Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOGÍAS DE LA INFORMACIÓN

Despliegue y configuración automatizada de un clúster kubernetes con alta disponibilidad

Estudante: Iago Domínguez Cameán

Dirección: Roberto Rey Expósito

A Coruña, junio de 2024.

Agradecimientos

A mi familia y amigos, especialmente a mis padres y mi hermano que me han apoyado durante todos estos años. A mi tutor, Roberto Rey Expósito, por darme la oportunidad de hacer un proyecto de acuerdo a mis gustos.

Resumen

Este Trabajo de Final de Grado (TFG) tiene como principal objetivo el despliegue de un clúster **Kubernetes** configurado en alta disponibilidad (High Availability, HA) con la ayuda de herramientas auxiliares como por ejemplo **Ansible**, **Keepalived** y **HAProxy**, entre otras. El propósito es que, tanto el despliegue como la configuración de **Kubernetes**, se hagan de manera centralizada y con la menor intervención manual posible, es decir, buscando la automatización de todos los procesos en la medida de lo posible. En cuanto a la arquitectura clúster que se desplegará inicialmente constará de cinco nodos, de los cuales tres de ellos tendrán el rol de *control plane* de **Kubernetes**, y serán los encargados de proveer un punto de acceso al clúster y tomar las decisiones en función de los eventos que ocurren dentro del mismo. De estos tres nodos *control plane*, uno de ellos será el encargado de inicializar el clúster **Kubernetes** y al cual se unirán el resto de nodos. Los dos nodos restantes jugarán el rol de *worker*, y serán los encargados de ejecutar las aplicaciones y servicios por parte de los usuarios finales. También se dispondrá de un servidor *Network File System* (NFS) para proveer el almacenamiento compartido de las aplicaciones que vayamos a desplegar. En nuestro caso, este servidor NFS será proporcionado por uno de los tres nodos con el rol de *control plane*. Además, se configurarán sistemas de autoescalado horizontal/vertical de los recursos del clúster, junto con su monitorización y el envío de notificaciones ante eventos importantes.

El despliegue y configuración del *clúster* se llevará a cabo siguiendo el paradigma Infrastructure as Code (IaC), el cual nos permite administrar infraestructuras y sistemas a partir de código fuente. De esta forma podemos, de forma centralizada y automatizada, administrar equipos, redes, bases de datos, almacenamiento y otros tipos de recursos, además de su aprovisionamiento. Para este propósito usaremos la herramienta **Ansible**, mencionada anteriormente.

Por último, debido al hardware requerido para realizar este proyecto, se llevará a cabo mediante máquinas virtuales haciendo uso de **VirtualBox** y **Vagrant**. En cualquier caso, todos los *playbooks* de **Ansible** que se desarrollarán en este proyecto serán perfectamente válidos en caso de que se quiera desplegar un clúster **Kubernetes** directamente sobre máquinas físicas.

Abstract

The main objective of this project consists of the deployment of a **Kubernetes** cluster configured in high availability (HA), with the help of auxiliary tools such as **Ansible**, **Keepalived**

and **HAProxy**, among others. The goal is that both the deployment and configuration of **Kubernetes** are performed in a centralized manner and with the least amount of manual intervention possible, that is, aiming to automate all processes as much as possible. Regarding the clúster architecture that will be deployed, it will consist of five nodes, three of them playing the role of *control plane* in **Kubernetes**. These *control plane* nodes are in charge of providing an access point to the cluster and make decisions based on the events that occur within it. One of these three *control plane* nodes will be responsible for initializing the cluster and to which the rest of the nodes will later join. The two remaining cluster nodes will play the role of *worker*, being in charge of executing the applications and services by the end users. There will also be a *Network File System* (NFS) server to provide shared storage for the applications. In our case, this NFS server will be hosted in one of the three *control plane* nodes. In addition, horizontal/vertical auto-scaling systems for the cluster resources will be configured, along with monitoring them and sending notification of important events.

The deployment and configuration of the cluster will be carried out using the Infrastructure as Code (IaC) paradigm, which allows us to manage systems and infrastructures from source code. This approach enable managing computers, networks, databases, storage and other type of resources, as well as their provisioning, in a centralized and automated manner. For this purpose we will use the **Ansible** tool, as mentioned above.

Finally, due to the hardware required to carry out this project, we will use Virtual Machines (VMs) managed by **VirtualBox** and **Vagrant**. However, all the **Ansible** *playbooks* developed in this project will remain usable in case of deploying a **Kubernetes** cluster directly on physical machines.

Palabras clave:

- Kubernetes
- Infraestructura como código
- Alta disponibilidad
- Ansible
- Vagrant
- Packer

Keywords:

- Kubernetes
- Infrastructure as Code
- High availability
- Ansible
- Vagrant
- Packer

Índice general

Índice de figuras

Listings

Capítulo 1
Introducción

En este primer capítulo de la memoria se pone en contexto el presente Trabajo de Final de Grado (TFG), explicando brevemente los principales motivos por los que se ha realizado este trabajo en concreto, los objetivos que se pretenden alcanzar y describiendo la estructura en capítulos de esta memoria.

1.1 Motivación

En el mundo de la informática es imprescindible disponer de algún tipo de infraestructura hardware, ya sea en local o en la nube, para proveer de aplicaciones y servicios a los clientes y/o trabajadores de las empresas y organizaciones, tanto públicas como privadas. Estas infraestructuras suelen estar compuestas por múltiples servidores interconectados para ofrecer mejoras en rendimiento, eficiencia y/o en disponibilidad de los servicios que proporcionan, y esta composición forma lo que llamamos un clúster. En este TFG desplegaremos y configuraremos un clúster de forma automatizada, haciendo uso principalmente de la plataforma de orquestación **Kubernetes** [1].

Kubernetes, el cual constituye el núcleo principal de este TFG, es el orquestador de contenedores más usado en el mundo. Esta herramienta nos permite la creación del clúster en el que es posible ejecutar aplicaciones contenerizadas de forma declarativa. Una característica de **Kubernetes** es que el clúster es modular, es decir, podemos empezar con las funcionalidades básicas e ir añadiendo nuevas según las vayamos necesitando. La razón de usar un orquestador como **Kubernetes** para la ejecución de aplicaciones es simple, nos ahorra tiempo automatizando tareas como reiniciar contenedores que han dejado de funcionar, instanciar nuevos recursos en función de la demanda y de las políticas que se definan (autoescalado), dirigir el tráfico de red al destino correspondiente dentro del clúster, balancear dicho tráfico entre múltiples contenedores, etc. En resumen, nos evita realizar tareas manuales que pueden ser engorrosas.

El principal motivo por el que se va a usar **Kubernetes** como orquestador en este trabajo se debe a que actualmente es la herramienta de este tipo más utilizada con mucha diferencia [2]. Además, está desarrollado por Google bajo una licencia de código abierto, tiene una enorme comunidad a su alrededor y ofrece una amplia variedad de funcionalidades a demanda del usuario. Cabe mencionar que existen otras plataformas de orquestación de contenedores [3, 4], como por ejemplo Docker Swarm, Apache Mesos u Openshift, siendo los principales competidores de **Kubernetes**, aunque cuentan con una menor cuota de mercado.

La tendencia actual en cuanto al despliegue de un clúster **Kubernetes** consiste en su despliegue en la nube, ya que ofrece ventajas como escalabilidad, bajo coste, facilidad de gestión y mantenimiento. Algunos servicios en la nube que se encargan de gestionar **Kubernetes** son Elastic Kubernetes Service (EKS) y Azure Kubernetes Service (AKS), proporcionados por los proveedores cloud Amazon Web Service (AWS) y Microsoft Azure, respectivamente. Por otro lado, un despliegue en infraestructura local conlleva una instalación y configuración mucho más costosa en términos de recursos humanos, y con un coste monetario que también suele ser mayor. Además, su escalabilidad es más limitada ya que en la nube solo se tienen que solicitar más recursos al proveedor mientras que en local se deberá adquirir más hardware si fuese necesario. Sin embargo, hay muchas empresas que por diferentes motivos no quieren o no pueden pasarse el modelo *cloud*, por lo que resulta interesante simplificar el despliegue de **Kubernetes** en infraestructura local.

En este trabajo lo que se persigue es facilitar el despliegue desde cero y la gestión inicial de un clúster **Kubernetes** en local configurado en alta disponibilidad (High Availability, HA) y usando el paradigma Infrastructure as Code (IaC), el cual nos permite tratar la infraestructura o servicios como si fueran código, favoreciendo su consistencia y reproducibilidad. Además, se hará extensivo uso de herramientas IaC para automatizar todo el proceso tanto como sea posible.

Por último, en el grado se estudian diferentes ámbitos de la informática gracias a la variedad de materias de las que dispone. Junto con todo lo mencionado anteriormente, se pretende poner en práctica el conocimiento adquirido en algunas de las asignaturas cursadas, principalmente aquellas relacionadas con la administración de sistemas, concretamente las cursadas en la rama de Tecnologías de la Información (TI).

1.2 Objetivos

Este trabajo propone los siguientes objetivos específicos:

- Desarrollo de un proyecto **Vagrant** que permita el despliegue y la configuración de un clúster virtual de manera automatizada.

- Desarrollo de playbooks de **Ansible** para el aprovisionamiento del software necesario para el despliegue y configuración de un clúster **Kubernetes**, así como el despliegue de las aplicaciones que se vayan a ejecutar.
- Envío de notificaciones relacionadas con eventos que ocurren en el clúster usando un bot de **Telegram**.
- Creación de un box de **Vagrant** personalizado usando la herramienta **Packer**. En esta imagen se instalarán los binarios necesarios para el despliegue del clúster (p.ej. **Kubernetes**, Containerd).

1.3 Estructura de la memoria

La estructura del presente documento se define de la siguiente manera:

1. **Introducción**: en este capítulo pondremos en contexto el TFG, explicando la motivación para llevarlo a cabo y los objetivos que se pretenden alcanzar.
2. **Tecnologías y herramientas**: descripción y explicación de las herramientas seleccionadas para el desarrollo del trabajo.
3. **Metodología**: se expone la metodología escogida, **Iterativa e Incremental**, y se explicarán sus fases y el motivo por el cual ha sido seleccionada.
4. **Desarrollo del clúster**: este capítulo detalla todo el trabajo de desarrollo llevado a cabo, en concreto se describe la implementación de cada *playbook* de Ansible. Además, también se comenta aquella configuración que no se ha conseguido automatizar.
5. **Conclusiones**: se expondrán las conclusiones obtenidas al finalizar el trabajo, así como identificar posibles mejoras.

Capítulo 2

Tecnologías y herramientas

En este segundo capítulo se detallan la principales herramientas y tecnologías que se han usado durante el desarrollo del TFG. Se han agrupado en varias secciones: tecnologías de virtualización (sección 2.1), infraestructura como código (sección 2.2), **Kubernetes** (sección 2.3), alta disponibilidad (sección 2.4) y software de monitorización y notificación (sección 2.5).

2.1 Tecnologías de virtualización

En esta sección se llevará a cabo un pequeño análisis sobre las tecnologías y soluciones software para la virtualización utilizadas en el desarrollo de este proyecto.

2.1.1 VirtualBox

VirtualBox [5] es un software de virtualización de código abierto y multiplataforma muy popular, desarrollado por Oracle. Permite la creación y gestión de máquinas virtuales (Virtual Machine, VM) en una misma máquina física (el anfitrión), las cuales ejecutan SOs completos que pueden ser diferentes. Proporciona la posibilidad de personalizar los recursos de cada VM como la cantidad de discos virtuales, el espacio que ocupa/dispone cada uno de ellos, el número de CPUs, la configuración de red o la cantidad de memoria. Cabe destacar que su popularidad se debe a lo sencillo que es de usar y que su interfaz de usuario resulta bastante intuitiva.

Además de VirtualBox existen otras soluciones que virtualizan SOs completos como puede ser VMware Player.

2.1.2 Contenedores

Los contenedores [6, 7, 8] son entornos virtuales de ejecución que permiten encapsular el código de una aplicación junto con todas sus dependencias y librerías, permitiendo que puedan desplegarse en cualquier entorno, ya sea en la nube o en infraestructura local. La

principal diferencia con las VMs es que los contenedores se ejecutan sobre el kernel de la máquina física por lo que no ejecutan un SO completo, solo necesitan de un entorno en donde puedan ser ejecutados. Existen varias soluciones que hacen uso de esta tecnología como por ejemplo **Docker** o **LXC**.

2.2 Infraestructura como código

La infraestructura como código [9], o Infrastructure as Code (IaC) en inglés, consiste en gestionar y aprovisionar infraestructura TI mediante código fuente y herramientas de automatización, en vez de realizar el proceso manualmente o de forma interactiva. De esta forma creamos ficheros que de forma declarativa, automática y reproducible, generan la infraestructura deseada y también nos permite llevar un registro de cambios y versiones de dicha infraestructura de manera más sencilla.

2.2.1 Vagrant

Es una herramienta de código abierto utilizada para crear y configurar entornos de desarrollo virtualizados de forma automatizada [10], siguiendo el paradigma IaC. Nos ofrece la posibilidad de crear y configurar VMs que son fácilmente reproducibles. Gracias a esta herramienta se facilita la configuración de entornos consistentes y listos para usar.

Su característica principal es que nos permite definir la configuración de una VM o un conjunto de estas en un fichero, el llamado *Vagrantfile* [11], del cual se muestra un ejemplo en el listado 2.1. Este fichero debe estar escrito en lenguaje Ruby, y contiene todos los detalles de configuración de las VMs que vayamos a desplegar, como su SO, la configuración de red (véase línea 16), el número de CPUs (línea 26), la cantidad de memoria y discos, o incluso la instalación de software adicional en la máquina virtual.

```
1  # -*- mode: ruby -*-
2  # vi: set ft=ruby :
3  require_relative 'provisioning/vbox.rb'
4  VBoxUtils.check_version('7.0.6')
5  Vagrant.require_version ">= 2.3.4"
6
7  Vagrant.configure("2") do |config|
8      config.vm.box = "ubuntu/focal64"
9      config.vm.hostname = "ubuntu"
10     config.vm.network "forwarded_port", guest: 80, host: 8080
11
12     # Configure hostmanager and vbguest plugins
13     config.hostmanager.enabled = true
14     config.hostmanager.manage_host = true
15     config.hostmanager.manage_guest = true
```

```
16      config.vbguest.auto_update = false
17
18      config.vm.provider "virtualbox" do |vb|
19          vb.name = "Prueba-#{config.vm.hostname}"
20          vb.cpus = 2
21          vb.memory = 2048
22          vb.gui = false
23      end
24  end
```

Listing 2.1: Ejemplo de un *Vagrantfile*

2.2.2 Packer

Se trata de otra herramienta IaC [12] de los mismos desarrolladores que **Vagrant**. Sirve para crear lo que en **Vagrant** se conoce como *box*. Estos elementos son imágenes máquina las cuales contienen SOs previamente configurados con el software adicional que se requiera y que permiten instanciar rápidamente nuevas máquinas virtuales. Para crear una *box* con el software deseado, necesitamos una plantilla de un *Vagrantfile*, una plantilla de **Packer** en lenguaje JavaScript Object Notation (JSON) o HashiCorp Configuration Language (HCL) donde especificamos como queremos que se cree la *box* y, por ejemplo, un script en bash en el que indiquemos el software adicional que queremos instalar. Este *script* también se especifica dentro de la plantilla de **Packer**. El aprovisionamiento de la imagen también podría realizarse mediante un *playbook* de Ansible, entre otras opciones. Cabe destacar que Packer permite crear imágenes máquina para distintos entornos virtuales, soportando también plataformas en la nube.

2.2.3 Ansible

Ansible [13] es otra herramienta IaC de código abierto, en este caso desarrollada y soportada por Red Hat, la cual nos ayuda a agilizar y simplificar la gestión de la configuración de los sistemas, el despliegue de aplicaciones y la administración de la infraestructura. Para cumplir este objetivo, Ansible utiliza unos ficheros de código escritos en lenguaje Yet Another Markup Language (YAML) denominados *playbooks*, los cuales contienen la descripción de las tareas para gestionar la configuración y las operaciones a realizar en la infraestructura.

En el listado 2.2 se muestra un ejemplo de como sería un *playbook* de Ansible, donde *hosts* indica los servidores en los que se ejecuta el *playbook*, y *tasks* indica donde comienza la definición de las tareas que se van a realizar. En este caso, se define una única tarea que copia el fichero */etc/hosts* en otra ruta a modo de respaldo, haciendo uso de un módulo de Ansible llamado *ansible.builtin.copy*.

```
- name: Intro to Ansible Playbooks
  hosts: all

  tasks:
  - name: Copy file hosts with permissions
     ansible.builtin.copy:
        src: ./hosts
        dest: /tmp/hosts_backup
        mode: '0644'
```

Listing 2.2: Ejemplo de un *playbook* de Ansible

Adicionalmente, Ansible utiliza un inventario proporcionado por el administrador. Este inventario es un fichero donde se indican los servidores o *hosts* con los que debe interactuar. Estos *hosts* se pueden agrupar para realizar tareas determinadas sobre ese grupo, e incluso es posible agrupar grupos. Los *hosts* pueden ser indicados con su dirección IP o por su *hostname*.

Ansible utiliza una arquitectura cliente-servidor "sin agentes" o agentless. Esto quiere decir que no es necesario la instalación de ningún software (agente) específico de Ansible en los clientes que se quieren gestionar con esta herramienta. De esta forma, desde una única máquina, denominada como el nodo controlador, podemos gestionar la configuración del resto usando simplemente el protocolo SSH. Para mayor comodidad, es muy habitual realizar las configuraciones apropiadas que le permitan al nodo controlador poder conectarse por SSH sin necesidad de contraseña a cada uno de los nodos cliente que queremos gestionar con Ansible. En nuestro contexto, esto lo podemos hacer al arrancar las máquinas virtuales con **Vagrant**, ya que es posible la ejecución automática de scripts una vez hayan arrancado por completo. En la figura 2.1 muestra un esquema de este tipo de arquitectura agentless.

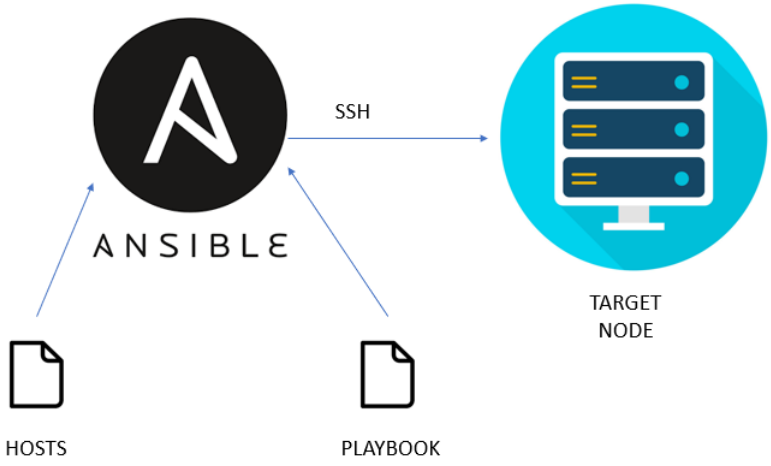


Figura 2.1: Arquitectura de Ansible

2.3 Kubernetes

Kubernetes es una herramienta de código abierto desarrollada por Google y diseñada para automatizar el despliegue, escalado y administración de aplicaciones ejecutadas en contenedores. Es el orquestador de contenedores más popular y utilizado en el mundo gracias a su eficiencia y fiabilidad. Funciona básicamente agrupando los contenedores que conforman una aplicación en unidades lógicas que son fáciles de administrar. De esta manera obtenemos una gestión más eficiente de los recursos computacionales del clúster, facilitando la portabilidad y escalabilidad de las aplicaciones. Además, proporciona características para la gestión de redes, el balanceo de carga y la auto-recuperación, entre muchas otras. Como se mencionó en el capítulo 1, **Kubernetes** puede desplegarse en la nube pública utilizando servicios gestionados como EKS y AKS, proporcionados por proveedores cloud, y también en local si se dispone de la infraestructura hardware necesaria para ello (despliegue *on-premises*), siendo este último el que nos interesa en este trabajo.

2.3.1 Arquitectura

Un clúster de **Kubernetes** está compuesto por múltiples servidores (nodos), los cuales pueden ser tanto máquinas virtuales como físicas. Estos nodos pueden formar parte del *control plane* o ser un *worker*. Cada nodo *worker* es gestionado por el *control plane* y dispone de

los servicios necesarios para poder ejecutar *pods* dentro de cada nodo. Un *pod* es la unidad de computación desplegable más pequeña que se puede crear y gestionar en **Kubernetes**, formada por un conjunto de uno o más contenedores agrupados.

Control Plane

El *control plane* es el componente de **Kubernetes** que se encarga de gestionar el correcto funcionamiento del clúster y de tomar decisiones en función de los eventos que ocurran dentro del mismo. Está compuesto por varios subcomponentes, que pueden ejecutarse en un solo nodo principal, o bien ser replicados en varios nodos con el rol de *control plane* con el fin de prorporciona alta disponibilidad.

Kubernetes proporciona una API REST que soporta principalmente operaciones Create, Read, Update and Delete (CRUD) para objetos, en su mayoría, persistentes. Desde esta API podemos orquestar el despliegue de objetos como *pods*, servicios, *ingress*, *replicaset* y *deployments*, entre otros. Tanto los usuarios del clúster como estos objetos interactúan con esta API, la cual sirve como punto de coordinación. La mayoría de los objetos contienen metadatos, y en estos se incluye información como etiquetas y anotaciones, su estado deseado o especificaciones del recurso.

Los componentes del *control plane* son:

- **API server**: Proporciona la API de **Kubernetes**. Está pensado para ser un servidor simple, con la mayoría de su lógica de negocio separada en plug-ins. Principalmente procesa y valida las operaciones REST y actualiza los objetos correspondientes en la base de datos (**etcd**). Adicionalmente, actúa como punto de entrada al clúster, por lo que debe ser accesible desde fuera del mismo.

- **etcd**: Todo estado persistente del clúster se almacena en esta base de datos. Proporciona una forma de guardar la configuración del clúster, el estado de los nodos, los metadatos de los *pods* y otros recursos.

- **Controller-manager server**: Consiste en varios controladores que observan el estado actual del clúster a través del API server. Su función principal es mantener el estado deseado del clúster.

- **Scheduler**: Este componente se encarga de asignar a cada *pod* un nodo del clúster de forma automática y en función de los recursos disponibles, requisitos de calidad de servicio, su afinidad con las especificaciones y otras restricciones.

Nodo

Como se mencionó anteriormente, un nodo contiene los servicios necesarios para poder ejecutar *pods* dentro del mismo y está controlado por el *control plane*. Existen dos tipos de nodos: aquellos que pertenecen al *control plane* y aquellos que no lo hacen, denominados nodos *worker*. Estos últimos son los que se encargan de ejecutar las aplicaciones y servicios de los usuarios del clúster. Los servicios que son esenciales ejecutar en un nodo *worker* son los siguientes:

- **Kubelet**: Es el controlador más importante en **Kubernetes**, responsable de que los *pods* que se están ejecutando en el nodo funcionen correctamente. Sirve como agente que interactúa con el *control plane* y se encarga principalmente de gestionar los contenedores (iniciar, detener, etc), supervisar el estado del nodo monitorizando continuamente su salud y reportando cualquier problema al *control plane*, además de intentar solucionarlos siempre que sea posible. También es el encargado de comunicarse con el API server, del montaje de volúmenes de almacenamiento y de la gestión de los recursos en el nodo (CPU, memoria, etc). Se encarga además de gestionar los *pods* estáticos, que a diferencia de los *pods* normales estos son gestionados directamente por kubelet sin que intervenga el API server.

