inicialmente la máquina que actuará como Plano de Control, la *IP* a asignar será `192.168.60.11`, como hemos podido observar en la tabla 4.2.

Simplemente seleccionamos la interfaz red correspondiente (`enp0s8`) e introducimos, mediante asignación manual, la dirección *IP* de la subred seleccionada junto con su máscara de red y la dirección de la máquina en cuestión (figura 4.4).

Es importante remarcar que, la puerta de enlace con *IP* `192.168.60.1`, no se debe configurar para ningún nodo. Si la configuramos (la puerta del adaptador de *Host-Only*), el nodo intentará usar esta puerta para todo el tráfico saliente, incluyendo el tráfico de Internet. Esto conllevará a la pérdida de conexión, ya que el adaptador de *Host-Only* no tiene acceso a Internet y tendremos que cambiar la configuración para establecer la puerta de la red *NAT* como predeterminada.

Una vez preparada la infraestructura necesaria para desplegar el clúster de *Kubernetes* vamos a instalar y configurar las herramientas necesarias para su funcionamiento. Una vez iniciada la *MV* es necesario deshabilitar la memoria `swap`.

Kubernetes depende de una gestión precisa de los recursos, especialmente de la memoria. El uso de `swap` puede causar problemas de rendimiento y estabilidad, ya que *Kubernetes* no está diseñado para manejar de manera eficiente la memoria intercambiada. Si se recurre a la memoria `swap` se podría ralentizar el rendimiento de los contenedores y desincronizar la asignación de recursos, lo que puede provocar problemas en la operación del clúster [18, 19].

Para ello tendremos que acceder al archivo `/etc/fstab` y comentar la línea correspondiente, como podemos observar en la figura 4.5.

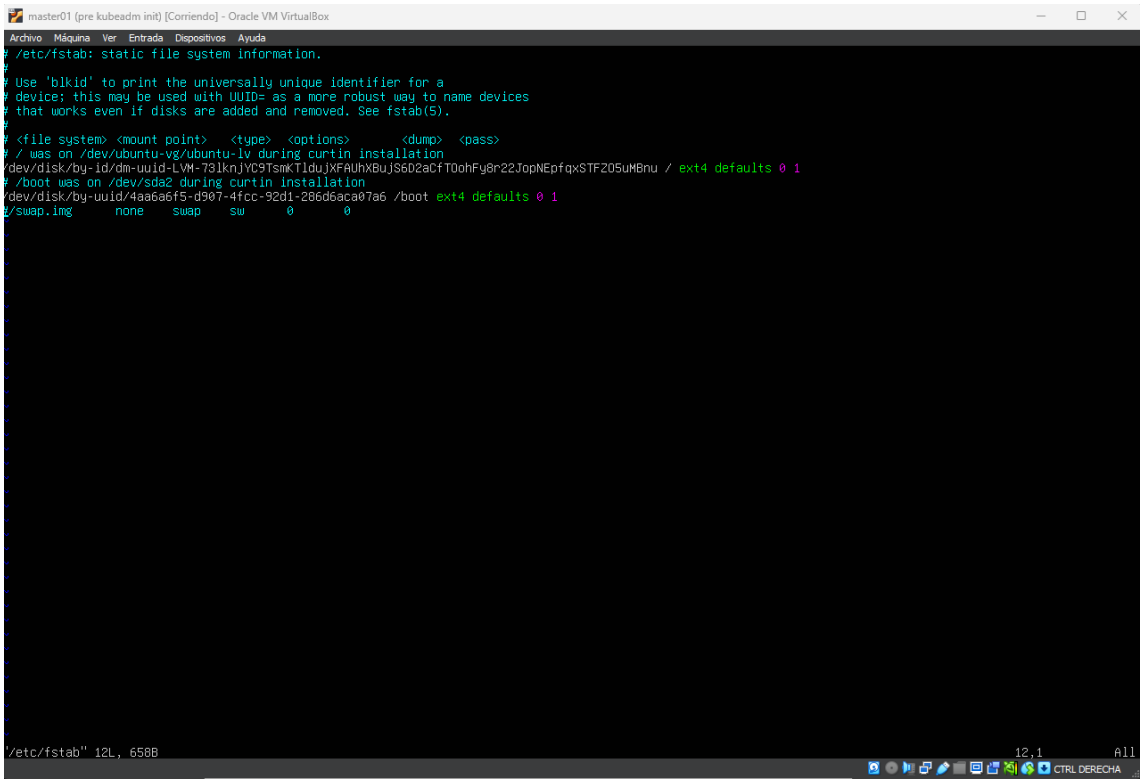

Figura 4.5: Inhabilitación de `swap`

El siguiente paso para asegurar un correcto funcionamiento del clúster es configurar el kernel de Linux modificando varios componentes que se expondrán a continuación. La primera acción a realizar sería habilitar los módulos `overlay` y `br_netfilter` mediante los siguientes comandos:

```
sudo modprobe overlay
```

```
sudo modprobe br_netfilter
```

El módulo `overlay` es esencial para el manejo de sistemas de archivos superpuestos utilizados por contenedores como los de *Docker*. Por otra parte, `br_netfilter`, necesario para el filtrado y el manejo de paquetes de red en redes *bridge*, que son comunes en configuraciones de *Kubernetes*.

El siguiente paso es configurar el *IP forwarding* y filtrado de paquetes, mediante la edición del archivo `/etc/sysctl.d/kubernetes.conf` y añadiendo los siguientes parámetros:

```
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
```

Estas configuraciones permiten que el tráfico de red en interfaces *bridge* sea filtrado por las reglas de `iptables`, lo cual es fundamental para que *Kubernetes* pueda aplicar políticas de red y seguridad, además de habilitar el reenvío de paquetes *IP* en IPv4, lo que es necesario para que los nodos del clúster puedan enrutar el tráfico entre diferentes redes y nodos. Sin esta configuración, los contenedores en diferentes nodos no podrían comunicarse correctamente.

Por otro lado `sudo sysctl --system` recarga todas las configuraciones de `sysctl`, el mecanismo de *Unix* que lee y modifica atributos de *kernel* del sistema, y aplica inmediatamente los cambios realizados.

Los módulos importados no se mantienen de manera persistente. Cargar los módulos del *kernel* en tiempo de ejecución significa que estarán activos únicamente hasta que el sistema se reinicie. Al reiniciar, los módulos cargados manualmente no se vuelven a cargar, por lo que se "pierden" a menos que se configuren. Esto se debe a que el comando `modprobe` carga módulos en el kernel de *Linux*, pero no hace cambios permanentes en el sistema de archivos para que sobrevivan a un reinicio.

Es por eso que se convierte en esencial el modificar los registros del sistema y añadir en la configuración de *Kubernetes* las entradas necesarias para mantener estos módulos. Para eso tendremos que modificar el archivo `/etc/modules-load.d/k8s.conf` y añadir:

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

Una vez realizada la modificación, volvemos a recargar las configuraciones de *sysctl*.

```
sudo sysctl --system
```

4.2.2. Instalación del *CRI: Container Runtime Interface* (Nodos Primarios y Secundarios)

Para ejecutar contenedores en Pods, *Kubernetes* utiliza una Interfaz de *Runtime* de Contenedores (*CRI*, por sus siglas en inglés). *CRI* es una interfaz que permite que `kubelet`

emplee una amplia variedad de *Container Runtimes* [20]. Es decir, el *Container Runtime Interface* es el protocolo principal para la comunicación entre el *kubelet* y el *Container Runtime*.

En nuestro caso escogeremos el *CRI Containerd*. *Containerd* es el *runtime* predeterminado utilizado por el motor de *Docker*. A menudo se lo considera un estándar de la industria debido a su amplia adopción [21]. El uso de *Containerd* es inevitable ya que, al contrario que versiones anteriores de *Kubernetes*, Docker pasó a un estado de desuso como *runtime* principal de contenedores en la versión *v1.20* (2020) [22].

Docker fue el primer *runtime* de contenedores utilizado por *Kubernetes*. El soporte para Docker estaba integrado mediante una necesidad conocida en el proyecto como *dockershim*. El incremento y adopción en el uso de contenedores culminó en la implementación de la interfaz de *runtime* de contenedores (*CRI*), y, como resultado, *dockershim* pasó a convertirse en una anomalía dentro del proyecto *Kubernetes*, resultando en código frágil [23].

Containerd, por otro lado, es un *runtime* de contenedores más liviano y enfocado únicamente en la ejecución y gestión de contenedores. Al hacer el cambio a *containerd*, *Kubernetes* pudo eliminar la necesidad de *dockershim*, reducir la complejidad del sistema y mejorar la interoperabilidad con otros *runtimes* de contenedores compatibles con *CRI*.

Para instalar Containerd dentro de tu sistema, necesitamos seguir los siguientes pasos reflejados en [24]:

Configurar el repositorio `apt` de Docker

1. Agregamos la clave *GPG* oficial de *Docker*:

   ```
   sudo apt-get update
   sudo apt-get install apt-transport-https ca-certificates curl gpg
   sudo install -m 0755 -d /etc/apt/keyrings
   sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
        -o /etc/apt/keyrings/docker.asc
   sudo chmod a+r /etc/apt/keyrings/docker.asc
   ```

   La clave *GPG (GNU Privacy Guard)* [25] de *Docker* es una clave criptográfica utilizada para firmar digitalmente los paquetes de software proporcionados por *Docker*. Esta firma asegura que los paquetes no hayan sido alterados desde que fueron firmados por el fabricante original y que provienen de una fuente confiable.

2. Agregar el repositorio a las fuentes de `apt`:

   ```
   echo \
     "deb [arch=$(dpkg --print-architecture) \
     signed-by=/etc/apt/keyrings/docker.asc] \
     https://download.docker.com/linux/ubuntu \
     $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
     sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
   sudo apt-get update
   ```

   Al agregar un repositorio, estamos indicando a la herramienta *apt* que busque paquetes de software en este. Esto es necesario cuando deseamos instalar software que no está disponible en los repositorios oficiales predeterminados del sistema. En el caso de *Docker*, agregar su repositorio permite instalar y actualizar sus componentes directamente desde los repositorios oficiales de *Docker*.

Instalar `containerd`

Instalamos el paquete de containerd desde el repositorio oficial mediante:

```
sudo apt-get install containerd.io
```

Tras esto necesitamos configurarlo mediante la ejecución de los siguientes comandos:

```
sudo mkdir -p /etc/containerd
sudo containerd config default | sudo tee /etc/containerd/config.toml
```

Guardamos la configuración para mantenerla de forma persistente en el kernel, concretamente en el archivo `/etc/containerd/config.toml`.

Tras esto, lo reiniciamos.

```
sudo systemctl restart containerd
sudo systemctl enable containerd
```

Una vez reiniciado ya tenemos *containerd* listo para su uso.

4.2.3. Instalación de `Kubeadm`, `Kubelet` y `Kubectl` (Nodos Primarios y Secundarios)

Existen varias maneras de configurar un clúster de *Kubernetes*, pero en esta ocasión utilizaremos la herramienta `kubeadm`. `Kubeadm` nos permite crear un clúster de *Kubernetes* que cumple con las pruebas de conformidad de la tecnología [26].

Cada nodo del clúster debe contar con los siguientes componentes[1]:

- `kubelet`: Es el agente principal del nodo. Puede registrar el nodo en el servidor de la *API* utilizando diferentes métodos: el nombre del *host*, una lógica específica para un proveedor de nube, etc.
- `kubectl`: Es la herramienta de línea de comandos de *Kubernetes* que te permite ejecutar comandos. Con `kubectl` puedes desplegar aplicaciones, inspeccionar y gestionar los recursos del clúster, ver los registros generados por los diferentes recursos, etc.
- `kubeadm`: La herramienta de creación del clúster.

Paso 1: Descargar la clave pública de firma

Primero, es necesario descargar la clave pública de firma para los repositorios de paquetes de *Kubernetes*. Para ello es necesario ejecutar el siguiente comando `curl` para descargar la clave y, más importante, convertirla al formato adecuado para ser procesado por `apt`:

```
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.30/deb/Release.key \\
    | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
```

[1]Para más información sobre los distintos procesos involucrados en los distintos nodos del clúster consultar Anexo A

Paso 2: Agregar el repositorio `apt` de *Kubernetes*

Después de agregar la clave, es necesario agregar el repositorio de Kubernetes a la lista de fuentes de `apt`.

```
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] \\
    https://pkgs.k8s.io/core:/stable:/v1.30/deb/ /' | \\
    sudo tee /etc/apt/sources.list.d/kubernetes.list
```

Paso 3: Actualizar el índice de paquetes e instalar *Kubernetes*

Una vez agregado el repositorio a `apt` es necesario actualizar el índice de paquetes de la herramienta para incluir las colecciones recientemente incorporadas al nuevo repositorio. Después de esto, se procede a instalar `kubelet`, `kubeadm` y `kubectl`.

```
sudo apt-get update
sudo apt-get install kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

Como se ha podido observar, hemos utilizado un comando adicional, `sudo apt-mark hold`. Esta orden se emplea para fijar las versiones de estos paquetes, evitando que se actualicen automáticamente y poder mantener la consistencia en el clúster.

Paso 4: Habilitar el servicio `kubelet`

Aunque este paso no es necesario, es recomendable habilitar el servicio `kubelet` antes de ejecutar `kubeadm`. Habilitar este servicio asegura que `kubelet` se inicie automáticamente con el sistema y comience a gestionar los Pods en los nodos una vez que el clúster esté configurado.

Esto lo realizamos mediante el siguiente comando:

