UNIVERSIDAD
COMPLUTENSE
MADRID

FACULTAD DE INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

TRABAJO DE FIN DE GRADO

TÍTULO: Despliegue de SQL Server sobre Kubernetes
TITLE: Deploying SQL Server on Kubernetes

AUTOR: Carlos Moisés Gil Solanas
DIRECTOR: Fernando Sáenz Pérez

CURSO ACADÉMICO: 2019-2020
CONVOCATORIA: Junio

Agradecimientos

Gracias a mis padres por haberme ayudado y apoyado durante todo el proceso de desarrollo de este trabajo.

Gracias a los profesores que he tenido a lo largo de mis estudios por ayudarme a obtener los conocimientos necesarios para poder afrontar este trabajo.

Gracias a mi director, Fernando Sáenz Pérez, por toda la ayuda durante la realización del trabajo.

Por último gracias a Enrique Catalá, quien me ayudó en la elección y el desarrollo de la idea de este trabajo.

Resumen

En pleno auge de los microservicios en las Tecnologías de la Información han ido surgiendo una serie de problemas que resolver. Uno de esos problemas es, sin duda, el de la orquestación y mantenimiento de estos microservicios. Para resolver este problema nace Kubernetes. Además, en torno a esta nueva tecnología surgen interrogantes. ¿Para qué puede ser usado? ¿*Cloud* u *On Premise*? ¿Es recomendable su uso para la gestión de bases de datos? ¿Cómo se puede saber si va a responder a nuestras necesidades?

Este trabajo de fin de grado tiene como objetivo intentar ayudar a responder a estas preguntas. Para abordar estos temas es conveniente entender bien la tecnología, así que debemos estructurar su análisis de manera que sea posible entender cada parte y también el conjunto. Por ello el trabajo constará de tres partes.

La primera parte se centrará en introducir la tecnología de Kubernetes, entendiendo para qué sirve. Se explicarán de manera clara los principales conceptos de la tecnología y se describirán cómo funcionan y qué debemos tener en cuenta para su uso. En la segunda parte analizaremos los pros y contras de utilizar esta tecnología en *Cloud* u *On Premise*. Para la plataforma *Cloud* haremos uso de Microsoft Azure, en concreto usaremos una cuenta de Microsoft Azure for Students.

Por último, en tercer lugar, desplegaremos SQL Server sobre nuestra arquitectura de Kubernetes para poder monitorizar el servicio y generar las métricas de uso y analizarlas. Para esta parte usaremos el propio *Dashboard* de Kubernetes y Power BI para poder hacer nuestros propios cuadros de mando.

Finalmente, se expondrán las conclusiones obtenidas del trabajo realizado en cada uno de los tres apartados.

Palabras clave: Kubernetes, SQL Server, Microsoft Azure, microservicios, Docker, automatización del despliegue.

Abstract

During the current rise of microservices in the Information and Communication Technologies some problems that need to be solved have emerged over the time. One of them is undoubtedly the orchestration and maintenance of these microservices. To solve this problem Kubernetes was born. In addition, some questions have appeared around this technology. Which would be the usages? Cloud or On Premise? Is it recommended its use for databases management? How can we know if it is going to cover the expectations?

The aim of this dissertation project is to try and find an answer to these questions, and to address these issues is convenient to understand in depth the technology, so we must structure its analysis in a way that we can understand each part as well as the whole set. That is why the work is constituted by three parts.

The first part will introduce the Kubernetes technology, mainly explaining its function. Also, the intention is to make clear the most important concepts of this technology. It is also important to know how they work and the important things to consider.

On the other hand the pros and cons of using this technology on Cloud or On Premise will be analysed. For this part I will use Microsoft Azure, specifically through a Microsoft Azure Students account.

Eventually SQL Server on our Kubernetes architecture will be deployed to later be able to analyze the metrics that we generate from the use. For this part we will use the Kubernetes Dashboard and Power BI to make our own Dashboards.

Finally the intention is to answer the questions previously asked extracting the appropriate conclusions from each one of the sections.

Keywords: Kubernetes, SQL Server, Microsoft Azure, microservices, Docker, deployment automation.

Índice general

Capítulo 1:

Introducción

La arquitectura de los sistemas informáticos ha sido desde siempre uno de los mayores quebraderos de cabeza a la hora de afrontar proyectos de Tecnologías de la Información. Hasta tal punto es así que una buena decisión en cuanto a la arquitectura en el inicio de un proyecto puede marcar la diferencia entre el éxito y el fracaso del mismo.

Debido a la importancia de este tema, a lo largo de la historia se han ido desarrollando diferentes tipos de arquitecturas, pero siempre limitadas a la tecnología del momento. En los últimos tiempos existen dos grandes ideas predominantes y contrapuestas en cuanto a la arquitectura y un extenso debate de por qué se debe usar una u otra en función de las circunstancias. Estas dos arquitecturas contrapuestas son la *Arquitectura monolítica* y la *Arquitectura orientada a microservicios*.

La arquitectura monolítica es la típica arquitectura que siguen la mayoría de las aplicaciones en las que existe un único programa autónomo e independiente de otros sistemas informáticos. Una arquitectura monolítica tiene toda su operativa, su flujo y sus datos incluidos en una única aplicación, aunque puedan existir diferentes operaciones a realizar dentro de la aplicación. De este modo podemos conseguir un grado de acoplamiento óptimo y una gran fiabilidad en los datos ya que están centralizados. Por otro lado, uno de los grandes problemas que surgen es que si algo falla, al estar toda la aplicación alojada en un único servidor, dejaría de funcionar toda la aplicación.

Otro gran problema es el acceso a los datos, un punto crítico en el que a menudo se producen cuellos de botella. Esto se debe a que todos los servicios alojados en la aplicación acceden al mismo repositorio de datos. En un sistema transaccional, donde se realizan muchas operaciones de inserción y modificación de datos, pueden llegar a colapsar los sistemas de bases de datos por problemas de concurrencia.

Por su parte, la arquitectura de microservicios presenta una estructura distribuida en la que el acceso a un servicio de la aplicación se hace de manera independiente al resto de funcionalidades de la aplicación. De este modo, la modularidad de las aplicaciones es mucho mayor y aumenta la facilidad con que se pueden desarrollar las diferentes partes y su propio mantenimiento.

Otra gran ventaja que presenta este tipo de arquitectura es que puede adaptar la oferta de un determinado servicio en función de la demanda; es decir, puede hacer un balanceo de carga para dar mayor importancia a un servicio en un determinado momento. Otra de las ventajas de los microservicios, en contraposición a la arquitectura monolítica, es que si deja de funcionar un servicio no deja de hacerlo toda la aplicación.

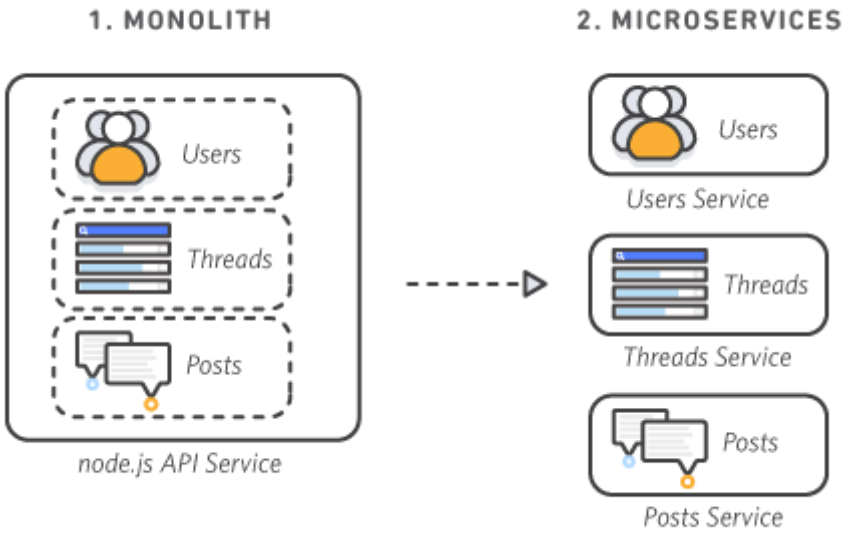

Figura 1.1. Diagrama monolítico *vs.* microservicios [1]

Para la arquitectura de microservicios surgen una serie de cuestiones a resolver, ya que pasamos de usar un único programa (arquitectura monolítica) a usar una serie de pequeños programas independientes entre sí que permiten una mayor autonomía. No obstante, al existir un mayor número de programas y flujos de información, se debe tener un mayor cuidado con la infraestructura que soporta a esta arquitectura.

Por ello, este trabajo de fin de grado tiene por objetivo investigar una tecnología que está mostrando gran solidez y fiabilidad: Kubernetes. Esta tecnología está muy ligada a la arquitectura de microservicios, ya que se usa para orquestarlos y garantizar el acceso a ellos en cualquier eventualidad.

Aparte de entender cómo funciona esta tecnología y ver qué problemas puede resolver, también queremos responder a más preguntas. En el nuevo paradigma *Cloud*, ¿tiene sentido seguir creando complejas infraestructuras *On Premise*? ¿Por qué? ¿Puede ser útil una tecnología como Kubernetes para desplegar nuestras plataformas de datos?

El término **Cloud Computing** o simplemente **Cloud** es el paradigma que permite a un usuario acceder a una serie de servicios a través de Internet. Estos servicios se dividen en tres categorías:

- ***IaaS*** (Infraestructura como Servicio): ofrecen una infraestructura básica sobre la que crear o desplegar el software que queramos. Un ejemplo serían las máquinas EC2 de AWS (Amazon Web Services) o el almacenamiento que ofrece S3, también de AWS.
- ***PaaS*** (Plataforma como Servicio): ofrecen una infraestructura con algún software ya instalado para facilitarnos el trabajo y no tener que configurar algunas características. En otras palabras ofrece un nivel más de abstracción. Algunos ejemplos son EMR (Elastic Map Reduce) de AWS que viene con Hadoop instalado o AKS de Microsoft Azure que viene con Kubernetes configurado.
- ***SaaS*** (software como Servicio): ofrecen una serie de utilidades ya listas para utilizar por usuarios sin necesidad de realizar ninguna configuración y está en la capa más alta de abstracción. Algunos ejemplos son Google Docs o DropBox, aplicaciones muy conocidas a nivel de usuario.