- ***Container runtime***: Su función principal es ejecutar imágenes de contenedores. Es el responsable de llevar a cabo las tareas necesarias para ejecutar un contenedor en un entorno virtual aislado, como la creación del espacio de nombres (namespace), la configuración de las interfaces de red, la gestión de procesos del contenedor, entre otras cosas. Cabe destacar que **Kubernetes** no está directamente vinculado a un *container runtime* específico. **Kubernetes** define una interfaz denominada Container Runtime Interface (CRI), gracias a la cual obtenemos la capacidad de poder intercambiar entre diferentes *container runtimes*. En nuestro caso hemos optado por **Containerd**, el cual es el motor de ejecución de Docker. Cuando kubelet recibe una petición para gestionar un nuevo *pod*, hará uso del *container runtime* configurado en **Kubernetes**.

- **Kube proxy**: La abstracción de los servicios proporciona una forma de agrupar *pods* bajo políticas de acceso comunes. Esta implementación crea una IP virtual a la cual los clientes pueden acceder y es redirigida transparentemente a los *pods* gestionados por un servicio. Cada nodo ejecuta kube-proxy, el cual programa reglas usando iptables que redirigen las peticiones de red al backend correcto.

Topología

Como vamos a crear un clúster configurado en alta disponibilidad, debemos tener en cuenta la topología de la base de datos etcd. Tenemos dos opciones: **stacked etcd**, mostrado en la figura 2.2, o **external etcd**, que se muestra en la figura 2.3.

Con la topología **stacked etcd**, el componente etcd se ejecuta en los mismos nodos que componen el *control plane*. Cada nodo del *control plane* ejecutaría una instancia de API server, Scheduler y Controlle-manager. En este caso, el API server se expone a los nodos *worker* a través de un balanceador de carga. Cada nodo del *control plane* crea localmente un componente etcd, el cual solo se comunica con el API server de dicho nodo. De igual forma ocurre para las instancias de Scheduler y Controller-manager. Con esta topología se reducen el número de nodos que se necesitan en total, lo cual hace que el clúster sea más sencillo de configurar y de replicar. Con la desventaja de que si un nodo se cae por algún motivo, también lo harán tanto etcd como el *control plane*, por lo que la redundancia queda comprometida y no tendríamos alta disponibilidad. Esto lo podemos solucionar añadiendo más nodos al *control plane*. Cabe destacar que esta es la topología por defecto al iniciar un clúster con kubeadm.

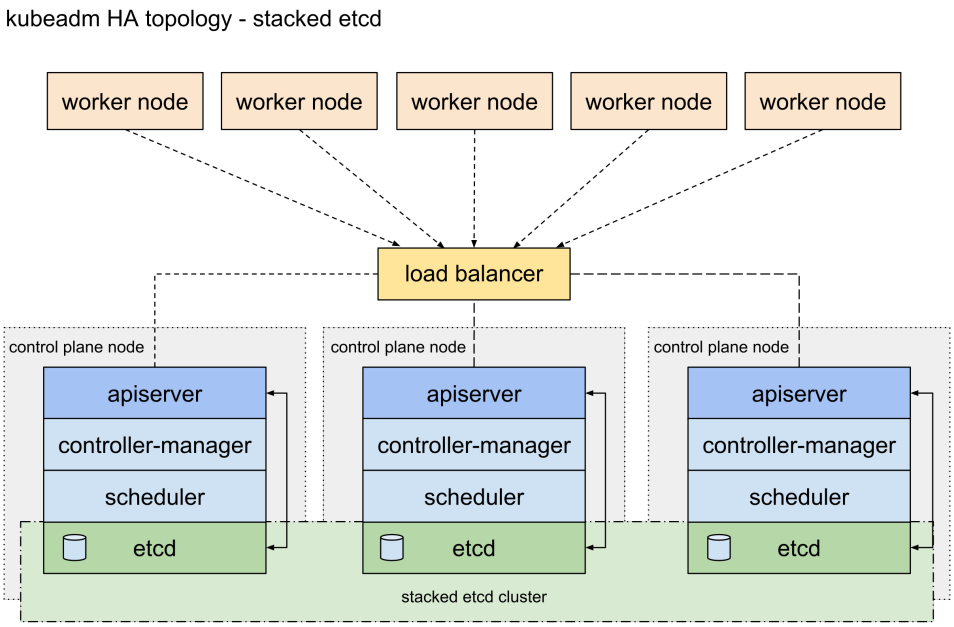


Figura 2.2: Topología stacked etcd

En la topología **external etcd** se necesita de un segundo clúster donde se alojan las instancias del componente etcd. En general funciona de la misma manera que la topología **stacked**,

la mayor diferencia es que los componentes etcd se ejecutan en nodos distintos a los del *control plane*, y que cada nodo etcd se comunica con la instancia de API server de cada nodo del *control plane*. Esta topología desacopla el *control plane* y etcd, proporcionando una configuración de alta disponibilidad en la que perder un nodo del *control plane* o uno de etcd tiene menos impacto y no afecta tanto a la redundancia del clúster como la topología **stacked**. Sin embargo, esta topología requiere el doble de nodos que la anterior, sin contar los nodos *worker*, es decir, un mínimo de tres nodos para el *control plane* y otros tres para etcd.

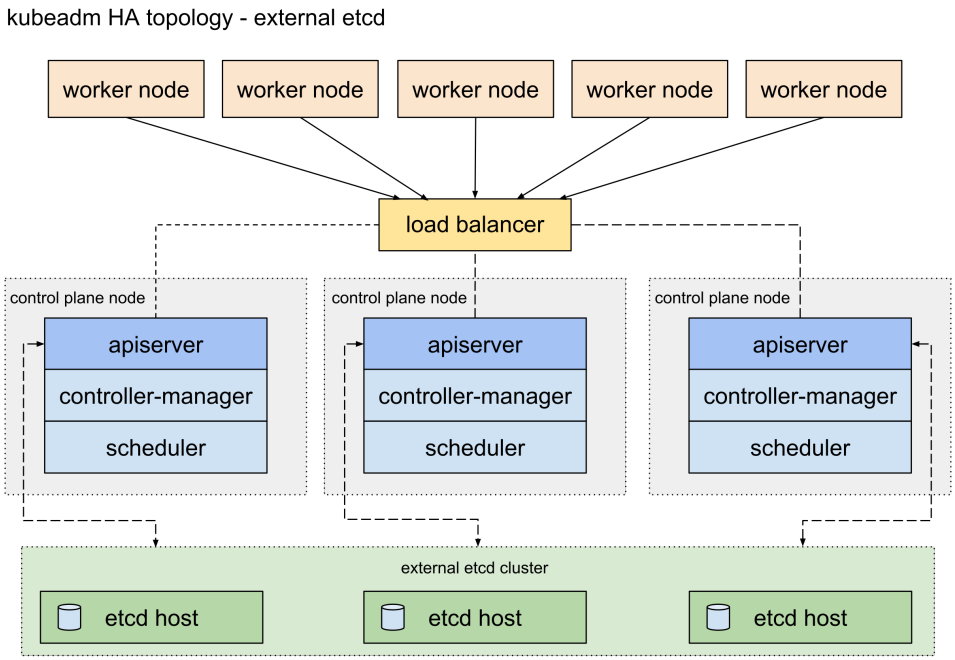


Figura 2.3: Topología external etcd

Para el desarrollo de este proyecto se ha optado por la topología **stacked**, ya que es la más sencilla de desplegar y configurar, además de que requiere del menor número de nodos a la vez que mantiene una configuración en alta disponibilidad.

2.3.2 *Pods*

Como se mencionó previamente, son la unidad de cómputo más pequeña que se puede desplegar en un clúster de **Kubernetes**. Más específicamente, un *pod* es un grupo de uno o más contenedores con almacenamiento y recursos compartidos, además de una especificación común de como deben ejecutarse. Esta especificación se describe en un fichero, típicamente en formato YAML, en el cual se indican como debe configurarse y ejecutarse el *pod* (nombre,

contenedores, imágenes, puertos, etc). Podemos ver un ejemplo de la especificación de un *pod* en el listado 2.3, donde se indica el contenedor que se va a crear y la imagen que se utiliza para ello, entre otras cosas como los metadatos.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Listing 2.3: Ejemplo de especificación de un *pod*

El entorno compartido de los contenedores de un *pod* es un conjunto de *namespaces*, cgroups y potencialmente otros aspectos del aislamiento, similar a aislar un contenedor individual. En **Kubernetes** los *pods* se pueden usar de dos formas, ejecutando un solo contenedor, siendo la forma más común, o con múltiples contenedores. En esta segunda opción todos los contenedores del *pod* pueden comunicarse entre sí a través de la dirección de *localhost*.

Workloads

Normalmente, no es recomendable crear y gestionar los *pods* directamente. En vez de crear un *pod* manualmente, es preferible usar objetos de tipo *workload*. De este tipo de objetos podemos destacar los dos más usados:

- ***Deployment***: Especifica como desplegar y gestionar aplicaciones dentro de **Kubernetes** proporcionando un nivel de abstracción superior al *pod*. Puede usarse para desplegar otros tipos de recursos, no solo *pods*, e incluso nos permite actualizar los *pods* y aumentar o disminuir el número de estos sin interrupción de servicio, facilitando además su autoescalado.

- ***Replicaset***: Su función es mantener un número estable de réplicas de un *pod* ejecutándose en todo momento. Su principal uso es garantizar la alta disponibilidad de un número determinado de *pods*. Cabe destacar que al crear un objeto de tipo *deployment* se crea y gestiona de forma automática un *replicaset*, por lo que el primero proporciona también una abstracción sobre el segundo.

En el listado 2.4 se nos muestra un ejemplo de un objeto *deployment*. Podemos ver que la especificación del objeto contiene: etiquetas en los metadatos para poder localizar más fácil-

mente los objetos, el número de réplicas que se desean (línea 8) y una pequeña especificación de los *pods* que se va a ejecutar, que será lo que se ejecute en todas las réplicas.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5    labels:
6      app: nginx
7  spec:
8    replicas: 3
9    selector:
10     matchLabels:
11       app: nginx
12   template:
13     metadata:
14       labels:
15         app: nginx
16     spec:
17       containers:
18       - name: nginx
19         image: nginx:1.14.2
20         ports:
21         - containerPort: 80
```

Listing 2.4: Ejemplo de especificación de un *deployment*

2.3.3 Servicios

Modelo de red de Kubernetes

En **Kubernetes** cada *pod* dispone de una dirección IP privada y única en el clúster. Normalmente no es necesario redireccionar los puertos de un contenedor a puertos del nodo donde se ejecuta [14]. Gracias a esto podemos pensar, a grandes rasgos, en los *pods* como máquinas virtuales. **Kubernetes** impone los siguientes requisitos fundamentales en cualquier implementación de red:

1. Todos los *pods* del clúster deben comunicarse entre sí sin necesidad de usar NAT.
2. Los agentes (p.ej. kubelet) deben comunicarse con todos los *pods* del nodo en el cual se estén ejecutando.

Este modelo de red nos permite que haya poca fricción al portear aplicaciones desde VMs a contenedores. El cómo se implementa este modelo depende del *container runtime* en uso.

¿Qué es un servicio en Kubernetes?

Un servicio es un recurso que ayuda a exponer grupos de *pods* a través de la red, proporcionando una forma de acceder a la aplicación que estén ejecutando. Los *pods* que son gestionados por un servicio suelen estar determinados por un ***selector***, funcionando a modo de etiqueta por la cual podemos filtrar recursos. Podemos ver un ejemplo de como se especifica un servicio en el listado 2.5.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Listing 2.5: Ejemplo de especificación de un servicio

Kubernetes dispone de varios tipos de servicios:

- **ClusterIP**: expone el servicio a una IP interna del clúster, esto hace que el servicio solo sea accesible desde dentro del mismo. Es el tipo por defecto a la hora de crear un nuevo servicio. Se puede exponer al exterior haciendo uso de un objeto *ingress*.
- **NodePort**: expone el servicio en la dirección IP de cada nodo del clúster en un puerto estático, por lo que todos los nodos exponen el mismo puerto. Para que este puerto esté disponible, **Kubernetes** expone el servicio en una dirección IP de la misma forma que si se hubiera solicitado un servicio de tipo ClusterIP.
- **LoadBalancer**: expone el servicio usando un balanceador de carga externo. **Kubernetes** no proporciona una implementación, por lo que hay que proporcionársela o integrarlo con un balanceador proporcionar por el proveedor cloud, si es que estamos en un entorno en la nube. En nuestro caso vamos a usar MetalLB [15, 16], un software con el cual podemos proveer a la infraestructura local de un balanceador de carga.
- **ExternalName**: Mapea el servicio a un nombre de DNS, en vez de a un *selector*, utilizando el parámetro *spec.externalName*.

2.3.4 Balanceo de carga y Networking

Networking

Para asignar las direcciones IP a los *pods*, permitir la comunicación entre ellos o incluso poder comunicarlos con servicios externos de manera eficiente, es necesario instalar un *plug-in* de red, el cual debe implementar la interfaz Container Network Interface (CNI) de **Kubernetes**. En este caso se ha optado por **Calico** [17], ya que es de los más populares y es posible su despliegue en un entorno local.

Balanceo de carga

Otra forma de poder exponer los servicios al exterior, es usando un objeto *ingress*. Este objeto nos permite definir reglas para enrutar tráfico HTTP y HTTPS a los servicios que tengamos operativos dentro del clúster, como se muestra en el listado 2.6. Para que *ingress* pueda dirigir el tráfico debemos indicar el servicio que queremos exponer y bajo qué ruta en concreto. Opcionalmente, podemos especificar el *host* bajo el cual queremos exponer el servicio. En cuanto a las rutas tenemos dos tipos:

- **Prefix**: este tipo permite hacer peticiones a subrutas. Por ejemplo, si hacemos una petición a */aa/bb/cc* y hemos definido que la ruta es */aa/bb*, podremos acceder a */cc*.
- **Exact**: como su nombre indica son rutas exactas, de modo que si hemos definido la ruta */foo* solo podemos acceder a dicha ruta.

También hay un tercer tipo el cual mezcla los dos anteriores.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: example-service
            port:
              number: 80
```

Listing 2.6: Especificación de un objeto *ingress*

Para poder hacer uso de objetos *ingress*, es necesario instalar un controlador *ingress* (*ingress-controller*) que implemente el enrutamiento en el clúster y aplique las reglas definidas en los objetos *ingress*. **Kubernetes** no proporciona ni instala por defecto un controlador, sino que nuevamente hay que proveer al clúster con una implementación. Dependiendo de si el *ingress-controller* instalado se expone a sí mismo como un servicio de tipo *NodePort* o *LoadBalancer*, necesitaremos hacer uso o no de un balanceador de carga externo. Con el balanceador de carga externo no hará falta exponer los servicios en el nodo en el que se estén ejecutando, ya que se le asignará una dirección IP externa al *ingress-controller* sobre la cual podremos acceder a los servicios que proporcionaría *ingress*. Para este despliegue se ha optado por usar Ingress-nginx [18], una implementación de controlador *ingress* comúnmente usada, que nos permite su despliegue en entornos locales en modo *LoadBalancer*.

Como hemos optado por exponer el controlador *ingress* mediante un servicio de tipo *LoadBalancer*, vamos a necesitar instalar un balanceador de carga externo, y para ello se va a usar MetalLB. Este software realiza la misma función que un balanceador de carga proporcionado por los proveedores en la nube cuando se despliega **Kubernetes** en dichos entornos.

2.3.5 Autoescalado

Kubernetes soporta tanto el autoescalado horizontal, Horizontal Pod Autoscaler (HPA), como el vertical, Vertical Pod Autoscaler (VPA). Es posible replicar los *pods* de una aplicación para atender la demanda que hay sobre esta, HPA, o bien aumentar/disminuir los recursos computacionales de los *pods* que ejecuten la aplicación, VPA. Para poder realizar este autoescalado, se utilizará **Metrics-server**[19], un componente que se puede añadir al clúster de **Kubernetes** para recopilar métricas de uso de recursos de los *pods* y de los nodos del clúster.

Una vez instalado **Metrics-server**, es posible aplicar el autoescalado a un *deployment* tal y como se observa en el listado 2.7, en este caso y por simplicidad nos limitamos a crear un objeto HPA. En este ejemplo, si un *pod* excede el 50% de uso de CPU se crearán nuevos *pods* para hacer frente a la demanda de recursos, hasta un máximo de 10 *pods*.

```
1 kubectl autoscale deployment dep-name --cpu-percent=50 --min=1 --max=10
```

Listing 2.7: Ejemplo de creación de un objeto HPA

2.3.6 Almacenamiento

Para este apartado necesitamos conocer los siguientes conceptos: Persistent Volume (PV) y Persistent Volume Claim (PVC). Por un lado, los PV son recursos de almacenamiento duraderos aprovisionados estáticamente por el administrador o dinámicamente por un objeto de tipo StorageClass (SC), el cual permite a los administradores indicar a los usuarios de qué