```
sudo systemctl enable --now kubelet
```

4.2.4. Creación de las máquinas `worker`

En este instante disponemos de una única máquina virtual con todo el software común instalado. Si recordamos el esquema planteado en la figura 4.3, el clúster dispone de tres máquinas: un nodo `master` y dos nodos `worker`. Para ello es necesario duplicar las máquinas virtuales dentro de *VirtualBox*, como observamos en la figura 4.6.

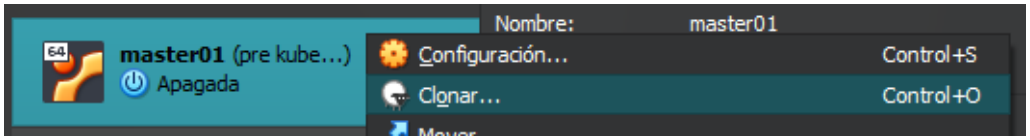

Figura 4.6: Opción de clonación de máquina virtual

Una vez en el menú de clonación tendremos que cambiar el nombre a `workerN` (entendiendo N como el número de worker deseado). En cuanto a la política de dirección *MAC* vamos a escoger "generar nuevas direcciones *MAC* para todos los adaptadores red"para evitar posibles conflictos en la comunicación red (figura 4.7).

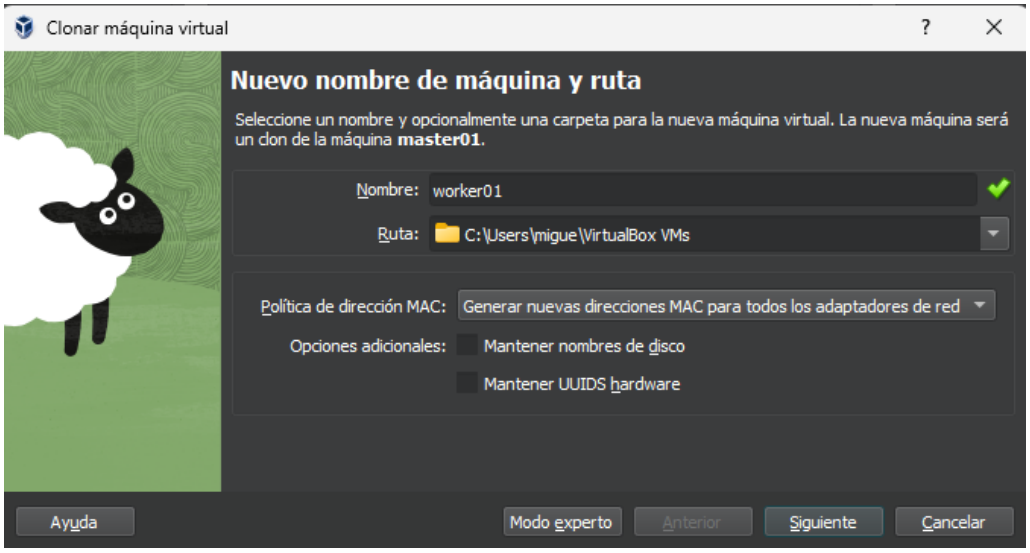

Figura 4.7: Clonación de máquina virtual

Una vez obtenidas las dos máquinas adicionales (`worker01` y `worker02`) necesitamos realizar una configuración adicional para obtener un clúster totalmente funcional. Si volvermos a recordar el esquema de red de la figura 4.3, observamos que los nodos worker conectados a la red *Host-Only* tienen una asignación manual de *IP* distinta a la del nodo `master`.

Sin embargo, en el proceso de clonación de *VirtualBox*, todas las configuraciones internas se copian tal y como se encuentran en el instante de inicio del proceso. Es por ello que es necesario acceder a los distintos nodos para editar una serie de archivos que, sin su modificación, harían imposible la comunicación.

El primer elemento a modificar será el nombre de *host*. Aunque sí es cierto que, si no modificamos este aspecto, la comunicación a nivel de red podría seguir funcionando (bajo ciertas condiciones[2]) se sugiere modificar esta propiedad para evitar conflictos de resolución de nombres o errores en servicios, por ejemplo.

Para modificar este aspecto será necesario editar dos archivos: `/etc/hostname` y `/etc/hosts`.

```
ubuntu@worker01:~$ sudo vim /etc/hosts
[sudo] password for ubuntu:
ubuntu@worker01:~$ cat /etc/hosts
127.0.0.1 localhost
#127.0.1.1 master01
127.0.1.1 worker01

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ubuntu@worker01:~$
```

Figura 4.8: Modificación de `/etc/hosts`

[2]Si las máquinas se comunican directamente a través de direcciones *IP*, el conflicto de `hostname` no afecta la comunicación a nivel de red. Estas seguirán siendo capaces de enviar y recibir datagramas, siempre y cuando usen las direcciones *IP* correctas.

```
ubuntu@worker01:~$ sudo vim /etc/hostname
ubuntu@worker01:~$ cat /etc/hostname
worker01
ubuntu@worker01:~$
```

Figura 4.9: Modificación de `/etc/hostname`

Una vez terminamos de modificar el nombre de *hosts* el siguiente elemento a modificar es la dirección *IP* de la interfaz de la red *Host-Only* (que se corresponde con la interfaz de red `enp0s8`) y, para ello, emplearemos la herramienta *Netplan*.

Netplan es una herramienta de configuración de red que se utiliza en sistemas operativos *Linux*, particularmente en *Ubuntu* (se incorporó por primera vez en la versión 17.10). Proporciona una forma simple y coherente de definir configuraciones de red utilizando archivos *YAML*, lo que facilita la configuración de interfaces de red, como la asignación de direcciones *IP*, configuraciones de *DNS*, *gateways*, y otras opciones de red [27].

Para cambiar la IP de la MV habrá que modificar `/etc/netplan/50-cloud-init.yaml` e incluir la nueva dirección en la interfaz deseada, como podemos observar en la figura 4.10.

```
# This file is generated from information provided by the datasource.  Changes
# to it will not persist across an instance reboot.  To disable cloud-init's
# network configuration capabilities, write a file
# /etc/cloud/cloud.cfg.d/99-disable-network-config.cfg with the following:
# network: {config: disabled}
network:
    ethernets:
        enp0s3:
            dhcp4: true
        enp0s8:
            addresses:
            - 192.168.60.22/24
            nameservers:
                addresses: []
                search: []
    version: 2
~
~
```

Figura 4.10: Modificación de `/etc/netplan/50-cloud-init.yaml` para el `worker02`

Tras esta modificación debemos emplear el comando `sudo netplan apply` para que los cambios surtan efecto. Tras la aplicación de la nueva configuración podemos observar mediante la orden `ip a` que la dirección de la interfaz de red *Host-Only* ha cambiado (figura 4.11).

```
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:34:50:87 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 metric 100 brd 10.0.2.255 scope global dynamic enp0s3
       valid_lft 86396sec preferred_lft 86396sec
    inet6 fe80::a00:27ff:fe34:5087/64 scope link
       valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:82:87:a3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.60.22/24 brd 192.168.60.255 scope global enp0s8
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe82:87a3/64 scope link
       valid_lft forever preferred_lft forever
```

Figura 4.11: Direcciones interfaces red `worker02`

4.2.5. Coordinación del clúster

Inicialización del nodo maestro

Una vez que todos los componentes requeridos han sido instalados y configurados, es necesario inicializar el nodo maestro. Dado que, en un principio, se agregó una red privada secundaria para facilitar la comunicación interna entre los nodos, debemos asegurarnos de enlazar la `api-advertisement-address` exclusivamente a dicha red. Este paso se puede llevar a cabo utilizando la herramienta `kubeadm` mediante el siguiente comando:

```
sudo kubeadm init --apiserver-advertise-address 192.168.60.11 \
    --control-plane-endpoint 192.168.60.11 --pod-network-cidr=10.244.0.0/16
```

Esta orden especifica que el servidor de la *API* de *Kubernetes* debe anunciarse en la dirección *IP* `192.168.60.11` y establece el mismo *endpoint* para el plano de control. Adicionalmente configura el rango de direcciones *IP* como `10.244.0.0/16`, lo que determina en qué rango de *IPs* se asignarán las direcciones de los Pods, lo que es necesario para implementar una red dentro del clúster (como observaremos más adelante).

Conexión de nodos de trabajo al clúster

Cuando el proceso de inicialización del nodo maestro haya terminado, `kubeadm` proporcionará los comandos necesarios para ejecutar en los nodos de trabajo y otros posibles nodos maestros para unirse al clúster.

Adicionalmente, para conectarse a nuestro clúster utilizando `kubectl`, necesitamos aplicar los siguientes cambios en la máquina donde se ejecuta el Plano de Control.

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
sudo chmod -R 755 /etc/kubernetes/admin.conf
```

Este conjunto de comandos crea un directorio oculto `.kube` en el directorio *home* del usuario para almacenar el archivo de configuración. Luego, copia el archivo `admin.conf`, generado durante la inicialización del nodo maestro a este directorio, y finalmente, ajusta los permisos del archivo para que el usuario actual tenga los derechos adecuados para acceder y gestionar el clúster con `kubectl`.

Una vez inicializado kubeadm necesitamos que los nodos de trabajo se conecten al Plano de Control. Para ello kubeadm nos proporciona una orden personalizada con un token y una identificación.

```
kubeadm join 192.168.0.11:6443 --token 1rc7cy.ln07dedededej0ghjr \
--discovery-token-ca-cert-hash sha256:a6sdsadasdswdeeedd33c7677 (...)
```

Esta orden es necesario ejecutarla en todos los nodos secundarios que queramos incluir en el clúster.

Instalación de la red de Kubernetes

Si accedemos al listado de nodos o *Pods* podemos observar que estos se encuentran en un estado de "`NotReady`"(en caso de los nodos) o "`Pending`" (en caso de algunos *Pods*).

Esto se debe a que es necesaria la instalación de un *plugin CNI*, que cree una red de *Kubernetes* para el clúster. Esta es diferente de la red de infraestructura, ya que es responsable de crear una red interna para que todos los componentes internos puedan comunicarse entre sí.

En nuestro caso emplearemos *Flannel*, una de las redes más simples. Existen otros complementos de red con más características y capacidades, pero *Flannel* no añade complejidad innecesaria. Para instalar el complemento de red *Flannel* simplemente tendremos que crear los objetos de *Kubernetes* necesarios que podemos encontrar en el repositorio de *GitHub* oficial:

```
kubectl apply -f \
    https://raw.githubusercontent.com/flannel-io/flannel/master
        /Documentation/kube-flannel.yml
```

Tras la descarga y un tiempo variable de espera podemos observar la totalidad de Pods necesarios operativos, como observamos a continuación en la figura 4.12.

```
ubuntu@master01:~$ kubectl get pods -A
NAMESPACE      NAME                                 READY   STATUS    RESTARTS        AGE
kube-flannel   kube-flannel-ds-f6rdl                1/1     Running   0               4d4h
kube-flannel   kube-flannel-ds-ltm6r                1/1     Running   1 (6m18s ago)   4d5h
kube-flannel   kube-flannel-ds-rqv5k                1/1     Running   1 (4m9s ago)    4d4h
kube-system    coredns-7db6d8ff4d-m76v1             1/1     Running   1 (6m18s ago)   4d5h
kube-system    coredns-7db6d8ff4d-w7jxv             1/1     Running   1 (6m18s ago)   4d5h
kube-system    etcd-master01                        1/1     Running   4 (6m18s ago)   4d5h
kube-system    kube-apiserver-master01              1/1     Running   4 (6m18s ago)   4d5h
kube-system    kube-controller-manager-master01     1/1     Running   4 (6m18s ago)   4d5h
kube-system    kube-proxy-jx9dg                     1/1     Running   1 (6m18s ago)   4d5h
kube-system    kube-proxy-qmvg5                     1/1     Running   0               4d4h
kube-system    kube-proxy-zjz64                     1/1     Running   1 (4m9s ago)    4d4h
kube-system    kube-scheduler-master01              1/1     Running   5 (6m18s ago)   4d5h
ubuntu@master01:~$
```

Figura 4.12: Respuesta de la orden `kubectl get pods -A`

Tras incorporar el *CNI* y todos los nodos de trabajo al clúster, el resultado de `kubectl get node` debería mostrar los nuevos nodos unidos y preparados (figura 4.13).

```
ubuntu@master01:~$ kubectl get nodes
NAME       STATUS   ROLES           AGE    VERSION
master01   Ready    control-plane   4d5h   v1.30.4
worker01   Ready    <none>          4d4h   v1.30.4
worker02   Ready    <none>          4d4h   v1.30.4
ubuntu@master01:~$
```

Figura 4.13: Respuesta de la orden `kubectl get nodos`

CAPÍTULO 5
Implementación de la solución

En esta sección, se detallará el proceso de implementación de la solución propuesta, enfocándose en la transición del entorno preliminar, basado en *Docker Compose*, a una infraestructura más robusta y escalable mediante el empleo de *Kubernetes*. Este cambio no solo implica la migración de los servicios existentes, sino la adaptación y optimización adicional de la arquitectura para aprovechar las capacidades avanzadas de orquestación y resiliencia que *Kubernetes* ofrece.

La implementación de la solución se desglosa en varias etapas clave. Primero, se abordará la configuración de contenedores y *Pods*, explicando cómo los componentes individuales de la aplicación se despliegan y gestionan dentro de un clúster.

A continuación, se explorará la orquestación de los distintos elementos, donde se destacarán las estructuras de soporte y los mecanismos de resiliencia que se implementan para asegurar que el sistema no solo sea capaz de manejar cargas variables y fallos, sino que también pueda escalar según sea necesario. Esta parte de la implementación se centrará en la utilización de varios elementos, como *Deployments*, *Services*, y otros recursos esenciales de *Kubernetes* para construir una solución altamente disponible y resiliente.

5.1 Configuración de contenedores y Pods

La configuración general de la aplicación se compone de cuatro componentes principales:

- OAuth
- Frontend
- NATS
- Middleware

Es por ello que, antes de desarrollar la configuración específica de cada servicio vamos a desglosar, brevemente, su funcionamiento:

5.1.1. OAuth

OAuth (Open Authorization) es un estándar abierto que permite a los usuarios otorgar a aplicaciones de terceros acceso limitado a sus recursos en un servicio, sin necesidad de compartir sus credenciales (como nombre de usuario y contraseña). En lugar de que la

aplicación de terceros acceda directamente a la cuenta del usuario, *OAuth* actúa como un intermediario que proporciona a la aplicación un *token* de acceso temporal, lo que permite que la aplicación interactúe con los datos del usuario de manera segura y controlada.