El término ***On Premise*** hace referencia a aquel software que se instala sobre un Hardware propio en vez de aquel que se instala de manera remota en granjas de servidores o entornos *Cloud*.

1.1. Motivación

La motivación para realizar este trabajo viene dada por el interés que me ha generado una tecnología como es Kubernetes, debido a que se trata de una de las tecnologías que más expectación genera en los últimos tiempos dentro del mundo IT pero cuyo uso no está suficientemente extendido fuera del ámbito académico o investigador.

Por este motivo, quería involucrarme en esta tecnología para entender de primera mano cuáles pueden llegar a ser sus casos de uso, prestando atención a aquellos en los que sería pertinente esta tecnología y aquellos en los que no.

Además, actualmente las aplicaciones son tan complejas que necesitan una infraestructura que sea robusta y fiable. A su vez, también se necesita que sea fácilmente manejable para reducir la complejidad de la orquestación, mantenimiento y actualización de los servicios que prestan las aplicaciones.

Aparte de todo esto, mi experiencia realizando prácticas en dos consultoras tecnológicas (Altia [2] y SolidQ [3]) me han ayudado a comprender lo complejo que es el proceso de pasar una aplicación del entorno de desarrollo al entorno de producción. Por este motivo, aprender a usar tecnologías como esta adecuadamente podrían ayudar a reducir la complejidad de ese proceso.

Como consecuencia a lo anteriormente mencionado, nace la figura del *ingeniero DevOps* y la *Cultura DevOps* para facilitar que los desarrolladores se centren en programar, desentendiéndose de la infraestructura subyacente en la aplicación a la hora de desplegarla.

1.2. Objetivos

Este trabajo tiene como objetivo realizar una extensa investigación sobre lo que Kubernetes puede ofrecer y qué problemáticas puede resolver. Esto se debe a que, como sabemos, existe una tentación muy grande de querer utilizar una tecnología que se pone de moda en cuanto nos enteramos de que existe y vemos lo que se puede hacer con ella (el comúnmente denominado efecto *hype*).

Por ello, queremos saber si realmente es adecuado usar este tipo de tecnología para albergar sistemas de bases de datos que demandan una alta disponibilidad y consistencia. Las bases de datos que se utilizan en entornos de producción son un ejemplo de ello, donde se debe tener una alta disponibilidad (*los famosos nueves*).

El acuerdo de nivel de servicio o SLA (del inglés, *Service Level Agreement*) define la cantidad de tiempo que puede estar no disponible nuestro servicio a lo largo de un año, siendo el número de nueves que contiene el SLA el porcentaje del tiempo que se asegura que el sistema estará disponible. Es decir, que cuanta mayor cantidad de nueves haya en ese número, mayor disponibilidad del servicio se tendrá asegurado a lo largo de un año.

Para llevar a cabo el análisis se estudiará primero el funcionamiento de Kubernetes y sus componentes. Posteriormente se procederá a desplegar pequeños servicios en un entorno local. Más adelante se desplegará SQL Server sobre esta infraestructura tanto *On premise* como en *Cloud*.

Finalmente, analizando las métricas del sistema a las que tendremos acceso, intentaremos discernir si es una buena decisión desplegar SQL Server sobre Kubernetes en un entorno de producción donde se requiere que nuestro servicio esté permanentemente disponible.

1.3. Estructura del trabajo

La estructura de trabajo está dividida en cinco capítulos principales que a su vez se dividirán en subapartados. Esta estructura de trabajo sirve para describir el plan de trabajo. Cada capítulo contendrá lo siguiente:

Capítulo 2: Kubernetes

En este capítulo se muestra cómo funciona Kubernetes y en qué consiste su arquitectura interna, así como cada uno de los componentes que permiten su funcionamiento. Esta parte es la base para entender el resto, por lo que goza de especial importancia.

Capítulo 3: ¿Cloud u On Premise?

En este capítulo se usará Kubernetes en entorno *Cloud* y entorno *On Premise* para mostrar las ventajas de su uso en cada uno de ellos. Analizaremos las herramientas que se deben usar en ambos paradigmas y los procesos para la creación de estos entornos.

Capítulo 4: Análisis del sistema

Para este capítulo desarrollaremos utilidades de monitorización y visualización de datos. Estas utilidades nos permitirán ver métricas que nos ayudarán a analizar el comportamiento y rendimiento la solución desplegada.

Capítulo 5: Conclusiones del trabajo

En este capítulo se expondrán las conclusiones a las que se han llegado después de realizar el trabajo de los apartados anteriores.

Capítulo 6: Trabajos futuros

En este último capítulo se mostrarán posibles trabajos futuros sobre esta materia. Además, también se hablará de algunas tecnologías ya existentes que pueden ser de gran interés con respecto a los temas tratados en el trabajo.

Además de estos capítulos, existe un apéndice en el que se detallan los procesos para instalar y manejar las herramientas que se utilizarán a lo largo de todo el trabajo.

Chapter 1:

Introduction

The information systems architecture has been one of the greatest troubles at the moment to face IT projects. To such and extent that a good decision about the architecture in the beginning can make the difference between success and failure.

Due to the relevance of the topic, different kinds of architectures have been developed along the history, but always limited by the technology of the moment. In recent times two big ruling ideas and opposite to architecture exist with a long discussion about the reason why it must be used one or the other according to the circumstances. These two opposite architectures are the Monolithic Architecture and the Microservices Oriented Architecture.

Monolithic architecture is the typical one used by most of the applications with an only autonomous program and independent to other computer systems. This kind of architecture has all his operational capacity, flow and data included in a single application, although they can exist different operations in it. This way we can achieve an optimus degree of coupling and a great data reliability since they are centralized. In the other hand one of the biggest problems is that if something fails, as the whole application is hosted in a lonely server, it would stop working the whole application.

Another one big problem is the access to data, a critical point wherein they are often bottlenecks produced. This results from the fact that all the hosted services by the application access the same data repository. In a transactional system, wherein there are a lot of insert and modification data operations made, they can even collapse the database systems with concurrency problems.

For its part, the microservices architecture introduces a distributed structure in which the one application service accessing is independent of the rest of the application functionalities. This way, the modularity of the applications is much bigger and it increases the ease within they can be developed different parts and its own maintenance.

Another great advantage of this kind of architecture is that it can adapt to the supply of a particular service according to its demand; that is to say, a load balancing can be done in order to give more importance to a service in a precise moment. Another advantage of microservices, in contraposition to monolithic architecture, is that if one service stops working, the whole application would not do so.

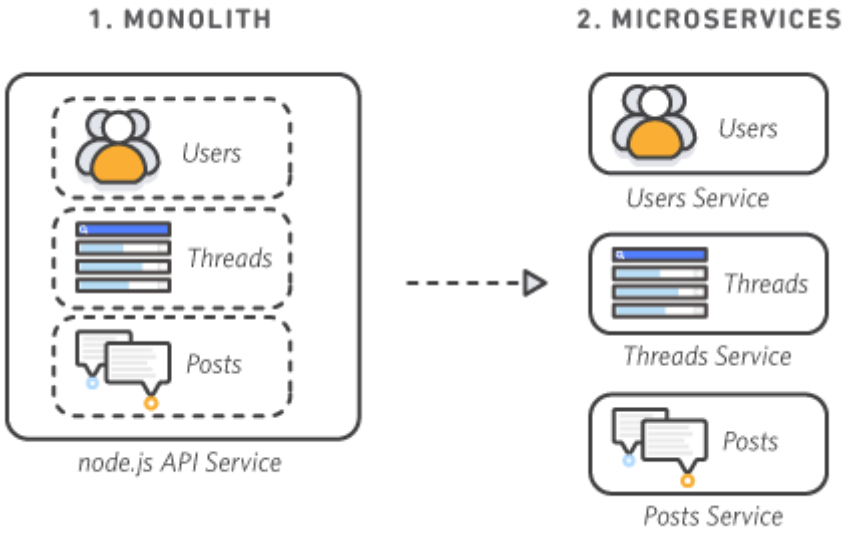

Figure 1.1. Monolithic diagram vs microservices [1]

For microservices architecture a few questions to solve arise, since we move from using one single program (monolithic architecture) to using a series of minor programs independent between them, that permit a bigger autonomy. Nevertheless, because of the existence of a greater number of programs and information flows, a particular care should be taken with the infrastructure that support this architecture.

For that reason, this dissertation has as objective investigating a technology that exhibits great solidity and fiability: Kubernetes. This technology it is highly connected to the microservices architecture, since it is used to orchestrate them and guarantee the access to them in any eventuality.

Aside from understanding how this technology works and see the problems it can solve, we want to answer more questions as well. In the new *Cloud* paradigm, which sense has continue creating complex infrastructures *On Premise*? Why? Can a technology as Kubernetes be useful to deploy our data platforms?

The term *Cloud Computing* or simply *Coud* is the paradigm that allows the user access to a series of services through the internet. These services are divided in three categories:

- ***IaaS*** (Infraestructure as a Service): they offer a basic infrastructure on which create or deploy the software we want. An example would be the EC2 machines from AWS (Amazon Web Services) or the storage which offers S3, from AWS as well.
- ***PaaS*** (Platform as a service): they offer a infrastructure with a preinstalled software to facilitate the work, not needing to configure some characteristics. In other words, it offers a greater level of abstraction. Some examples are EMR (Elastic Map Reduce) from AWS, which comes with Hadoop preinstalled or AKS from Microsoft Azure, which comes with Kubernetes configured.
- ***SaaS*** (Software as a service): they offer a series of utilities ready to use by the user without the necessity of doing any configuration, and it is in the highest layer of abstraction. Some examples are Google Docs or DropBoc, well known applications at user level.

The term *On Premise* makes reference to the software installed on a personal hardware, instead of the one installed remotely in server farms or *Cloud* environments.