tipos de almacenamiento disponen. Permiten que los datos guardados en ellos persistan aunque los *pods* que los utilizan se hayan eliminado. Por otro lado, un PVC es una petición de almacenamiento de tipo PV. De forma similar a un *pod* que solicita recursos computacionales del nodo (CPU y memoria), un PVC solicita espacio en disco y un modo de acceso a un PV. El listado 2.8 muestra un ejemplo de la especificación de un PVC.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: kube-prometheus-stack-pvc
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: nfs-client
  resources:
    requests:
      storage: 10Gi
```

Listing 2.8: Especificación del objeto PVC

Antes de nada aclarar que en **Kubernetes**, resumidamente, los **volúmenes** son directorios que pueden inicialmente contener datos o no. Existen muchos tipos de volúmenes, pero en nuestro contexto nos interesa únicamente el tipo de volumen *Network File System* (NFS), ya que desplegaremos un servidor NFS con el que proporcionar almacenamiento compartido a los nodos del clúster. Este tipo de volumen permite montar un directorio NFS dentro de un *pod*. En el caso de este proyecto no será necesario especificar ningún objeto PV ni SC manualmente, ya que usaremos **Kubernetes NFS Subdir External Provisioner** [20], el cual creará dinámicamente un objeto de tipo SC. Más específicamente, este componente utiliza un servidor NFS que debe haber sido previamente configurado por un administrador, para proporcionar aprovisionamiento dinámico de PVs haciendo uso de PVCs. De esta forma solo tenemos que crear los objetos PVC que sean necesarios, en los cuales indicaremos el espacio que se va a solicitar y el modo de acceso.

2.3.7 Kubeadm y kubectl

Estas son las herramientas que nos permitirán realizar toda la instalación y configuración del clúster. Por un lado, **kubeadm** permite simplificar todas las tareas que son necesarias para tener un clúster mínimamente funcional, encargándose de inicializar el clúster y de permitir la unión de nuevos nodos al mismo, así como de la gestión de los certificados o actualizar **Kubernetes** a una nueva versión. Por otro lado, **kubectl** proporciona una interfaz de línea de comandos que nos permite realizar acciones sobre un clúster **Kubernetes** para, entre otras cosas, instanciar nuevos objetos, ya sean *pods*, servicios, *ingress*, añadir componentes como

Calico [17] y el controlador Ingress-nginx [18], u obtener información del estado del clúster.

2.4 Alta disponibilidad

La **alta disponibilidad** es una práctica que nos permite reducir el tiempo de inactividad de un sistema ante fallos hardware o software, sin que el usuario llegue a percatarse de estos. Esto se consigue eliminando puntos únicos de fallo, ya sea añadiendo redundancia en hardware o software, balanceo de carga, replicación de datos en distintas zonas geográficas, añadir sistemas de monitorización y alerta, entre otros métodos.

2.4.1 HAProxy

Es una herramienta de código abierto que permite proporcionar **alta disponibilidad** [21], balanceo de carga y funcionalidad de *proxy* inverso para aplicaciones basadas en TCP o HTTP. Utiliza una arquitectura dirigida por eventos para manejar múltiples conexiones simultáneamente con gran eficiencia, permitiendo que el sistema reaccione rápidamente a las solicitudes entrantes. Es conocido por su uso eficiente de recursos, minimizando el uso de CPU y memoria, lo que permite un rendimiento óptimo incluso bajo cargas pesadas. Implementa un planificador eficiente que gestiona las tareas con baja latencia y alto rendimiento, optimizando la distribución del tráfico y manteniendo la rapidez en el procesamiento de solicitudes.

2.4.2 Keepalived

Es un software de enrutamiento cuyo objetivo es proporcionar de forma simple y robusta **alta disponibilidad** y balanceo de carga para sistemas basados en Linux [22]. Implementa un sistema de comprobaciones que de forma dinámica y adaptable maneja un grupo de servidores en función de su salud (disponibilidad), con el objetivo de balancear su carga. Utiliza el protocolo Virtual Router Redundancy Protocol (VRRP) para crear *routers* redundantes, lo que garantiza que, si uno falla, otro puede asumir rápidamente el control, manteniendo la alta disponibilidad del servicio. Para la detección rápida de fallos en la red implementa el protocolo Bidirectional Forwarding Detection (BFD), el cual permite que los dispositivos de red detecten rápidamente cuando hay un fallo en la comunicación entre ellos.

De **HAProxy** vamos a usar su balanceo de carga con los tres nodos que formarán el *control plane* de **Kubernetes**, mientras que usaremos el protocolo VRRP de **keepalived** para asignar una dirección IP virtual a uno de los tres nodos del *control plane*, con el fin de que el clúster sea accesible bajo esa IP.

2.5 Software de monitorización y notificación

Las herramientas de monitorización nos permiten supervisar y controlar el funcionamiento de sistemas informáticos, servicios y redes. Se encargan de recopilar constantemente datos y métricas sobre el rendimiento, disponibilidad y el estado de los recursos que están monitorizando. Estos datos en tiempo real nos brindan información sobre el rendimiento de los sistemas, posibles cuellos de botella, problemas y fallos con el objetivo de tomar decisiones para optimizar su rendimiento o corregir dichos problemas. En estas métricas podemos ver cosas como el uso de CPU, memoria, el tráfico de red y muchos otros indicadores que consideremos relevantes. Además, estas herramientas pueden integrar también algún mecanismo de notificación o bien permitir su integración con otras herramientas de notificación, facilitando así la detección precoz de errores y problemas principalmente.

Las herramientas que veremos a continuación las instalaremos usando **Helm** [23], un gestor de paquetes que simplifica la definición, instalación y actualización de aplicaciones en un clúster **Kubernetes**.

2.5.1 Prometheus

Es una herramienta de monitorización y alerta de código abierto [24], muy utilizada en la actualidad para obtener métricas de sistemas, aplicaciones y servicios, a través de series temporales que podemos gestionar haciendo uso del lenguaje PromQL, el cual nos permite realizar consultas complejas fácilmente. Destaca por su arquitectura flexible y escalable, con un modelo de datos multidimensional. Ofrece la capacidad de añadir etiquetas a las métricas, facilitando la organización y análisis de estas. **Prometheus** puede adaptarse a diferentes tipos de sistemas, permitiendo la integración de diferentes fuentes de datos.

Otra de sus funciones son las alertas, las cuales puede generar en tiempo real. Es posible definir alertas personalizadas que se activarán cuando se den las condiciones especificadas. Estas alertas se pueden mandar mediante aplicaciones de mensajería haciendo que los administradores puedan actuar más rápidamente.

Destacar también que dispone de una interfaz web en la cual podemos añadir las gráficas en las que representar visualmente las métricas que queramos analizar.

2.5.2 Grafana

Es una herramienta de visualización de datos ampliamente utilizada gracias a su facilidad para crear paneles y gráficas visualmente vistosas e interactivas a partir de datos obtenidos de diversas fuentes, como pueden ser Prometheus o ElasticSearch, entre muchas otras [25].

Dispone de una interfaz web muy intuitiva, la posibilidad de personalizar los paneles con tablas, ofreciendo diferentes tipos de gráficas, alertas y otros elementos. El dinamismo de los

paneles nos permite explorar los datos de forma muy cómoda, pudiendo hacer zoom en los periodos de tiempo que queramos e interactuar con elementos visuales proporcionándonos más información. También posee un sistema para la gestión de usuarios y permisos, con el que poder compartir paneles con otros miembros del equipo o usuarios externos, fomentando la colaboración entre compañeros.

Por último, destacar que **Grafana** dispone de una gran variedad de *plug-ins* y extensiones permitiendo expandir sus funcionalidades y adaptarse a las necesidades específicas del usuario.

2.5.3 *Bot* de Telegram

La conocida aplicación de mensajería, **Telegram**, dispone de *bots* que podemos crear y configurar a gusto del consumidor y de forma gratuita. Debido a esto también vamos a crear un *bot* que se encargará de enviar notificaciones a un grupo que crearemos para este propósito. De esta forma se puede añadir a los administradores del clúster **Kubernetes** a un único grupo y así mantenerlos informados de todas las alertas y notificaciones que requieran de su atención.

Capítulo 3
Metodología

En este capítulo veremos la metodología que se ha seguido para el desarrollo del TFG, junto con un diagrama de Gantt donde se refleja el tiempo dedicado a cada parte del mismo.

3.1 Modelo iterativo e incremental

Esta metodología de desarrollo combina el enfoque iterativo y el incremental, proporcionando así flexibilidad y capacidad para manejar cambios y mejoras continuas a lo largo del ciclo de vida de un proyecto software, siendo estas sus principales ventajas.

Por un lado, en el desarrollo iterativo, en cada ciclo se realizan las actividades de planificación, diseño, implementación y pruebas. Al finalizar cada iteración se crea una versión funcional del producto final para ser evaluada y entregada al cliente para obtener retroalimentación.

Por otro lado, el desarrollo incremental se basa en la creación de un producto en múltiples etapas sucesivas. Cada incremento añade nuevas funcionalidades al producto hasta que se alcanza la versión final del mismo. Esto nos permite entregar al cliente el producto por etapas y de esta forma proporcionar valor de manera temprana y continua.

Esta combinación de ambos enfoques nos proporciona de una metodología más flexible y adaptable durante el desarrollo. Según se vaya obteniendo *feedback*, se pueden realizar ajustes y mejoras continuas al producto.

3.2 Metodología aplicada al proyecto

Al aplicar esta metodología al contexto de este TFG, se definieron los siguientes incrementos:

1. **Creación de VMs**: en este incremento se desarrolla el fichero *Vagrantfile* en el que se definen las características de las VMs que jugarán el rol de los nodos del clúster.

También se creará un fichero de variables, *settings.yml*, en el cual es posible configurar como será el despliegue del clúster, permitiendo establecer el número de VMs que se van a crear y de qué tipo, las direcciones IP y sus rangos, etc.

2. **Cluster inicial y Packer**: en este incremento se despliega el clúster de **Kubernetes**, instalando el orquestador y comprobando que los nodos tengan conectividad de red entre ellos. Se crearán las primeras versiones de los *playbooks* de Ansible que automaticen todas estas tareas, y crearemos también la primera *box* personalizada para Vagrant usando una plantilla de Packer.

3. **Cluster HA**: este incremento se encarga de instalar y configurar las herramientas Keepalived y HAproxy para proporcionar alta disponibilidad al clúster **Kubernetes**.

4. **Almacenamiento y monitorización**: se realiza un análisis de las formas de proveer de almacenamiento a Prometheus y Grafana. Se configura uno de los nodos del clúster como servidor NFS y se añade un nuevo disco para usarlo como almacenamiento. También se creará el *bot* de Telegram en este incremento.

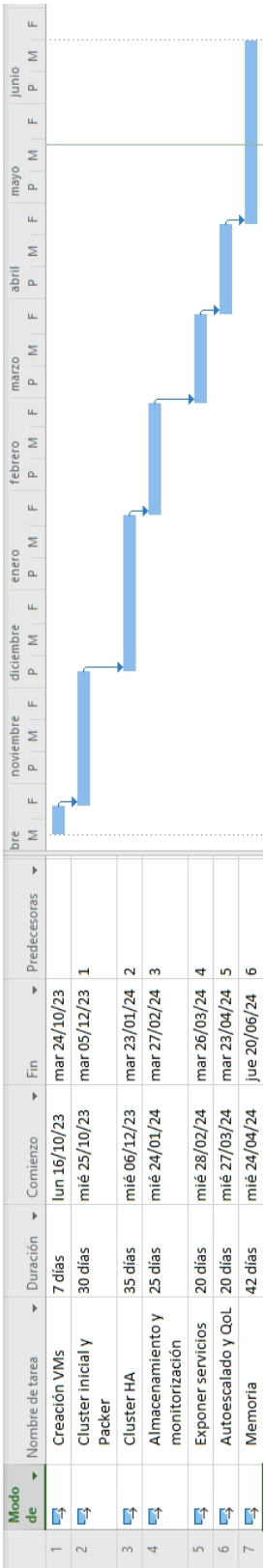
5. **Exponer servicios**: Se utilizan recursos *ingress* y balanceo de carga para exponer Prometheus y Grafana.

6. **Autoescalado y mejoras el los *playbooks***: añadimos la instalación y configuración Metrics-server para poder obtener métricas que nos permitan autoescalar los recursos de cómputo y añadimos mejoras en el código si son posibles, como puede ser el uso de módulos específicos de Ansible para determinadas tareas.

7. **Memoria**: redacción y revisión de la memoria.

Cabe destacar que en cada uno de los incrementos, a excepción del primero y el último, se modifican o crean nuevos *playbooks* de Ansible. Igualmente, la *box* personalizada de Vagrant y la plantilla de Packer también se actualizan según sea necesario a lo largo del desarrollo.

En la figura 3.1 se muestra un diagrama de Gantt con la estimación inicial de la duración de cada incremento y del trabajo en general. En total, se han dedicado unas **300 horas** de esfuerzo al proyecto, lo que junto con el sueldo medio de un ingeniero, que se ha estimado en 17 €/h aproximadamente [26], se obtiene un coste total de **5100 €**. Cabe destacar que no solo está incluido el coste del esfuerzo, sino que también se incluye el conocimiento y las habilidades de un ingeniero de sistemas.



	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		Creación VMs	7 días	lun 16/10/23	mar 24/10/23	
2		Cluster inicial y Packer	30 días	mié 25/10/23	mar 05/12/23	1
3		Cluster HA	35 días	mié 06/12/23	mar 23/01/24	2
4		Almacenamiento y monitorización	25 días	mié 24/01/24	mar 27/02/24	3
5		Exponer servicios	20 días	mié 28/02/24	mar 26/03/24	4
6		Autoescalado y QoL	20 días	mié 27/03/24	mar 23/04/24	5
7		Memoria	42 días	mié 24/04/24	jue 20/06/24	6

Figura 3.1: Diagrama de Gantt

Capítulo 4

Desarrollo del clúster

Durante este capítulo se va a detallar la implementación de cada *playbook* de Ansible que ha sido desarrollado en este TFG. También se describe la construcción del *Vagrantfile* para el despliegue de las máquinas virtuales del clúster y como se crea la *box* personalizada con la herramienta Packer. Finalmente, se explica como se configura el sistema de monitorización y notificación, además de detallar las pruebas con las que se ha verificado el correcto despliegue y configuración del clúster en su conjunto. Además, se explicará el porqué de algunas decisiones que se han tomado durante el desarrollo del TFG.

4.1 Configuración inicial de los nodos

En primer lugar, fue necesario seleccionar el hipervisor que se usaría para la virtualización de los nodos del clúster, en nuestro caso VirtualBox, así como la herramienta para gestionar estas máquinas virtuales con el objetivo de obtener una mayor automatización, la cual fue Vagrant. Debido a que VirtualBox no sigue el paradigma IaC, sin usar Vagrant sería necesario realizar la creación y configuración de las máquinas virtuales (nodos del clúster) de forma manual e interactiva a través de su interfaz gráfica o mediante comandos de terminal. Gracias a Vagrant podemos automatizar no solo los scripts que se ejecutan al terminar de arrancar una máquina virtual sin tener que añadirlos directamente a esta, sino también su despliegue y configuración. A continuación, se decidió el SO que se utilizará en los nodos del clúster. Se ha optado por **Rocky Linux**, una versión de código abierto compatible con la distribución empresarial más utilizada en la actualidad: Red Hat Entreprise Linux (RHEL). En un principio se empezó usando RHEL 9, pero por problemas con la configuración de Containerd con *cgroupfs* (un driver de *cgroups*), se decidió utilizar la versión 8 y usar *systemd* como driver de *cgroups*. Cabe destacar que esto no supone ningún problema a largo plazo, puesto que RHEL 8 tiene soporte garantizado hasta 2029 [27].

4.1.1 *Vagrantfile*

Antes de abordar el desarrollo del *Vagrantfile* fue necesario la instalación de dos *plug-ins* de Vagrant: *vagrant-hostmanager* [28] y *vagrant-vbguest* [29]. El primero se encarga de gestionar el fichero */etc/hosts* en las VMs y, opcionalmente, en la máquina anfitrión. El segundo *plug-in* sirve para instalar y actualizar automáticamente las *VirtualBox Guest Additions*, un complemento de VirtualBox muy utilizado para mejorar la experiencia de uso de las máquinas virtuales proporcionando funcionalidades como las carpetas sincronizadas, entre otras.

Una vez decidido el sistema operativo de las VMs, podemos comenzar a desarrollar el *Vagrantfile*, el fichero que nos permite desplegar y configurar de forma automatizada las máquinas virtuales que queremos ejecutar usando Virtualbox (aunque cabe destacar que Vagrant soporta también otros hipervisores). Haciendo uso del comando *vagrant init* se puede crear un fichero *Vagrantfile* inicial con una configuración mínima que procederemos a completar.

Al principio del *Vagrantfile* cargaremos un fichero *settings.yml*, creado previamente, el cual contiene la definición de todas las variables que permiten al usuario configurar los recursos que serán asignados a cada nodo del clúster, como se muestra en el listado 4.1. Mediante estas variables es posible indicar el número de CPUs, la cantidad de memoria, el número de nodos del *control plane* y nodos *worker* que vamos a desplegar junto con sus direcciones IP. Así como la red en la que se encuentra el servidor NFS y todas las constantes que serán necesarias más adelante. En el listado 4.2 se muestra un extracto del *Vagrantfile* donde podemos ver como se obtienen las variables del fichero de configuración que ha sido cargado (líneas 7-21), además de definir otras variables que se modifican tras la creación de cada VM (líneas 23-25).

```
1 require 'yaml'
2 custom_settings = YAML.load_file('settings.yml')
```

Listing 4.1: Carga del fichero de variables en el *Vagrantfile*

```
1  # Hostnames for master and worker nodes
2  MASTER_HOSTNAME = "idc-tfg-k8s-master"
3  MASTER_SEC_HOSTNAME = "idc-tfg-k8s-sec-master"
4  WORKER_HOSTNAME = "idc-tfg-k8s-worker"
5
6  # Cluster settings
7  MASTER_IP = custom_settings['master_ip']
8  MASTER_CORES = custom_settings['master_cores']
9  NUM_SEC_MASTERS = custom_settings['num_sec_masters']
10 NUM_WORKERS = custom_settings['num_workers']
11 WORKER_CORES = custom_settings['worker_cores']
12 MASTER_MEMORY = custom_settings['master_mem']
13 WORKER_MEMORY = custom_settings['worker_mem']
```

```
14 POD_NETWORK = custom_settings['pod_network']
15 CLUSTER_ENDPOINT = custom_settings['cluster_endpoint']
16 NFS_SERVER_IP = custom_settings['nfs_server_ip']
17 NFS_SERVER_NET = custom_settings['nfs_server_net']
18 NFS_SERVER_NAME = custom_settings['nfs_server_name']
19 NFS_SERVER_PATH = custom_settings['nfs_server_path']
20 CLUSTER_ENDPOINT_IP = custom_settings['cluster_endpoint_ip']
21 CLUSTER_ENDPOINT_NAME = custom_settings['cluster_endpoint_name']
22
23 require 'ipaddr'
24 CLUSTER_IP_ADDR = IPAddr.new MASTER_IP
25 CLUSTER_IP_ADDR = CLUSTER_IP_ADDR.succ
```

Listing 4.2: Variables generales

A continuación, empezamos con la configuración que es común a todo el entorno virtual, mostrada en el listado 4.3. Se define en primer lugar la *box* de Vagrant que vamos a usar para todas las VMs del clúster. Esta *box* es una imagen máquina que contiene un SO base preinstalado y listo para ser utilizado por una máquina virtual. Como se ha mencionado previamente, se utilizará una *box* basada en la distribución Rocky Linux 8 (véase la línea 1). También podemos indicar si queremos que Vagrant busque posibles actualizaciones de las *VirtualBox Guest Additions*, que en nuestro caso hemos desactivado simplemente por una cuestión de acelerar el despliegue durante la fase de desarrollo. Se configura también una carpeta sincronizada indicando que queremos compartir el directorio actual de la máquina anfitrión bajo la ruta */vagrant* de cada VM (línea 3). Por último, permitimos que el *plug-in hostmanager* pueda editar el fichero */etc/hosts* de cada uno de los nodos para añadir las direcciones IP y los *hostnames* del resto según terminan de ser desplegados (véanse las líneas 5-7).

```
1 config.vm.box = "generic/rocky8"
2 config.vm.box_check_update = false
3 config.vbguest.auto_update = false
4 config.vm.synced_folder ".", "/vagrant", type: "virtualbox"
5 config.hostmanager.enabled = true
6 config.hostmanager.manage_host = true
7 config.hostmanager.manage_guest = true
```

Listing 4.3: Configuración común

A partir de este punto empieza la configuración particular del nodo principal del clúster, los nodos secundarios que forman parte del *control plane* y los nodos *worker*. La configuración de los nodos secundarios y los *workers* es esencialmente la misma, tan solo se cambia la cantidad de recursos de los que van a disponer. En el listado 4.4 podemos ver como se configura un nodo *worker*. Podemos observar que la variable *CLUSTER_IP_ADDR* se modifica cada vez que se crea una VM (líneas 5-7) para asignar las distintas IPs, mientras que los recursos son

asignados en función de las variables que se han definido anteriormente (líneas 11-12).

```
1  # Worker nodes
2    (1..NUM_WORKERS).each do |i|
3      config.vm.define "worker-#{i}" do |worker|
4          worker.vm.hostname = "#{WORKER_HOSTNAME}-#{i}"
5          IP_ADDR = CLUSTER_IP_ADDR.to_s
6          CLUSTER_IP_ADDR = CLUSTER_IP_ADDR.succ
7          worker.vm.network "private_network", ip: IP_ADDR
8
9          worker.vm.provider "virtualbox" do |prov|
10             prov.name = "TFG-#{worker.vm.hostname}"
11             prov.cpus = WORKER_CORES
12             prov.memory = WORKER_MEMORY
13             prov.gui = false
14       end
15     end
16   end
```

Listing 4.4: Configuración de los nodos *worker*

A diferencia del resto, el nodo principal dispone de configuración adicional para instalar y ejecutar Ansible y la adición de un nuevo disco, esta configuración adicional la podemos ver en el listado 4.5. En primer lugar se le añade un disco adicional (líneas 1-11), ya que se ha decidido que en vez de usar otra máquina como servidor NFS se usará este nodo principal para proveer el almacenamiento compartido, para ahorrar recursos de la máquina anfitrión en nuestro despliegue (CPU y RAM principalmente). Para crear el disco hacemos uso de los comandos que nos permiten interactuar con VirtualBox sin usar su interfaz gráfica. Inicialmente ,definimos las variables que contienen la ruta en la que se va a guardar el nuevo disco y el tipo de controlador que se usará (líneas 1-2). A continuación, comprobamos si el disco existe para evitar crear uno nuevo (véase línea 5). Mediante el comando *createmedium* de VirtualBox, le indicamos la ruta que hemos especificado anteriormente, escogemos el formato con el que lo queremos guardar (en la máquina anfitrión) y el tamaño que va a ocupar de forma dinámica, es decir, al momento de ser creado no ocupará el total que le hemos asignado (línea 6). Y ya por último asignamos el nuevo disco a la máquina virtual usando el comando *storageattach* de VirtualBox (véase la línea 10). A este comando le pasamos el identificador de la máquina, el controlador de almacenamiento que hemos definido previamente, el número de puerto del controlador y el del dispositivo, el tipo de disco que queremos añadir y la ruta de la máquina anfitrión donde se ha creado.

La instalación y configuración de Ansible (líneas 14-31), se realiza usando pip3. Entre otras cosas, se configura el *playbook* principal que será ejecutado, el cual solo realiza la importación del resto de *playbooks* para ejecutar las tareas que contienen. También se le especifica el fichero

de inventario que contiene los *hosts* y los grupos definidos que se usarán para especificar qué *playbook* ejecuta cada nodo. Por último, se le pasan una serie de variables que se han definido al principio del *Vagrantfile* y que nos permitirán automatizar la configuración de las herramientas y de los nodos.

```
      disk = ".vagrant\\machines\\master\\nfsdsk.vmdk"
      sataController = "SATA Controller"

      # Create the virtual disk if doesn't exist
      unless File.exist?(disk)
        prov.customize ["createmedium", "disk", "--filename", disk,
    "--format", "VMDK", "--size", 20480]
      end

      # Attach the virtual disk into the storage SAS controller
      prov.customize ["storageattach", :id, "--storagectl",
    sataController, "--port", 1, "--device", 0, "--type", "hdd",
    "--medium", disk]
    end

    # Install and setup K8s using ansible
    master.vm.provision "ansible_local", run: "once" do |ansible|
      ansible.install = "true"
      ansible.install_mode = "pip3"
      ansible.playbook = "provisioning/playbook-main.yml"
      ansible.inventory_path = "ansible.inventory"
      ansible.limit = "all"
      ansible.extra_vars = {
        master_ip: MASTER_IP,
        master_hostname: MASTER_HOSTNAME,
        pod_network: POD_NETWORK,
        cluster_endpoint: CLUSTER_ENDPOINT,
        nfs_server_ip: NFS_SERVER_IP,
        nfs_server_net: NFS_SERVER_NET,
        nfs_server_path: NFS_SERVER_PATH,
        nfs_server_name: NFS_SERVER_NAME,
        cluster_endpoint_ip: CLUSTER_ENDPOINT_IP,
        cluster_endpoint_name: CLUSTER_ENDPOINT_NAME,
      }
```

Listing 4.5: Configuración extra del nodo principal

Para finalizar el *Vagrantfile*, una vez que termina de arrancar cada máquina virtual se ejecutará el *script bootstrap.sh*. Este aprovisionamiento configura la conexión por SSH sin contraseña desde el nodo principal al resto de nodos mediante el uso de claves público/privadas. Esto es muy recomendable ya que de lo contrario tendríamos que introducir manualmente la con-

traseña cada vez que se ejecute una tarea de Ansible en cada uno de los nodos. Resumidamente, el *script* lo que hace es copiar el fichero de inventario de Ansible (*ansible.inventory*) en la ruta */etc/ansible/hosts*, ruta que después usará Ansible. A continuación, se genera una nueva clave *rsa* para el usuario *vagrant*, copiándola a la carpeta compartida si el nodo que lo está ejecutando es el principal. Al finalizar, añade la clave al fichero */home/vagrant/.ssh/authorized_keys*, para poder iniciar sesión sin proporcionar la contraseña. Para ejecutar el *script* al terminar de crear una VM, se añade al *Vagrantfile* el código que se muestra en el listado 4.6.

```
1
2 # Global provisioning bash script
3 config.vm.provision "shell", run: "once", path:
      "provisioning/bootstrap.sh"
4 end
```

Listing 4.6: Configuración en el *Vagrantfile* del aprovisionamiento de las máquinas virtuales

4.1.2 Creación de la *box* personalizada

Como vamos a instalar los binarios de **Kubernetes** en las máquinas virtuales, el *container runtime* (Containerd), las herramientas Keepalived y Haproxy, el servidor NFS, además de realizar alguna configuración adicional, resulta muy recomendable disponer de una *box* de Vagrant personalizada que contenga ya todo el software necesario y configurado, listo para su uso. Esta aproximación ayuda a reducir el tiempo de despliegue desde cero de forma significativa. Para crear esta *box* usaremos la herramienta Packer, la cual nos permite hacerlo de forma automatizada y declarativa, y para lo cual es necesario instalar su *plug-in* correspondiente para Vagrant.

Para aprovisionar la *box* con todo el software usaremos un *script* que, partiendo de la *box* de base Rocky Linux 8, se encargue de instalar y configurar todo lo que sea necesario. Para instalar Containerd y **Kubernetes** es necesario añadir sus respectivos repositorios (véase el listado 4.7). En el caso de Containerd es suficiente con un único comando (línea 2), el cual creará automáticamente el archivo *.repo* en la ruta */etc/yum.repos.d* usando la URL apropiada. El repositorio que vamos a añadir pertenece a Docker, y es de donde podremos descargar su *container runtime*. En cambio, para añadir el repositorio de **Kubernetes** debemos crear manualmente un archivo *.repo* en la misma ruta de antes (*/etc/yum.repos.d*, ver líneas 5-11). En este archivo especificamos la URL del repositorio y donde encontrar las claves GNU Privacy Guard (GPG) para la validación de los paquetes, como mínimo. Además, añadimos la opción *enabled*, para que el gestor de paquetes Yum sepa si puede usar el repositorio o no, y *gpgcheck*, que le indica a dicho gestor si debe realizar la validación de las firmas GPG de los paquetes. Ahora que hemos añadido ambos repositorios podemos instalar los binarios necesarios haciendo uso del comando *yum install -y <package>* (líneas 14-15).