En el sistema diseñado, el único punto de entrada es a través de este sistema. Para lograr la autenticación e identificación de los usuarios mediante *Google* en la aplicación, se ha implementado el uso de OAuth2-proxy.

Con estos datos, el cliente OAuth2-proxy puede utilizar Google para la autenticación, sin embargo, es necesario especificar la *URI* de redireccionamiento en el proyecto de *Google Cloud* para que *Google* pueda redirigir al usuario a nuestra aplicación. En este caso, la *URI* de redireccionamiento debe ser `http://localhost:80/oauth2/callback`.

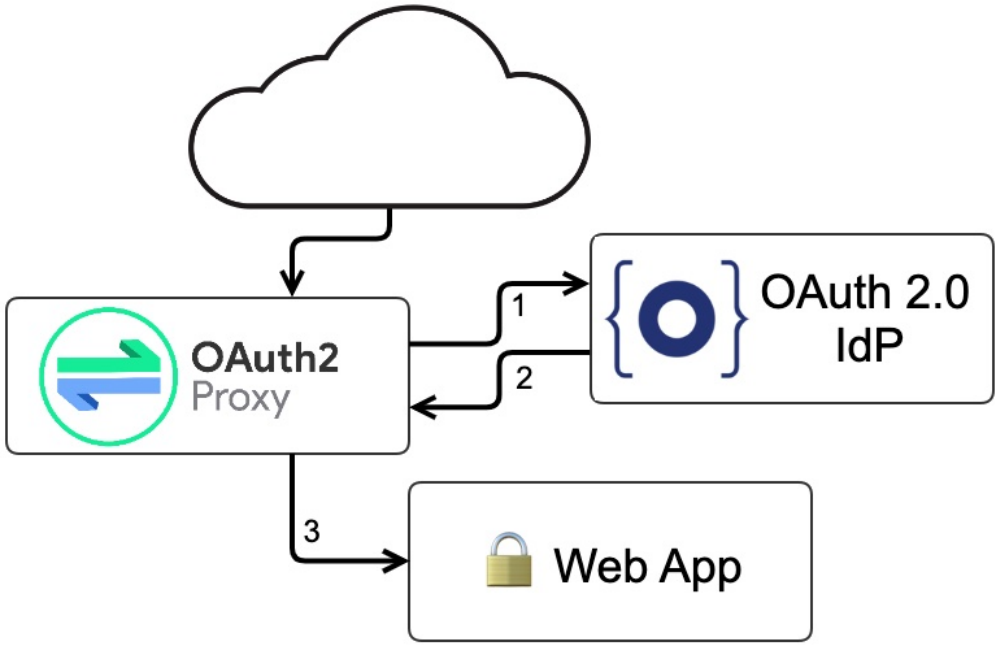

Figura 5.1: Flujo del OAuth2-proxy

Este paso es crucial para completar el flujo de autenticación, ya que `OAuth2-proxy` solo cuenta con un código de autorización en este punto, el cual se utiliza para obtener información del usuario, como su correo electrónico y un *ID* único. Una vez que se ha obtenido esta información, el *proxy* redirige al usuario al Pod *frontend* utilizando la dirección especificada en la configuración guardada en el archivo `oauth2-proxy.cfg`.

Migración a Kubernetes

Para mejorar la escalabilidad y la gestión de despliegues en la nueva infraestructura dispuesta necesitamos encapsular el contenedor *Docker* original donde se empaquetaba el contenedor *OAuth* en un Pod, ya que, originalmente, el componente se gestionaba de manera manual, lo que implicaba un esfuerzo significativo en su mantenimiento y escalabilidad.

Para ello simplemente necesitamos especificar el contenedor dentro de la estructura de un Pod:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: oauth
```

```
spec:
  containers:
  - name: oauth
    image: miguellomot/cc-oauth:latest
    ports:
        - containerPort: 8080
```

Listing 5.1: Manifiesto YAML para el despliegue del Pod OAuth

Como se puede comprobar, la imagen correspondiente al contenedor se obtiene, al no especificar ningún *hostname* de un registro alternativo, de *Docker Hub*, ya que Kubernetes asume que las imágenes se alojan ahí de no indicarse lo contrario en la declaración del recurso [28]. Por defecto, y en todas las componentes que componen el servicio, `kubelet` descarga las imágenes remotas de forma secuencial. Esto significa que envía una solicitud de descarga de imagen al servicio de de una en una y, por consecuente, demás solicitudes de descarga deben esperar hasta que la actual finalice [28].

5.1.2. Frontend

Para el desarrollo del *frontend*, se ha utilizado el *framework* Next.js, debido a su capacidad para crear tanto una interfaz que se ejecuta en el lado del cliente como un servidor que se comunica con otros servicios.

Next.js facilita la creación de rutas de manera eficiente mediante su funcionalidad *App Router*. Este enrutador opera dentro de la carpeta `app` del proyecto, donde todos los archivos denominados `page` representan rutas accesibles para el usuario. Como se ilustra en la figura 5.2, el archivo `page` ubicado en la carpeta `app` permite que el usuario acceda a la ruta `http://localhost/`. De manera similar, si se coloca un archivo `page` dentro de las carpetas `app/job/[id]`, se genera una ruta en `http://localhost/job/[id]`, donde `id` es un parámetro dinámico que se utiliza posteriormente para obtener la información correspondiente al identificador especificado.

Figura 5.2: Captura de pantalla de las rutas en el *frontend*.

Es importante destacar que el cliente no tiene una conexión directa con otros componentes de la arquitectura, excepto con OAuth y el *frontend*. *Next.js* permite ejecutar funciones desde el servidor utilizando la directiva `use server`. De este modo, cuando un cliente crea un trabajo, los datos del formulario se envían al servidor web, que luego se encarga de publicarlos en la cola correspondiente de *NATS*.

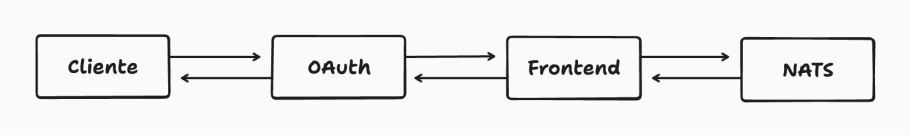

Figura 5.3: Esquema de la comunicación entre los diferentes servicios.

En la vista principal del *frontend*, el usuario puede visualizar un listado de todos los trabajos. Esta información se obtiene mediante una petición automática que se realiza cada 5 segundos, en la cual el servidor *frontend* se conecta a la *Key-Value Store* de *NATS*, busca en el *bucket* correspondiente al `userId` (identificador del usuario conectado) todas las claves, y consulta su contenido.

Al hacer clic en uno de estos trabajos, el usuario puede acceder a una vista detallada con información adicional, como la salida por consola o el *output* generado, accediendo al *Object Store*.

Desde esta misma vista principal, es posible crear nuevos trabajos utilizando el método descrito anteriormente.

Migración a Kubernetes

Al igual que el componente anterior, para realizar un despliegue básico del componente simplemente tendremos que encapsular el contenedor en un Pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: frontend
    image: miguellomot/cc-fronted:latest
```

Listing 5.2: Manifiesto YAML para el despliegue del Pod frontend

Como podemos ver en la sección de código anterior, la encapsulación del contenedor en un Pod es elemental. En la siguiente sección ampliaremos más estos despliegues para ampliar su capacidad de resiliencia, escalado y comunicación.

5.1.3. NATS

El servicio *NATS* es una solución de mensajería distribuida, ligera y de alto rendimiento, diseñada para facilitar la comunicación entre diferentes componentes de un sistema. Se utiliza ampliamente para conectar servicios de microservicios, *IoT*, y sistemas distribuidos, permitiendo el intercambio de mensajes en tiempo real. Ofrece diversas funcionalidades clave, como colas de mensajes, almacenamiento de clave/valor (*Key-Value Store*) y almacenamiento de objetos (*Object Store*), las cuales son fundamentales para el correcto funcionamiento de sistemas distribuidos y escalables.

Cuando un usuario crea un trabajo, el servidor *frontend* publica un mensaje en la cola correspondiente, siguiendo el patrón `cola.[lenguaje]`. Este mensaje es recibido por el servicio *middleware*, que se suscribe a cada una de las colas según los lenguajes disponibles (como *Go*, *JavaScript* y el lenguaje *C*, entre otros).

El almacenamiento de clave/valor proporcionado por *NATS* se utiliza para registrar y mantener los trabajos de los usuarios, permitiendo acceder al estado de estos en cual-

quier momento. El esquema de este almacenamiento es el siguiente: se crea un *bucket* identificado por el *ID* del usuario, y dentro de este se almacenan los trabajos del usuario. La clave es el identificador único del trabajo, y el valor es el objeto correspondiente convertido a una cadena de texto.

Por otro lado, el *Object Store* de *NATS* se utiliza para almacenar los resultados generados por los trabajos de los usuarios. El formato es similar al del *Key-Value Store*: el nombre del *bucket* corresponde al identificador del usuario, mientras que el nombre del objeto sigue el patrón `[jobId]-output.zip`, donde `jobId` es el identificador del trabajo.

Migración a Kubernetes

La migración de NATs a su homólogo en *Kubernetes* se compone:

```
apiVersion: v1
kind: Pod
metadata:
  name: nats-1
spec:
  containers:
        - name: nats-1
          image: nats:latest
          command: ["nats-server", "-js", "-DV", "-http_port", "8222"]
          ports:
            - containerPort: 4222
            - containerPort: 8222
          env:
            - name: NATS_JETSTREAM
              value: "1"
```

Listing 5.3: Manifiesto YAML para el despliegue del Pod NATS

Como se puede observar existen secciones diferentes al resto de componentes. Primeramente se puede observar que la imagen se obtiene de un repositorio de imágenes distinto. Se trata de la imagen oficial del proveedor de *NATS*. Como se obtiene simplemente la imagen oficial, no se realiza ningún tipo de configuración en el contenedor que se lanza. Es por ello que tenemos que emplear la sección `command` para iniciar el servicio de *NATS* en el puerto indicado. Adicionalmente establecemos puertos alternativos para servicios complementarios y variables de entorno para la correcta configuración del servicio.

5.1.4. Middleware

El *middleware* es un servicio desarrollado en *Go* que gestiona los eventos provenientes de la cola para lanzar las imágenes *Docker* correspondientes a los trabajos a realizar. Al iniciar el servicio, este se conecta a la cola de NATS y se suscribe a los eventos asociados con `cola.go`, `cola.javascript` y `cola.c`. Dichos eventos contienen un objeto de tipo *Trabajo*, previamente definido, con los parámetros seleccionados por el usuario.

Este componente genera un identificador único y almacena el objeto en el *Key-Value Store* de *NATS*, específicamente en el *bucket* cuyo nombre coincide con el identificador del usuario y utilizando el identificador del trabajo como clave. Una vez que el trabajo ha sido registrado en el *Key-Value Store* con el estado *Pendiente*, se procede a lanzar el ejecutor correspondiente, dependiendo del lenguaje requerido.

Migración a Kubernetes

La migración a *Kubernetes* de este componente resulta la más compleja. Esto se debe a que este recurso, como se ha expuesto en la sección 3.1.2, el *middleware* emplea, originalmente, el mecanismo de ejecución *Docker-in-Docker* (*DinD*).

DinD permite la ejecución de un contenedor *Docker* dentro de otro contenedor, lo que implica que el contenedor externo tiene acceso al *daemon* de *Docker* en el sistema anfitrión y puede llevar a cabo operaciones como la creación y ejecución de contenedores adicionales. Esto se logra mediante el uso del *socket* de Unix asociado a *Docker* (`/var/run/docker.sock`), que actúa como el principal punto de comunicación con el *daemon* en el *host*. Para habilitar esta comunicación, se enlaza dicho *socket* con el servicio de *middleware* dentro de la plantilla de infraestructura a través de un volumen de *Docker*.

Este diseño permite optimizar, en la arquitectura original, la carga sobre el *middleware*, ya que su única responsabilidad es procesar los mensajes de la cola de entrada y lanzar el contenedor correspondiente.

Sin embargo, al migrar esta lógica a *Kubernetes* nos encontramos varios problemas. En primer lugar la inserción directa de contenedores en un clúster es imposible en este contexto, ya que la unidad básica de ejecución es el *Pod*, y no el contenedor.

Para poder migrar esta lógica de negocio tendremos que generar nuevos *Pods* ejecutores dentro del clúster de *Kubernetes*. Para ello, necesitaremos incorporar objetos adicionales al *Deployment* del *middleware*. Principalmente emplearemos las conocidas como *ServiceAccount*. Una *Service Account* es una identidad asociada a un *Pod* (o un conjunto de los mismos) que les permite interactuar con el clúster de forma segura, utilizando permisos específicos. Son útiles para gestionar de manera granular los permisos que cada aplicación o servicio en ejecución dentro del clúster puede tener. A través del *Role-Based Access Control (RBAC)*, se pueden definir roles y permisos específicos para una cuenta de servicio, limitando su acceso solo a los recursos necesarios.