1.1. Motivation

The motivation to do this dissertation comes from the interest that a technology as Kubernetes has generated on myself, due to the fact that it is one of the technologies which more expectation creates in the last few years in the world of IT, but whose use it is not sufficiently extended outside of the academic and researcher ambit.

For that reason, I would like to involve in this technology in order to understand first-hand which are the potential usages, paying attention to those in which this technology would be pertinent and the ones which would not.

Additionally, at the moment applications are so complex that need a robust and reliable infrastructure. Likewise, it is needed it to be easy to handle in order to reduce the orchestration complexity, maintenance and update of the services which the applications offer.

Apart from this, my experience in internships in two technological consultancies (Altia [2] and SolidQ [3]) have helped me understand how complex is the process of moving an application from the development to the production environment. For that reason, learning to use technologies such as this one appropriately could help reduce the complexity of this process.

As a consequence to the above mentioned, the figure of the *DevOps* engineer and the *DevOps* culture is born to facilitate that programmers can focus on programming, drawing back from the subjacent infrastructure of the application at the moment of deploying it.

1.2. Objectives

This paper has as objective make an wide research about what Kubernetes can offer and the problematics it can solve. This is because, as we know, there is a huge temptation to use a trending technology at the moment we find out it exists, and we see what can be done with it (the commonly denominated *hype* effect).

Thus, we want to know if it is really appropriate using this kind of technology to host database systems that need a high disponibility and consistency. The database used in in production environments are an example of it, in which there has to be a high disponibility (*the famous nines*).

> The Service Level Agreement or SLA defines the amount of time that can be unavailable our service along one year, being the number of nines that contains the SLA the percentage of the time our service will be available for sure. It is to say the more nines in this number the more insured availability along a year.

To perform the analysis we will study the functioning of Kubernetes and its components first. After that we will deploy small services in a local environment. Further on we will deploy SQL Server on this infrastructure both *On Premise* and *On Cloud*.

Finally, analyzing the metrics of the system we can access, we will try to discern whether it is a good choice to deploy SQL Server on Kubernetes in a production environment where it's necessary that our service is permanently available.

1.3. Dissertation structure

The dissertation structure is divided in five main chapters, which are divided in subsections. This structure is used to describe the work plan. Each chapter will contain the following:

Chapter 2: Kubernetes

In this chapter it is showed how Kubernetes works and its internal architecture, as well as each one of the components that get its operation. This part is the basis to understanding the rest of the dissertation, that is the reason why this chapter is so important.

Chapter 3: Cloud or On Premise?

In this chapter we will use Kubernetes in a Cloud environment and in an On Premise environment to show the advantages of its use in each one. We will analyze the necessary tools for both environments and the processes of creation for these environments.

Chapter 4: System analysis

For this chapter we will develop monitoring and data view utilities. These utilities will allows us to see metrics that will help us analyzing the behavior and performance of the deployed solution.

Chapter 5: Conclusions of the dissertation

In this chapter we will show the final conclusions reached after developing the above chapters work.

Chapter 6: Future works

In this last chapter there will be showed any possible future works about this topic. In addition, we will talk about existing technologies that can be very interesting with regard to the dissertation topics.

In addition to these chapters, there is an attachment where set of information to install and manage the tools that will be used along the whole dissertation are detailed.

Capítulo 2:

Kubernetes

2.1. Introducción

Como ya se ha mencionado en la estructura del trabajo, este capítulo tiene como objetivo hacer un amplio análisis del funcionamiento de Kubernetes desde un contexto más general a los componentes más concretos de la tecnología.

Según la definición de la página oficial de Kubernetes [4], “es una plataforma de código abierto para automatizar la implementación, el escalado y la administración de aplicaciones en contenedores”.

Por lo tanto, haciendo caso de esta definición, podemos ver que puede ser una herramienta muy útil para la gestión de una arquitectura basada en microservicios. Uno de los principales motivos para su utilidad es el hecho de que podemos actualizar una determinada funcionalidad, sin necesidad de desconectar todo nuestro sistema para después conectarlo de nuevo con la última actualización.

Con el uso de Kubernetes podríamos desplegar nuestra última actualización de un microservicio sin necesidad de detener el mismo. ¿Cómo es posible esto? Aquellos usuarios que están en ese momento accediendo al servicio podrán acabar sus acciones sobre ese servicio y, solo cuando esas sesiones terminen, se detendrá el servicio. Inmediatamente después se relanzará con la nueva actualización para ese servicio determinado.

¿Qué pasaría si, por algún motivo, se detuviese uno de los puntos de acceso a un servicio? Kubernetes es capaz de volver a reiniciar automáticamente aquellos puntos que se hayan detenido. Lo cual es muy útil a la hora de mantener nuestra arquitectura de microservicios.

¿Cómo podría hacer para arrancar un servicio en mi infraestructura de manera que existan distintos puntos por los que acceder y, de este modo, distribuir los recursos? Se tendrá que definir un fichero de configuración con el que se le dirá a la herramienta la imagen de la aplicación que se quiere usar, el tipo de acceso, el número de réplicas… Una vez definido este fichero, ejecutaremos un comando y tendremos nuestro servicio activo ejecutándose con las características que hemos definido para él. En otras palabras, simplifica mucho la labor de orquestación de los microservicios en nuestra arquitectura.

En los siguientes apartados se detalla el uso de Kubernetes:

2.2. Creación de un cluster de Kubernetes
2.3. Despliegue de un servicio Web
2.4 Despliegue de SQL Server

2.2. Creación de un cluster de Kubernetes

Para esta primera parte crearemos un pequeño entorno *StandAlone* (es decir, con un único nodo) para realizar algunos despliegues simples que nos sirvan para entender cómo funciona Kubernetes.

En este punto, partimos de la base de que ya se han realizado las acciones que se indican en el Apéndice A para instalar las herramientas que vamos a usar.

Lo primero que debemos hacer es arrancar nuestra máquina virtual Ubuntu Server con las herramientas instaladas y conectarnos mediante *ssh* a la misma. En mi caso, como uso Ubuntu, lo haré desde un terminal. En caso de que se tenga Windows, se puede hacer desde un CMD, una consola de Power Shell o desde un cliente como PuTTY.

Existe la posibilidad de ejecutar los comandos directamente sobre la máquina sin conectarnos por *ssh* en esta parte. En cualquier caso, tendremos que usar un cliente *sftp* para transferir ficheros a la máquina virtual y, sobre todo, más adelante al transferirlos a máquinas en la nube. Entonces será indispensable conectarnos por *ssh* (siempre que no se use la consola que proporciona la propia plataforma).

Una vez conectados, podemos arrancar nuestro *cluster* de Kubernetes utilizando la herramienta minikube con el siguiente comando:

```
$ sudo minikube start --vm-driver=none
```

Minikube es la herramienta que proporciona Kubernetes para crear y administrar *clusters* de un solo nodo. Permite realizar acciones de arranque, reinicio, estado del *cluster* y además, también permite realizar otro tipo de acciones como, por ejemplo, mostrar la *url* por la que tendríamos que acceder a un *Service*.

La opción de `--vm-driver=none` la usamos ya que no vamos a virtualizar de ningún modo nuestro *cluster*. Otras opciones serían `virtualbox`, `hyper-v` o `vmware` entre otras, dependiendo del sistema operativo nativo de nuestra máquina. Los controladores para máquinas virtuales no son los mismos para todos los sistemas operativos. Por ejemplo, podemos usar `hyper-v` en Windows pero no podemos usarlo en Linux.

Para poder ver que todo está en orden y nuestro *cluster* se está ejecutando adecuadamente ejecutamos el siguiente comando:

```
$ sudo minikube status
```

Esto nos devolverá información acerca de nuestro *cluster* y debería aparecer todo como se encuentra en la Figura 2.1; es decir, con todos los elementos en el estado *running / configured*.

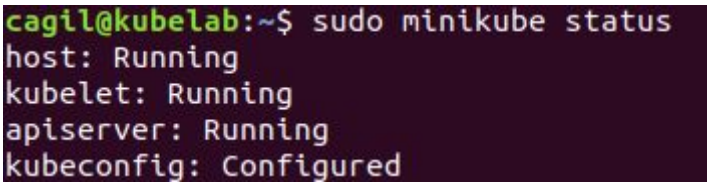

Figura 2.1. Estado de nuestro cluster.

2.3. Despliegue de un servicio Web

Para esta parte vamos a realizar una pequeña aplicación en el lenguaje de programación Python. Será una aplicación web que realizaremos mediante el uso del framework *Flask* [5].

> ***Flask*** es un framework del lenguaje Python que permite crear aplicaciones web ligeras con pocas líneas de código. En nuestro caso con apenas 10 líneas de código tenemos una Web a la que hacer peticiones GET.

La idea es hacer un despliegue de esa aplicación en varios *pods* (se detalla más adelante) para acceder a la misma y que en cada petición nos devuelva el ID del *Pod* al que estamos accediendo.

Para realizar esta tarea, primero generamos una imagen de Docker [6]. Para generar esta imagen necesitamos un archivo Dockerfile en el que indicamos las acciones a realizar para crear nuestra imagen. En la Figura 2.2 podemos ver la estructura de nuestro Dockerfile.

> ***Docker*** es una herramienta que permite empaquetar un software en un entorno con un sistema operativo diferente al de nuestro sistema nativo sin necesidad de tener que configurar máquinas virtuales

```
FROM alpine:3.5

RUN apk add --update py2-pip

COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

EXPOSE 8000

CMD ["python", "/usr/src/app/app.py"]
```

Figura 2.2. Archivo Dockerfile

En este archivo se definen los pasos a ejecutar para crear una imagen de Docker para que nuestra aplicación funcione como queremos. El fichero *requirements.txt* contiene aquellas dependencias que queremos instalar. En nuestro caso, este fichero contiene exclusivamente *Flask* con la versión que queremos instalar.

También copiamos el script de python de nuestra aplicación y el fichero *.html* al directorio */usr/src/app* para ejecutar nuestra aplicación con el último comando. este ejecuta un comando de shell para lanzar nuestra aplicación. La instrucción `expose` indica el puerto por el que accederemos a la aplicación.