```
# Add Docker repo
sudo yum-config-manager --add-repo
    https://download.docker.com/linux/centos/docker-ce.repo

# Add K8s repo
sudo echo "[kubernetes]" | sudo tee -a /etc/yum.repos.d/kubernetes.repo
sudo echo "name=Kubernetes" | sudo tee -a
    /etc/yum.repos.d/kubernetes.repo
sudo echo "baseurl=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/" | sudo
    tee -a /etc/yum.repos.d/kubernetes.repo
sudo echo "enabled=1" | sudo tee -a /etc/yum.repos.d/kubernetes.repo
sudo echo "gpgcheck=1" | sudo tee -a /etc/yum.repos.d/kubernetes.repo
sudo echo "gpgkey=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/repodata/
    repomd.xml.key" | sudo tee -a /etc/yum.repos.d/kubernetes.repo

# Install binaries
sudo yum install -y docker-ce docker-ce-cli containerd.io
    docker-buildx-plugin docker-compose-plugin
sudo yum install -y kubelet kubeadm kubectl
```

Listing 4.7: Añadir repositorios de Docker y Kubernetes

Referente a la configuración, mostrada en el listado 4.8, tal y como viene indicado en la documentación de **Kubernetes** debemos deshabilitar el *swap* en todos los nodos (líneas 2-3). También desactivamos SELinux (líneas 6-7), ya que en la documentación de kubeadm [30] se recomienda hacer esto en los sistemas basados en RHEL, aunque debería ser posible usarlo si se sabe configurar correctamente [31]. También creamos una configuración por defecto de Containerd, y le asignamos el valor *true* a la variable *SystemdCgroup* con el propósito de que Containerd hago uso de *systemd* para manejar los *cgroups* [32]. Desactivamos también el cortafuegos (firewalld) y eliminamos las reglas que haya en iptables (líneas 14-17). Esto se hace porque al iniciar el clúster, **Kubernetes** generará las entradas adecuadas en iptables. Instalamos también otras herramientas que vamos a usar más adelante: nfs-utils, HAProxy y Keepalived. Y para finalizar nos descargamos las imágenes que nos harán falta para el despliegue del clúster de **Kubernetes**, de esta forma evitamos que se tengan que descargar al arrancar el clúster por primera vez.

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

# SELinux as Permissive
sudo setenforce 0
sudo sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/'
    /etc/selinux/config
```

```
8
9  # Configure Containerd
10 sudo Containerd config default | sudo tee
       /etc/Containerd/config.toml
11 sudo sed -i 's/SystemdCgroup = false/SystemdCgroup = true/'
       /etc/Containerd/config.toml
12
13 # Delete Iptables entries
14 sudo iptables -F && sudo iptables -t nat -F && sudo iptables -t
       mangle -F && sudo iptables -X
15
16 # Disable firewalld
17 sudo systemctl disable firewalld
18
19 # Install extra tools
20 sudo yum install -y nfs-utils keepalived haproxy
21
22 # Fetch kubernetes images
23 sudo kubeadm config images pull
```

Listing 4.8: Personalización del *box*

A continuación, creamos una plantilla de *Vagrantfile* que usará Packer para iniciar una máquina virtual en la que instalará el software para crear la *box* resultante. Esta plantilla se parece mucho a la parte de los nodos *worker* que se mostró anteriormente en el listado 4.4, solo que se elimina la interfaz de red que se añade en ese código y se define un *output*, el cual guardará la *box* resultante, y al igual que las máquinas que se van a desplegar, monta un directorio compartido.

Y por último, tenemos que crear la plantilla de Packer usando el lenguaje HCL. En esta plantilla indicamos de forma declarativa como se va a construir la *box*, en el listado 4.9 se muestra su contenido. Primero se define un *source*, donde se indica la plataforma para la que se crea el artefacto y el directorio de destino en el que se guarda una vez ha finalizado el proceso. Dentro de este *source*, le indicamos varias cosas. Primero, que se debe comunicar por SSH con la máquina virtual que se va a crear, luego la *box* base de la que se parte y su versión. También le indicamos el software de virtualización que se usará, en este caso al igual que para las VMs desplegadas será VirtualBox (opción *provider*, línea 5). Por último, se le indica la plantilla que usará como *Vagrantfile*, explicada previamente. A continuación, se añade un bloque *build* (líneas 9-15), en el que se especifica que *sources* se van a construir, además del tipo de aprovisionamiento que van a ejecutar. En este caso, construimos el *source* que hemos definido previamente, el cual mediante un aprovisionamiento de tipo *shell* ejecutará el *script* de configuración que se explicó en este mismo apartado, encargado de instalar y configurar todo el software necesario.

```
source "vagrant" "tfg_rocky8" {
  communicator = "ssh"
  source_path  = "generic/rocky8"
  box_version  = "2.0.0"
  provider     = "virtualbox"
  template     = "provisioning/Vagrantfile.template"
}

build {
  sources = ["source.vagrant.tfg_rocky8"]

  provisioner "shell" {
    script  = "provisioning/prov_packer.sh"
    timeout = "30m"
  }
}
```

Listing 4.9: Plantilla de Packer

4.2 *Playbooks*

Previamente a ejecutar Ansible debemos desplegar las máquinas virtuales del clúster. Para ello usamos el comando *vagrant up -provision-with shell*. El argumento de este comando indica que se ejecute el *script* que configura la conexión por SSH sin contraseña, explicado previamente. Esto es necesario hacerlo de esta forma ya que, en caso contrario, en el momento que la primera máquina virtual terminase de arrancar se instalaría Ansible sin que el resto de máquinas hayan sido creadas, e intentaría ejecutar los *playbooks* los cuales fallarían.

Una vez que las máquinas se hayan desplegado y la configuración de SSH haya sido realizada, ejecutaremos el comando *vagrant provision -provision-with ansible_local*. Esto hará que el nodo principal instale Ansible y comience a ejecutar los *playbooks* que hemos desarrollado, como se indicó en el *Vagrantfile* mostrado en el listado 4.5. La implementación de estos *playbooks* se explica a continuación.

4.2.1 *Playbook* común

Este es el primer *playbook* en ser ejecutado. Como se muestra en el listado 4.10, se llevan a cabo las tareas que deben realizar todos los nodos del clúster como usuario *root*, usando la opción *become: yes* al inicio del *playbook* forzamos esta configuración. No sin antes indicar que grupo realizará estas tareas, el grupo *cluster*. Se trata de un grupo formado por los demás grupos que se han creado (*masters*, *workers* y *sec_masters*), que abarcan todos los nodos.

Normalmente todas las tareas se ejecutarán como usuario *root* a excepción de las tareas del último *playbook* y alguna tarea puntual. Para estas tareas puntuales se configura *become: false* en la propia definición de las mismas para ejecutarlas con el usuario *vagrant*.

```
1 - hosts: cluster
2   become: yes
```

Listing 4.10: Selección de grupo y usuario para ejecutar el *playbook*

Inicialmente, vamos a añadir el repositorio de Docker, ya que lo necesitamos para instalar Containerd, y el repositorio de **Kubernetes** (véase el listado 4.11). Esto se realiza de forma similar a como se hizo a la hora de personalizar la *box* con Packer usando un *script*. La diferencia es que esta vez usaremos un módulo de Ansible (*ansible.builtin.shell* o *shell* para abreviar) que nos permite interactuar con la línea de comandos del nodo en el que se ejecuta, añadiendo así el repositorio de Docker (línea 2). También añadimos el repositorio de **Kubernetes** haciendo uso del módulo *ansible.builtin.yum_repository* (líneas 5-12), en el que añadimos la URL del repositorio (*baseurl*), y la clave GPG para la validación de los paquetes (*gpgkey*). Adicionalmente, añadimos los parámetros *enabled*, para indicarle al gestor de paquetes que use el repositorio, y *gpgcheck*, con el que especificamos que queremos verificar los paquetes con la clave que le indicamos anteriormente. Además, con el parámetro *state* le indicamos el estado deseado del fichero que contiene el repositorio, en este caso, debe crearse si no existe.

```
1  - name: Add docker repo
2        shell: yum-config-manager --add-repo
     https://download.docker.com/linux/centos/docker-ce.repo
3
4  - name: Add an rpm signing key for k8s & repo
5        ansible.builtin.yum_repository:
6          name: k8s
7          description: Add K8s repo
8          baseurl: https://pkgs.k8s.io/core:/stable:/v1.28/rpm/
9          gpgkey:
     https://pkgs.k8s.io/core:/stable:/v1.28/rpm/repodata/repomd.xml.key
10         enabled: yes
11         gpgcheck: yes
12         state: present
```

Listing 4.11: Añadir los repositorios de Docker y Kubernetes con Ansible

Una vez que se han añadido los repositorios, instalamos los paquetes usando un gestor de paquetes (Dnf o Yum). Podemos hacerlo de dos formas: o bien invocamos el módulo *shell* e introducimos el comando requerido, o usamos los módulos *ansible.builtin.yum* o *ansible.builtin.dnf*, como se ve en el listado 4.12. Además de los paquetes de Containerd y **Kubernetes** (kubelet, kubeadm y kubectl), aprovechamos e instalamos los correspondiente para

Keepalived, HAProxy y nfs-utils, que necesitaremos más adelante.

```
- name: Install Docker and other dependencies
      yum:
        name: "{{ packages }}"
        state: present
      vars:
        packages:
          - docker-ce
          - docker-ce-cli
          - containerd.io
          - docker-buildx-plugin
          - docker-compose-plugin
          - keepalived
          - haproxy
          - nfs-utils

- name: Install Kubernetes binaries
      yum:
        name: "{{ packages }}"
        state: present
      vars:
        packages:
          - kubelet
          - kubeadm
          - kubectl
```

Listing 4.12: Instalación de paquetes

Una vez instalado Containerd, vamos a eliminar su configuración por defecto (o la que existiera previamente), usando para ello el módulo *ansible.builtin.file*. En este módulo indicamos la ruta del fichero de configuración (*path*) y el estado deseado (*state*), en este caso *absent* (para eliminar dicho fichero). Acto seguido, generaremos una nueva configuración en la cual le asignamos el valor *true* a la variable *SystemdCgroup*, haciendo uso del comando *sed* y del módulo *shell*. Esto hará que Containerd hago uso de *systemd* como driver para manejar los *cgroups*. Sumado a esto, y tras eliminar la configuración de Containerd, haciendo uso del módulo *notify* y un *handler* de Ansible, reiniciamos el *container runtime* una vez que se hayan completado todas las tareas del *playbook*. Toda esta configuración la podemos observar en los listados 4.13 y 4.14.

```
- name: Remove default Containerd file
      file:
        path: /etc/Containerd/config.toml
        state: absent
      notify:
        - restart containerd
```

```
7
8  - name: Create default Containerd config.toml
9        shell: containerd config default | tee
       /etc/Containerd/config.toml
10
11 - name: Edit config.toml
12       shell: sed -i 's/SystemdCgroup = false/SystemdCgroup = true/'
       /etc/Containerd/config.toml
```

Listing 4.13: Editar la configuración de Containerd

```
1  handlers:
2      - name: restart docker
3        service: name=docker state=restarted
4
5      - name: restart containerd
6        service: name=containerd state=restarted
```

Listing 4.14: Uso de los *handlers* de Ansible

Tras haber configurado Containerd, seguimos con los pasos mostrados en el listado 4.15. Con el comando *kubeadm reset -f* borramos la configuración de **Kubernetes** con el fin de eliminar cualquier rastro de despliegues previos, en caso de que hubiese fallado y queremos que se haga desde cero. Asimismo, eliminamos el archivo de configuración que nos permite usar Kubectl para administrar el clúster con el usuario *vagrant*. También modificamos el archivo */etc/hosts* (líneas 9-17), para añadir la dirección IP del servidor NFS y la IP del *endpoint* a través del cual accedemos al clúster, usando para ello el módulo *lineinfile* de Ansible. Gracias a las variables que le pasamos a Ansible desde el *Vagrantfile*, podemos realizar cambios en la arquitectura final del clúster. Por ejemplo, en el caso de que quisiéramos usar un servidor NFS distinto, tan solo tendríamos que cambiar la dirección IP que cargamos en Ansible, como podemos ver en la tarea que se define en las líneas 14-17.

```
1  - name: Reset Kubernetes cluster
2        shell: kubeadm reset -f
3
4  - name: Remove kube config file
5        file:
6          path: /home/vagrant/.kube/config
7          state: absent
8
9  - name: Add cluster-endpoint ip to /etc/hosts
10       lineinfile:
11         path: /etc/hosts
12         line: "{{ cluster_endpoint_ip }}\t{{ cluster_endpoint_name
       }}"
```

```
13
14 - name: Edit /etc/hosts
15       lineinfile:
16         path: /etc/hosts
17         line: "{{ nfs_server_ip }}\t{{ nfs_server_name }}"
```

Listing 4.15: Reset con Kubeadm y modificación de */etc/hosts*

Para poder desplegar el clúster, **Kubernetes** nos obliga a desactivar el *swap* [33], ya que perdemos propiedades de aislamiento. Para esto usamos los módulos *mount* y *shell* de Ansible, como se muestra en el listado 4.16 (líneas 1-12). Además de deshabilitar el *swap*, también eliminamos las entradas existentes en iptables (línea 15), ya que **Kubernetes** añadirá las suyas propias automáticamente, y deshabilitamos firewalld (líneas 17-21), para que no interfiera con el despliegue del clúster.

```
1  - name: Remove swapfile from /etc/fstab
2        mount:
3          name: "{{ item }}"
4          fstype: swap
5          state: absent
6        loop:
7          - swap
8          - none
9
10 - name: Disable swap
11       shell: swapoff -a
12       when: ansible_swaptotal_mb > 0
13
14 - name: Reset iptables
15       shell: iptables -F && iptables -t nat -F && iptables -t
       mangle -F && iptables -X
16
17 - name: Disable firewalld
18       systemd_service:
19         enabled: false
20         name: firewalld
21         state: stopped
```

Listing 4.16: Deshabilitar swap, firewalld y eliminar entradas en iptables

Y para finalizar ponemos SELinux en modo permisivo (véase el listado 4.17), ya que se recomienda desde la documentación de kubeadm [30] para sistemas basados en RHEL. Además, a la hora de configurar posteriormente el balanceo de carga, HAProxy entra en conflicto con SELinux en el modo restrictivo (*enforcing*) y las peticiones que se intentan hacer al clúster no se redirigen correctamente a los API server de los nodos que conforman el *control plane* de

Kubernetes. Destacar que debería ser posible usar SELinux en modo restrictivo (*enforcing*), si se configura correctamente [31].

```
- name: Set SELinux to permissive
      ansible.posix.selinux:
        policy: targeted
        state: permissive
```

Listing 4.17: Desactivar SELinux

4.2.2 *Playbook* del nodo principal del *control plane*

En este *playbook* vamos a configurar las herramientas que nos proporcionan alta disponibilidad en el nodo principal, así como inicializar el clúster.

Configuración de Keepalived y HAProxy

Empezamos copiando la configuración que hemos generado para Keepalived en la ruta destino */ect/keepalived* desde el directorio compartido con la máquina anfitrión (*/vagrant/provisioning*). Para ello hacemos uso del módulo *copy* de Ansible como se muestra en el listado 4.18 (líneas 1-5). Cabe recordar que Keepalived nos va a proporcionar una dirección IP virtual para los nodos del *control plane*. Cuando Keepalived detecte que el nodo con la IP virtual asignada no esté disponible, automáticamente se la asignará a uno de los nodos de respaldo. Para esta detección creamos un *script* con el que hacemos peticiones continuamente a la IP virtual por el puerto en el que se expone el clúster haciendo uso del comando *curl*, como se muestra en el listado 4.19. Este *script* también lo copiamos desde el directorio compartido a la ruta */etc/keepalived*, usando el mismo módulo de Ansible.

```
- name: Copy config file
      copy:
        src: /vagrant/provisioning/keepalived.conf
        dest: /etc/keepalived/
        remote_src: yes

- name: Copy check_apiserver.sh
      copy:
        src: /vagrant/provisioning/check_apiserver.sh
        dest: /etc/keepalived/
        remote_src: yes
```

Listing 4.18: Copia de la configuración de Keepalived y del *script* check_apiserver.sh

```
#!/bin/sh
APISERVER_VIP=192.168.56.30
```

```
APISERVER_DEST_PORT=8443
errorExit() {
    echo "*** $*" 1>&2
    exit 1
}
curl --silent --max-time 2 --insecure
    https://${APISERVER_VIP}:${APISERVER_DEST_PORT}/ -o /dev/null ||
    errorExit "Error GET
    https://${APISERVER_VIP}:${APISERVER_DEST_PORT}/"
```

Listing 4.19: *Script* para comprobar conexión con el nodo

En el archivo de configuración *keepalived.conf*, que se muestra en el listado 4.20, vamos a definir un primer objeto con la siguiente configuración (líneas 4-10): el *script* de detección que hemos creado, el intervalo de tiempo en segundos (*interval*) que hay entre las ejecuciones del *script*, cuanto se le sustrae a la prioridad (*weight*) en caso de que el *script* falle, el número de veces que tiene que fallar (*fall*) el *script* para que se considere que el nodo no está disponible y cuanto aumenta la prioridad (*rise*) en caso de que el *script* sea exitoso.

```
global_defs {
    router_id LVS_DEVEL
}
vrrp_script check_apiserver {
  script "/etc/keepalived/check_apiserver.sh"
  interval 3
  weight -2
  fall 10
  rise 2
}
vrrp_instance VI_1 {
    state MASTER
    interface eth1
    virtual_router_id 151
    priority 150
    authentication {
        auth_type PASS
        auth_pass P@##D321!
    }
    virtual_ipaddress {
        192.168.56.30/24
    }
    track_script {
        check_apiserver
    }
}
```

Listing 4.20: Configuración de Keepalived

A continuación, creamos otro objeto (líneas 11-25) que será el encargado de la asignación de la IP virtual. En este objeto definimos el estado: *Master* o *Backup*. En caso de que el estado sea *Master*, indica que ese nodo será el que tenga la IP virtual inicialmente. Si el estado es *Backup*, entonces Keepalived decidirá cuál es el nodo que dispondrá de la IP cuando el *Master* no esté disponible. También indicamos la interfaz a la que se le asignará la IP, en este caso es *eth1*, que es donde tenemos la red privada con el resto de nodos. Añadimos también la prioridad inicial del nodo, un ID de *router* virtual que identifica una instancia del protocolo Virtual Router Redundancy Protocol (VRRP) en un segmento de red, la autenticación para los mensajes del protocolo VRRP, la IP virtual con la máscara de subred y una referencia al objeto anterior para seguir sus actualizaciones.

En la figura 4.1, podemos ver una representación de como funciona Keepalived, la línea continua simboliza que la IP virtual está asignada a ese nodo, mientras que las líneas discontinuas que van hacia los nodos simbolizan que están a la espera de que el nodo que tiene la IP virtual no esté disponible. Esto es lo que se conoce como una configuración activo-pasivo.

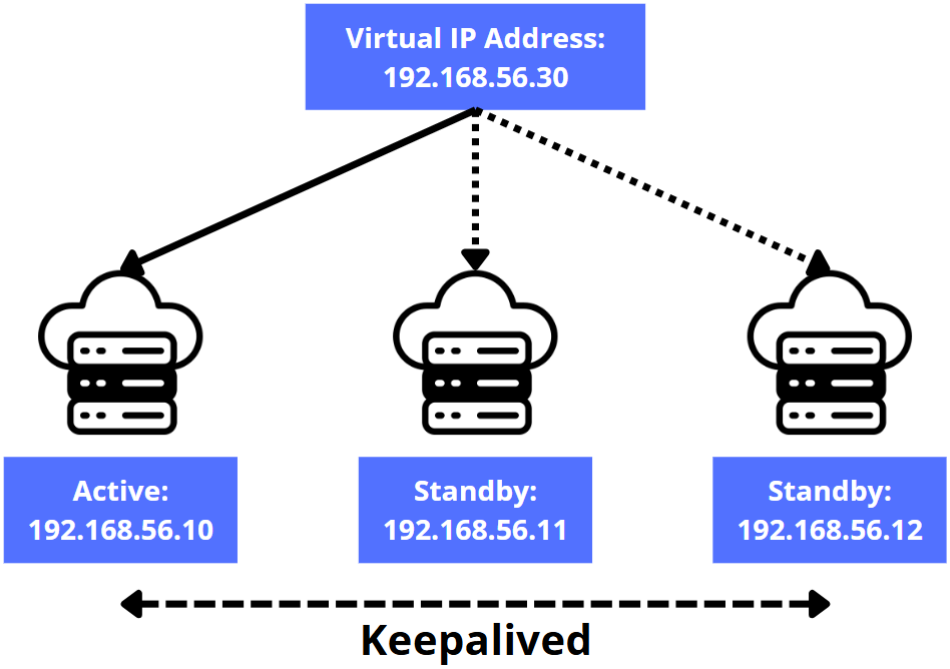


Figura 4.1: Representación de la IP virtual con Keepalived

En el caso de HAProxy haremos lo mismo que para Keppalived, empezando por copiar su fichero de configuración *haproxy.cfg* desde el directorio compartido */vagrant/provisioning* al

directorio */etc/haproxy*, como se muestra en el listado 4.21.

```
- name: Copy haproxy.cfg
      copy:
        src: /vagrant/provisioning/haproxy.cfg
        dest: /etc/haproxy/
        remote_src: yes
```

Listing 4.21: Copia del fichero haproxy.cfg en /etc/haproxy

De este fichero de configuración solo nos interesa la parte mostrada en el listado 4.22, ya que el resto son valores por defecto [34, 35]. En esta sección tenemos dos bloques: el *frontend*, desde donde vamos a exponer el clúster, y el *backend*, compuesto por las instancias del API server en cada nodo del *control plane*. En el *frontend*, se redirige el tráfico entrante por el puerto 8443 de todas las direcciones IP al *backend*, especificado como *backend k8s*. También le indicamos el modo en el que opera este *frontend*, que es mediante el protocolo TCP. En el *backend* indicamos la IP y el puerto en los que se expone la instancia de API server en los nodos del *control plane* de **Kubernetes**. Además, para comprobar que el servidor está disponible añadimos la opción *option httpchk GET /healthz* (línea 7), para que haga una petición a dicha ruta. El código de respuesta esperado por la comprobación anterior se configura con *http-check expect status 200*, y para que haga una comprobación básica de que está escuchando en el puerto correcto usamos *option ssl-hello-chk*. Adicionalmente, especificamos qué algoritmo se usa para el balanceo de carga (*roundrobin*, línea 11).