Es por esto que, como observamos a continuación, empleamos *Service Accounts*, *Role* y *RoleBinding* para asociar el permiso a la *Service Account*:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: middleware-serviceaccount
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-creator-role
rules:
- apiGroups: [""]  # "" --> Core API group
  resources: ["pods"]
  verbs: ["create"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-creator-rolebinding
  namespace: default
subjects:
- kind: ServiceAccount
  name: middleware-serviceaccount
  namespace: default
roleRef:
```

```
27   kind: Role
28   name: pod-creator-role
29   apiGroup: rbac.authorization.k8s.io
```

Listing 5.4: Manifiesto YAML para creación de la infaestructura adicional

Como podemos observar dentro del conjunto de permisos creados, en el apartado de reglas, facilitamos la generación de nuevos Pods dentro del clúster mediante el verbo `create`. En el *RoleBinding* vinculamos la *ServiceAccount* con el rol generado.

5.2 Orquestación con Kubernetes: Estructuras de soporte y resiliencia

En la sección anterior hemos observado la migración más básica del servicio desde *Docker Compose* hacia *Kubernetes*. Esto se ha realizado mediante el empleo de la unidad básica de ejecución, los *Pods*. Como se menciona en la sección homóloga del anexo, A.3.1, los *Pods* representan uno o más contenedores que comparten el mismo entorno de red y almacenamiento.

Sin embargo, en entornos reales, gestionar únicamente *Pods* individuales sería insuficiente, debido a las necesidades de escalabilidad, resiliencia y gestión de aplicaciones complejas. Es aquí donde entran en juego estructuras adicionales para manejar estos desafíos de manera efectiva, como son los *Services*, *ReplicaSets* y *Deployments*, entre otros.

Por una parte, necesitamos poder comunicar los distintos *Pods* que componen el servicio, es decir, cada *Pod* debe ser capaz de interactuar con otros dentro de la misma red. Concretamente, en un entorno de red proporcionado por *Kubernetes*, los *Pods* pueden intercambiar datos si conocen la dirección *IP* del *Pod* destino. Sin embargo, este enfoque se complica cuando se utilizan mecanismos de escalado y actualización avanzados, que implican la creación y eliminación dinámica de réplicas. Esta funcionalidad viene dada, en nuestro caso, por los *Deployments* (que desarrollaremos más adelante) y, como resultado, las direcciones *IP* de los *Pods* son asignan de manera dinámica y pueden cambiar con frecuencia.

Este fenómeno se debe a que cada nuevo *Pod* creado por un *Deployment* recibe una dirección *IP* aleatoria dentro de un rango determinado al momento de su creación. Este dinamismo en la asignación de *IP* puede complicar la comunicación y gestión de los *Pods*, ya que las direcciones pueden cambiar continuamente.

Para abordar esta problemática y asegurar una comunicación estable y confiable entre componentes, *Kubernetes* utiliza el componente conocido como *Services*. Los *Services* proporcionan una capa de abstracción que encapsula la lógica de comunicación, donde cada *Pod* puede ser referenciado mediante un nombre de servicio en lugar de una dirección *IP* específica. Esta abstracción permite que el *Service* maneje el descubrimiento de *Pods* (entre otros) de manera transparente, garantizando que las aplicaciones permanezcan accesibles y funcionales a pesar de los cambios en la infraestructura subyacente[1].

En nuestro caso, el empleo de servicios es muy similar en todos los elementos de la aplicación, ya que emplearemos el mismo tipo, `ClusterIP`, para todos los componentes. Tomaremos como ejemplo el *Service* empleado para el componente *NATS*:

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: nats-1
```

[1]Para ampliar información, consultar la sección A.3.5

```
spec:
  selector:
    app: nats-1
  ports:
    - name: port1
      protocol: TCP
      port: 4222
      targetPort: 4222
    - name: port2
      protocol: TCP
      port: 8222
      targetPort: 8222
  type: ClusterIP
```

Listing 5.5: Manifiesto YAML para el despliegue del Servicio de NATS

En este servicio en concreto, la sección `ports`, que se emplea para definir puertos en uso define el puerto 4222 y el puerto 8222, que son expuestos. Estos puertos están asociados tanto al puerto interno del *Service* como al puerto objetivo en los *Pods*.

El tipo `ClusterIP` indica que el *Service* es accesible solo dentro del clúster de *Kubernetes*, proporcionando una dirección *IP* estable para los *Pods* seleccionados sin exponer el servicio al exterior.

Una vez resuelto el mecanismo de comunicación interna entre los distintos componentes que conforman la aplicación, el siguiente paso crítico es definir el punto de entrada o acceso a la misma. En un entorno gestionado por *Kubernetes*, se dispone de varios enfoques para establecer este punto de ingreso, cada uno adaptado a diferentes necesidades y arquitecturas de red.

Entre las opciones más comunes se encuentra el uso del recurso conocido como *Ingress Controller*, encargado de gestionar el tráfico externo hacia los servicios internos. Por otra parte, el recurso *LoadBalancer* es otra alternativa, especialmente en infraestructuras de nube pública, permitiendo que el tráfico entrante sea distribuido eficientemente entre las instancias disponibles.

Podríamos profundizar extensamente en este tema, dado que ofrece un abanico considerable de opciones y complejidades en términos de posibilidades técnicas y arquitectónicas. Sin embargo, para mantener el enfoque en la eficiencia y simplicidad del proyecto, optaremos por la solución más directa y adecuada a los requisitos actuales, evitando una sobrecarga innecesaria de complejidad. Para ello emplearemos el método mas elemental para acceso a la aplicación, y es el empleo del mecanismo de *port-forwarding*.

Este mecanismo permite establecer una conexión directa entre un puerto local en la máquina del usuario y un puerto específico, en nuestro caso, de un *Service* dentro del clúster. Al emplearlo, el tráfico que se envía desde el puerto local se redirige al puerto correspondiente del *Pod* o *Service* seleccionado [29].

Para ello empleamos el siguiente comando:

```
kubectl port-forward service/oauth-service 8087:80
```

Listing 5.6: Redirección del puerto local 8087 al 80 del servicio

Esta redirección vincula el puerto local 8087 con el 80 del servicio OAuth. Una vez establecido podemos acceder al servicio, como se observa en la figura 5.4

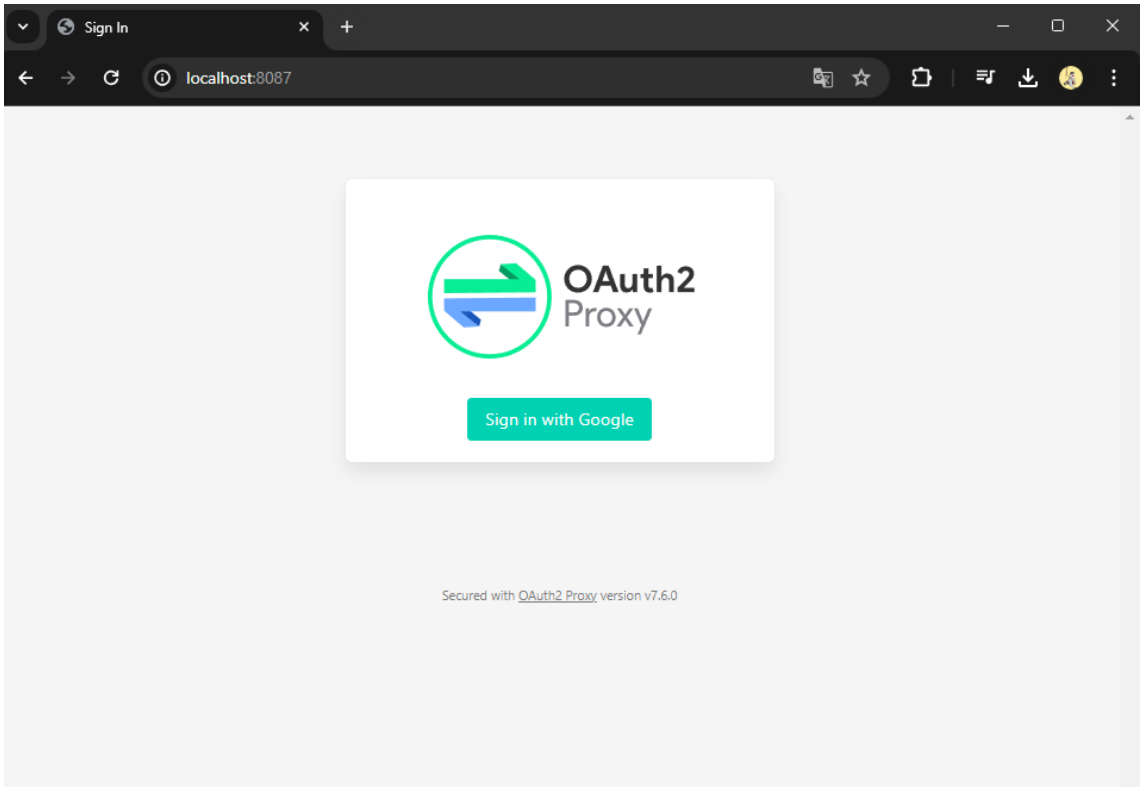

Figura 5.4: Acceso al servicio desde navegador.

Y, si consultamos la traza del comando expuesto anteriormente, comprobamos que efectivamente redirige el tráfico, como se visualiza en la figura 5.5.

```
miguellomot@DESKTOP-OP85O51:~ $ kubectl port-forward service/oauth-service 8087:80
Forwarding from 127.0.0.1:8087 -> 8080
Forwarding from [::1]:8087 -> 8080
Handling connection for 8087
Handling connection for 8087
```

Figura 5.5: Redirección de tráfico via *port-forwarding*.

Por último, una vez definido definido el mecanismo de comunicación entre los distintos componentes dentro del clúster, es necesario determinar los mecanismos de escalado y resiliencia. Para ello se hace prioritario el empleo de *ReplicaSets* y *Deployments*, ya que, como se determina en la sección del anexo A.3.2, un *ReplicaSet*, por ejemplo, garantiza que siempre haya una cantidad específica de *Pods* en ejecución, reemplazando automáticamente aquellos que fallan.

Por desgracia, los *ReplicaSets* no son suficiente para realizar mejoras significativas en la resiliencia y escalabilidad del clúster. Esto se debe a que, el principal problema con los *ReplicaSets*, es su falta de capacidad para gestionar adecuadamente el ciclo de vida de las actualizaciones y cambios en los *Pods*[2].

Este problema se resuelve mediante el uso de *Deployments*, que son una capa superior sobre los *ReplicaSets*. Un *Deployment* proporciona herramientas para realizar actualizaciones controladas de las aplicaciones, implementaciones sin interrupciones y retrocesos

[2]Consultar sección A.3.2 para ampliar más información al respecto

automáticos en caso de fallos (como se puede observar en la figura A.5), lo que facilita la operación de aplicaciones complejas en producción[3].