Una vez hemos definido todo, debemos construir la imagen de nuestra aplicación. Para ello ejecutamos el siguiente comando:

```
$ docker build .
```

Al indicarle a Docker el directorio actual, automáticamente busca el fichero Dockerfile y lo ejecuta. Una vez generada la imagen Docker vamos a subirla a nuestro repositorio de Docker Hub.

Para poder subirlo primero debemos hacer *login* con nuestras credenciales. Una vez hecho esto debemos asignarle una etiqueta a nuestra imagen y finalmente hacer un *push*. Los comandos para esta parte son los siguientes:

```
$ docker login
$ docker tag <image-id> <tag>
$ docker push <tag-given> <docker-hub-username>/<image-name>:tag
```

Una vez hemos hecho todo esto, podemos conectarnos a la máquina y definir las condiciones para nuestro despliegue de la aplicación en Kubernetes.

Ahora debemos definir nuestro archivo de configuración *.yaml* para establecer las condiciones de nuestro despliegue en el *cluster* de Kubernetes. Es aquí donde debemos introducir los primeros conceptos de la arquitectura de Kubernetes.

Primero ejecutaremos un comando para crear un *namespace* (espacio de nombres). En Kubernetes, el *namespace* sirve para crear un entorno aislado en el que realizar nuestros despliegues y después gestionar de manera más sencilla los recursos del mismo.

```
$ kubectl create namespace flask-app
```

Kubectl es la herramienta que proporciona Kubernetes para gestionar los recursos internos de un *cluster* de Kubernetes. Con esta herramienta tenemos un acceso sencillo a la API de Kubernetes para consultar, crear, eliminar o modificar *Pods, Deployments, Services* y todos los elementos propios de Kubernetes.

Una vez creado este *namespace*, al crear nuestros despliegues podremos hacer que estos se generen en dicho *namespace*. De este modo no lo harán en el *namespace* por defecto y será más sencillo, por ejemplo, eliminar o consultar *Pods* o *Services*.

A continuación vamos a definir en un fichero de configuración *.yaml* los parámetros para realizar nuestro despliegue y el acceso al servicio que corre sobre él. Este fichero *.yaml* se divide en dos partes, la que define el *Service* y la que define el *Deployment*.

Service

Imaginemos que tenemos una serie de *pods* que se han desplegado por una definición de *Deployment* y tenemos otra serie de *pods* que se han desplegado por otra definición de *Deployment*. Estas series de *pods* pueden ser unas, con sus características, en un momento dado, pero si los analizamos en otro instante pueden haber cambiado completamente. Imaginemos, por ejemplo, que estas dos series de *pods* son parte de un mismo flujo de un servicio, un backend y un frontend. Al cambiar las características pueden cambiar sus IPs.

Entonces, ¿cómo podemos seguir mapeando las IPs de los *pods* para que se encuentren entre sí? Para eso sirve *service*, actúa como un *service discovery*. Un *service discovery* es un protocolo de red que sirve para encontrar de manera automática las IPs que nuestro servicio necesita para ejecutarse de manera adecuada.

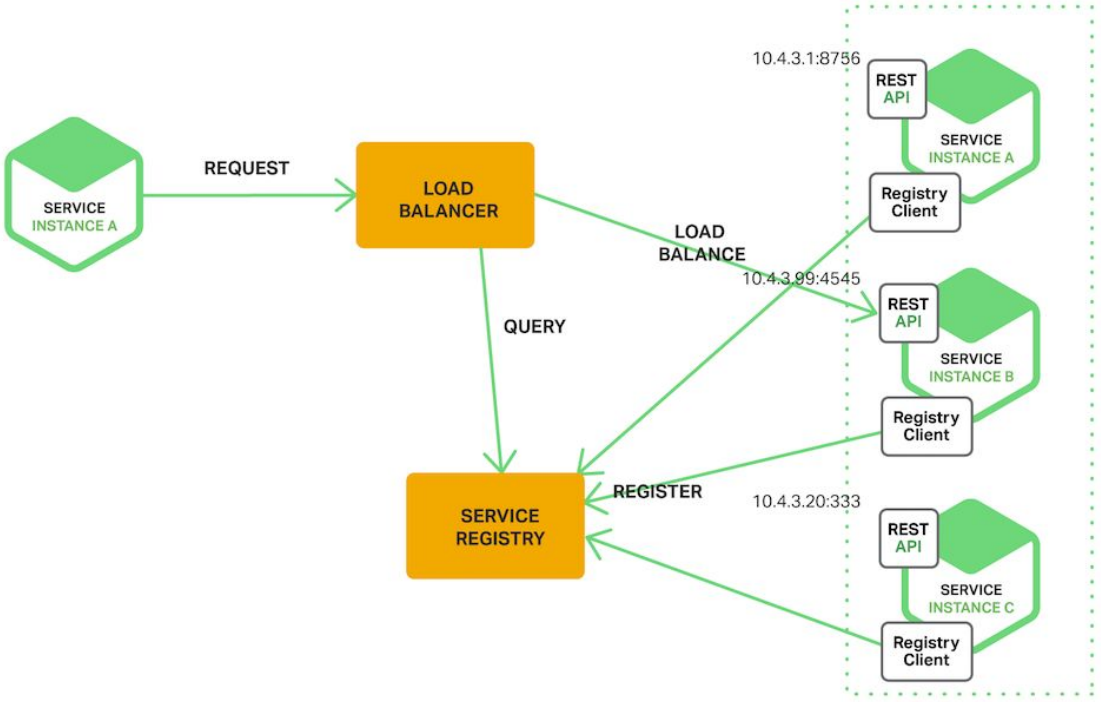

Figura 2.3. Diagrama de *Service Discovery* [7]

En la definición de nuestro *Service* debemos decir de qué tipo va a ser (ClusterIp, NodePort, LoadBalancer…). En nuestro caso vamos a usar NodePort. Este tipo de servicio expone un puerto de la máquina virtual para el acceso a nuestra aplicación, que después vamos a mapear desde VirtualBox para poder accederlo desde nuestro Host.

Por defecto, estos puertos van desde el 30000 al 32767, aunque se pueden modificar en la configuración de nuestro *cluster*. En nuestro caso nos da igual, por lo tanto respetaremos este rango.

La declaración de nuestro servicio quedaría como se muestra en la Figura 2.4. Esta configuración está dentro del fichero *flask-deployment.yaml*, que también contiene la configuración del *Deployment*.

```
1   apiVersion: v1
2
3   kind: Service
4
5   metadata:
6
7     name: flaskapp
8
9   spec:
10
11    selector:
12      app: flaskapp
13
14    ports:
15      - nodePort: 31525
16        protocol: TCP
17        port: 8000
18        targetPort: 8000
19
20    type: NodePort
21
```

Figura 2.4. Definición de un Service

Deployment

Un *Deployment* sirve para establecer los parámetros de nuestro despliegue. Estos parámetros van desde el número de *pods* que queremos para nuestro despliegue hasta el las especificaciones para los contenedores que se ejecutan en ellos.

En este punto es donde indicaremos la imagen de Docker que queremos ejecutar sobre nuestros pods, que actuarán como puntos de acceso a nuestro servicio. En el caso de querer cambiar la imagen de Docker que queremos ejecutar sobre nuestros pods, solo deberemos cambiar la referencia en nuestro archivo de configuración y relanzar el despliegue. De este modo, como ya comentamos anteriormente, los *pods* con la imagen antigua irán siendo eliminados conforme dejen de ser usados y se lanzarán los nuevos para los accesos que se produzcan a la aplicación a partir de ese momento.

La imagen que vamos a usar para nuestro despliegue es la que anteriormente hemos subido a mi repositorio de Docker Hub. Para poder traer la imagen de Docker sin problemas debemos crear un Secret que guarde nuestras credenciales de *login* para Docker Hub.

Como paso previo a la creación del Secret, debemos hacer lo siguiente:

```
$ docker login #introduciendo nuestras credenciales
```

Una vez iniciada la sesión, debemos volcar el contenido del fichero donde se guardan esos datos de autenticación en el Secret que creamos. Una vez hecho esto, ya podemos hacer pull de imágenes de Docker para ejecutar en nuestros pods.

La configuración del *Deployment* del servicio Web con *Flask* se muestra en la Figura 2.5.

```yaml
23
24  apiVersion: apps/v1
25
26  kind: Deployment
27
28  metadata:
29
30    name: flaskapp
31
32  spec:
33
34    replicas: 3
35
36    selector:
37
38      matchLabels:
39
40        app: flaskapp
41
42    template:
43
44      metadata:
45
46        labels:
47
48          app: flaskapp
49
50      spec:
51
52        containers:
53
54        - name: flaskapp
55
56          image: cagil04/kubepod_flaskapp:latest
57          imagePullPolicy: Always
58          ports:
59
60          - containerPort: 8000
61
62        imagePullSecrets:
63
64        - name: regcred
```

Figura 2.5. Definición de un *Deployment*.

Pods

Ya hemos hablado de los *pods* con anterioridad, a continuación vamos a definir lo que son. Podemos entender los *pods* como la unidad mínima de la arquitectura; es aquí donde se va a ejecutar la imagen que contiene nuestra aplicación. Pero esto no es así siempre, es decir que puede haber más de un contenedor ejecutándose dentro de un pod.

Por ejemplo, en un *Pod* podríamos tener definida una aplicación web con un volumen para almacenar los datos que permiten el correcto funcionamiento de la aplicación, estando ambas ejecutándose en distintos contenedores.

A continuación, en la Figura 2.6 se expone gráficamente la arquitectura de un *cluster* de Kubernetes y el acceso a los pods.