La figura 4.2 muestra una representación visual del funcionamiento de HAProxy.

```
frontend apiserver
    bind *:8443
    mode tcp
    option tcplog
    default_backend k8s
backend k8s
    option httpchk GET /healthz
    http-check expect status 200
    mode tcp
    option ssl-hello-chk
    balance     roundrobin
        server master-1 192.168.56.10:6443 check
        server master-2 192.168.56.11:6443 check
        server master-3 192.168.56.12:6443 check
```

Listing 4.22: Configuración de HAProxy

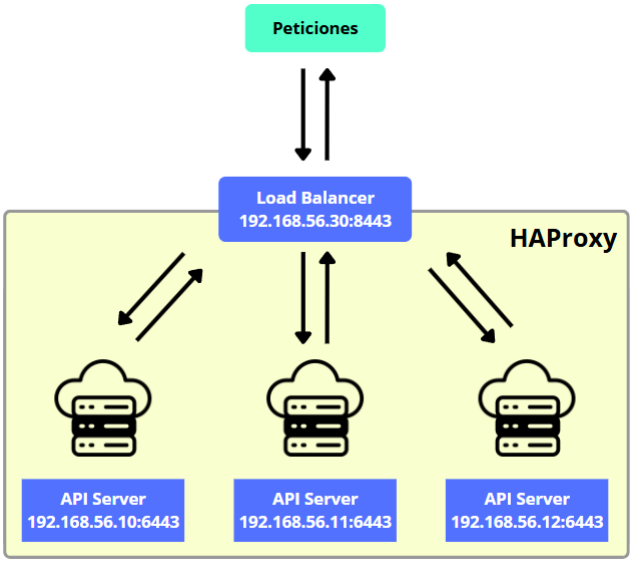


Figura 4.2: Representación de del balanceo de carga con HAProxy

Para terminar con Keepalived y HAProxy en este nodo principal, debemos hacer que se inicien los servicios correspondientes durante el arranque del sistema, haciendo uso del módulo *systemd_service* de Ansible, como se muestra en el listado 4.23.

```
- name: Enable keepalived
      systemd_service:
        name: keepalived
        enabled: true

- name: Enable haproxy
      systemd_service:
        name: haproxy
        enabled: true
```

Listing 4.23: Habilitar Keepalived y HAProxy en el arranque del sistema

Inicialización de Kubernetes

Ahora que tenemos Keepalived y HAProxy configurados y funcionando correctamente, es momento de inicializar el clúster de **Kubernetes**. Para ello usaremos la herramienta kubeadm, la cual nos ayuda simplificando los pasos para obtener un clúster mínimamente funcional, haciendo uso de las operaciones *kubeadm init* y *kubeadm join*. Con el comando *kubeadm init* ejecutado en el nodo principal inicializamos el clúster junto con una serie de parámetros que se explican a continuación. Con *control-plane-endpoint* indicamos que queremos exponer el clúster a través de un *endpoint*. En nuestro caso, el *endpoint* será *idc-cluster-endpoint:8443*,

nótese que también especificamos el puerto por el cual queremos exponerlo y que debe coincidir con el que usamos para configurar el *frontend* del balanceador de carga de HAProxy, como se vio en el listado 4.22. También añadimos el parámetro *apiserver-advertise-address*, que especifica la dirección IP desde la cual se puede acceder a la instancia de API server de este nodo, la cual será la IP que se le ha asignado al nodo desde el *Vagrantfile* con la variable *MASTER_IP*. Otro parámetro que necesitamos añadir es *pod-network-cidr*, el cual especifica la red interna del clúster por la cual los *pods* se comunicarán entre sí. Para que la comunicación entre *pods* funcione es necesario configurar un *plug-in* que cumpla con la especificación CNI y que instalaremos posteriormente (usaremos Calico [17]). Adicionalmente, el parámetro *upload-certs* se usa para añadir los certificados del *control plane* a un objeto de tipo *Secret* temporalmente. Estos certificados serán necesarios para permitir que los nodos secundarios del *control plane* puedan adquirir su rol correctamente. Por último, incluimos el parámetro *ignore-preflight-errors=NumCPU* simplemente para permitir arrancar el clúster con una sola CPU, ya que por defecto saldrá un error indicando que no disponemos de al menos dos CPUs como viene indicado en la documentación de **Kubernetes**. Sumado a esto, las direcciones IP y redes que se usan se obtienen a través de variables de Ansible, que le proporcionamos desde el *Vagrantfile*. El uso de estas variables se muestra en el listado 4.24.

```
- name: Iniciar el cluster con kubeadm
      shell: kubeadm init --control-plane-endpoint {{ cluster_endpoint
   }} --apiserver-advertise-address="{{ master_ip }}"
   --pod-network-cidr="{{ pod_network }}" --upload-certs
   --ignore-preflight-errors=NumCPU
```

Listing 4.24: Iniciar el clúster **Kubernetes**

Cuando haya terminado de ejecutarse el comando *kubeadm init*, tendremos que crear el directorio *.kube* en */home/vagrant*, y copiar en esa ruta el fichero */etc/kubernetes/admin.conf* haciendo propietario del mismo al usuario *vagrant* con permisos exclusivos de lectura/escritura, como se muestra en el listado 4.25. Esto lo hacemos así para poder interactuar con el clúster usando Kubectl desde los nodos del *control plane*. Para esta tarea usamos los módulos de Ansible *file* y *copy*, con el primero creamos el directorio en cuestión y con el segundo copiamos el fichero necesario, a la vez que modificamos el usuario y grupo al que pertenece y le proporcionamos los permisos adecuados.

```
- name: Create .kube dir for vagrant user
     file:
       path: /home/vagrant/.kube
       state: directory
       mode: 0755

- name: Copy admin.conf to user´s directory
```

```
8       copy:
9         src: /etc/kubernetes/admin.conf
10        dest: /home/vagrant/.kube/config
11        remote_src: yes
12        owner: vagrant
13        group: vagrant
14        mode: 0600
```

Listing 4.25: Creación del directorio *.kube* y copia del fichero *admin.conf*

Ahora que ya podemos interactuar con el clúster de **Kubernetes** a través de Kubectl podemos instalar el *plug-in* Calico, el cual se encargará de controlar la red interna que hemos especificado al iniciar el clúster. Para instalar Calico simplemente usamos el módulo *kubernetes.core.k8s* de Ansible. A este módulo le indicamos que queremos instalar Calico (*state: present*) y el repositorio de Github (*src: <repo>*) desde el que obtener la especificación de los objetos necesarios para su instalación, como se ve en el listado 4.26.

```
1 - name: Install calico pod network
2       become: false
3       kubernetes.core.k8s:
4         state: present
5         src: https://raw.githubusercontent.com/projectcalico/
    calico/v3.25.0/manifests/calico.yaml
```

Listing 4.26: Instalación de Calico en el clúster

Para poder añadir el resto de nodos al clúster vamos a crear los comandos de unión necesarios para ello, como se muestra en el listado 4.27. Estos comandos los copiaremos a la carpeta compartida haciendo uso del módulo *local_action* (líneas 6 y 17) de Ansible. Tendremos dos comandos: uno para nnir los nodos adicionales que formarán parte del *control plane* y otro para los nodos *worker*. En ambos casos haremos uso del comando *kubeadm join*, en el cual indicamos el *endpoint* y el puerto en el que está expuesto el clúster, un *token* que usaremos para autenticar el clúster al añadir un nuevo nodo, además de añadir un *hash* con el parámetro *discovery-token-ca-cert-hash* que usaremos para validar la configuración.

Para que los nodos secundarios del *control plane* obtengan su rol correctamente e instancien los *pods* de los componentes esenciales del *control plane*, debemos añadir el parámetro *control-plane*. También le indicamos la dirección IP en la que deben exponer el API server, usando el parámetro *apiserver-advertise-address*. Adicionalmente, añadimos una clave con la que descifrar las claves de los certificados subidos por *kubeadm init*. Esta clave se muestra por pantalla al ejecutar *kubeadm init phase upload-certs -upload-certs*. Acto seguido, la añadimos con el parámetro *certificate-key*. Por último, añadimos *ignore-preflight-errors=NumCPU* para que pueda iniciarse el nodo como parte del *control plane* sin necesitar al menos dos CPUs. Mencionar que obtenemos la IP que le queremos asignar al API server de cada nodo usando

los comandos *ip a*, el cual lista la información de las interfaces de red, y *grep*, que nos permite filtrar la salida del comando anterior (véase línea 17).

```
1  - name: Generate new join command
2        shell: kubeadm token create --print-join-command
3        register: kube_join_cmd
4
5  - name: Copy command to local file
6        local_action: copy content="{{ kube_join_cmd.stdout_lines[0]
       }}" dest="/vagrant/provisioning/join-command.sh"
7
8  - name: Generate join command for ControlPlane
9        shell: kubeadm token create --print-join-command
10       register: kube_join_cp
11
12 - name: Generate new certificate-key
13       shell: kubeadm init phase upload-certs --upload-certs | grep
       -vw -e certificate -e Namespace
14       register: cert_key
15
16 - name: Copy the command to file
17       local_action: copy content="{{ kube_join_cp.stdout_lines[0]
       }} --apiserver-advertise-address $(ip a | grep -oP 'inet
       \K[\d.]+' | grep 192.168.56.1) --control-plane --certificate-key
       {{ cert_key.stdout_lines[0] }} --ignore-preflight-errors=NumCPU
       --v=5"
       dest="/vagrant/provisioning/control_plane_join_command.sh"
```

Listing 4.27: Generación de los comandos de unión al clúster

Para terminar con este *playbook*, vamos a instalar la herramienta Helm, un gestor de paquetes para **Kubernetes**. Para ello nos descargamos el binario correspondiente usando el módulo *get_url* de Ansible. Lo extraemos usando el módulo *unarchive* y lo copiamos en la ruta */usr/bin* para que esté añadido a la variable de entorno *PATH*, dándole permisos al usuario *vagrant* para que pueda utilizarlo, como se muestra en el listado 4.28.

```
1  - name: Download helm
2        ansible.builtin.get_url:
3          dest: /home/vagrant/
4          url: https://get.helm.sh/helm-v3.14.0-linux-amd64.tar.gz
5
6  - name: Extract helm
7        ansible.builtin.unarchive:
8          src: /home/vagrant/helm-v3.14.0-linux-amd64.tar.gz
9          dest: /home/vagrant
10         mode: 0755
11
```

```
- name: Copy helm binary into path dir
      copy:
        src: /home/vagrant/linux-amd64/helm
        dest: /usr/bin/
        owner: vagrant
        group: vagrant
        mode: 0755
```

Listing 4.28: Descarga e instalación de Helm

4.2.3 *Playbooks* para añadir el resto de nodos

En esta sección tenemos dos *playbooks*, el primero destinado a los nodos secundarios del *control plane* y el segundo a los nodos *workers*.

Para configurar los nodos del *control plane*, debemos configurar Keepalived y HAProxy al igual que en el nodo principal, con ligeras diferencias. Inicialmente, copiamos la configuración que tenemos guardada de cada herramienta en el directorio compartido a su respectivo directorio destino. Una vez que lo hayamos copiado, debemos realizar algunos cambios en la configuración de Keepalived para que no haya conflictos a la hora de asignar la IP virtual. Primero debemos cambiar el estado o rol del nodo, asignándole el rol de *Backup* en este caso, como se muestra en el listado 4.29. A continuación, bajamos la prioridad del nodo al menos una unidad. Con estos cambios hacemos que el nodo principal sea el que tenga la dirección IP virtual siempre que esté disponible.

```
- name: Change keepalived state
      shell: sudo sed -i 's/state MASTER/state BACKUP/'
   /etc/keepalived/keepalived.conf

- name: Change keepalived priority
      shell: sudo sed -i 's/priority 150/priority 100/'
   /etc/keepalived/keepalived.conf
```

Listing 4.29: Modificación de la configuración de Keepalived

Después de haber modificado la configuración de Keepalived, reiniciamos las herramientas y las habilitamos para que se ejecuten al arrancar las máquinas. Posteriormente, ejecutamos el comando que generamos en el *playbook* anterior para unir nodos como parte del *control plane*, como se muestra en el listado 4.30. Al igual que con el nodo principal, creamos el directorio *.kube* en la ruta */home/vagrant/* y copiamos el fichero */etc/kubernetes/admin.conf* con los permisos correspondientes, como se mostró anteriormente en el listado 4.25. Este último paso es necesario para poder seguir interactuando con el clúster desde el resto de nodos del *control plane* en caso de que el nodo principal dejase de funcionar.

```
- name: Join cluster as ControlPlane node
      shell: sh /vagrant/provisioning/control_plane_join_command.sh
```

Listing 4.30: Agregar los nodos al *control plane*

Por último, en el segundo *playbook*, para añadir los nodos *worker* al clúster simplemente ejecutamos lo mostrado en el listado 4.31. Tan solo es necesario ejecutar el comando de unión que generamos en la sección anterior.

```
- hosts: workers
  become: yes

  tasks:
    - name: Join cluster
      shell: sh /vagrant/provisioning/join-command.sh
```

Listing 4.31: *Playbook* para añadir nodos *worker*

4.2.4 *Playbook* del servidor NFS

En este TFG, el nodo principal del clúster también hará las funciones de servidor NFS. Destacar que este *playbook* está desarrollado de forma que, si se quiere usar otro nodo como servidor NFS, tan solo es necesario cambiar la dirección IP del servidor guardada en el archivo *settings.yml* por la IP del nuevo servidor, además de otros cambios en el *playbook*.

Como vamos a usar el nodo principal como servidor NFS [36], primero le añadiremos un nuevo disco al nodo que servirá para el almacenamiento de los datos generados por las aplicaciones desplegadas dentro de **Kubernetes**, como se explicó en la sección 4.1.1. Una vez que el disco esté creado y conectado al nodo principal, tendremos que formatearlo, como podemos ver en el listado 4.32. Para ello haremos uso de tres módulos de Ansible: *community.general.parted*, *shell* y *filesystem*.

Inicialmente, creamos una partición nueva mediante el sistema de ficheros *xfs*. Para ello usamos *community.general.parted*, módulo con el que primero indicamos el esquema de particiones que vamos a usar en el disco (líneas 1-4). Con el parámetro *device* se configura el disco en el que queremos crear el esquema de particiones, en nuestro caso */dev/sdb*, y el tipo de esquema con el parámetro *label*, siendo GUID Partition Table (GPT) en este caso. A continuación, creamos la partición (líneas 6-12), nuevamente indicamos el disco con el parámetro *device*, el número de la partición con *number*, el estado deseado, que en nuestro caso es crear la partición (*present*), y la alineación óptima de la partición (*optimal*), parámetro relacionado con el rendimiento del disco. Añadir que si ya está creada la partición el módulo mandará un error a Ansible y este terminará su ejecución. Para evitar esto añadimos el parámetro *ignore_errors* al módulo, de esta forma si ya está creada la partición simplemente continua la ejecución.

Cuando esto haya terminado se comprueba que la partición esté alineada correctamente. Esta tarea no es estrictamente necesaria, se usa a modo de comprobación. Por último, con el módulo *filesystem* formateamos la partición creada (*/dev/sdb1*) con el sistema de ficheros *xfs* (líneas 17-20).

```
1  - name: Create partition and filesystem
2        community.general.parted:
3          device: /dev/sdb
4          label: gpt
5
6  - name: Create primary partition
7        community.general.parted:
8          device: /dev/sdb
9          number: 1
10         state: present
11         align: optimal
12       ignore_errors: yes
13
14 - name: Verify partition alignment
15       shell: "parted -s -- /dev/sdb align-check optimal 1"
16
17 - name: Format partition to XFS
18       filesystem:
19         fstype: xfs
20         dev: /dev/sdb1
```

Listing 4.32: Particionado y formateo del disco

Con el nuevo disco formateado, creamos el directorio con el módulo *file* de Ansible donde montaremos la única partición que acabamos de crear. Posteriormente con el módulo *ansible.posix.mount* realizamos la operación de montaje de dicha partición, el cual también configurará su montaje automático al inicio del sistema modificando el fichero */etc/fstab*, como se muestra en el listado 4.33. Destacar que esto solo es necesario en caso de que se tenga un disco secundario en la máquina que hace de servidor NFS y se quiera usar dicho disco secundario. Esta parte tendrá que ser modificada para adaptarse a la infraestructura que se quiere desplegar.

```
1  - name: Create data dir
2        file:
3          path: /data
4          state: directory
5
6  - name: Mount partition
7        ansible.posix.mount:
8          path: /data
```

```
9         src: /dev/sdb1
10        fstype: xfs
11        opts: defaults
12        state: mounted
```

Listing 4.33: Montaje del nuevo disco

Ahora que tenemos el disco preparado, creamos el directorio que queremos exportar con NFS y modificamos el fichero */etc/exports* con ayuda del módulo *lineinfile*, como se muestra en el listado 4.34. Con esta modificación se configura el directorio que se quiere exportar (*nfs_server_path*), la red a la que se quiere exportar (*nfs_server_net*) y las opciones de exportación. A continuación, ejecutamos el comando *exportfs -av*, el cual exporta todas las entradas existentes en el fichero */etc/exports*. Por último, reiniciamos el servicio NFS y lo habilitamos en el arranque junto con *rpcbind* [37], haciendo uso del módulo *service*.

```
1  - name: Export directory
2        lineinfile:
3          path: /etc/exports
4          line: "{{ nfs_server_path }}\t{{ nfs_server_net
       }}(rw,sync,no_subtree_check,no_root_squash)"
5
6  - name: Exportfs
7        shell: exportfs -av
8
9  - name: Restart nfs-server
10       service:
11         name: nfs-server
12         state: restarted
13
14 - name: Enable nfs-server
15       service:
16         name: nfs-server
17         enabled: true
18
19 - name: Enable rpcbind
20       service:
21         name: rpcbind
22         enabled: true
```

Listing 4.34: Configuración del directorio a exportar

4.2.5 *Playbook* del cliente NFS

Este *playbook* lo ejecutan todos los nodos a excepción del principal. Lo único que hacen estos nodos es montar el directorio exportado por el servidor NFS y lo añaden al fichero

/etc/fstab, como se puede ver en el listado 4.35. Es decir, realizan la función de clientes NFS.

Inicialmente, listamos desde el cliente los sistemas de ficheros que exporta el servidor NFS que hemos configurado, no es estrictamente necesario ejecutar esta tarea ya que solo se hace a modo de comprobación. Tras esto montamos en la ruta */mnt* el directorio que hemos compartido desde el servidor usando el módulo *ansible.posix.mount* de Ansible, al cual le indicamos el tipo de sistema de ficheros que debe usar con el parámetro *fstype* (*nfs* en este caso), el servidor y el directorio que se quiere montar haciendo uso del parámetro *src*. Como el estado deseado es *mounted*, además de añadir la configuración necesaria al fichero */etc/fstab*, se monta automáticamente el directorio.

```
1 - name: Update exportfs list
2       shell: showmount --exports {{ nfs_server_name }}
3
4 - name: Mount NFS share
5       ansible.posix.mount:
6         path: /mnt
7         state: mounted
8         src: "{{ nfs_server_name }}:{{ nfs_server_path}}"
9         fstype: nfs
```

Listing 4.35: Montaje del directorio compartido en los clientes

4.2.6 *Playbook* para el despliegue de aplicaciones/servicios

Este *playbook* puede ser ejecutado por cualquier nodo del *control plane*, en nuestro caso será el principal.

En el listado 4.36 se muestra la definición de las tareas descritas a continuación. Comenzamos por añadir el repositorio del proveedor de almacenamiento (nfs-subdir-external-provider) a Helm, usando el módulo *kubernetes.core.helm_repository*. A continuación, usando el módulo *kubernetes.core.helm* instalamos el proveedor, pasándole como parámetro el servidor NFS que hemos configurado y el directorio que hemos exportado. Además, se configura que cuando el proveedor sea desinstalado también elimine la SC que ha creado [38]. Ahora que tenemos el proveedor de almacenamiento instalado, junto con el PVC que añadiremos posteriormente (líneas 17-18), podemos asignar dinámicamente espacio en disco a las aplicaciones y servicios que despleguemos en el clúster de **Kubernetes**. Básicamente, se crearán volúmenes persistentes (PVs) de acuerdo a las peticiones a los PVCs que se realicen.

```
1 - name: Add Helm Repository for nfs-subdir-external-provisioner
2       kubernetes.core.helm_repository:
3         name: nfs-subdir-external-provisioner
4         repo_url:
    https://kubernetes-sigs.github.io/nfs-subdir-external-provisioner
```

```
5
6  - name: Install nfs-subdir-external-provisioner
7        kubernetes.core.helm:
8          name: nfs-subdir-external-provisioner
9          namespace: default
10         chart_ref:
       nfs-subdir-external-provisioner/nfs-subdir-external-provisioner
11         atomic: true
12         set_values:
13           - value: nfs.server={{ nfs_server_ip }}
14           - value: nfs.path={{ nfs_server_path }}
15           - value: storageClass.onDelete=true
16
17 - name: Apply kube-nfs-pvc
18       shell: kubectl apply -f
       /vagrant/provisioning/manifests/kube-nfs-pvc.yaml
```

Listing 4.36: Repositorio para aprovisionar espacio en disco y añadir PVC

Para preparar la instalación de MetalLB vamos a seguir los comandos mostrados en el listado 4.37. Comenzamos modificando el valor de *strictARP: false* por *strictARP: true* en el componente kube-proxy (ver línea 2), como se indica en la documentación de MetalLB [16]. Esta operación puede realizarse mientras el clúster está iniciado. Con este pequeño cambio hecho, instalamos MetalLB usando Kubectl, comando al cual le pasamos la URL del repositorio donde está alojado (línea 5). De la misma forma que instalamos MetalLB, instalamos el controlador de *ingress*, Ingress-nginx, en modo *LoadBalancer* (ver línea 8).

```
1  - name: Prepate MetalLB installation
2        shell: "kubectl get configmap kube-proxy -n kube-system -o yaml |
       sed -e \"s/strictARP: false/strictARP: true/\" | kubectl apply -f -
       -n kube-system"
3
4  - name: Install MetalLB
5        shell: kubectl apply -f https://raw.githubusercontent.com/
       metallb/metallb/v0.14.3/config/manifests/metallb-native.yaml
6
7  - name: Install ingress-nginx controller
8        shell:  kubectl apply -f
       https://raw.githubusercontent.com/kubernetes/ingress-nginx
       /controller-v1.10.0/deploy/static/provider/cloud/deploy.yaml
```

Listing 4.37: Instalación de MetalLB y del controlador de *ingress*

A continuación creamos, dos nuevos objetos de **Kubernetes** para MetalLB: *IPAddressPool* y *L2Advertisment*, cuyas especificaciones se muestran en el listado 4.38. El primer objeto se usa para asignar direcciones IP externas a los servicios del clúster **Kubernetes** que se des-

plieguen en modo *LoadBalancer*, y el segundo se utiliza para anunciar qué direcciones IP se han asignado y a qué servicio [39]. Para instanciar estos objetos, nuevamente utilizamos el comando *kubectl apply -f <file>*, sustituyendo *file* por el fichero en formato YAML en el que hemos guardado su especificación, el cual en nuestro caso tenemos almacenado en la carpeta compartida (*/vagrant/provisioning/manifests*).

```
1  ---
2  apiVersion: metallb.io/v1beta1
3  kind: IPAddressPool
4  metadata:
5    name: default
6    namespace: metallb-system
7  spec:
8    addresses:
9    - 192.168.56.40-192.168.56.50
10   autoAssign: true
11 ---
12 apiVersion: metallb.io/v1beta1
13 kind: L2Advertisement
14 metadata:
15   name: default
16   namespace: metallb-system
17 spec:
18   ipAddressPools:
19   - default
```

Listing 4.38: Configuración para MetalLB

Continuamos añadiendo funcionalidades al clúster instalando a continuación Metrics-server, un servicio que recoge métricas de **Kubernetes** y que permiten crear políticas de autoescalado. Como no disponemos de un certificado firmado por una autoridad certificadora, debemos añadir el argumento *kubelet-insecure-tls* para poder ejecutarlo sin obtener errores, tal y como se ve en la línea 15 el listado 4.39. Para instalar Metrics-server, descargamos primero el fichero YAML desde su repositorio [40], el cual especifica todos los objetos que son necesarios para su correcto funcionamiento. En el objeto *deployment*, apartado *args*, es donde debemos añadir el argumento mencionado previamente. Una vez hecho esto, instalamos Metrics-server mediante el comando *kubectl apply -f <file>*.

```
1  ---
2  apiVersion: apps/v1
3  kind: Deployment
4  metadata:
5    labels:
6      k8s-app: metrics-server
7    name: metrics-server
```

```
8  ..............................
9  - args:
10         - --cert-dir=/tmp
11         - --secure-port=10250
12         -
       --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
13         - --kubelet-use-node-status-port
14         - --metric-resolution=15s
15         - --kubelet-insecure-tls
```

Listing 4.39: Modificación del *deployment* de Metrics-server

Finalmente, solo nos queda instalar y configurar Prometheus y Grafana usando nuevamente Helm, como se muestra en el listado 4.40. Para ello añadimos un nuevo repositorio (prometheus-community) usando el módulo *kubernetes.core.helm_repository* de Ansible (líneas 1-4). Este repositorio nos permite instalar el kube-prometheus-stack, el cual contiene tanto Prometheus como Grafana además de otros servicios necesarios. Una vez añadido el repositorio, instalamos dicho stack con ayuda del módulo *kubernetes.core.helm* (líneas 6-15). En este módulo se configura nombre que le queremos dar, el *namespace* con el que podremos organizar los servicios, y el *chart* de Helm que queremos instalar con el parámetro *chart_ref* (línea 10). Además, con el parámetro *atomic* indicamos que se elimine la instalación en caso de que se produzca algún error, y configuramos el PVC que deben usar los servicios que van a hacer uso del almacenamiento compartido.