Es por eso que encapsularemos todos los distintos Pods que componen el servicio en Deployments. Para ejemplarizarlo, emplearemos el deployment de NATS[4]:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nats-1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nats-1
  template:
    metadata:
      labels:
        app: nats-1
    spec:
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: kubernetes.io/hostname
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              app: nats-1
      containers:
        - name: nats-1
          image: nats:latest
          command: ["nats-server", "-js", "-DV", "-http_port", "8222"]
          ports:
            - containerPort: 4222
            - containerPort: 8222
          env:
            - name: NATS_JETSTREAM
              value: "1"
```

Listing 5.7: Manifiesto YAML para el despliegue del Deployment de NATS

Este *Deployment* tiene varias secciones clave, cada una con un propósito específico. A continuación describiremos las principales junto con su función:

- `kind: Deployment`: Define el tipo de objeto que se está creando. En este caso, `Deployment`, que se encarga de gestionar y actualizar automáticamente los *Pods* y las réplicas necesarias.
- `metadata`: Contiene metadatos del objeto, como el nombre (`name: nats-1`). Este nombre es importante para referenciar el objeto dentro del clúster.
- `spec`: Define la especificación del *Deployment*. La sección `spec` define el estado deseado del objeto, es decir, donde se determinan las características principales, incluyendo cantidad de réplicas, plantilla de los *Pods*, selector de etiquetas, etc.

  Dentro de `spec`, las subsecciones clave son las siguientes:

  - `replicas: 1` indica cuántas réplicas del *Pod* se desean ejecutar. En este caso, sólo se especifica 1 réplica.

[3] Al igual que los ReplicaSets, se puede ampliar más información en la sección del anexo A.3.3

[4] Todos los *Deployments* planteados para la ejecución del servicio son similares, únicamente varía (aparte de campos selectores como nombre y etiquetas) la definición del Pod interno.

- `selector`: Utiliza `matchLabels` para indicar las etiquetas con las que Kubernetes identificará los *Pods* gestionados por este *Deployment*. En este caso la etiqueta se correspondería con `app: nats-1`.
- `template`: Define la plantilla del *Pod* que se usará para las réplicas del *Deployment*. Al igual que la sección `spec`, `template` se compone de varias secciones clave (adicionales a la sección de metadata, que ya ha sido comentada):
  - `containers`: Define el contenedor que se ejecutará dentro del *Pod*. En este caso, se especifica el contenedor exactamente igual que se ha expuesto en la sección 5.1.3.
  - `topologySpreadConstraints`: Esta sección configura las reglas para la distribución de los *Pods* en diferentes nodos dentro del clúster. Al ser una sección especial, se explorará a continuación.

5.2.1. topologySpreadConstraints: Distribución uniforme de réplicas

Dentro del clúster propuesto existe un defecto. En caso de aumento del número de réplicas, y de no indicar lo contrario, todas se planificarían en el mismo nodo hasta saturarlo.

Para ampliar más información y entender la lógica detrás de la implementación de esta etiqueta y su relación con la distribución de las réplicas debemos saber un poco más sobre el proceso de programación (mejor conocido como *scheduling*).

En *Kubernetes*, el proceso de programación de Pods a nodos es gestionado por el *kube-scheduler*. Este componente evalúa el estado del clúster y la disponibilidad de recursos, considerando varios factores, como la capacidad de *CPU* y memoria, las etiquetas de `affinity` y `anti-affinity`, y las restricciones de `taints` y `tolerations`. `kube-scheduler` primero identifica los nodos candidatos y luego aplica políticas de programación, como `topologySpreadConstraints`, para seleccionar el nodo más adecuado para el *Pod*.

Una vez que se ha tomado una decisión, el `kube-scheduler` asigna el *Pod* al nodo seleccionado, actualizando el estado del *Pod* en el *API server*. El proceso es dinámico y continuo; el *scheduler* monitorea el clúster y puede reevaluar y reasignar *Pods* en respuesta a cambios en la disponibilidad de recursos o políticas, asegurando una asignación eficiente y equilibrada de los *Pods* en el clúster.

En el contexto del proyecto actual, y como se ha mencionado antes, se ha observado que el comportamiento del *scheduler* ha mostrado una tendencia a asignar todos los *Pods* a un único nodo hasta que este alcance su capacidad máxima. Esta asignación continua puede llevar al nodo a una saturación de recursos, dado que las máquinas virtuales utilizadas poseen capacidades computacionales limitadas. Cuando el nodo se encuentra completamente saturado, su capacidad para gestionar nuevas cargas de trabajo se ve comprometida, lo que resulta en la caída del servicio que proporciona, es decir, el nodo queda inoperativo y no puede seguir desempeñando su función dentro del clúster.

Es por ello que, mediante la aplicación de las políticas definidas en esta sección del archivo *YAML* podemos realizar una distribución uniforme entre ambos nodos secundarios. Para entender cómo definimos esta política y aplicamos esta distribución vamos a profundizar más en su funcionamiento.

Como se ha mencionado antes, `topologySpreadConstraints` define las reglas para distribuir los *Pods* a través de diferentes unidades de topología, como nodos o zonas de disponibilidad [30].

- `maxSkew`: El valor de `maxSkew` especifica la diferencia máxima permitida en el número de *Pods* entre las distintas unidades de topología. En este caso, `maxSkew: 1` indica que la diferencia en la distribución de *Pods* no debe exceder 1. Esto ayuda a mantener un equilibrio en la asignación de *Pods* entre los nodos.
- `topologyKey`: Define la clave utilizada para identificar las unidades de topología en las que se aplicará la distribución. En este caso, `kubernetes.io/hostname` se refiere al nombre del nodo en el clúster. Esto significa que la distribución de *Pods* se basa en los nodos individuales.
- `whenUnsatisfiable`: Este campo determina la acción a tomar si no se puede cumplir la restricción especificada. Concretamente, `ScheduleAnyway` indica que, si no se puede cumplir con la distribución ideal, los *Pods* serán programados de todos modos, asegurando que se mantenga la operación del clúster incluso si el equilibrio no es perfecto.
- `labelSelector`: Como se ha mencionado en anteriores ocasiones, `labelSelector` define un selector de etiquetas que se usa para identificar qué *Pods* deben considerar la restricción de distribución.

Conocido en profundidad la definición y lógica detrás de esta sección, pongamos un ejemplo de funcionamiento. Supongamos que tenemos un clúster con dos nodos: `Nodo A` y `Nodo B`. El `Deployment` tiene configuradas 3 réplicas para el *Pod* con la etiqueta `app: nats-1`. Primero, el `scheduler` evaluará la distribución de los *Pods* basándose en el *hostname* de cada nodo. Por lo tanto, el `scheduler` intentará distribuir las réplicas de manera que la diferencia en el número de las mismas entre `Nodo A` y `Nodo B` sea como máximo 1 (basándose en `maxSkew: 1`).

El `scheduler` comienza asignando *Pods* a los nodos disponibles. En este caso, con 3 Pods y 2 nodos, la asignación ideal sería distribuir 2 Pods en un nodo y 1 Pod en el otro para cumplir con la restricción de `maxSkew`. Supongamos que se decide asignar 2 Pods al `Nodo A` y 1 Pod al `Nodo B`. Esto cumple con la restricción de `maxSkew` ya que la diferencia en la cantidad de Pods entre `Nodo A` y `Nodo B` es 1, lo cual es aceptable.

Si, en cambio, el `scheduler` intenta asignar 3 Pods a `Nodo A` y ninguno a `Nodo B`, incumpliría con la restricción, ya que la diferencia sería de 3.[5]

[5]Cabe destacar que, en este caso y debido a la configuración de `whenUnsatisfiable: ScheduleAnyway`, el `scheduler` aún podría programar los *Pods* a pesar de no cumplir completamente con la restricción, aunque esta no es la distribución ideal [30].

CAPÍTULO 6
Conclusión y Futuras Implementaciones

A lo largo de este trabajo, se ha logrado cumplir los objetivos delineados en el primer capítulo, además de cumplir con la meta de diseñar e implementar un servicio de ejecución de código en la nube basado en una arquitectura de contenedores orquestada mediante *Kubernetes*, garantizando su alta disponibilidad, escalabilidad y resiliencia.

El diseño de la infraestructura, esencial para soportar el servicio, ha sido un componente crítico en la implementación de la solución. La configuración de las máquinas virtuales y la instalación del software necesario han establecido una base robusta que asegura el correcto funcionamiento del clúster de *Kubernetes*, permitiendo que los servicios se ejecuten de manera eficiente y sin interrupciones. Esta infraestructura ha sido esencial para cumplir con los requisitos del proyecto, garantizando la estabilidad y la capacidad de expansión del sistema.

Adicionalmente, la efectividad de *Kubernetes* como herramienta para la orquestación y gestión de contenedores ha quedado demostrada a través de esta implementación, resultando en mejoras significativas en la resiliencia y agilidad del sistema. Si bien se han alcanzado la mayoría de los objetivos establecidos, la implementación ha revelado áreas donde se puede continuar mejorando, especialmente en la integración de soluciones avanzadas de monitoreo, optimización de la infraestructura de red y el aumento del grado de autonomía en el escalado, tanto en infraestructura como en componentes internos de cada nodo.

Es por esto que, en lo que respecta a etapas futuras, se sugiere continuar con el desarrollo y optimización de la infraestructura, siguiendo las directrices y líneas de acción expuestas en este documento.

Primeramente, resulta fundamental abordar la integración de soluciones avanzadas de monitoreo y la optimización de la infraestructura de red para asegurar un rendimiento óptimo y una transición exitosa hacia fases de desarrollo posteriores.

Una de estas soluciones es la integración de la conocida herramienta *Prometheus*, empleada para el monitoreo sobre recursos *software* y *hardware*. La principal característica de esta herramienta es la posibilidad de obtención de métricas de todos los elementos dentro de *Kubernetes*. Por ejemplo, mediante el uso de *operators* (que esencialmente son controladores implementados por la comunidad cuyo objetivo es la obtención y exportación de métricas en formato *JSON* de un determinado recurso), *Prometheus* es capaz de obtener datos de infinidad de recursos.

Una vez los datos son obtenidos tendremos la capacidad de implementar el escalado de *Pods* automático, mediante el empleo de un recurso de *Kubernetes* llamado *Horizon-*

talPodAutoscaler. Esta característica es capaz de actualizar el número de réplicas de un *Deployment* o *Statefulset* dependiendo de la carga de trabajo (que se determina empleando las métricas obtenidas de los controladores mencionados en el párrafo anterior).

Además, es recomendable realizar una revisión periódica de la arquitectura y la estrategia implementada, con el fin de asegurar su alineación con los objetivos y desafíos emergentes. Para ello es interesante observar la posibilidad de migrar la gestión del *Plano de Control* hacia entornos en la nube como *Amazon Web Services* o *Microsoft Azure*. Esto permitirá que la fiabilidad del servicio se incremente y emplear herramientas secundarias como la orquestación e incorporación (o eliminación) dinámica de nuevos nodos de computación al clúster, dependiendo de la demanda.

Glosario

asincronía La asincronía se define como un suceso que no tiene lugar en total correspondencia temporal con otro suceso. En mensajería computacional, se establece un sistema asíncrono como aquel en el que no existen garantías en las velocidades relativas de ejecución en los distintos nodos del sistema ni en el tiempo que un canal tarda en entregar el mensaje transmitido [31].. 51