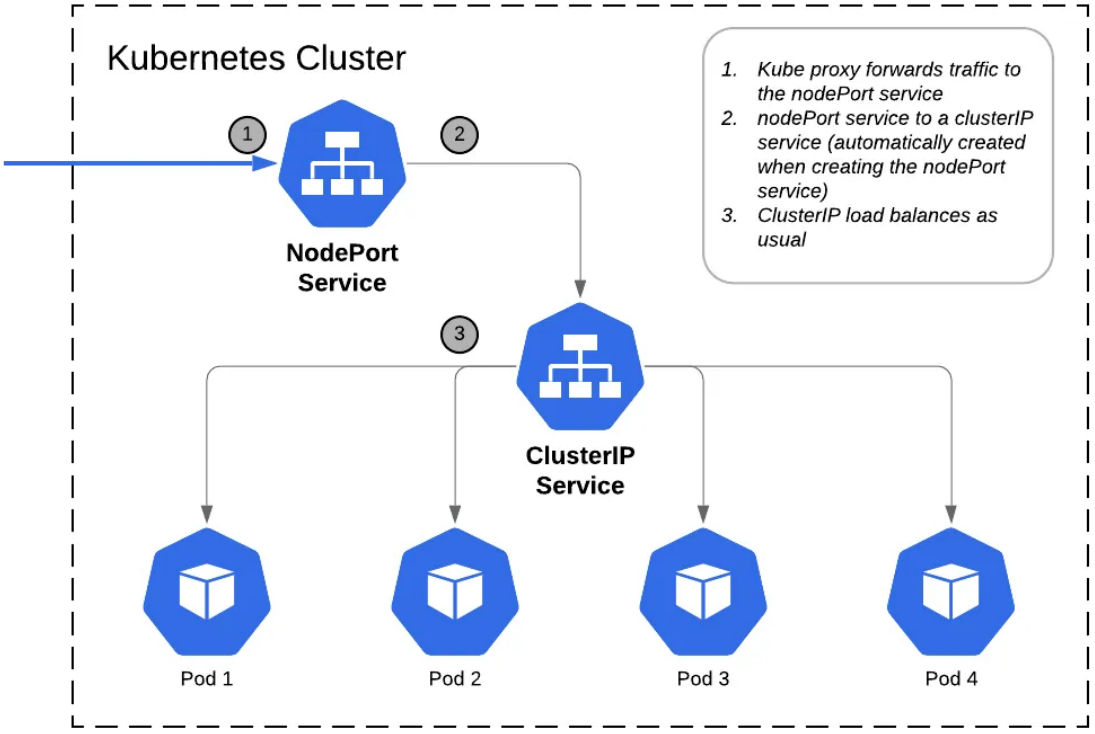

Figura 2.6. Topología de Kubernetes para llegar a los pods [8] .

Dentro del nodo, cada *Pod* contiene una dirección IP única para evitar que existan conflictos de puertos entre los distintos pods. Esto también sirve para que el servicio pueda encontrar de manera inequívoca a todos y cada uno de los pods.

Despliegue y prueba

Una vez definidas las configuraciones del *Service* y el *Deployment* debemos ejecutar un comando para poner en marcha nuestra aplicación.

```
$ kubectl apply -f flask-deployment.yaml -n flask-app
```

Al ejecutar este comando se crean el *Service* y el *Deployment*. Podemos ver que todo funciona correctamente ejecutando lo siguiente:

```
$ kubectl get pods -n flask-app
```

esto debería devolver una salida como la que se muestra en la Figura 2.7. En ella se muestran las tres réplicas o *pods* que indicamos en nuestro despliegue con sus respectivos nombres, en estado running que indica que están funcionando correctamente.

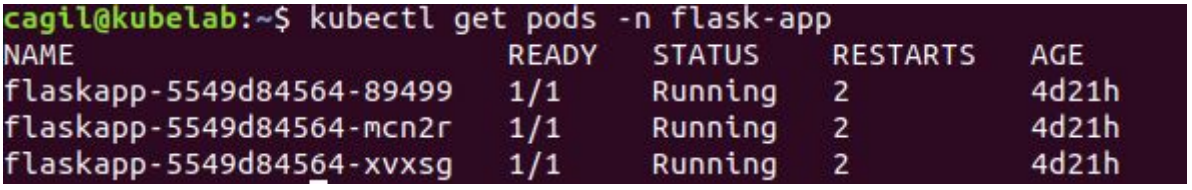

Figura 2.7. Pods donde se ejecuta la aplicación Flask

Si todo funciona correctamente ya podemos abrir un navegador y ver si realmente podemos acceder a nuestra aplicación Web. Abrimos un navegador e introducimos la siguiente URL:

http://localhost:2024/?name=your_name

En mi caso he usado el puerto 2024 para el reenvío de puertos de `VirtualBox`, pero puede ser cualquier otro puerto libre. Hacemos un GET Request con nuestro nombre y debería devolver lo que se muestra en la figura 2.8.

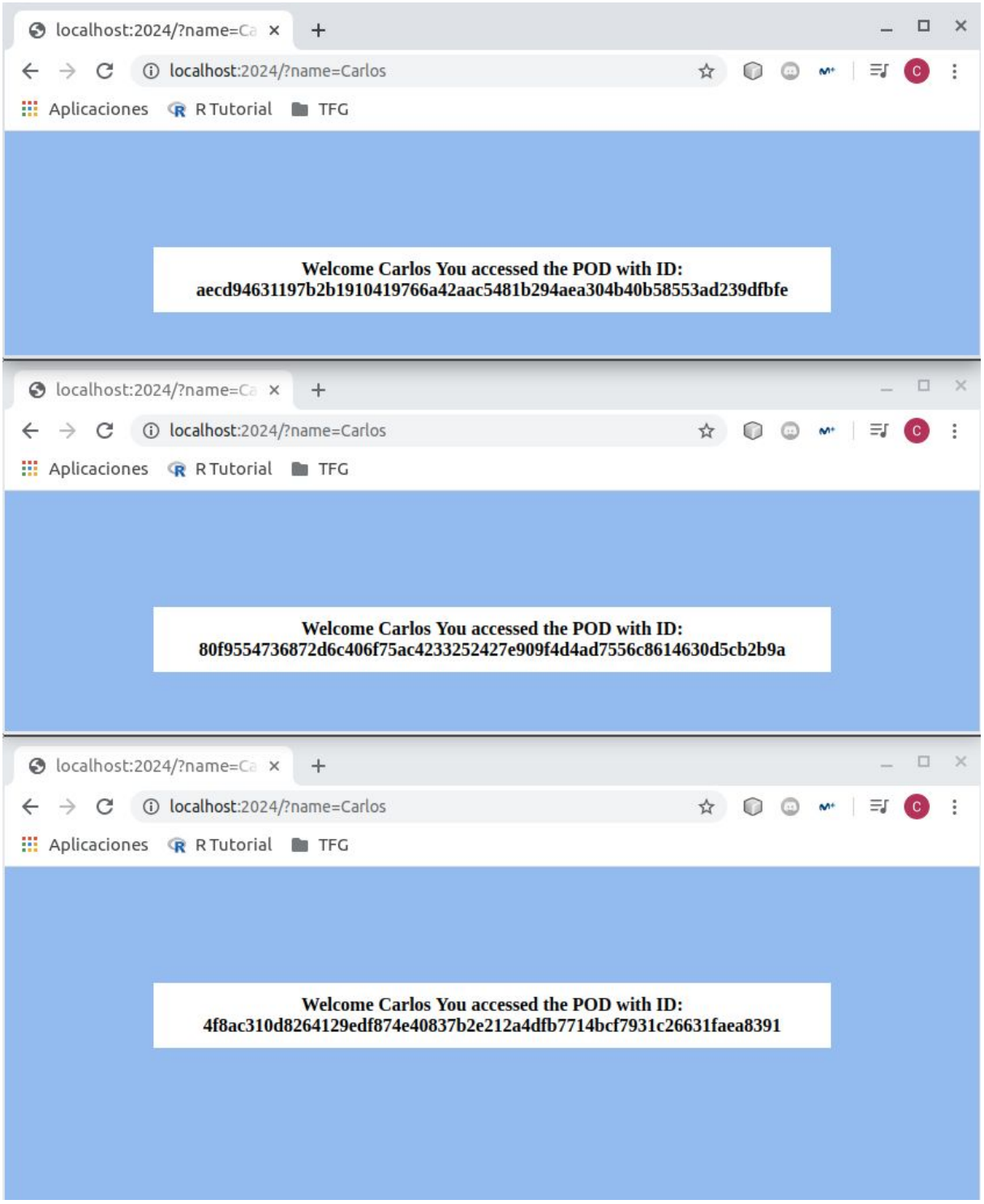

Figura 2.8. Aplicación web con Flask.

Como se observa en la Figura 2.8 accedemos tres veces al mismo servicio y nos da cada vez un ID distinto. Recordemos que teníamos tres *pods*, así que en cada uno de los navegadores estamos accediendo a un *Pod* distinto.

2.4. Despliegue de SQL Server

En los puntos anteriores ya hemos visto algunos de los conceptos más importantes sobre Kubernetes. Ya estamos listos para desplegar una instancia de SQL Server sobre nuestro *cluster*. Pero antes vamos a introducir brevemente SQL Server.

He decidido usar SQL Server como SGDB (Sistema de Gestión de Bases de Datos) por ser el SGBD en el que más experiencia tengo y, además, se adecúa mejor al entorno *Cloud* de Azure que es el que usaremos más adelante. La causa de que sea más adecuado SQL Server con el *Cloud* de Azure es que ambas tecnologías son Microsoft y se adapta mejor al entorno.

SQL Server

SQL Server es el sistema de gestión de bases de datos relacional de Microsoft. Su lenguaje es T-SQL (Transact-SQL), que extiende el lenguaje SQL, contiene ciertas funciones propias y permite utilizar o crear procedimientos para ejecutar en una instancia de SQL Server.

Un aspecto importante para su funcionamiento en Docker es que, a partir de SQL Server 2017, funciona en sistemas operativos Linux. Debido a ello, podremos crear una imagen de Ubuntu con SQL Server funcionando para poder ejecutar sobre nuestra arquitectura.

En caso de conectarnos desde una máquina Windows podemos usar SQL Server Management Studio. De no ser así, y estar usando una máquina Ubuntu, podemos usar clientes de línea de comandos como `sqlcmd` o `mssql-cli`.