Para acceder a Prometheus y Grafana desde el exterior, creamos un objeto *ingress* que nos permita redireccionar las peticiones a cada servicio expuesto bajo la IP externa que MetalLB ha asignado al controlador Ingress-nginx, tal y como se muestra en la última tareas del listado 4.40. Esto se hace después de eliminar el objeto *ValidatingWebhookConfiguration ingress-nginx-admission*, ya que de lo contrario daría problemas a la hora de instanciar el nuevo objeto *ingress*. Adicionalmente, debemos editar el fichero */etc/hosts* en distribuciones Linux (*C:\Windows\System32\drivers\etc* en Windows) para añadir la IP externa y los nombres de *host* que se especifican dentro del objeto *ingress*. No hay ningún problema por tener dos servicios bajo la misma IP ya que *ingress* se encargará de dirigir el tráfico entrante hacia el destino correspondiente. La especificación del recurso *ingress* se muestra en el listado 4.41.

```
1  - name: Add Helm Repository for prometheus
2        kubernetes.core.helm_repository:
3          name: prometheus-community
4          repo_url: https://prometheus-community.github.io/helm-charts
5
6  - name: Install and deploy prometheus
7        kubernetes.core.helm:
8          name: kube-prometheus-stack
9          namespace: default
```

```
10          chart_ref: prometheus-community/kube-prometheus-stack
11          atomic: true
12          set_values:
13            - value: alertmanager.persistentVolume.existingClaim=nfs-pvc
14            - value: server.persistentVolume.existingClaim=nfs-pvc
15            - value: grafana.persistentVolume.existingClaim=nfs-pvc
16
17  - name: Delete ingress-nginx-admission ValidatingWebhookConfiguration
18        shell: kubectl delete -A ValidatingWebhookConfiguration
      ingress-nginx-admission
19
20  - name: Deploy ingress
21        shell: kubectl apply -f
      /vagrant/provisioning/manifests/kube-ingress.yaml
```

Listing 4.40: Instalar kube-prometheus-stack y recurso *ingress*

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: ingress
5  spec:
6    ingressClassName: nginx
7    rules:
8    - host: grafana.idctfg.k8s.es
9      http:
10       paths:
11       - path: /
12         pathType: Prefix
13         backend:
14           service:
15             name: kube-prometheus-stack-grafana
16             port:
17               number: 80
18   - host: prometheus.idctfg.k8s.es
19     http:
20       paths:
21       - path: /
22         pathType: Prefix
23         backend:
24           service:
25             name: kube-prometheus-stack-prometheus
26             port:
27               number: 9090
```

Listing 4.41: Especificación de las reglas del recurso *ingress*

4.3 Sistema de notificaciones

Esta parte del trabajo no pudo ser automatizada. En cuanto a Grafana, dispone de una API con la cual se puede interactuar, pero añadir algunos elementos (puntos de contacto, alertas, ...) desde esta API hace que no sean visibles o usables desde la interfaz web. En cuanto al *bot* de Telegram como solo necesitamos uno y lo vamos a reusar cada vez que se realicen cambios en el trabajo, no es necesario automatizar su despliegue.

4.3.1 *Bot* de Telegram

Para crear el *bot*, debemos abrir Telegram y buscar al usuario *BotFather*, el cual es un *bot* especial que nos permite crear y manejar nuestros propios *bots*. Solo es necesario mandarle el comando */newbot* a este usuario para crear nuestro bot. *BotFather* nos pedirá que le demos un nombre al nuevo *bot* y un *username* terminado en *_bot*. Una vez hecho este paso, nos generará un *token* con el que podremos interactuar con la API de Telegram y usar así el nuevo *bot*. En la figura 4.3 se muestra la interacción con este *bot*. A continuación, creamos un grupo de Telegram y añadimos el *bot* que acabamos de crear. Para saber cuál es el identificador del grupo, le mandamos un mensaje por el grupo y accedemos a la siguiente URL: *https://api.telegram.org/bot{your_bot_api_token}/getUpdates*. Aquí podremos ver los mensajes que han llegado al *bot* y el identificador del grupo desde el cual han sido enviados. Con el *token* y el identificador del grupo ya es posible configurar las alertas desde Grafana para que el *bot* las envíe al grupo que se ha creado.

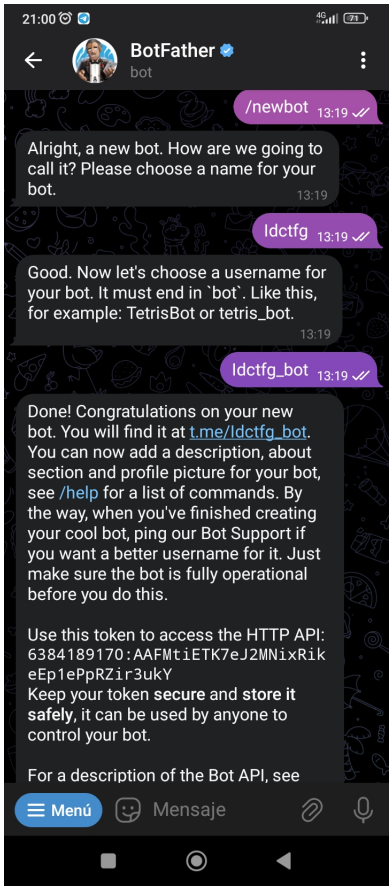


Figura 4.3: Interacción con *BotFather*

4.3.2 Configuración de Grafana

Para hacer que el *bot* envíe las notificaciones, tenemos que añadirlo como punto de contacto en la interfaz de Grafana. Para ello accedemos a Grafana a través del *hostname* que hemos especificado en el objeto *ingress* y buscamos la opción *contact points*. Para crear el nuevo punto de contacto, debemos añadir el *token* que obtuvimos al crear el *bot* y el identificador del grupo al que añadimos al *bot*. Adicionalmente, tendremos que poner un *0* en *message thread id*, para que no muestre los mensajes como una respuesta a un mensaje anterior. Una vez rellenados los campos, le damos al botón de *Test* para comprobar que funciona correctamente.

Con esto configurado ya solo falta añadir las alertas que se requieran, además de obtener y visualizar las métricas que sean convenientes desde Grafana.

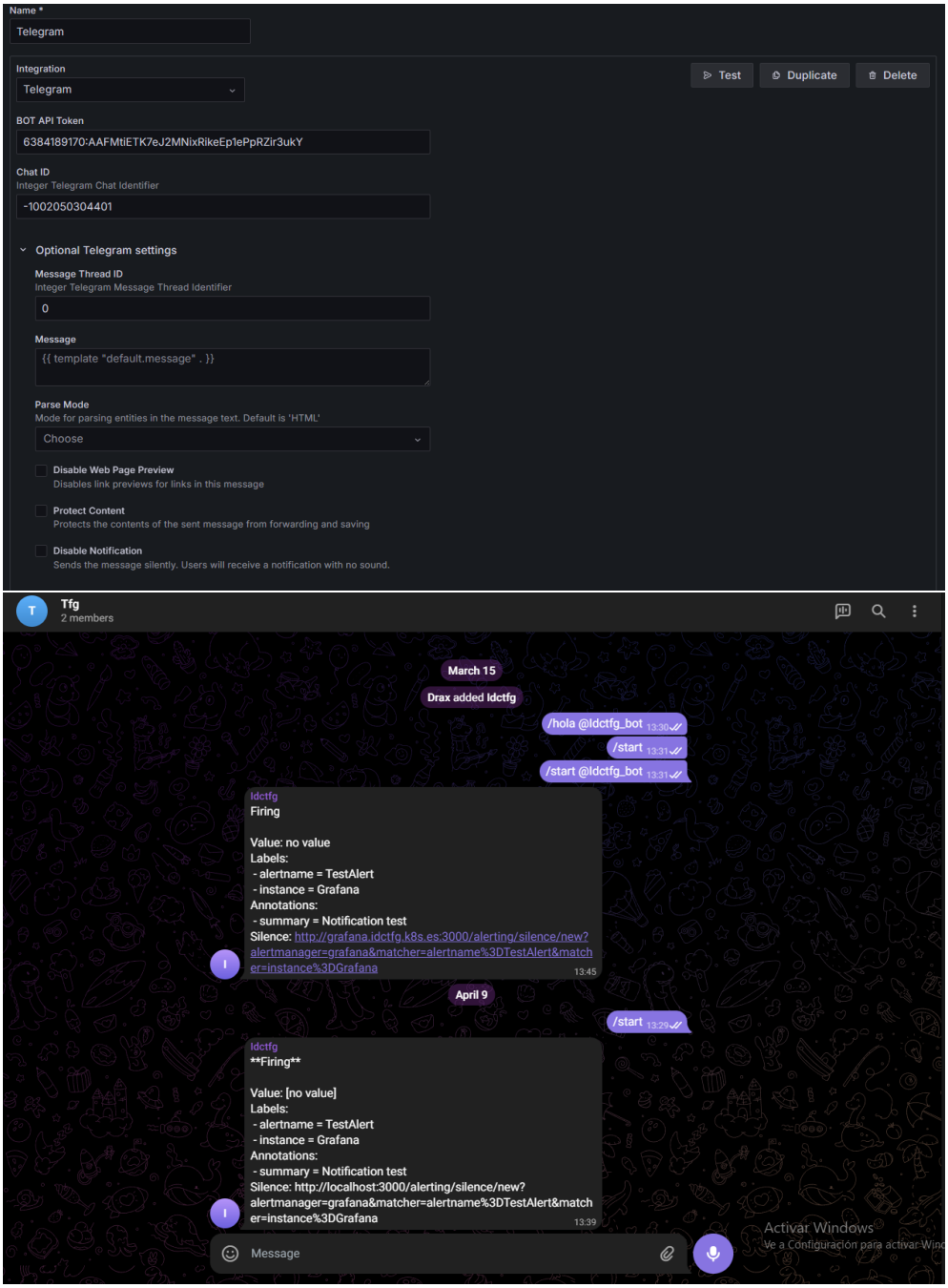


Figura 4.4: Añadir el *bot* de Telegram como punto de contacto en Grafana

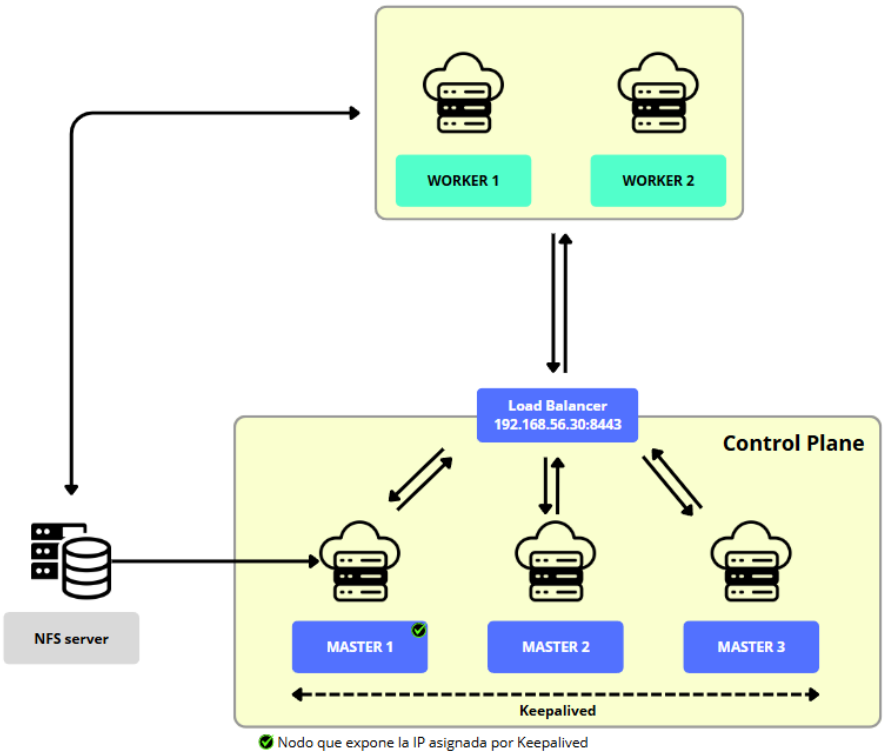


Figura 4.5: Arquitectura final

4.4 Comprobación del clúster

En esta sección comprobaremos todos los recursos que hemos instalado y configurado hasta el momento, así como los nodos y su rol en el clúster. El resultado final de la arquitectura desplegada se muestra en la figura 4.5. Los nodos *worker* se comunican con el *control plane* a través de la dirección IP asignada con Keepalived, en este caso está asignada a *Master 1*, y por el puerto que usa HAProxy para proporcionar balanceo de carga entre los distintos API server. También se conectan con el servidor NFS usando la dirección IP del nodo *Master 1*.

Para comprobar que hay conexión con el clúster ejecutamos el comando *kubectl cluster-info*, como se muestra en la figura 4.6. Esto nos devolverá la URL desde la cual se expone el clúster y su DNS. Esta URL está compuesta por el *endpoint* que se añadió al fichero */etc/hosts* anteriormente, *idc-cluster-endpoint*, que se corresponde con la IP virtual que asigna Keepalived (192.168.56.30), y con el puerto por el cual se expone, que efectivamente se corresponde con el que se indicó en la configuración de HAProxy (8443).

```
[vagrant@idc-tfg-k8s-master ~]$ kubectl cluster-info
Kubernetes control plane is running at https://idc-cluster-endpoint:8443
CoreDNS is running at https://idc-cluster-endpoint:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Figura 4.6: Ejecución de *kubectl cluster-info*

A continuación, comprobamos que los nodos se han añadido al clúster con su rol asignado correctamente. Esto lo sabremos al ejecutar el comando *kubectl get nodes -o wide*, la opción *-o wide* nos proporciona información adicional del nodo, como su dirección IP tanto interna como externa (si tiene), su versión de SO y kernel, y el *container-runtime* que está utilizando así como su versión. También comprobamos que desde todos los nodos del *control plane* podamos usar Kubectl para poder modificar el estado del clúster y que los nodos *worker* no puedan usar dicho comando, comportamiento que podemos observar en la figura 4.7.

```
[vagrant@idc-tfg-k8s-master ~]$ kubectl get nodes -o wide
NAME                 STATUS   ROLES           AGE   VERSION   INTERNAL-IP     EXTERNAL-IP   OS-IMAGE                        KERNEL-VERSION                 CONTAINER-RUNTIME
idc-tfg-k8s-master   Ready    control-plane   58m   v1.28.10  192.168.56.10   <none>        Rocky Linux 8.6 (Green Obsidian)   4.18.0-372.26.1.el8_6.x86_64   containerd://1.6.32
idc-tfg-k8s-master-2 Ready    control-plane   53m   v1.28.10  192.168.56.11   <none>        Rocky Linux 8.6 (Green Obsidian)   4.18.0-372.26.1.el8_6.x86_64   containerd://1.6.32
idc-tfg-k8s-master-3 Ready    control-plane   30m   v1.28.10  192.168.56.12   <none>        Rocky Linux 8.6 (Green Obsidian)   4.18.0-372.26.1.el8_6.x86_64   containerd://1.6.32
idc-tfg-k8s-worker-1 Ready    <none>          53m   v1.28.10  192.168.56.13   <none>        Rocky Linux 8.6 (Green Obsidian)   4.18.0-372.26.1.el8_6.x86_64   containerd://1.6.32
idc-tfg-k8s-worker-2 Ready    <none>          53m   v1.28.10  192.168.56.14   <none>        Rocky Linux 8.6 (Green Obsidian)   4.18.0-372.26.1.el8_6.x86_64   containerd://1.6.32
[vagrant@idc-tfg-k8s-master ~]$ ssh idc-tfg-k8s-master-2
Last login: Wed Jun  5 13:06:14 2024 from 192.168.56.10
[vagrant@idc-tfg-k8s-master-2 ~]$ kubectl cluster-info
Kubernetes control plane is running at https://idc-cluster-endpoint:8443
CoreDNS is running at https://idc-cluster-endpoint:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
[vagrant@idc-tfg-k8s-master-2 ~]$ exit
logout
Connection to idc-tfg-k8s-master-2 closed.
[vagrant@idc-tfg-k8s-master ~]$ ssh idc-tfg-k8s-master-3
Last login: Wed Jun  5 13:28:38 2024 from 192.168.56.10
[vagrant@idc-tfg-k8s-master-3 ~]$ kubectl cluster-info
Kubernetes control plane is running at https://idc-cluster-endpoint:8443
CoreDNS is running at https://idc-cluster-endpoint:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
[vagrant@idc-tfg-k8s-master-3 ~]$ exit
logout
Connection to idc-tfg-k8s-master-3 closed.
[vagrant@idc-tfg-k8s-master ~]$ ssh idc-tfg-k8s-worker-1
Last login: Wed Jun  5 13:06:14 2024 from 192.168.56.10
[vagrant@idc-tfg-k8s-worker-1 ~]$ kubectl cluster-info
E0605 14:02:28.947110   29611 memcache.go:265] couldn't get current server API group list: Get "http://localhost:8080/api?timeout=32s": dial tcp 127.0.0.1:8080: connect: connection refused
E0605 14:02:28.947358   29611 memcache.go:265] couldn't get current server API group list: Get "http://localhost:8080/api?timeout=32s": dial tcp 127.0.0.1:8080: connect: connection refused
E0605 14:02:28.952173   29611 memcache.go:265] couldn't get current server API group list: Get "http://localhost:8080/api?timeout=32s": dial tcp 127.0.0.1:8080: connect: connection refused
E0605 14:02:28.952345   29611 memcache.go:265] couldn't get current server API group list: Get "http://localhost:8080/api?timeout=32s": dial tcp 127.0.0.1:8080: connect: connection refused
E0605 14:02:28.965544   29611 memcache.go:265] couldn't get current server API group list: Get "http://localhost:8080/api?timeout=32s": dial tcp 127.0.0.1:8080: connect: connection refused

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
The connection to the server localhost:8080 was refused - did you specify the right host or port?
[vagrant@idc-tfg-k8s-worker-1 ~]$
```

Figura 4.7: Comprobación del estado de los nodos

Ahora que hemos comprobado que hay conexión y que todos los nodos se han añadido correctamente con su rol, toca mirar si los servicios que hemos desplegado están funcionando como es debido. Con el comando *kubectl get pod -all-namespaces* obtenemos todos los *pods* que han sido desplegados, organizados según su *namespace*. Al igual que ocurre con los nodos, si añadimos la opción *-o wide* podemos ver información adicional, como el nodo en el que se están ejecutando o su dirección IP, que puede ser la misma que la del nodo o una privada que solo puede accederse desde dentro del clúster. En la figura 4.8 observamos que todos los *pods* se encuentran funcionado correctamente. Mencionar que los dos *pods* cuyo estado es *Completed* no son *pods* realmente. Estos son *jobs*, los cuales se ejecutan solo una vez y cuando terminan su tarea aparecen como completados. Cabe mencionar que fueron ejecutados como parte del proceso de instalación de algunos servicios.

```
NAMESPACE       NAME                                                        READY   STATUS      RESTARTS        AGE
default         pod/alertmanager-kube-prometheus-stack-alertmanager-0       2/2     Running     0               11m
default         pod/kube-prometheus-stack-grafana-7cf5785ff8-v5t6l          3/3     Running     0               12m
default         pod/kube-prometheus-stack-kube-state-metrics-65594f9476-nsd5m 1/1   Running     0               12m
default         pod/kube-prometheus-stack-operator-6b4cc67b8b-7wj7r         1/1     Running     0               12m
default         pod/kube-prometheus-stack-prometheus-node-exporter-7smx4    1/1     Running     0               12m
default         pod/kube-prometheus-stack-prometheus-node-exporter-8sblw    1/1     Running     0               12m
default         pod/kube-prometheus-stack-prometheus-node-exporter-dkj6t    1/1     Running     2 (10m ago)     12m
default         pod/kube-prometheus-stack-prometheus-node-exporter-hx7wk    1/1     Running     0               12m
default         pod/kube-prometheus-stack-prometheus-node-exporter-ntzvf    1/1     Running     0               12m
default         pod/nfs-subdir-external-provisioner-64d7fd64c5-fjgcx        1/1     Running     0               19m
default         pod/prometheus-kube-prometheus-stack-prometheus-0           2/2     Running     1 (85s ago)     11m
ingress-nginx   pod/ingress-nginx-admission-create-l5tvq                    0/1     Completed   0               18m
ingress-nginx   pod/ingress-nginx-admission-patch-wwtvc                     0/1     Completed   3               18m
ingress-nginx   pod/ingress-nginx-controller-6bfd765bdc-rxtmd               1/1     Running     0               18m
kube-system     pod/calico-kube-controllers-658d97c59c-kvxhj                1/1     Running     0               23m
kube-system     pod/calico-node-49rfv                                       1/1     Running     0               20m
kube-system     pod/calico-node-lf2pp                                       1/1     Running     0               21m
kube-system     pod/calico-node-qhmrt                                       1/1     Running     0               20m
kube-system     pod/calico-node-s95d9                                       1/1     Running     0               21m
kube-system     pod/calico-node-zpdx4                                       1/1     Running     0               23m
kube-system     pod/coredns-5dd5756b68-mxhcc                                1/1     Running     0               23m
kube-system     pod/coredns-5dd5756b68-v6pxs                                1/1     Running     0               23m
kube-system     pod/etcd-idc-tfg-k8s-master                                 1/1     Running     1               23m
kube-system     pod/etcd-idc-tfg-k8s-master-2                               1/1     Running     0               21m
kube-system     pod/etcd-idc-tfg-k8s-master-3                               1/1     Running     0               21m
kube-system     pod/kube-apiserver-idc-tfg-k8s-master                       1/1     Running     8 (10m ago)     23m
kube-system     pod/kube-apiserver-idc-tfg-k8s-master-2                     1/1     Running     3 (21m ago)     21m
kube-system     pod/kube-apiserver-idc-tfg-k8s-master-3                     1/1     Running     4               21m
kube-system     pod/kube-controller-manager-idc-tfg-k8s-master              1/1     Running     9 (15m ago)     23m
kube-system     pod/kube-controller-manager-idc-tfg-k8s-master-2            1/1     Running     9               21m
kube-system     pod/kube-controller-manager-idc-tfg-k8s-master-3            1/1     Running     11              21m
kube-system     pod/kube-proxy-4lnkk                                        1/1     Running     0               21m
kube-system     pod/kube-proxy-8mdn7                                        1/1     Running     0               20m
kube-system     pod/kube-proxy-h9w6h                                        1/1     Running     0               21m
kube-system     pod/kube-proxy-hb28j                                        1/1     Running     0               20m
kube-system     pod/kube-proxy-kcvxx                                        1/1     Running     0               23m
kube-system     pod/kube-scheduler-idc-tfg-k8s-master                       1/1     Running     8 (21m ago)     23m
kube-system     pod/kube-scheduler-idc-tfg-k8s-master-2                     1/1     Running     7               21m
kube-system     pod/kube-scheduler-idc-tfg-k8s-master-3                     1/1     Running     10              21m
kube-system     pod/metrics-server-84989b68d9-62zxs                         1/1     Running     0               15m
metallb-system  pod/controller-5f56cd6f78-dtx9z                             1/1     Running     1 (17m ago)     18m
metallb-system  pod/speaker-4m9mg                                           1/1     Running     0               18m
metallb-system  pod/speaker-4sr2n                                           1/1     Running     0               18m
metallb-system  pod/speaker-58q9r                                           1/1     Running     1 (95s ago)     18m
metallb-system  pod/speaker-99fsj                                           1/1     Running     4 (9m37s ago)   18m
metallb-system  pod/speaker-s6w5l                                           1/1     Running     0               18m
```

Figura 4.8: Estado de todos los *pods* desplegados

Con los *pods* ya comprobados pasamos a los servicios. En la figura 4.9 podemos ver los servicios operativos actualmente. Lo más destacable es el servicio *ingress-nginx-controller*, el cual está configurado en modo *LoadBalancer* como se mencionó anteriormente, teniendo asignado una dirección IP externa proporcionada por MetalLB (10.111.242.67 en la figura). Por el resto, vemos que Prometheus, Grafana y Metrics-server parecen estar funcionando, aunque estos servicios los comprobaremos más adelante. MetalLB sabemos que está funcionando ya que el controlador de *ingress* tiene su dirección IP externa asignada correctamente.