comunicación de tipo publicación/suscripción (pub/sub) La comunicación de tipo publicación/suscripción (pub/sub) es un patrón de mensajería que permite que los sistemas intercambien mensajes de manera **asincronía** y **desacoplada** [32]. Este modelo es ampliamente utilizado en sistemas distribuidos, arquitecturas de microservicios, y aplicaciones de mensajería en tiempo real.. 9

Container Runtime Un runtime de contenedores es un software que ejecuta contenedores y gestiona sus imágenes en un nodo de despliegue [21]. En la actualidad existen múltiples opciones en el mercado pero la más conocida lleva el nombre de *Containerd*. Es el *runtime* utilizado por el motor de *Docker* y, a menudo, se lo considera un estándar de la industria debido a su amplia adopción [21] [33]. . 29

Daemon Un daemon o servicio es un tipo especial de programa que se ejecuta en segundo plano, en vez de ser controlado directamente por el usuario. No disponen de interfaz y no utilizan las entradas y salidas estándar para registrar su funcionamiento, sino que emplean, en su mayoría, archivos del sistema. [34]. 59

rollback El término *rollback* se refiere a la acción de "deshacer" o "revertir" un conjunto de operaciones para restaurar un estado previo. En el contexto del desarrollo de software, la operación de *rollback* actúa como una estrategia de recuperación crítica que permite a los desarrolladores volver a una versión anterior (normalmente estable) en caso de que la nueva versión del software genere problemas inesperados [35].. 22, 64

Bibliografía

[1] Adrián Barbáchano Cirión. Comparativa de prestaciones de servidores virtualizados. *ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN. Universidad Pública de Navarra*, 2010. Consultado el 30 de julio de 2024.

[2] Dharma Web Studio. ¿Conoces los entornos de trabajo utilizados en el desarrollo de una aplicacion web? Consultado el 24 de julio de 2024.

[3] Wikipedia contributors. High availability — Wikipedia, the free encyclopedia, 2024. [Online; accessed 3-September-2024].

[4] Wikipedia. Equilibrio de carga — wikipedia, la enciclopedia libre, 2024. [Internet; descargado 19-enero-2024].

[5] J. Laprie. From dependability to resilience. In *Dependable Systems and Networks*, 2008. Consultado el 25 de agosto de 2024.

[6] ResiliNetsWiki. Definitions - resilinetswiki, 2014. Consultado el 25 de agosto de 2024.

[7] The Go Authors. The go programming language, 2024. Consultado el 07 de agosto de 2024.

[8] The NATS Authors. Nats documentation, 2024. Consultado el 07 de agosto de 2024.

[9] Mike Roberts. Serverless architectures. Blog de Martin Fowler, 2016. URL: https://martinfowler.com/articles/serverless.html, Consultado el 07 de agosto de 2024.

[10] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017. Consultado el 14 de agosto de 2024.

[11] Docker, Inc. Docker compose documentation, 2024. Consultado el 07 de agosto de 2024.

[12] Arsys. Cómo hacer documentación técnica de software, 2024. Accedido: 2024-08-26.

[13] Rafi Mojabi. Create a kubernetes cluster using virtualbox and without vagrant. `https://medium.com/@mojabi.rafi/create-a-kubernetes-cluster-using-virtualbox-and-without-vagrant-90a14d791617`, 2020. Accessed: 2024-08-28.

[14] Md Abdullah Al Rabbi. Kubernetes cluster setup on ubuntu 24.04 lts server. https://medium.com/@rabbi.cse.sust.bd/kubernetes-cluster-setup-on-ubuntu-24-04-lts-server-c17be85e49d1, 2022. Consultado el 28 de agosto de 2024.

[15] Kubernetes Documentation. Creating a cluster with kubeadm. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/. Consultado el 28 de agosto de 2024.

[16] Abhishek Verma. Create a kubernetes cluster with kubeadm on virtualbox. https://medium.com/@abhishek261291/create-a-kubernetes-cluster-with-kubeadm-on-virtualbox-8268d6525b, 2020. Consultado el 28 de agosto de 2024.

[17] The VirtualBox Authors. Virtualbox official documentation: Chapter 6. virtual networking (inglés). Consultado el 07 de agosto de 2024.

[18] GitHub Contributors. Issue #53533: volume deletion does not delete container volume directory. https://github.com/kubernetes/kubernetes/issues/53533, 2017. Consultado el 27 de agosto de 2024.

[19] Erik Svendsen. Swap off, why is it necessary? https://discuss.kubernetes.io/t/swap-off-why-is-it-necessary/6879/2, 2019. Consultado el 27 de agosto de 2024.

[20] The Kubernetes Authors. Kubernetes official documentation: Container runtime interface (cri) (inglés). Consultado el 27 de agosto de 2024.

[21] Ramzi Debab and Walid Khaled Hidouci. Containers runtimes war: a comparative study. In *Proceedings of the Future Technologies Conference (FTC) 2020, Volume 2*, pages 135–161. Springer, 2021. Consultado el 27 de agosto de 2024.

[22] J. Castro, D. Cooley, K. Cosgrove, J. Garrison, N. Kantrowitz, B. Killen, R. Lejano, D. Papandrea, J. Sica, and D. Srinivas. Don't panic: Kubernetes and docker. https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/, 2020. Consultado el 28 de agosto de 2024.

[23] Sergey Kanzhelev, Jim Angel, Davanum Srinivas, Shannon Kularathna, Chris Short, and Dawn Chen. Kubernetes is moving on from dockershim: Commitments and next steps. https://kubernetes.io/blog/2022/01/07/kubernetes-is-moving-on-from-dockershim/, 2022. Consultado el 28 de agosto de 2024.

[24] Containerd Authors. Getting started with containerd. https://github.com/containerd/containerd/blob/main/docs/getting-started.md, 2024. Consultado el 28 de agosto de 2024.

[25] Wikipedia. Gnu privacy guard — wikipedia, la enciclopedia libre, 2024. Consultado el 28 de agosto de 2024.

[26] The Kubernetes Authors. Kubernetes official documentation: Creating a cluster with kubeadm (inglés). Consultado el 27 de agosto de 2024.

[27] Canonical. Netplan documentation: Introduction to netplan. https://netplan.readthedocs.io/en/stable/structure-id/, 2024. Consultado el 28 de agosto de 2024.

[28] The Kubernetes Authors. Kubernetes official documentation: Images (inglés). Consultado el 04 de septiembre de 2024.

[29] The Kubernetes Authors. Kubernetes official documentation: Use port forwarding to access applications in a cluster (inglés). Consultado el 04 de septiembre de 2024.

[30] The Kubernetes Authors. Kubernetes official documentation: Pod topology spread constraints (inglés). Consultado el 04 de septiembre de 2024.

[31] Pablo Galdámez Saiz. *Fundamentos de los Algoritmos Distribuidos*. Máster Universitario en Computación en la Nube y de Altas Prestaciones. Universidad Politécnica de Valencia, 2024. Consultado el 07 de agosto de 2024.

[32] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus: an architecture for extensible distributed systems. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 58–68, 1993.

[33] The Kubernetes Authors. Kubernetes official documentation: Container runtimes (inglés). Consultado el 27 de agosto de 2024.

[34] Wikipedia. Daemon (informática) — wikipedia, la enciclopedia libre, 2024. Consultado el 24 de julio de 2024.

[35] Team Split. Software rollback, 2024. Consultado el 25 de agosto de 2024.

[36] DigitalOcean. What is Kubernetes? (inglés). Consultado el 24 de julio de 2024.

[37] The Kubernetes Authors. Kubernetes official documentation: Control plane components (inglés). Consultado el 24 de julio de 2024.

[38] Docker, Inc. Containerd vs. docker: Understanding the differences, 2022. Consultado el 24 de julio de 2024.

[39] J Hale. Key kubernetes concepts, 2019. Consultado el 25 de julio de 2024.

[40] The Kubernetes Authors. Kubernetes official documentation: Pods (inglés). Consultado el 25 de julio de 2024.

[41] The Kubernetes Authors. Kubernetes official documentation: Replicaset (inglés). Consultado el 30 de julio de 2024.

[42] The Kubernetes Authors. Kubernetes official documentation: Deployment (inglés). Consultado el 30 de julio de 2024.

[43] The Kubernetes Authors. Kubernetes official documentation: Statefulset (inglés). Consultado el 30 de julio de 2024.

[44] The Kubernetes Authors. Kubernetes official documentation: Service (inglés). Consultado el 30 de julio de 2024.

[45] The Kubernetes Authors. Kubernetes official documentation: Configmap (inglés). Consultado el 07 de agosto de 2024.

[46] The Kubernetes Authors. Kubernetes official documentation: Secret (inglés). Consultado el 07 de agosto de 2024.

[47] The Kubernetes Authors. Kubernetes official documentation: Namespaces (inglés). Consultado el 07 de agosto de 2024.

[48] The Kubernetes Authors. Kubernetes official documentation: Volume (inglés). Consultado el 07 de agosto de 2024.

[49] The Kubernetes Authors. Kubernetes official documentation: Persistent volumes (inglés). Consultado el 07 de agosto de 2024.

[50] The Kubernetes Authors. Kubernetes official documentation: Ingress (inglés). Consultado el 07 de agosto de 2024.

APÉNDICE A
Kubernetes: Introducción y Estructuras elementales

A.1 Conceptos básicos de Kubernetes

Kubernetes es una tecnología de código abierto empleada para manejar y orquestar contenedores de forma portable y extensible. Su principal aplicación es el manejo de servicios y cargas de trabajo basadas en contenedores que faciliten aproximaciones de configuración de forma declarativa y automatizada.

La idea principal tras *Kubernetes*, mediante la orquestación de software basado en contenedores (que ayuda a empaquetar el software desarrollado), es permitir que las aplicaciones se liberen y actualicen sin tiempo de inactividad. Brinda herramientas para escalar, mantener alta disponibilidad y monitorizar estas aplicaciones.

A.1.1. Arquitectura general de Kubernetes

Para entender la lógica detrás de *Kubernetes* tendremos que visualizar su funcionamiento con un nivel de abstracción relativamente elevado, y después profundizar en cada componente. Como se menciona en [36], *Kubernetes* puede entenderse como un sistema complejo compuesto de capas, donde cada capa tiene un nivel de abstracción más elevado que la anterior.

En el nivel más físico de *Kubernetes* observamos que la tecnología se encarga de coordinar varias máquinas (que pueden ser obtenidas de varias formas, sean ya equipos físicos o virtualizados mediante tecnologías como *VirtualBox*) para formar un clúster computacional. Esto lo realiza empleando una red común de comunicación para interconectar los distintos nodos, convirtiendo así este clúster en un espacio unificado e interconectado donde se lanzarán las distintas componentes adicionales que dan soporte a la aplicación y todas las características que brinda esta tecnología.

Una vez intercomunicados, todos los componentes computacionales del clúster reciben un rol, ya que se sigue un modelo **primario-secundario**, donde uno de los nodos (o en su defecto un subconjunto) actúa como el maestro, incluyendo componentes que toman decisiones (como asignar y dividir tareas [1]), detectan y responden a eventos en la totalidad del clúster, esencialmente orquestando al completo su funcionamiento[37]. El servidor principal, conocido como Plano de Control, actúa esencialmente como un punto primario de contacto con el clúster y se hace responsable, de forma centralizada, de toda la lógica que proporciona la tecnología.

[1] A este fenómeno se le denomina *scheduling* y se desarrollará más adelante

Por otra parte el resto de máquinas ajenas al ya mencionado Plano de Control actúan con un papel de nodos secundarios. Su única tarea es la de alojar y dar soporte todas las cargas de trabajo que aloja *Kubernetes*. Esto se realiza, como ya se ha mencionado antes, mediante el empleo de contenedores. Es por ello que en cada nodo secundario se necesita incluir una serie de componentes adicionales que dan soporte al funcionamiento. Estas herramientas adicionales se componen de: procesos y comunicación con el Plano de Control (para recibir instrucciones de trabajo del nodo maestro y repartir tráfico de forma apropiada) y procesos para la ejecución y control de contenedores (para traducir las instrucciones recibidas en creación y destrucción de contenedores), que desarrollaremos más adelante.