Características de la imagen de Docker

Como ya vimos en el caso del servicio web del Apartado 2.3, debemos crear una imagen de Docker que después desplegaremos sobre nuestros *pods*. En el caso del despliegue de SQL Server es un poco más complejo porque vamos a necesitar configurar más aspectos como, por ejemplo, las bases de datos que importaremos en nuestra instancia.

En esta ocasión necesitaremos algunos ficheros más, no solo un *Dockerfile* y un *requirements.txt*. A continuación vamos a ver los ficheros necesarios:

- **Dockerfile**: ya sabemos que en este fichero establecemos una serie de instrucciones a ejecutar en el proceso de creación de la imagen. Para este caso es algo más complejo, usaremos el usuario `root` e indicaremos las bases de datos a importar tras haber definido qué versión de SQL Server usaremos (en nuestro caso SQL Server 2017). Finalmente se ejecuta un script *entrypoint.sh* para configurar algunos detalles y se duerme indefinidamente (comando `sleep infinity`) el proceso.

- **docker-compose.yml**: en este archivo indicamos algunos parámetros de configuración como el puerto que exponemos para acceder a SQL Server o las credenciales para acceder al mismo.

- **entrypoint.sh**: este script contiene dos intrucciones. La primera es arrancar el funcionamiento del motor de SQL Server y la segunda es para ejecutar el script *setup.sh*.

- **setup.sh**: en este script se comprueba si las bases de datos están importadas o no. En caso de no estarlo se ejecuta el script *setup.sql*.

- **setup.sql**: finalmente en este script se importan las bases de datos en la instancia de SQL Server desde los ficheros *.bak* que extraemos en la ejecución del *Dockerfile*.

La ejecución del comando para la creación de la imagen llevará algo más de tiempo que la del ejemplo de la aplicación de *Flask*. Una vez completada la creación de la imagen, sigue el mismo proceso: etiquetamos la imagen y la subimos al repositorio para después poder hacer un pull desde el despliegue de Kubernetes.

Storage Class

El *Storage Class* es un elemento propio de Kubernetes que, como su propio nombre indica, establece el tipo de almacenamiento que vamos a usar. Este es uno de los aspectos que menos relación directa guarda con Kubernetes, ya que es un tema más relacionado con los proveedores de servicios Cloud. Es decir, según el servicio Cloud que usemos podremos usar unos tipos de almacenamiento u otros. En este primer caso, On Premise, usaremos el *Storage Class* por defecto de Kubernetes y no nos hará falta definir un archivo *.yaml* con la configuración de nuestro Storage Class para usar en nuestro *Persistent Volume Claim*.

Persistent Volume Claim

El *Persistent Volume Claim* es un elemento de Kubernetes que usamos para guardar datos de manera persistente en nuestro sistema, en este caso nuestras bases de datos. Nos interesa poder tener este tipo de elementos para almacenar toda aquella información que no sea volátil en nuestro sistema.

Un *Persistent Volume Claim* es similar a un *Pod* pero, en vez de solicitar recursos al nodo del *cluster*, solicita recursos al *Persistent Volume*. El *Persistent Volume* es, como su propio nombre indica, un almacenamiento persistente que ha provisto el administrador del sistema. Es independiente de los despliegues, los *pods* mueren y se crean otros pero el *Persistent Volume* debe mantener su información intacta.

No hay mucho que definir en esta configuración, a diferencia de otras como los *Deployment*. Especificamos el tipo de acceso al almacenamiento, que en nuestro caso será de `ReadWriteOnce`. Esto quiere decir que solo permite un acceso de lectura/escritura a la vez al almacenamiento desde ese punto. Otro modo de acceso es `ReadOnly` que permite muchos accesos de solo lectura.

Despliegue

Para esta parte tenemos dos ficheros de configuración *.yaml*, uno para definir el *Persistent Volume Claim* y otro para el *Deployment*. Lo primero que haremos es aplicar la configuración del *Persistent Volume Claim*.

```
$ kubectl apply -f sql_pvc.yaml -n mssql
```

Ahora habría que hacer lo mismo con el despliegue, pero antes debemos crear un *secret* para guardar la clave de acceso SQL Server. Aunque no la vamos a usar porque ya lo gestionamos a nivel interno del Docker.

```
$ kubectl create secret MSSQL_SA_PASSWORD="<password>"
```

Después de haber realizado este paso ya se puede realizar el despliegue ejecutando un comando similar al que se usó con la aplicación *Flask*:

```
$ kubectl apply -f sql_deployment.yaml -n mssql
```

Conexión a la instancia

Ahora que ya tenemos nuestra instancia de SQL Server ejecutándose en nuestro *cluster* de Kubernetes, debemos crear un puente de nuestra máquina host a la máquina virtual en el reenvío de puertos de VirtualBox.

Ahora ya podemos ejecutar nuestro cliente para conectarnos a la instancia a través de línea de comandos. El comando para conectarse con *mssql-cli* sería el siguiente:

```
$ sudo mssql-cli -S localhost,2026 -U sa -P PaSSw0rd
```

Al ejecutar ese comando debemos conectarnos sin problemas a la instancia de SQL Server. Una vez dentro veremos algo similar a lo que se muestra en la Figura 2.9.

Figura 2.9. Conexión a la instancia de SQL Server

Capítulo 3:

Cloud u On Premise

3.1. Introducción

Los modelos de los sistemas de la información han evolucionado a lo largo de la historia. En la última década ha surgido una nueva tendencia de *Cloud Computing*, en contraposición a los sistemas clásicos de cliente-servidor con *hosting On Premise*.

A diferencia de los sistemas clásicos, con el uso de *Cloud* no necesitamos disponer de grandes y costosos servidores para almacenar y procesar nuestra información. Además, podemos olvidarnos de tener que administrar estas máquinas, así como de su optimización y conexión adecuada. Con las tecnologías Cloud podemos crear una máquina remota con las características que queramos en pocos clicks y sin necesidad de tener que configurar sistemas operativos, drivers…

Aparte de todo lo anterior existe un detalle aún más importante, y es que permite hacer un escalado de infraestructura de manera rapída y sencilla, sin necesidad de interrumpir nuestra infraestructura actual.

Todo esto hace de la tecnología Cloud una opción muy interesante para desplegar nuestras arquitecturas, pero debemos saber si las características de Kubernetes y sus problemáticas se adaptan y se resuelven de manera correcta sobre este paradigma.

En los siguientes apartados se detalla la creación de un *cluster On Premise* y *Cloud*:

3.2. Creación de un *cluster On Premise*
3.3. Creación de un *cluster AKS*

3.2. Creación de un cluster On Premise

La creación de un *cluster* On Premise necesita el aprovisionamiento adecuado de los siguientes recursos:

- Memoria RAM
- Discos o almacenamiento
- CPU
- Ancho de banda
- Número de nodos
- Etc

Debemos hacer una buena previsión de los recursos necesarios ya que ampliarlos en un futuro será una tarea laboriosa, en contraposición a lo que ocurre con las plataformas Cloud.

Por otro lado, existe la complejidad que provoca el uso de las herramientas propias de Kubernetes para gestionar los *clusters* y su instalación en cada uno de los nodos que los conforman. Debido a ello, voy a explicar una nueva herramienta que es necesaria para entender la gestión de los *clusters* con más de una máquina virtual. Esta herramienta es kubeadm.

Kubeadm es una herramienta que permite la creación y gestión de *clusters* de Kubernetes. A diferencia de la que ya habíamos visto, Minikube, esta herramienta permite manejar *clusters* con varios nodos.

```
cagil@kubemaster:~$ sudo kubeadm init --apiserver-advertise-address=192.168.99.1
00 --pod-network-cidr=192.168.0.0/16
W0428 07:40:06.957577    4639 version.go:101] could not fetch a Kubernetes versi
on from the internet: unable to get URL "https://dl.k8s.io/release/stable-1.txt"
: Get https://dl.k8s.io/release/stable-1.txt: net/http: request canceled while w
aiting for connection (Client.Timeout exceeded while awaiting headers)
W0428 07:40:06.957902    4639 version.go:102] falling back to the local client v
ersion: v1.17.4
W0428 07:40:06.958165    4639 validation.go:28] Cannot validate kubelet config -
 no validator is available
W0428 07:40:06.958317    4639 validation.go:28] Cannot validate kube-proxy confi
g - no validator is available
[init] Using Kubernetes version: v1.17.4
[preflight] Running pre-flight checks
```

Figura 3.1. Ejemplo de ejecución de Kubeadm en el nodo maestro.

En la Figura 3.1 se puede observar el comando que se debe ejecutar en el nodo maestro para desplegar Kubernetes en un *cluster* de más de un nodo. En los nodos esclavos, en vez de usar `kubeadm init`, se usaría `kubeadm join`. No vamos a detallar más en profundidad sobre este aspecto técnico ya que no tiene especial importancia en lo que vamos a desarrollar.

Existen multitud de tutoriales sobre cómo hacer un *cluster* de kubernetes de más de un nodo desde cero con máquinas virtuales en nuestra propia máquina. Algunos en los que me he basado para ello son:

- *Building a Kubernetes cluster in VirtualBox* [9]
- *Install and Deploy Kubernetes in Ubuntu 18.04* [10]
- *Installing Kubeadm* [11]

El proceso de creación de este tipo de *clusters* es costoso en tiempo y recursos necesarios. Por lo tanto, se deben valorar las diferentes alternativas antes de comenzar un proyecto en el que se quiera usar esta tecnología. También es cierto que hay algunas ocasiones en las que, debido al tiempo y la disponibilidad necesaria, es imprescindible que nuestros servicios estén funcionando *On Premise*. Para este tipo de situaciones puede resultar interesante una arquitectura híbrida de *On Premise- Cloud.*

Cuando se habla de arquitectura híbrida, en este contexto, se refiere al uso de tecnologías *Cloud* sobre infraestructura *On Premise*. Es decir, en nuestro caso sería usar las herramientas de Azure como `az` y las herramientas propias de gestión de *clusters* de Kubernetes como `kubeadm` sobre una infraestructura *On Premise*. Esta solución puede resultar de interés para proyectos que no pueden ser lanzados en un entorno *Cloud* por seguridad, pero que necesitan una serie de recursos exclusivos de este tipo de tecnologías. Un ejemplo de este tipo de proyecto serían aquellos que tuvieran que ver con la banca.