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
default	service/alertmanager-operated	ClusterIP	None	<none>	9093/TCP,9094/TCP,9094/UDP	11m
default	service/kube-prometheus-stack-alertmanager	ClusterIP	10.98.164.136	<none>	9093/TCP,8080/TCP	12m
default	service/kube-prometheus-stack-grafana	ClusterIP	10.105.42.248	<none>	80/TCP	12m
default	service/kube-prometheus-stack-kube-state-metrics	ClusterIP	10.107.208.29	<none>	8080/TCP	12m
default	service/kube-prometheus-stack-operator	ClusterIP	10.99.31.68	<none>	443/TCP	12m
default	service/kube-prometheus-stack-prometheus	ClusterIP	10.102.94.14	<none>	9090/TCP,8080/TCP	12m
default	service/kube-prometheus-stack-prometheus-node-exporter	ClusterIP	10.102.216.242	<none>	9100/TCP	12m
default	service/kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	23m
default	service/prometheus-operated	ClusterIP	None	<none>	9090/TCP	11m
ingress-nginx	service/ingress-nginx-controller	LoadBalancer	10.111.242.67	192.168.56.40	80:30556/TCP,443:30582/TCP	18m
ingress-nginx	service/ingress-nginx-controller-admission	ClusterIP	10.103.218.93	<none>	443/TCP	18m
kube-system	service/kube-dns	ClusterIP	10.96.0.10	<none>	53/UDP,53/TCP,9153/TCP	23m
kube-system	service/kube-prometheus-stack-coredns	ClusterIP	None	<none>	9153/TCP	12m
kube-system	service/kube-prometheus-stack-kube-controller-manager	ClusterIP	None	<none>	10257/TCP	12m
kube-system	service/kube-prometheus-stack-kube-etcd	ClusterIP	None	<none>	2381/TCP	12m
kube-system	service/kube-prometheus-stack-kube-proxy	ClusterIP	None	<none>	10249/TCP	12m
kube-system	service/kube-prometheus-stack-kube-scheduler	ClusterIP	None	<none>	10259/TCP	12m
kube-system	service/kube-prometheus-stack-kubelet	ClusterIP	None	<none>	10250/TCP,10255/TCP,4194/TCP	11m
kube-system	service/metrics-server	ClusterIP	10.106.90.243	<none>	443/TCP	15m
metallb-system	service/webhook-service	ClusterIP	10.96.149.215	<none>	443/TCP	18m

Figura 4.9: Servicios desplegados

Todos estos servicios y *pods* que hemos comprobado hasta el momento son desplegados en su mayoría junto con su *deployment* y *replicaset* correspondientes, como se observa en la figura 4.10. Para ver estos recursos podemos ejecutar el comando *kubectl get all -all-namespaces* o *kubectl get deployment -all-namespaces* y *kubectl get replicaset -all-namespaces*. Como ya se explicó anteriormente (sección 2.3.2), cuando se crea un *deployment* también se crea un *replicaset* por defecto.

NAMESPACE	NAME	READY	UP-TO-DATE	AVAILABLE	AGE
default	deployment.apps/kube-prometheus-stack-grafana	1/1	1	1	12m
default	deployment.apps/kube-prometheus-stack-kube-state-metrics	1/1	1	1	12m
default	deployment.apps/kube-prometheus-stack-operator	1/1	1	1	12m
default	deployment.apps/nfs-subdir-external-provisioner	1/1	1	1	19m
ingress-nginx	deployment.apps/ingress-nginx-controller	1/1	1	1	18m
kube-system	deployment.apps/calico-kube-controllers	1/1	1	1	23m
kube-system	deployment.apps/coredns	2/2	2	2	23m
kube-system	deployment.apps/metrics-server	1/1	1	1	15m
metallb-system	deployment.apps/controller	1/1	1	1	18m

NAMESPACE	NAME	DESIRED	CURRENT	READY	AGE
default	replicaset.apps/kube-prometheus-stack-grafana-7cf5785ff8	1	1	1	12m
default	replicaset.apps/kube-prometheus-stack-kube-state-metrics-65594f9476	1	1	1	12m
default	replicaset.apps/kube-prometheus-stack-operator-6b4cc67b8b	1	1	1	12m
default	replicaset.apps/nfs-subdir-external-provisioner-64d7fd64c5	1	1	1	19m
ingress-nginx	replicaset.apps/ingress-nginx-controller-6bfd765bdc	1	1	1	18m
kube-system	replicaset.apps/calico-kube-controllers-658d97c59c	1	1	1	23m
kube-system	replicaset.apps/coredns-5dd5756b68	2	2	2	23m
kube-system	replicaset.apps/metrics-server-84989b68d9	1	1	1	15m
metallb-system	replicaset.apps/controller-5f56cd6f78	1	1	1	18m

Figura 4.10: *Deployments* y *replicasets* desplegados

Para probar que tanto Prometheus como Grafana están funcionando, debemos conectarnos a ellos a través de la dirección IP asignada al controlador de *ingress* (modificando para ello */etc/hosts*). En la figura 4.11, la dirección IP se resuelve con dos nombres, ya que *ingress* se encargará de filtrar el tráfico hacia el servicio correspondiente. Ahora que tenemos todo listo podemos ir al navegador y probar a conectarnos a ambos *hosts* para comprobar que tanto los servicios de Prometheus y Grafana como *ingress* funcionan correctamente, como se ve en la figura 4.12.

Ahora que sabemos que ambos funcionan, vamos a probar el *bot* de Telegram. Como se vio en la figura 4.4, después de añadir el *contact point* desde la web de Grafana y probar si

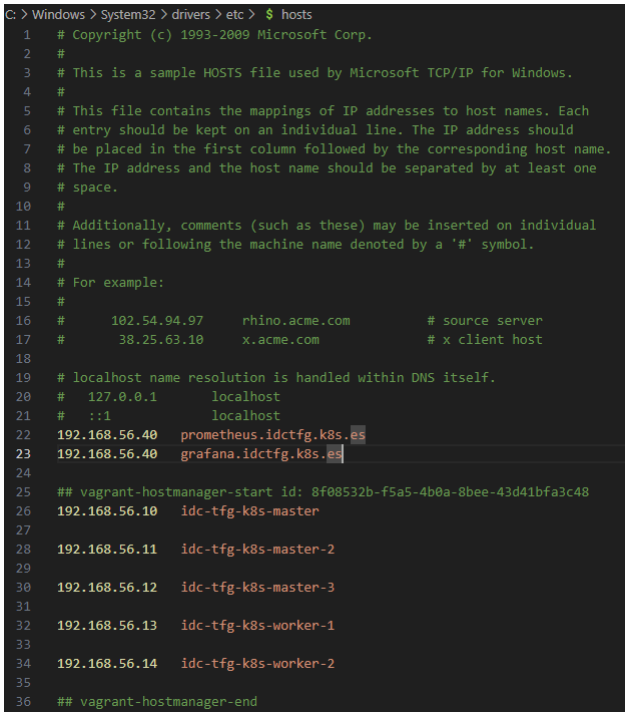


Figura 4.11: Modificación del fichero *etc/hosts* de Windows

realmente puede mandar notificaciones, vemos que el grupo que se ha creado para este trabajo recibe mensajes del *bot* con información sobre una alerta de prueba. Una vez que sabemos que el *bot* envía las alertas del clúster, procedemos a crear nuestras propias alertas. En la imagen superior de la figura 4.13 se muestra un ejemplo de una posible alerta. Para crearla, vamos al apartado de *Alerting* de Grafana, donde tendremos que seleccionar la métrica que queremos monitorizar y las condiciones para que salte la alerta. Podemos añadirle etiquetas para que nos ayude a organizar y filtrar mejor las alertas. También tendremos que crear un directorio donde se guardan las alertas creadas y un grupo de evaluación, el cual sirve para indicar el tiempo mínimo que se debe esperar entre cada evaluación. También podemos añadirle un resumen y una descripción, así como cualquier campo adicional que queramos. Sumado a esto, cuando se resuelve un incidente relacionado con una alerta, se envía adicionalmente un mensaje notificando que se ha resuelto la alerta, como se muestra en la imagen inferior de la figura 4.13, en la cual vemos como llega una alerta de uso excesivo de CPU y al cabo de un rato se manda el mensaje de que se ha resuelto.

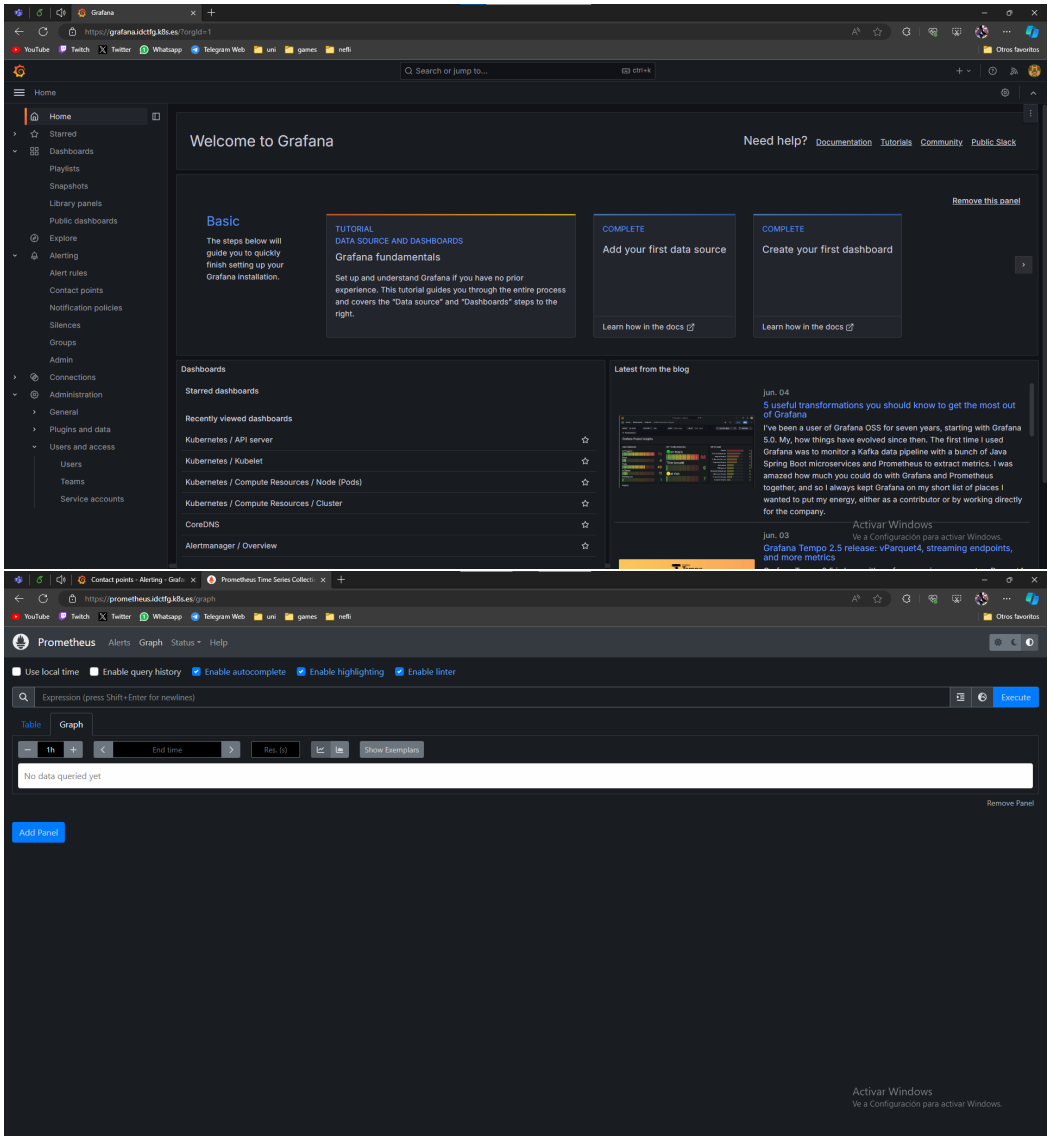


Figura 4.12: Comprobación de los servicios de Grafana y Prometheus

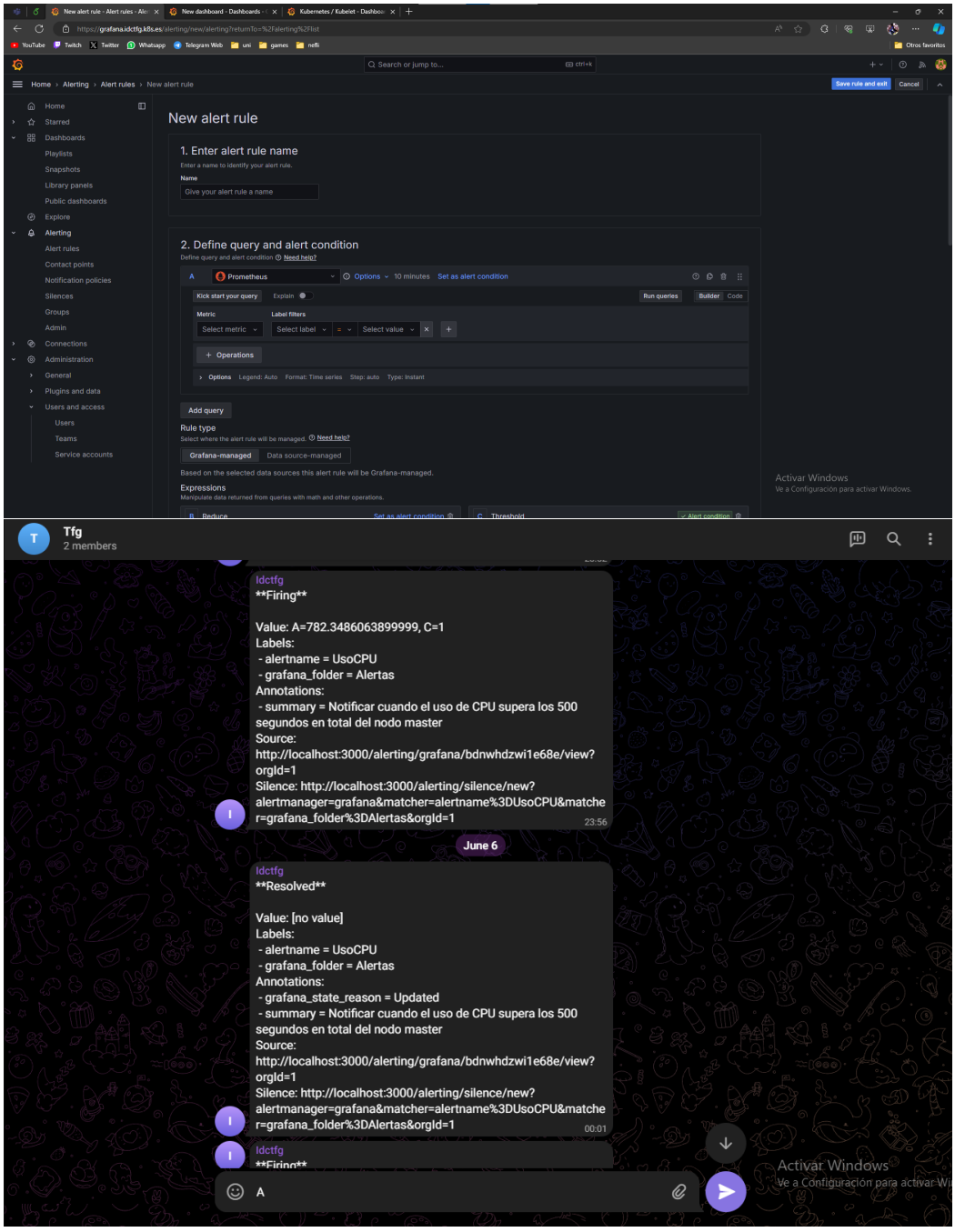


Figura 4.13: Creación de una alerta en Grafana y envío de mensajes cuando salta una alerta y cuando se resuelve

Usando Prometheus y Grafana, además de alertas podemos visualizar el estado del clúster con una gran variedad de métricas. Por defecto, la instalación de Prometheus y Grafana usando Helm configura varios paneles y métricas que nos muestran información relevante del clúster.

Por ejemplo, la figura 4.14 muestra el uso de CPU y memoria de cada *pod* en cada nodo y los objetos que se encuentran en ejecución, como pueden ser el total de nodos, *pods*, el total de *containers* que se ejecutan dentro de los *pods*, y mucho más.

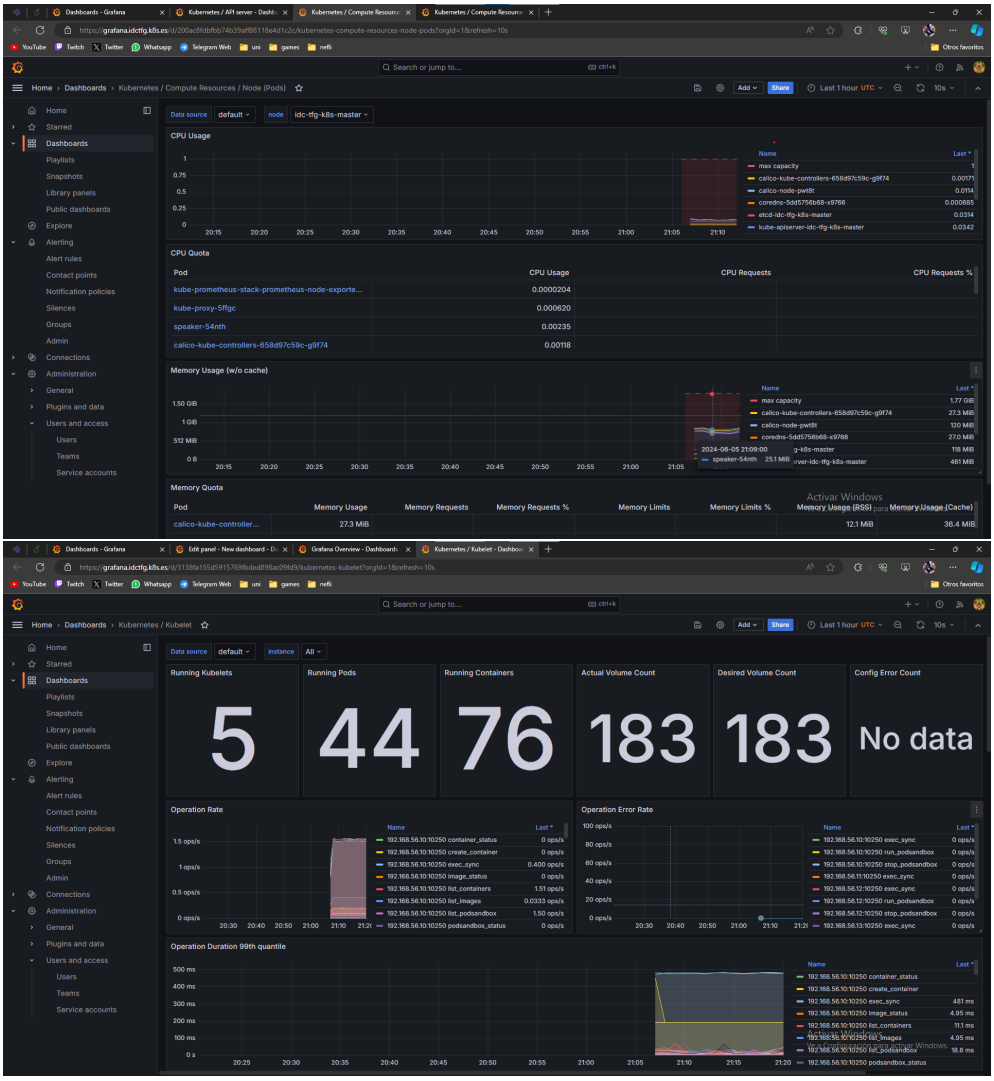


Figura 4.14: Uso de CPU de cada *pod* desplegado en el nodo principal y elementos ejecutándose en el clúster

Además de los paneles que ya están creados por defecto, podemos crear nuestros propios paneles con la información que nos resulte más útil y representarla como sea más conveniente, ya que Grafana dispone de una gran variedad de gráficos que podemos personalizar, hasta cierto punto. Como ejemplo hemos seleccionado todas las peticiones que se han hecho al PVC que hemos creado (véase la figura 4.15) y que han devuelto el código *200 ok*. Escogemos esta ya que de momento no se ha visto ninguna petición que no devuelva otro código, en caso

de querer comprobar si alguna ha devuelto otro código de error solo sería necesario cambiar la evaluación de la etiqueta *code*.

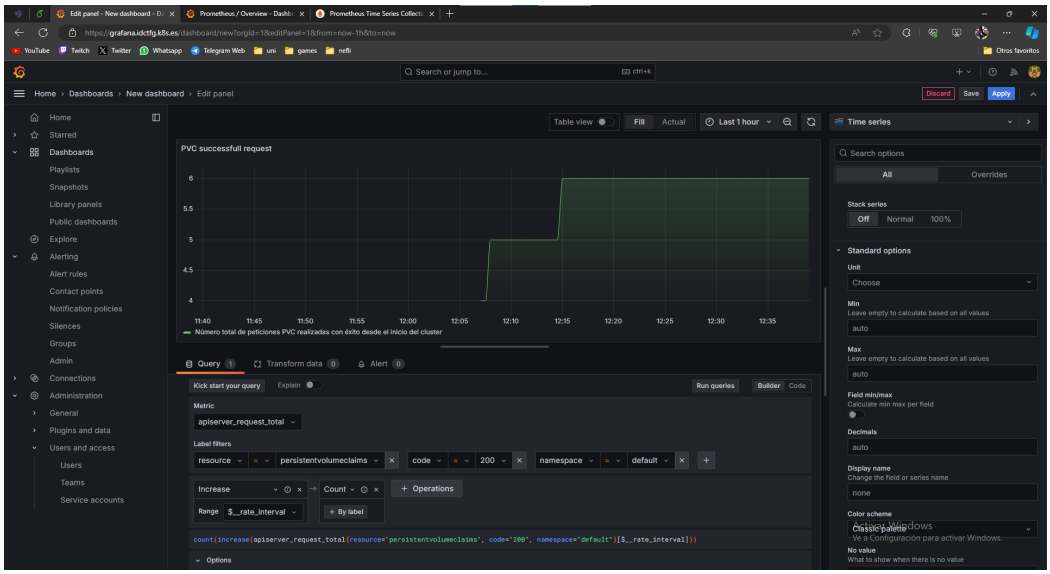


Figura 4.15: Número total de peticiones PVC realizadas con éxito

A continuación, vamos a comprobar que el autoescalado horizontal de recursos funciona correctamente. Como se muestra en la figura 4.16, vamos a seguir el ejemplo que proporciona la documentación de **Kubernetes** [41]. Ejecutando el comando *kubectl apply -f https://k8s.io/examples/application/php-apache.yaml*, se crearán todos los recursos necesarios para desplegar un servidor web Apache con soporte para PHP.

```
[vagrant@idc-tfg-k8s-master ~]$ kubectl get pods
NAME                                                READY   STATUS    RESTARTS      AGE
alertmanager-kube-prometheus-stack-alertmanager-0   2/2     Running   0             67m
kube-prometheus-stack-grafana-7cf5785ff8-9jz9l      2/3     Running   0             68m
kube-prometheus-stack-kube-state-metrics-84958579f9-xdrpx   1/1     Running   0             68m
kube-prometheus-stack-operator-7d5857db64-7j24n     1/1     Running   0             68m
kube-prometheus-stack-prometheus-node-exporter-92tlk   1/1     Running   0             68m
kube-prometheus-stack-prometheus-node-exporter-jpsrl   1/1     Running   0             68m
kube-prometheus-stack-prometheus-node-exporter-ntrkr   1/1     Running   0             68m
kube-prometheus-stack-prometheus-node-exporter-rz89t   1/1     Running   1 (17m ago)   68m
kube-prometheus-stack-prometheus-node-exporter-t9cm9   1/1     Running   0             68m
nfs-subdir-external-provisioner-64d7fd64c5-lvdxj    1/1     Running   1 (18m ago)   77m
php-apache-598b474864-nnc7k                         0/1     Pending   0             6s
prometheus-kube-prometheus-stack-prometheus-0       2/2     Running   2 (67s ago)   67m
```

Figura 4.16: Despliegue del servidor web Apache

Cuando Apache esté en funcionamiento, ejecutamos el siguiente comando: *kubectl autoscale deployment php-apache –cpu-percent=50 –min=1 –max=10* para crear una regla de autoescalado. Esta regla indica que cuando el uso de CPU del *pod* supere el 50%, se crearán nuevas réplicas para poder hacer frente a la demanda con un máximo de 10 *pods* creados. En la figura

4.17 podemos ver como se ha creado correctamente la regla.

```
[vagrant@idc-tfg-k8s-master ~]$ kubectl get hpa
NAME         REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   0%/50%    1         10        1          3m11s
```

Figura 4.17: Comprobación que la regla de autoescalado se ha creado correctamente

Ahora, desde un terminal ejecutamos el comando del listado 4.42, el cual aumenta la carga (uso de CPU) del *pod* con el objetivo de demostrar que la regla se aplica. Mientras tanto ejecutamos en otro terminal el comando *kubectl get hpa php-apache –watch*, para observar como aumenta la carga del *pod* y se crean las nuevas réplicas de acuerdo a la regla que hemos definido. Para detener este comando solo hace falta teclear la combinación *Ctrl+C*.

```
1 kubectl run -i --tty load-generator --rm --image=busybox:1.28
    --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O-
    http://php-apache; done"
```

Listing 4.42: Comando para generar carga en el servidor web

En la figura 4.18 podemos observar que el autoescalado funciona correctamente: cuando aumentamos la carga del *pod* que ejecuta el servidor web, se crean automáticamente nuevas réplicas para hacer frente a la demanda (hasta 6 se llegan a crear, ver parte izquierda de la imagen), y dichas réplicas se eliminan cuando disminuye la carga quedando solo el *pod* inicial.