Por último, como se ha mencionado en el párrafo anterior, todas las cargas de trabajo de la aplicación se basan en el uso y orquestación de servicios en contenedores. Es por ello que el resto de componentes subyacentes existen para asegurar que el estado de los contenedores del clúster es óptimo. Para ello se emplea el siguente modo de funcionamiento: para iniciar un servicio, se presenta uno o varios archivos en formato JSON o YAML en los que se contiene, de forma declarativa, el plan de ejecución que define qué crear y cómo debe gestionarse. El Plano de Control toma este plan y determina cómo ejecutarlo en la infraestructura, examinando los requisitos y el estado actual del sistema. Este grupo de objetos y aplicaciones definidas por el usuario, que se ejecutan según un plan especificado, representa la capa final de *Kubernetes* [36].

A.2 Componentes principales de *Kubernetes*

Una vez descrito el modo general de funcionamiento y desarrollada, de forma general, la lógica empleada subyacentemente, vamos a profundizar más en los distintos elementos que componen las diferentes capas expuestas en la sección anterior.

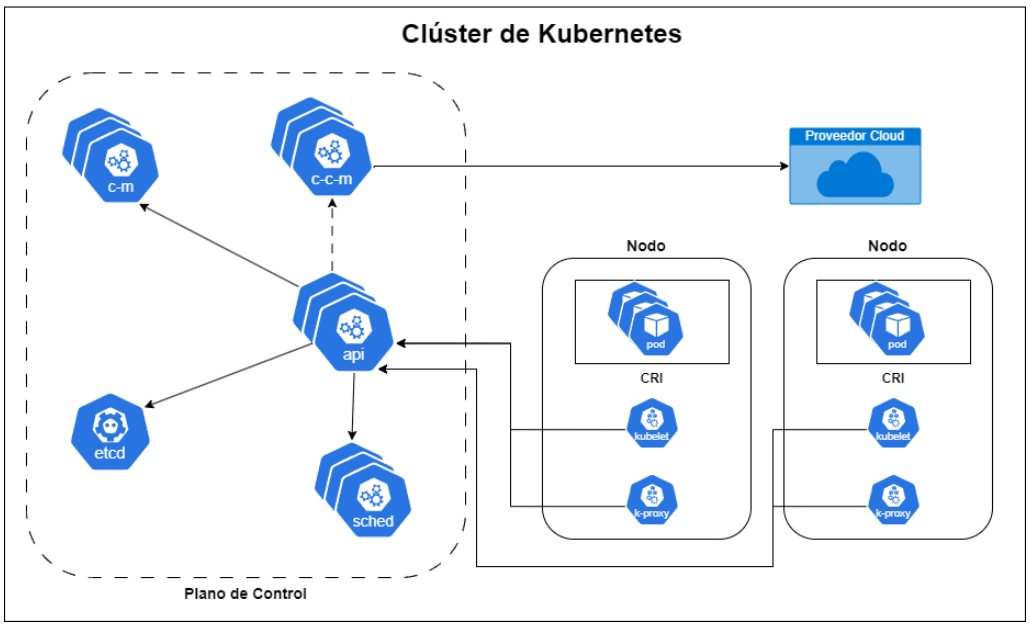

Figura A.1: Componentes de un clúster de *Kubernetes*

Como podemos observar en la figura A.1, un clúster de *Kubernetes* está típicamente compuesto de varios nodos físicos y virtuales con distintos roles. Dependiendo del rol que toman en el funcionamiento del clúster, cada nodo queda compuesto por unos componentes u otros. Estos componentes se emplean para describir el estado deseado del

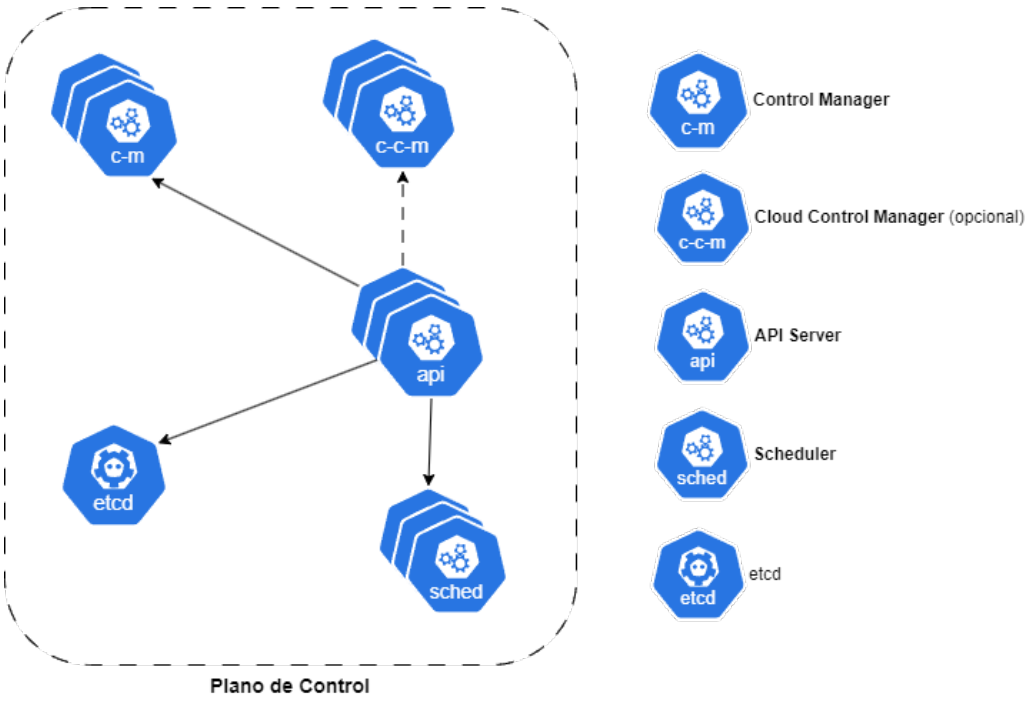

Figura A.2: Componentes del Plano de Control

clúster: qué aplicaciones u otras cargas de trabajo se quieren ejecutar, qué imágenes de contenedores utilizan, el número de réplicas, la red y los recursos de almacenamiento que deben estar disponibles, etc. De forma más concreta, cabe destacar que, el estado deseado del clúster, se especifica mediante la creación de objetos usando la API de *Kubernetes*, típicamente a través de la interfaz de línea de comandos, `kubectl`, o tecnologías de automatización externas. También es posible utilizar la API de *Kubernetes* directamente para interactuar con el clúster y especificar o modificar el estado deseado.Para obtenerlo, *Kubernetes* realiza diversas tareas de forma automática, como detener e iniciar contenedores o escalar el número de réplicas de una aplicación, entre otras.

A.2.1. Plano de Control (Nodo Maestro)

Los diferentes componentes del Plano de Control de *Kubernetes* gestionan la comunicación entre *Kubernetes* y el clúster. Este mantiene un registro de todos los objetos presentes en el sistema y ejecuta bucles de control de forma continua para gestionar su estado. En algún momento, los bucles responden a los cambios realizados en el clúster y realizan las acciones necesarias para que el estado actual de todos los objetos del sistema converja hacia el estado deseado que se proporciona en las especificaciones del servicio que se expone. Todo esto se realiza mediante el empleo de distintos *Daemons*. Los *daemons* que componen el Plano de Control, como quedan reflejados en la figura A.2 son:

- **API Server** (`kube-apiserver`): Interfaz *frontend* que expone la API de *Kubernetes*. Desde aquí los usuarios interactúan con el clúster.
- `etcd`: Almacenamiento consistente y altamente disponible de tipo clave-valor donde se mantienen los datos necesarios para manejar el clúster.
- **Scheduler** (`kube-scheduler`): Encargado de distribuir la carga de trabajo entre los diferentes nodos. Busca contenedores nuevos y los asigna a los distintos nodos disponibles.

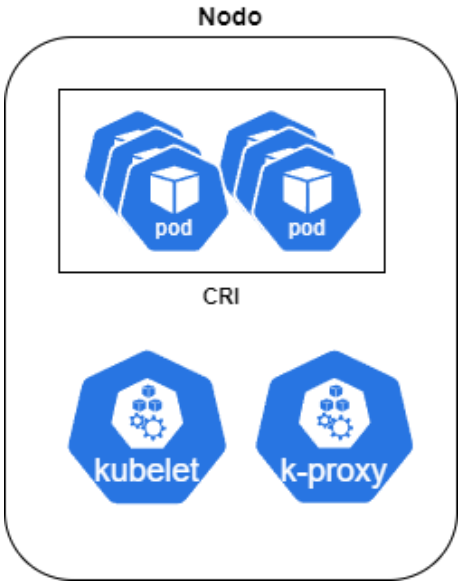

Figura A.3: Componentes de un Nodo

- **Controladores**: Responsables de monitorizar los nodos y responder en caso de que dejen de funcionar. En este caso pueden tomar decisiones para lanzar nuevos contenedores. Estos controladores son:

  - **Controller Manager** (`kube-controller-manager`): Encargado de ejecutar todos los procesos controladores del cluster. Cada controlador es un proceso lógico separado, pero para reducir la complejidad se compilan y lanzan como un único proceso. Existen varios tipos de controladores integrados en este componente (`node-controller`, `job-controller`, etc).
  - **Cloud Controller Manager** (`cloud-controller-manager`): Este controlador contiene embebida toda la lógica de control de los elementos en la nube. Permite enlazar el clúster en el que se ejecuta con las APIs de los proveedores cloud. Al igual que su homólogo *On-premise*, el c-c-m (denominado así por las siglas de su nombre), combina varios procesos lógicos independientes específicos de cada proveedor cloud en un único proceso para reducir la complejidad.

A.2.2. Nodos (Nodos secundarios o workers)

Como se ha mencionado en la sección de arquitectura del apartado A.1, los nodos secundarios (conocidos simplemente como nodos) son equipos físicos o máquinas virtuales que sirven como nodos de trabajo dentro del clúster. Estos nodos, internamente, tienen dedicados una serie de *daemons* que corren en todos los nodos del clúster. Estos componentes tienen como objetivo mantener los Pods[2] en ejecución y proveer a *Kubernetes* de un entorno de ejecución.

Como podemos observar en la figura A.3, los componentes que se ejecutan en cada nodo son:

- `kubelet`: Responsable de asegurar que los contenedores funcionan como deberían en los nodos.

[2]Este concepto se desarrollará más adelante en el apartado A.3

- `kube-proxy`: Proxy de red que tiene como función mantener todas las reglas de red dentro de los nodos. Es el componente que, junto a una serie de objetos adicionales, implementa el concepto de Servicio [3].
- **Container Runtime**: Es el componente subyacente encargado de lanzar los entornos de ejecución de contenedores, como el estándar de industria `containerd` (que es el utilizado por la tecnología Docker[4]).

A.2.3. Complementos

Dentro de *Kubernetes* existen componentes adicionales que no forman parte del núcleo del sistema, pero que proporcionan funcionalidades esenciales o útiles para la operación y gestión del clúster. Suelen ser desplegados como aplicaciones en el clúster de *Kubernetes* y pueden ser gestionados y actualizados utilizando las mismas herramientas y procesos que otras aplicaciones de la tecnología.

Existen varios complementos disponibles para ser integrados dentro de los clústeres, pero vamos a comentar únicamente los dos más notables y que son empleados dentro de este mismo proyecto. Nos referimos a los siguientes:

- **DNS**: Proporciona servicios de nombres dentro del clúster para que los Pods puedan encontrarse unos a otros por nombre.
- **Plugins de red (CNI)**: Plugins de red que implementan la especificación de interfaz de red de contenedores (CNI), como Calico, Flannel, o Weave, para gestionar la conectividad de red entre Pods.

A.3 Objetos básicos en *Kubernetes*

Una carga de trabajo es una aplicación que se ejecuta en *Kubernetes*, ya sea mediante un solo componente o varios que funcionan juntos, dentro de un conjunto de **Pods**. Un Pod es la unidad mínima de gestión dentro de *Kubernetes*, ya que no existe ningún otro objeto más pequeño que se podrá iniciar, parar o eliminar mediante las órdenes disponibles dentro de la tecnología. Por consecuente, ya que el objetivo principal dentro de *Kubernetes* es el de orquestar flujos de trabajo basados en contenedores, podemos representar un Pod como un conjunto de uno o varios contenedores en ejecución dentro de un clúster.

Los Pods en *Kubernetes* tienen un ciclo de vida definido. Por ejemplo, una vez que un pod está en ejecución, una falla crítica en el nodo donde se está ejecutando significa que todos los Pods en ese nodo fallarán. *Kubernetes* trata ese nivel de falla como definitivo: se necesita crear un nuevo Pod para volver al estado óptimo de funcionamiento, incluso si el nodo vuelve a estar operativo.

Sin embargo, para hacer la vida considerablemente más fácil, no se necesita gestionar cada Pod directamente. En su lugar, existen una serie de recursos de carga de trabajo que gestionan los conjuntos de Pods de forma autónoma. Estos recursos configuran controladores que aseguran que el número correcto del tipo correcto de Pods esté en ejecución, para que coincida con el estado especificado en el Plano de Control.

Según Hale [39], podemos diferenciar, dentro de los distintos objetos básicos empleados en *Kubernetes*, seis niveles de abstracción conceptuales:

[3] *Grosso modo*, un Servicio es una forma de exponer una aplicación que está siendo ejecutada en un conjunto de Pods como un servicio de red. Se desarrolla en profundidad en el apartado A.3

[4] Para más información al respecto sobre `containerd` consultar [38]

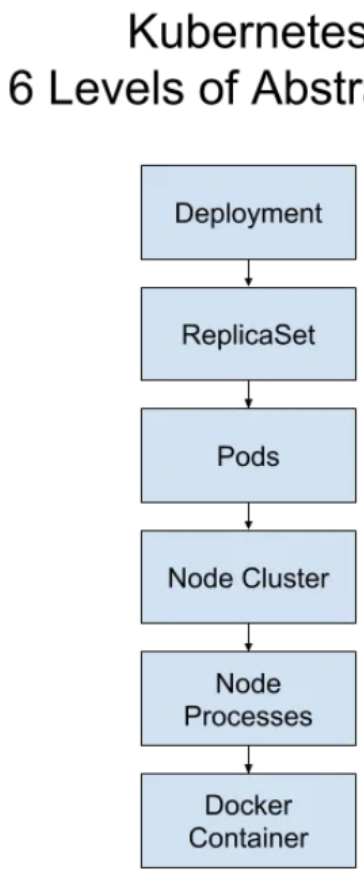

Figura A.4: Abstracciones de *Kubernetes* para un Deployment © Jeff Hale 2019

Estos niveles permiten a los usuarios interactuar con el clúster a diferentes grados de detalle y control, usando una relación jerárquica entre los distintos niveles. Como podemos observar en la Figura A.4, los *Deployments* crean y gestionan *ReplicaSets*, que crean y gestionan *Pods*, que se ejecutan en los *Nodes*, los cuales tienen un entorno de ejecución de contenedores que ejecutan el código de la aplicación [39].

A continuación, observaremos cada nivel de forma individual, comenzando por los niveles más elementales a partir del *Pod*, ya que, por una parte, los procesos implicados en el funcionamiento de los Nodos han sido descritos en el apartado anterior. Por otra parte y como observaremos a continuación, *Kubernetes* no gestiona los contenedores como un objeto como tal, ya que se integran dentro de la unidad básica de ejecución del clúster: el Pod.

A.3.1. Pod

Los Pods son las unidades de cómputo más pequeñas que se pueden crear y manejar dentro de *Kubernetes*. Esencialmente, un Pod es un grupo de uno o más contenedores, con almacenamiento y red compartidos, además de estar definidos por unas especificaciones que indican cómo ejecutarlos. Según la documentación oficial de *Kubernetes* [40], los contenidos de un Pod son siempre **coubicados**, **coprogramados**[5] y **ejecutados en un contexto compartido**.

[5] Al estar programados juntos, los contenedores dentro de un Pod pueden iniciarse y detenerse juntos. Esto asegura que los contenedores que dependen unos de otros se ejecuten en el mismo entorno y se gestionen como una única unidad, facilitando la sincronización entre ellos.

Esencialmente, un Pod modela un "host lógico" específico para la aplicación: contiene uno o más contenedores relativamente entrelazados [40]. Antes de la llegada de los contenedores, ejecutarse en la misma máquina física o virtual implicaba ser ejecutado en el mismo host lógico, ya que implicaba compartir un entorno y recursos comunes.

Por otra parte, el contexto compartido de un Pod se compone, esencialmente, de un conjunto de namespaces de Linux, cgroups y, potencialmente, otras facetas de aislamiento, las mismas cosas que aíslan un contenedor Docker.

Además, los contenedores dentro de un Pod comparten dirección IP, puerto y pueden encontrarse a través de `localhost`. También pueden comunicarse entre sí mediante mecanismos estándar de comunicación entre procesos, como semáforos o la memoria compartida POSIX, y tienen acceso a volúmenes compartidos que se definen como parte del mismo Pod y están disponibles para ser montados en el sistema de archivos de cada aplicación.

A.3.2. ReplicaSet

Un ReplicaSet es un recurso en *Kubernetes* que garantiza que un número especificado de réplicas de un Pod esté en ejecución en todo momento. Si un Pod falla o es eliminado, el ReplicaSet creará o eliminará automáticamente el número de Pods que sea necesario para alcanzar el número esperado, mediante el uso de la plantilla Pod definida en el manifiesto del objeto ReplicaSet. [41]