3.3. Creación de un cluster AKS en Azure

Para crear un *cluster* de Kubernetes en una plataforma *Cloud* existen muchas alternativas entre los mayores proveedores *Cloud* del mercado (Amazon Web Services, Microsoft Azure, Google Cloud…). Todos ellos incorporan productos ya predefinidos que evitan la labor de tener que configurar las máquinas virtuales de los entornos *Cloud*. Debido a ello, resulta bastante sencillo desplegar un *cluster* de Kubernetes en estos entornos.

Se ha decidido usar Microsoft Azure porque es el único que proporciona crédito y recursos suficientes a cuentas de estudiantes para realizar los despliegues que se necesitan para este trabajo. Se ha intentado con Google y AWS, aparte de Azure, pero en sus cuentas de estudiantes no proporcionan las herramientas necesarias para trabajar.

En el portal de Microsoft Azure se puede acceder a todos los productos que Microsoft ofrece para su plataforma *Cloud*. En este caso el producto elegido es AKS (*Azure Kubernetes Service*). Con el uso de este producto no es necesario preocuparse de la instalación de ningún software ni de la configuración de ninguna máquina. Al crear el servicio se establecen ciertos parámetros y en pocos minutos ya está el *cluster* configurado con las características que se deseen.

En la Figura 3.2 se puede ver la página de configuración básica de AKS que se ha utilizado para crear el *cluster*. Además de esta página hay otras para configurar la autenticación, la red, el escalado… En el caso de esas otras páginas se dejaron las opciones por defecto, ya que no tenían especial relevancia en este trabajo.

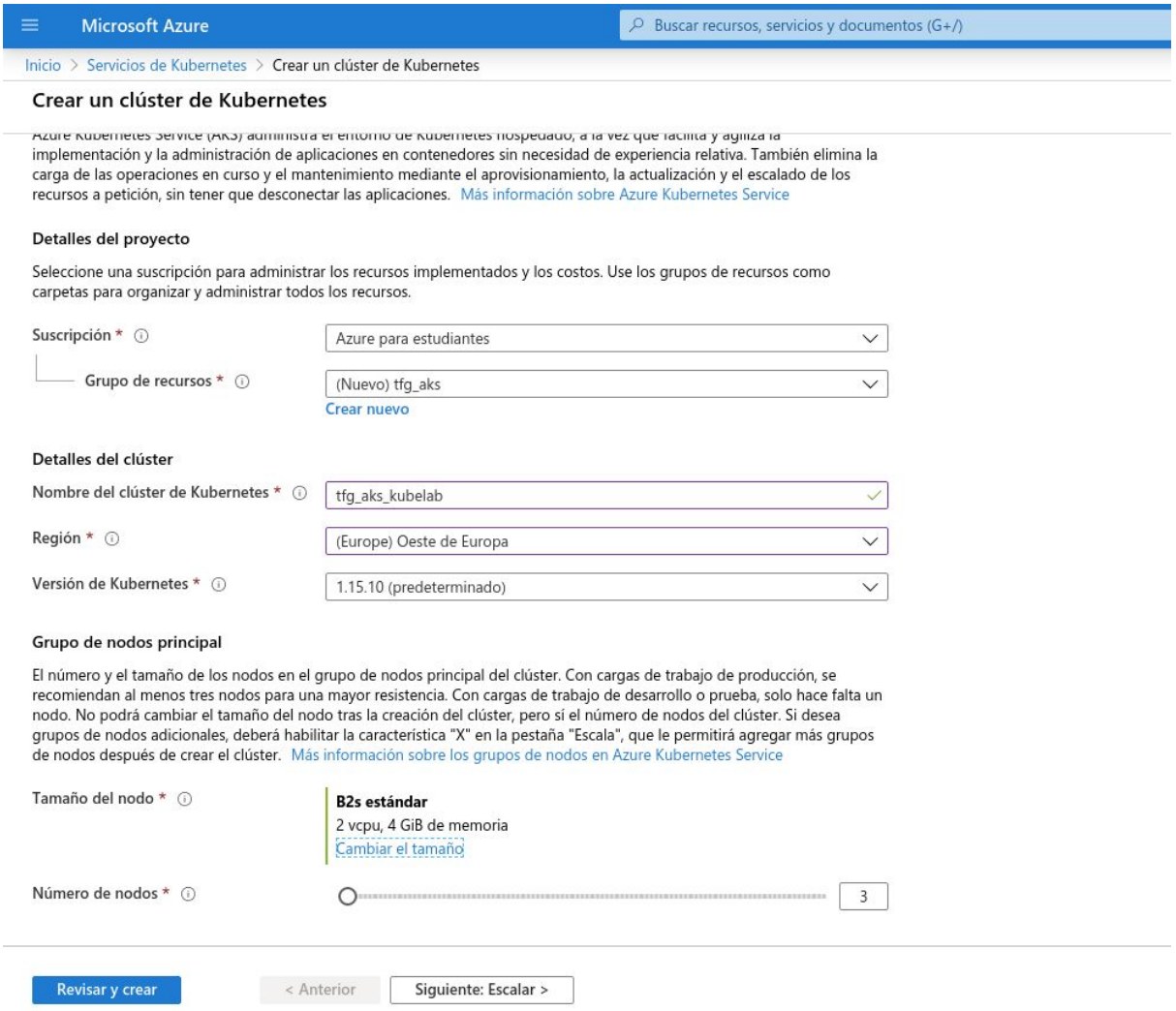

Figura 3.2. Página de configuración de un *cluster* AKS

Una vez se ha creado el *cluster* ya podemos conectarnos a él. Azure ofrece la posibilidad de conectarse desde el propio portal a través de una consola de comandos que tiene incorporada, como se muestra en la Figura 3.3.

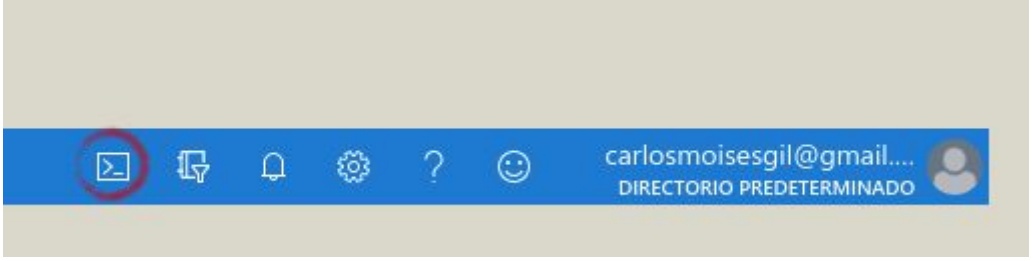

Figura 3.3. Acceso a Azure Cloud Shell

Una vez aparece el símbolo del sistema, para conectarnos a nuestro *cluster* de Kubernetes solo hay que ejecutar el siguiente comando:

```
$ az aks get-credentials --resource-group myResourceGroup --name myAKScluster
```

Ahora ya podemos hacer uso de Kubectl para gestionar nuestro *cluster* de Kubernetes como se muestra en la Figura 3.4, en la que se muestran los nodos activos de nuestro *cluster*.

```
carlos@Azure:~$ kubectl get nodes
NAME                                STATUS   ROLES   AGE     VERSION
aks-agentpool-19307125-vmss000000   Ready    agent   9m59s   v1.15.10
aks-agentpool-19307125-vmss000001   Ready    agent   10m     v1.15.10
```

Figura 3.4. Ejecución de kubectl get nodes en AKS

Una vez se ha hecho todo este proceso, las instrucciones a seguir para lanzar nuestros servicios son muy parecidas a como se hacía con nuestro *cluster* de Minikube. Al desplegar nuestro *cluster* en una plataforma Cloud, existe una novedad muy importante como es el hecho de poder usar *LoadBalancer*. *LoadBalancer* es una alternativa a *NodePort*, que usábamos en *Minikube*, para el tipo de servicio que se usa en nuestros despliegues. *LoadBalancer* permite externalizar una IP para acceder a nuestro servicio desde cualquier parte, sin necesidad de hacer puentes como ocurre con *NodePort.*

Usar *LoadBalancer* para poder asignar una IP externa a un servicio es una de las grandes ventajas que ofrece el uso de tecnologías *Cloud*, ya que con *Minikube* no se puede utilizar este tipo de servicio.

Ahora vamos a desplegar nuestro SQL Server en nuestro servicio AKS para ver si funciona correctamente también aquí. En este caso se va a desplegar una instancia de SQL Server vacía, sin ninguna base de datos importada. Además, se desplegará como un servicio de tipo *LoadBalancer,* y también se puede hacer uso de algunas características, como los *Azure Disk* para crear la *StorageClass,* para mejorar el rendimiento de la instancia en Kubernetes.

```
carlos@Azure:~/mssql-deployment$ kubectl apply -f pvc.yaml
storageclass.storage.k8s.io/azure-disk created
persistentvolumeclaim/mssql-data created
```

Figura 3.5. Creación de los elementos para el almacenamiento (StorageClass y PersistentVolumeClaim)

En la Figura 3.5 se ve como creamos los elementos para almacenar la información persistente de nuestra instancia. No tienen diferencias significativas con respecto a las usadas en el Capítulo 2. Únicamente el detalle mencionado antes del *Azure Disk.*

Azure Disk sirve para crear un elemento de disco de almacenamiento dentro del grupo de recursos donde se encuentra el Servicio Kubernetes alojado. Estos discos SSD agilizan los procesos de acceso a los datos a través del uso de los *PersistentVolumeClaim* desde los despliegues realizados.