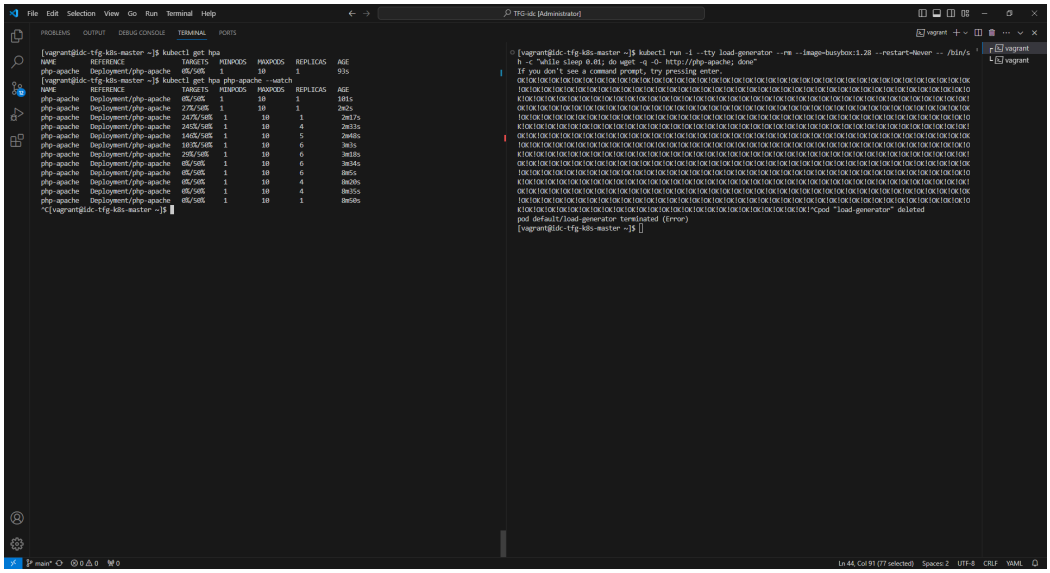


Figura 4.18: Prueba del autoescalado horizontal

Por último, nos falta comprobar que cuando el nodo que expone la dirección IP virtual no está disponible se le asigne a otro nodo del *control plane*. Esto lo comprobaremos simulando

un fallo del nodo principal, para lo cual suspendemos temporalmente la ejecución de su VM mediante el comando de Vagrant: *vagrant suspend master*, y esperando a que se reasigne la IP a alguno de los otros dos nodos del *control plane*. Como se ve en la figura 4.19, cuando la IP está en el nodo *Master* tenemos conexión con el clúster, pues está operativo en este momento. Al suspender este nodo, comprobamos que el nodo *Master-2* no tiene la IP pero seguimos teniendo conexión con el clúster, lo que significa que es el nodo *Master-3* el que expone la IP.

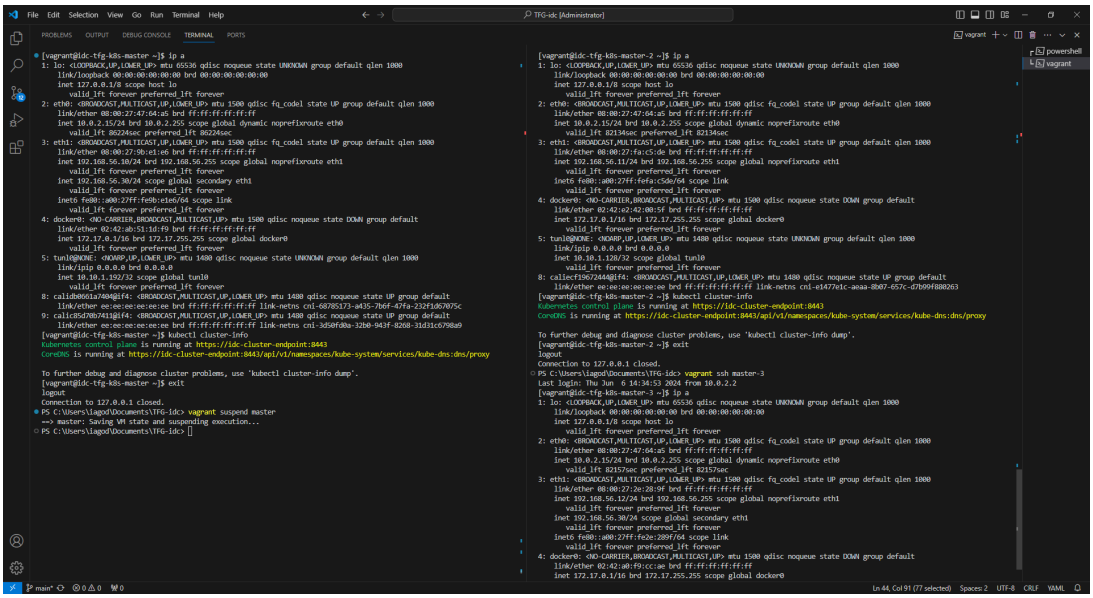


Figura 4.19: Comprobación de que la IP virtual cambia de nodo

Por tanto, hemos visto que la IP es automáticamente asignada a otro nodo del *control plane* cuando el nodo principal no está disponible. Ahora nos falta comprobar lo contrario, que cuando el nodo principal vuelva a estar en línea, sea éste el que vuelve a exponer la IP. Con el comando *vagrant resume master* reanudamos de nuevo la VM del nodo principal y realizamos la comprobación. En la figura 4.20 podemos observar que efectivamente cuando el nodo principal vuelve a estar disponible, es el nodo el encargado de exponer la IP virtual.

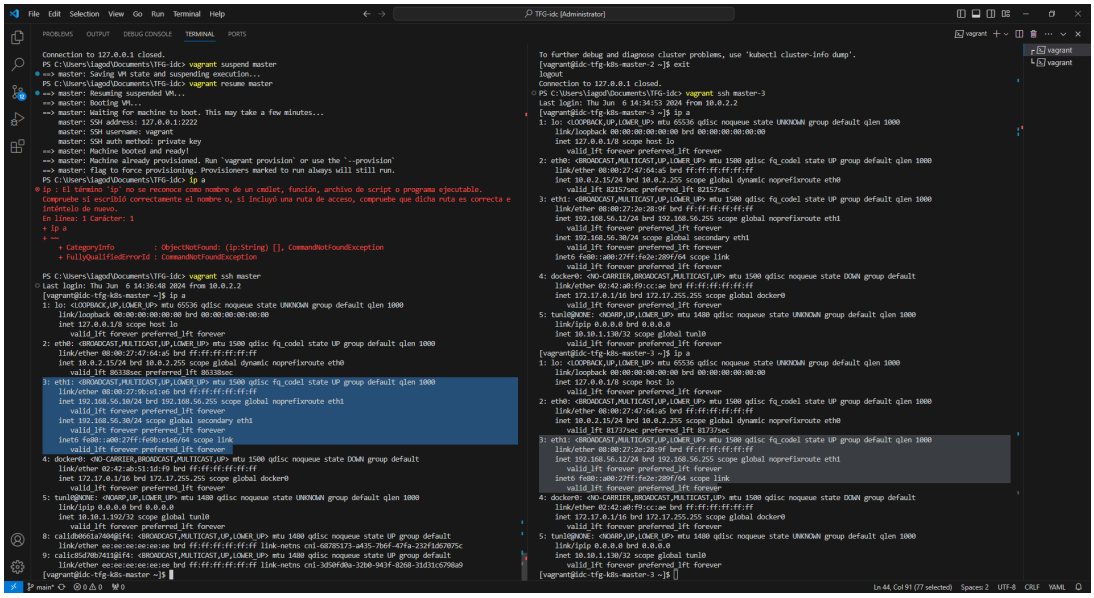


Figura 4.20: Comprobación de que la IP virtual vuelve al nodo principal

Capítulo 5

Conclusiones

Una vez finalizado el TFG, se puede afirmar que se ha logrado cumplir con los objetivos propuestos inicialmente para el despliegue automatizado en infraestructura local de un clúster de **Kubernetes** completamente funcional y configurado en alta disponibilidad. Todo ello gracias a la combinación de múltiples herramientas como por ejemplo Ansible, la cual facilita la replicación del despliegue y la configuración del clúster de una manera declarativa, así como Vagrant y Packer que, en conjunto, simplifican el proceso de gestionar varias VMs y preconfigurarlas de forma que aumenten el grado de automatización.

Personalmente, ha sido un placer profundizar en el uso de una plataforma de orquestación de contenedores tan utilizada en la actualidad por las empresas como es **Kubernetes**, o en las herramientas relacionadas con el paradigma IaC que se imparten con poca profundidad en el grado, resultándome muy fascinante el poder desplegar de forma replicable, amigable y automatizada tanto recursos de infraestructura como aplicaciones y servicios.

Destacar también que ha sido muy satisfactorio poder agrupar y hacer uso de los conocimientos adquiridos durante el grado en materias como **Administración de Infraestructuras y Sistemas Informáticos**, asignatura en la que se abordó el paradigma IaC y el uso de la mayoría de las herramientas que se han utilizado en este TFG, **Ingeniería de Infraestructuras Informáticas**, en la cual aprendí acerca de la alta disponibilidad y la importancia de incorporar redundancia en los sistemas críticos, y **Administración de Sistemas Operativos**, materia donde profundicé en varias distribuciones de Linux como Fedora o Debian, además de otros sistemas como Solaris u OpenBSD. También se han puesto en práctica conocimientos sobre la configuración de la red que aprendí en las asignaturas **Redes** y **Legislación y Seguridad Informática**.

Por último, mencionar que cualquiera que quiera puede desplegar este mismo clúster a través del siguiente enlace a GitHub: `https://github.com/iagoDominguezCamean`.

5.1 Trabajo futuro

Tras haber cumplido con los objetivos principales del trabajo, algunas mejoras a futuro que se podrían realizar, entre otras, son:

- **Mejora de la seguridad**. Como se mencionó anteriormente, SELinux se puso en modo permisivo ya que tenía algún tipo de conflicto con HAproxy y no permitía que este se expusiera en el puerto 6443 aunque no hubiese otro servicio usando dicho puerto. Sería conveniente identificar y solucionar el problema entre ambos para poder configurar SELinux en un modo más restrictivo. Adicionalmente, sería recomendable hacer uso de certificados SSL/TLS para los servicios desplegados y en general seguir buenas prácticas en cuanto a seguridad se refiere.

- **Nube híbrida**. Realizar un despliegue de nube híbrida para aprovechar alguna de las ventajas del despliegue en la nube como pueden ser la escalabilidad y resiliencia del clúster, permitiéndonos crear estrategias para *backup* de forma más sencilla. Además, usar la nube nos abre la posibilidad de hacer uso de servicios gestionados y/o reducir los costes de disponer de infraestructura on-premises.

- **Actualizar a RHEL 9**. Como se indicó al inicio del capítulo de desarrollo, inicialmente se había pensado en usar la ultima versión de RHEL, pero se descartó debido a problemas con el *container runtime*. Sería interesante tratar de resolver estos errores o usar otro *container runtime*, como puede ser CRI-O, para desplegar el clúster en un SO más actual. Aunque RHEL 8 será soportado hasta 2029, usar la versión 9 nos garantizaría dicho soporte hasta 2032.

Apéndices

Lista de acrónimos

AKS Azure Kubernetes Service. 2, 8

AWS Amazon Web Service. 2

BFD Bidirectional Forwarding Detection. 19

CNI Container Network Interface. 16, 43

CRI Container Runtime Interface. 10

CRUD Create, Read, Update and Delete. 9

EKS Elastic Kubernetes Service. 2, 8

GPG GNU Privacy Guard. 30, 34

GPT GUID Partition Table. 47

HA High Availability. 1, 2

HCL HashiCorp Configuration Language. 6, 32

HPA Horizontal Pod Autoscaler. 17

IaC Infrastructure as Code. 1, 2, 5, 6, 25, 70

JSON JavaScript Object Notation. 6

NFS *Network File System.* 1, 2, 18, 23, 26, 28, 30, 36, 47–50, 58

PV Persistent Volume. 17, 18, 50

PVC Persistent Volume Claim. 17, 18, 50, 53, 65

RHEL Red Hat Entreprise Linux. 25, 31, 37, 71

SC StorageClass. 17, 18, 50

TFG Trabajo de Final de Grado. 1, 3, 4, 22, 25, 47, 70

TI Tecnologías de la Información. 2, 5

VM Virtual Machine. 4, 5, 26, 27, 30, 68

VMs Virtual Machines. 2, 5, 14, 22, 23, 26, 27, 32, 70

VPA Vertical Pod Autoscaler. 17

VRRP Virtual Router Redundancy Protocol. 19, 40

YAML Yet Another Markup Language. 6, 12

Glosario

agentless Tipo de arquitectura en la que no es necesario instalar software en el equipo cliente. 7

backend Parte de una aplicación que se expone a través de un servicio, normalmente es donde se lleva a cabo toda la lógica de negocio de la aplicación. 10

bash Interfaz de usuario en linea de comandos, así como un lenguaje de scripting. 6

bot Proceso que realiza acciones predefinidas automáticamente, en este caso manda mensajes a traves de la API de Telegram. 3

box Se trata de una imagen con un sistema operativo preinstaldo y listo para ser usado por Vagrant. 3

cgroups Característica del kernel de Linux, que añade limitaciones a un conjunto de procesos. 13

clúster Es un conjunto de ordenadores interconectados. 1

iptables Software que sirve como cortafuegos. 10

namespace Los espacios de nombres se utilizan para organizar el código en grupos lógicos y prevenir colisiones de nombres, especialmente cuando un proyecto incluye varias bibliotecas o módulos. 10

pip3 Software usado para la instalación de módulos de Python. 28

playbook Es un fichero que contiene las tareas que debe realizar Ansible. 3

plug-in Aplicación o programa que permite extender la funcionalidades de otro sin modificar el código. 9

PromQL Es un lenguaje de consultas para el sistema de monitorización Prometheus. Está diseñado para crear consultas poderosas pero sencillas para gráficos, alertas o series temporales derivadas.. 20

script Archivo normalmente escrito en bash que ejecuta la secuencia de comandos especificados dentro del archivo. 6

Bibliografía

[1] kubernetes.io, "Kubernetes overview." [En línea]. Disponible en: https://kubernetes.io/docs/concepts/overview/

[2] "Why Kubernetes? benefits of using Kubernetes." [En línea]. Disponible en: https://www.geeksforgeeks.org/why-kubernetes-benefits-of-using-kubernetes/

[3] J. Cuesta, "Los 5 orquestadores de contenedores que deberías conocer," *OpenExpo Europe*. [En línea]. Disponible en: https://openexpoeurope.com/es/los-5-orquestadores-de-contenedores-que-deberias-conocer/

[4] A. Cloud, "Kubernetes vs Swarm vs Mesos: ¿Cuál es el mejor orquestador de contenedores?" [En línea]. Disponible en: https://ausum.cloud/kubernetes-swarm-mesos-mejor-orquestador-contenedores/

[5] Oracle, "Welcome to Virtualbox.org!" [En línea]. Disponible en: https://www.virtualbox.org/wiki/WikiStart

[6] S. Susnjara, "What are containers?" [En línea]. Disponible en: https://www.ibm.com/topics/containers

[7] T. Rodríguez, "De Docker a Kubernetes: entendiendo qué son los contenedores y por qué es una de las mayores revoluciones de la industria del desarrollo," *Xataka*, 10 Septiembre 2019. [En línea]. Disponible en: https://www.xataka.com/otros/docker-a-kubernetes-entendiendo-que-contenedores-que-mayores-revoluciones-industria-desarrollo

[8] M. Améndola, "Contenedores de software: Qué son y qué ventajas ofrecen." [En línea]. Disponible en: https://openwebinars.net/blog/contenedores-de-software-que-son-y-que-ventajas-ofrecen/

[9] redhat.cpm, "What is infrastructure as code (IaC)?" [En línea]. Disponible en: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac

[10] HashiCorp, "What is Vagrant?" [En línea]. Disponible en: https://developer.hashicorp.com/vagrant

[11] "Documentación de Vagrantfile." [En línea]. Disponible en: https://developer.hashicorp.com/vagrant/docs/vagrantfile

[12] HashiCorp, "What is Packer?" [En línea]. Disponible en: https://developer.hashicorp.com/packer

[13] M. DeHaan, "What is Ansible?" [En línea]. Disponible en: https://www.ansible.com/

[14] "Kubernetes network model." [En línea]. Disponible en: https://v1-28.docs.kubernetes.io/docs/concepts/services-networking/#the-kubernetes-network-model

[15] kubernetes.github.io, "A pure software solution: MetalLB." [En línea]. Disponible en: https://kubernetes.github.io/ingress-nginx/deploy/baremetal/#a-pure-software-solution-metallb

[16] metallb.universe.tf, "Instalando MetalLB." [En línea]. Disponible en: https://metallb.universe.tf/installation/

[17] L. Briggs, "What is Calico?" [En línea]. Disponible en: https://www.tigera.io/blog/kubernetes-networking-with-calico/

[18] Kubernetes, "Ingress nginx controller." [En línea]. Disponible en: https://github.com/kubernetes/ingress-nginx/tree/main

[19] kubernetes.io, "Metrics-server overview." [En línea]. Disponible en: https://github.com/kubernetes-sigs/metrics-server#readme

[20] kubernetes sigs, "Kubernetes NFS subdir external provisioner." [En línea]. Disponible en: https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner

[21] haproxy.org, "HAProxy description." [En línea]. Disponible en: https://www.haproxy.org/#desc

[22] keepalived.org, "What is Keepalived ?" [En línea]. Disponible en: https://keepalived.org/

[23] helm.sh, "What is Helm?" [En línea]. Disponible en: https://helm.sh/

[24] prometheus.io, "Prometheus overview." [En línea]. Disponible en: https://prometheus.io/docs/introduction/overview/

[25] grafana.com, "Documentación de Grafana." [En línea]. Disponible en: https://grafana.com/

[26] "Sueldo medio para ingeniero de sistemas." [En línea]. Disponible en: https://es.talent.com/salary?job=ingeniero+de+sistemas

[27] "Soporte rhel 8." [En línea]. Disponible en: https://access.redhat.com/support/policy/updates/errata

[28] "Repositorio de vagrant-hostmanager." [En línea]. Disponible en: https://github.com/devopsgroup-io/vagrant-hostmanager

[29] "Repositorio de vagrant-vbguest." [En línea]. Disponible en: https://github.com/dotless-de/vagrant-vbguest

[30] "Installing kubeadm." [En línea]. Disponible en: https://v1-28.docs.kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#installing-runtime

[31] C. Zaccagnino, "Does Kubernetes support SELinux?" *DEV*, 7 jul 2022. [En línea]. Disponible en: https://dev.to/carminezacc/does-kubernetes-support-selinux-3oop

[32] "Configuring cgroups drivers." [En línea]. Disponible en: https://kubernetes.io/docs/setup/production-environment/container-runtimes/#cgroup-drivers

[33] "Swap-off why is it necessary?" [En línea]. Disponible en: https://discuss.kubernetes.io/t/swap-off-why-is-it-necessary/6879

[34] Kubernetes, "High availability considerations." [En línea]. Disponible en: https://github.com/kubernetes/kubeadm/blob/main/docs/ha-considerations.md

[35] P. Kumar, "How to setup Kubernetes(k8s) cluster in HA with kubeadm," *LinuxTechi*, December 29, 2020. [En línea]. Disponible en: https://www.linuxtechi.com/setup-highly-available-kubernetes-cluster-kubeadm/#:~:text=How%20to%20Setup%20Kubernetes%20%28k8s%29%20Cluster%20in%20HA,Install%20Kubeadm%2C%20kubelet%20and%20kubectl%20...%20M%C3%A1s%20elementos

[36] J. Mutai, "Install and configure NFS server on RHEL 8 / Centos 8," *Computing for Geeks*, August 22, 2023. [En línea]. Disponible en: https://computingforgeeks.com/install-and-configure-nfs-server-on-centos-rhel/

[37] "NFS y rpcbind." [En línea]. Disponible en: https://access.redhat.com/documentation/es-es/red_hat_enterprise_linux/8/html/deploying_different_types_of_servers/nfs-and-rpcbind_exporting-nfs-shares

[38] H. Bayraktar, "How to setup dynamic NFS provisioning in a Kubernetes cluster," *Medium.com*, Nov 3, 2023. [En línea]. Disponible en: https://hbayraktar.medium.com/how-to-setup-dynamic-nfs-provisioning-in-a-kubernetes-cluster-cbf433b7de29#:~:text=Dynamic%20NFS%20storage%20provisioning%20in,intervention%20or%20pre-provisioned%20storage.)

[39] "Configuración de MetalLB." [En línea]. Disponible en: https://metallb.universe.tf/configuration/

[40] "Especificación de Metrics-server." [En línea]. Disponible en: https://github.com/kubernetes-sigs/metricsserver/releases/latest/download/components.yaml

[41] kubernetes.io, "HorizontalPodAutoscaler walkthrough." [En línea]. Disponible en: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/#run-and-expose-php-apache-server