A la hora de proporcionar un determinado servicio dentro de un clúster, necesitamos obtener la mayor disponibilidad posible. Mientras que los Pods son la unidad básica de ejecución, los ReplicaSets proporcionan una capa adicional de control y aseguramiento de la disponibilidad, lo que es crucial para mantener aplicaciones robustas y escalables en un entorno de producción.

Por otra parte, y como se indica en la documentación oficial [41], el uso de ReplicaSets en un entorno de producción es muy escaso. Esto se debe a que existen conceptos de más alto nivel cuya único objetivo es la gestión de ReplicaSets de forma eficiente y proporcionando características adicionales. Estos objetos serán desarrollados a continuación.

A.3.3. Deployment

Los Deployments son objetos avanzados que proporcionan un mecanismo para ofrecer actualizaciones declarativas para el control de replicasets y, por consecuente, los Pods subyacentes, como podemos observar en la figura A.5.

La finalidad de este objeto es definir un estado deseado, especificado en su propia configuración, que describe el número de réplicas que se pretende ejecutar. Por otro lado, el controlador interno del *Deployment* mantiene un .estado actual", que refleja el número de réplicas que están activamente en ejecución. El objetivo principal del *Deployment* es gestionar la transición del estado actual al estado deseado, utilizando distintas estrategias de despliegue, como *Recreate* o *RollingUpdate*, para lograr esta adaptación de forma controlada.

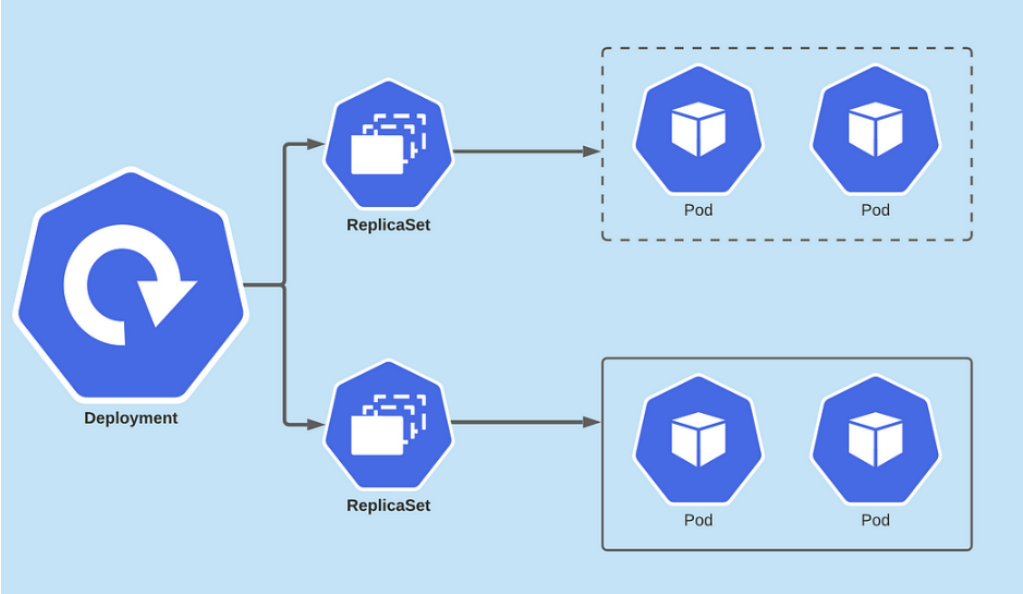

Figura A.5: Estructura de un *Deployment*

Para comprender mejor el funcionamiento del Deployment podemos establecer varios casos de común uso que vienen reflejados en la documentación oficial [42]:

- Crear un Deployment para desplegar un replicaset y comprobar su estado (si ha sido fructífero o no). Si despliegue no es satisfactorio o, en algún momento, pasa a un estado inestable, el Deployment tiene la capacidad de realizar operaciones de rollback.

- Declarar un nuevo estado de los pods. Esto se debe a que, si se necesita actualizar la declaración fundamental del modelo de Pod contenido dentro del Deployment (un ejemplo sería para cambiar la imagen que usan sus contenedores) implica la creación de un nuevo ReplicaSet, por lo que debemos transitar del objeto antiguo al nuevo.

- Realizar operaciones de escalado horizontal.

A.3.4. StatefulSets

En un servicio expuesto, es común enfrentar la necesidad de gestionar grupos de *Pods* que requieren identificación única y consistente, especialmente cuando se producen cambios en el número de réplicas debido a ajustes en la configuración o escalado dinámico. Para garantizar que estos *Pods* mantengan identificadores persistentes a lo largo de sus ciclos de vida y entre diferentes eventos de programación, se deben emplear mecanismos especializados.

En este contexto, los *StatefulSets* son cruciales. A diferencia de los *Deployments*, que están diseñados para aplicaciones sin estado (*stateless*) y no garantizan la persistencia de los identificadores de los *Pods* durante los cambios, los *StatefulSets* proporcionan un control más robusto para aplicaciones con estado (*stateful*). Esto asegura que cada *Pod* mantenga un nombre único y persistente, así como almacenamiento persistente, crucial para aplicaciones que requieren consistencia en sus identificadores y datos a lo largo del tiempo [43].

A.3.5. Service

Un Servicio (Service) es un recurso que proporciona una forma de exponer una aplicación en ejecución como una red accesible dentro del clúster o, incluso, externamente. Los Servicios permiten que los conjuntos de pods internos se comuniquen entre sí y que los clientes externos accedan a aplicaciones dentro del clúster. [44]

Para comprender el concepto de forma más intuitiva podemos establecer un caso de uso. Supongamos que, para ejecutar una aplicación, usamos un Deployment. Este puede crear y destruir Pods dinámicamente, por lo que, de un momento a otro, no podemos conocer cuántos de esos Pods están funcionando (incluso podemos desconocer su nombre).

Esto se debe a que, los Pods, se crean y destruyen para coincidir con el estado deseado, es decir, son recursos efímeros. Esto lleva a un problema: si un conjunto de Pods proporciona funcionalidad a otros dentro de tu clúster, necesitamos algún mecanismo que permita la comunicación entre ambos grupos de forma transparente y subsanando esa efimeridad. Es ahí cuando interviene el concepto de Servicio.

A.3.6. ConfigMap y Secret

En Kubernetes, un ConfigMap y un Secret son dos tipos de objetos utilizados para almacenar datos de configuración que se pueden usar en los Pods. Ambos objetos ayudan a gestionar la configuración de las aplicaciones de manera centralizada y desacoplada, pero se usan en contextos diferentes, dependiendo de si los datos son sensibles o no.

Por una parte, un ConfigMap se utiliza para almacenar datos de configuración no confidenciales en pares clave-valor. Permiten desacoplar la configuración de un entorno específico de una imagen de contenedor, permitiendo facilitar la portabilidad de las aplicaciones. El problema que presentan estos objetos es la ausencia de encriptación. Si se necesita almacenar objetos que contienen información sensible necesitamos emplear otro tipo de objeto (anteriormente mencionado) conocido como Secret [45].

Otra opción, como se ha mencionado en el párrafo anterior, es el uso de Secrets. Un Secret se utiliza para almacenar datos sensibles, como contraseñas, tokens de acceso, y claves SSH. Similar a un ConfigMap, un Secret almacena datos en pares clave-valor, pero estos datos quedan codificados para mayor seguridad.[46]

A.4 Otros objetos

A.4.1. Namespaces

Los *namespaces* en Kubernetes son una forma de segmentar lógicamente el clúster. Estos proporcionan un mecanismo para dividir los recursos en múltiples entornos aislados. Cada *namespace* puede contener sus propios pods, servicios, volúmenes, y otros recursos, como se puede observar en la figura A.6. Esto es especialmente útil en escenarios donde se quiere separar entornos de desarrollo, pruebas y producción, o cuando se desea delegar la gestión de recursos a diferentes equipos dentro de una misma organización. Además, los *namespaces* facilitan la aplicación de políticas de seguridad y límites de recursos de manera específica, al separar lógicamente las aplicaciones dentro del mismo clúster [47].

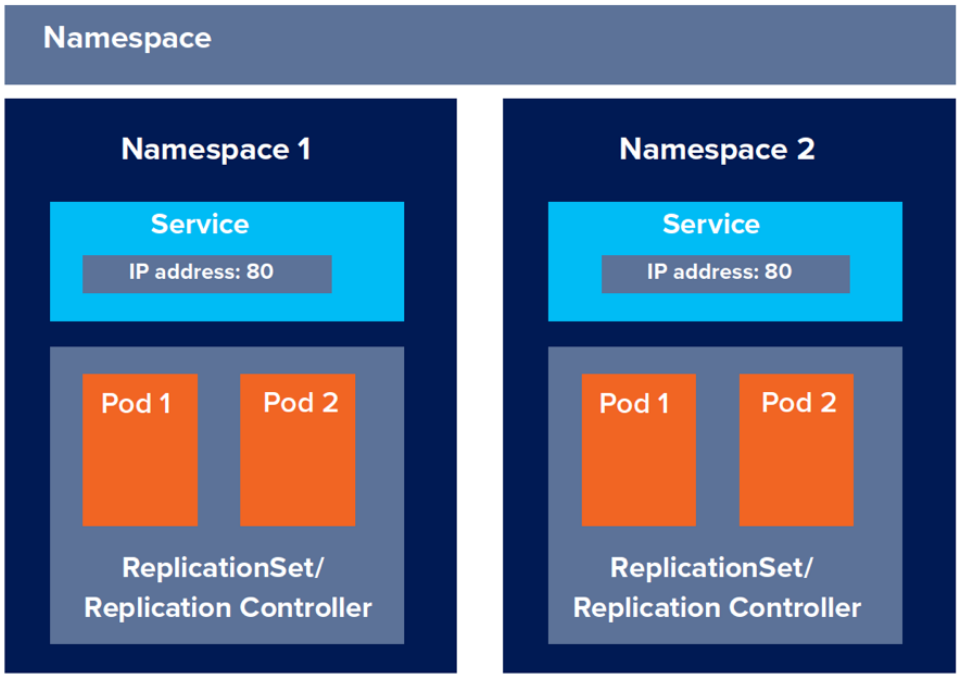

Figura A.6: Namespaces en Kubernetes

A.4.2. Volumes y Persistent Volumes

En Kubernetes, los *volumes* y *persistent volumes* son componentes esenciales para la gestión del almacenamiento persistente. Los *volumes* [48] se asocian a un pod y proporcionan un espacio de almacenamiento temporal que existe mientras el pod está en ejecución. Por otro lado, los *persistent volumes* (PV) [49] son recursos de almacenamiento que existen de manera independiente de los pods, permitiendo que los datos persistan más allá del ciclo de vida de los mismos. Los PV son generados por los administradores del clúster y se pueden reclamar mediante *Persistent Volume Claims* (PVC) por los usuarios de Kubernetes.

Este sistema permite la separación de las preocupaciones de almacenamiento entre los administradores del clúster y los desarrolladores de aplicaciones, como podemos observar en la figura A.7.

A.4.3. Ingress

El *ingress* [50] en Kubernetes es un objeto que gestiona el acceso externo a los servicios dentro del clúster. Proporciona reglas para enrutamiento de tráfico HTTP y HTTPS, permitiendo la configuración de balanceo de carga, terminación de SSL, y enrutamiento basado en nombre de host o ruta. Un *ingress controller* es el componente responsable de implementar estas reglas en el entorno del clúster. Esto facilita la exposición de aplicaciones a usuarios externos de manera segura y eficiente, reduciendo la necesidad de configuraciones individuales de servicios de distinto tipo (por ejemplo *NodePort* o *LoadBalancer* [44]) para cada aplicación.

Para comprender mejor el funcionamiento del *Ingress*, podemos establecer un ejemplo sencillo donde un Ingress envía todo el tráfico a un único Servicio, como se puede ver reflejado en la figura A.8, obtenida de la documentación oficial [50].

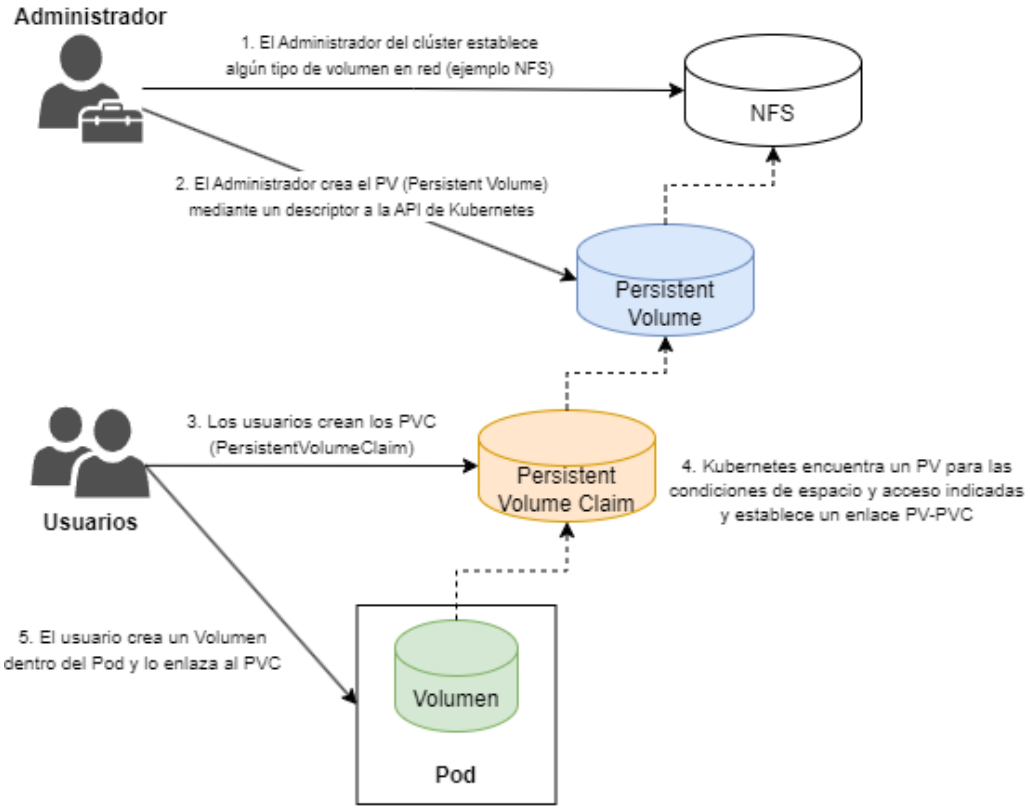

Figura A.7: Proceso de vinculación de volúmenes mediante el sistema PV-PVC, por DEV Community, 2021

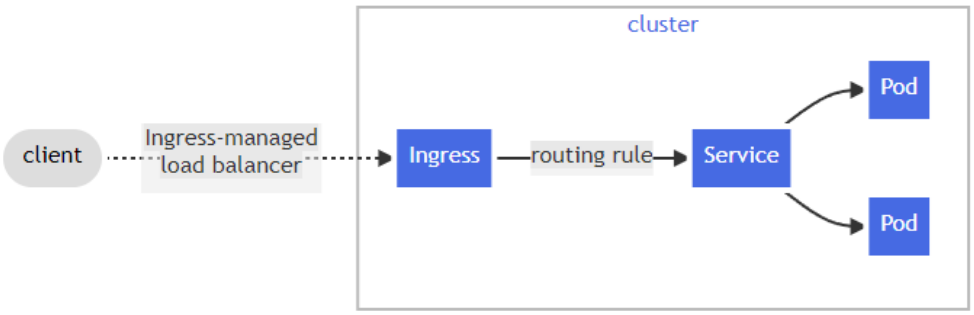

Figura A.8: Ejemplo de funcionamiento del Ingress

APÉNDICE B
Objetivos de Desarrollo Sostenible

B.1 Grado de relación con los ODS

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**		X		
ODS 12. **Producción y consumo responsables.**			X	
ODS 13. **Acción por el clima.**			X	
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

B.2 Reflexión sobre la relación del trabajo con los *ODS*

La implementación de un servicio de ejecución de código altamente disponible basado en contenedores tiene un impacto significativo en el desarrollo sostenible y se alinea con varios Objetivos de Desarrollo Sostenible (*ODS*) de las Naciones Unidas. Al analizar detenidamente las ventajas de este proyecto, se pueden identificar conexiones claras con los objetivos de **industria, innovación e infraestructura** (*ODS* 9), **ciudades y comunidades sostenibles** (*ODS* 11), **producción y consumo responsables** (*ODS* 12), y **acción por el clima** (*ODS* 13).

El *ODS* 9 promueve la construcción de infraestructuras resilientes, la industrialización inclusiva y sostenible, y el fomento de la innovación. Este proyecto se alinea direc-

tamente con este objetivo al enfocarse en el desarrollo de una infraestructura tecnológica avanzada que mejora la resiliencia y escalabilidad de los servicios en la nube. Además, el uso de tecnologías de orquestación y virtualización fomenta la innovación en el sector tecnológico, lo cual es esencial para el avance de la industria 4.0.

Por otra parte, aunque el *ODS* 11 se centra en la sostenibilidad de las ciudades, la infraestructura tecnológica desempeña un papel crucial en la creación de ciudades inteligentes. Un servicio de ejecución de código en la nube que sea altamente disponible y escalable puede ser fundamental para las soluciones tecnológicas utilizadas en ciudades inteligentes, facilitando servicios eficientes y sostenibles. Esto establece una relación moderada con este *ODS*, especialmente al considerar la importancia de la infraestructura tecnológica en la gestión eficiente de los servicios urbanos.

Adicionalmente, el *ODS* 12 aborda la necesidad de garantizar patrones sostenibles de producción y consumo. La optimización de recursos en la nube, que es un aspecto clave de este proyecto, minimizará el uso innecesario de recursos computacionales. En la Industria 4.0 pasará a tener un valor importante el soporte informático de algunos procesos productivos. Por ello, nuestros resultados tendrán una repercusión indirecta (y, por tanto, baja), pero positiva, sobre los procesos productivos responsables.

Si nos referimos al *ODS* 13, centrado en la acción por el clima, observamos que también está relacionado con este proyecto. La escalabilidad y eficiencia energética en el uso de servicios en la nube pueden contribuir a reducir las emisiones de carbono asociadas al funcionamiento de centros de datos. Al implementar servicios de orquestación más eficientes y con una arquitectura optimizada, este proyecto podría contribuir indirectamente a la mitigación del cambio climático mediante un uso más sostenible de los recursos computacionales. Aunque la conexión con este *ODS* es más indirecta, relacionada principalmente con la eficiencia energética y la reducción de emisiones, su relevancia en el contexto del proyecto es evidente.

En conclusión, la implementación de un servicio de ejecución de código altamente disponible no solo mejora la infraestructura tecnológica y fomenta la innovación, sino que también contribuye, parcialmente, a la sostenibilidad urbana, la producción y consumo responsables, y la acción por el clima. Estos impactos positivos demuestran cómo el proyecto se alinea con varios *ODS*, destacando su potencial para avanzar en el desarrollo sostenible.