En cuanto al *service*, la única novedad, como ya hemos comentado, es que ahora será de tipo *LoadBalancer* en lugar de *NodePort.* De este modo podremos acceder desde cualquier máquina ya que se expondrá una IP pública. En el *Deployment* solo habrá una diferencia, se creará una instancia vacía sin bases de datos dentro. Debido a esto, usaremos una imagen propia de Microsoft de SQL Server 2017 para Linux.

```
carlos@Azure:~/mssql-deployment$ kubectl apply -f deployment.yaml
deployment.apps/mssql-deployment created
service/mssql-deployment created
```

Figura 3.6. Creación del Deployment y el Service

Después de ejecutar la sentencia de la Figura 3.6 se crea el *Pod* que tardará unos segundos y después se debe ver algo similar a lo que se ve en la Figura 3.7.

```
carlos@Azure:~/mssql-deployment$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
mssql-deployment-89c7d4666-wv74x    1/1     Running   0          24s
```

Figura 3.7. Vista del Pod

Ahora ya solamente queda comprobar que se puede acceder a la instancia de SQL Server conectándonos desde la máquina que estemos usando. Para ello se necesita la IP externa como se muestra en la Figura 3.8.

```
carlos@Azure:~/mssql-deployment$ kubectl get svc
NAME               TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)          AGE
kubernetes         ClusterIP      10.0.0.1      <none>          443/TCP          4m59s
mssql-deployment   LoadBalancer   10.0.5.202    51.124.17.197   1433:31749/TCP   4m3s
```

Figura 3.8. IP externa del servicio mssql

Una vez tenemos esta IP únicamente queda conectarse usando `sqlcmd` y ejecutar alguna instrucción para comprobar que funciona, como ocurre en la Figura 3.9.

```
cagil@pc-carlos:~$ sqlcmd -S  51.124.17.197  -U sa -P PaSSw0rd
1> select @@VERSION
2> select @@SERVERNAME
3> GO




------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------------------------------------------------------
Microsoft SQL Server 2017 (RTM-CU20) (KB4541283) - 14.0.3294.2 (X64)
        Mar 13 2020 14:53:45
        Copyright (C) 2017 Microsoft Corporation
        Developer Edition (64-bit) on Linux (Ubuntu 16.04.6 LTS)


(1 rows affected)


------------------------------------------------------------------------------
----------------------------------------------
mssql-deploymen


(1 rows affected)
```

Figura 3.9. Conexión con la instancia de SQL Server

Capítulo 4:

Análisis del sistema

4.1. Introducción

Una vez definidas las características de la creación de *clusters On Premise* y *Cloud* debemos empezar a estudiar si nuestra arquitectura de Kubernetes responde bien a nuestras necesidades. Como se dijo en el capítulo de Introducción, la cuestión es si esta tecnología, a día de hoy, es fiable para una plataforma de datos en producción. Para que sea fiable, como ya dijimos, debemos saber la disponibilidad del servicio (menor número posible de veces que no está disponible el servicio a lo largo de un tiempo).

En este capítulo se verá cómo funciona Kubernetes Dashboard y si nos sirve para nuestro propósito, es decir si podemos monitorizar los recursos y logs que genera nuestro servicio. Aparte de esto se desplegará un servicio demonio que monitoree nuestra arquitectura y almacene la información para su posterior análisis con herramientas de visualización de datos como PowerBI.

Para esta parte se seguirá usando la instancia de SQL Server desplegada en el Capítulo 2. Se almacenarán los logs generados en el *Pod* de la instancia y las correspondientes métricas para su posterior análisis.

Para esa parte se introducirá un nuevo concepto en la arquitectura de Kubernetes que es el *DaemonSet.* Se verá cómo funciona y para qué podemos usarlo en el caso que estamos tratando.

En los siguientes apartados se detallan:

4.2. Kubernetes Dashboard
4.3. Definición de la arquitectura del sistema
4.4. Registro de logs
4.5. Registro de métricas
4.6. Análisis con PowerBI

4.2. Kubernetes Dashboard

Kubernetes ofrece una API gráfica para el manejo de los recursos de nuestra arquitectura Kubernetes. Desde la versión 1.13 esta API se ejecuta automáticamente al arrancar nuestro *cluster* de Kubernetes sin necesidad de hacer nada. Para acceder a esta API solo necesitamos abrir un navegador web y escribir la URL que nos proporcione nuestro *cluster* para el acceso.

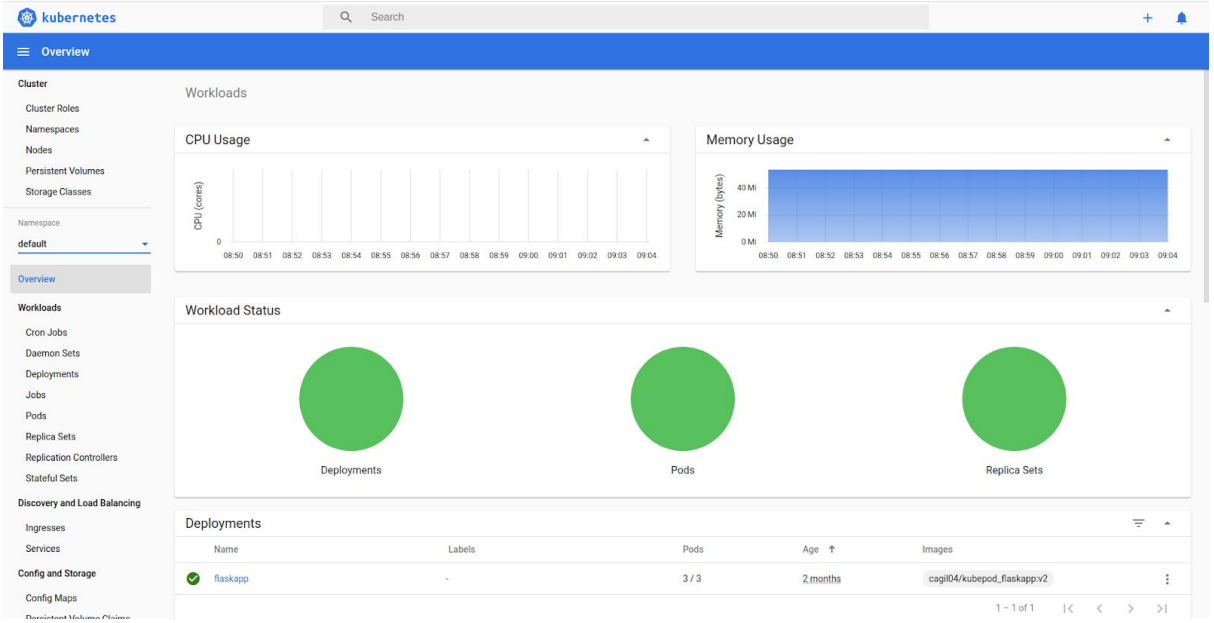

Figura 4.1. Kubernetes Dashboard Overview

El aspecto del *Dashboard* es el que se muestra en la Figura 4.1. Ahí se puede ver que muestra una imagen general del *cluster* de Kubernetes al que accedemos. En la parte izquierda se ve que se puede acceder a todos los recursos propios de la arquitectura que se han visto anteriormente, *Pods, Deployments, Services, StorageClass…*

En la parte de Workload Status se muestra la situación actual de los *Deployments*, *Pods* y *Replica Sets*. En la figura se muestran los tres círculos completamente verdes y eso quiere decir que están funcionando todos correctamente (su estado es *running)*. En el caso de que, por ejemplo, un *Pod* fallara aparecería la porción del círculo correspondiente en rojo. Si, por el contrario, no se hubiera puesto todavía en funcionamiento y no ha ocurrido ningún problema aparecería en un color amarillo.

En la parte superior derecha se ve un símbolo de "+" que sirve para añadir nuevos elementos a nuestro *cluster*, por ejemplo, podemos crear un nuevo despliegue desde ese apartado. Para este cometido se puede hacer de manera guiada o a partir de un fichero yaml, como hemos hecho en los capítulos anteriores siempre que hemos querido crear algún nuevo elemento. En la Figura 4.2 se muestra la primera opción.

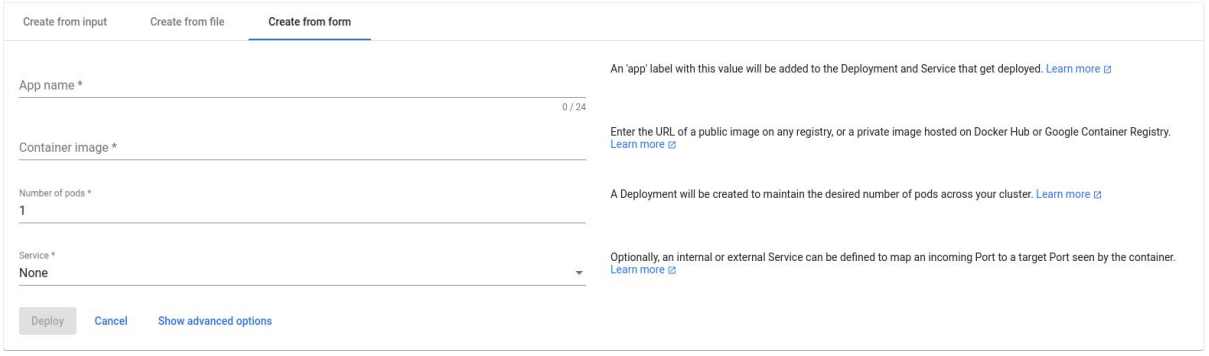

Figura 4.2. Creación de un despliegue mediante un formulario

También se puede ver que nos ofrece información acerca del uso de recursos por parte del *cluster*, CPU y memoria. Además se puede acceder a los logs de un *Pod* determinado. Pero a la hora de analizar tanto las métricas de recursos como los logs surgen dos problemas:

- Los logs se muestran de manera desestructurada. Por lo tanto si hay pocos logs no debería suponer un problema, pero si hubiese muchos el análisis se complicaría considerablemente.
- Las métricas de uso de recursos almacenan información de los últimos diez minutos. Si quisiéramos ver si un fallo tuvo que ver con el uso de recursos hace una hora, no tendríamos forma de saberlo.

Ante estos dos problemas habría que llevar a cabo dos tareas, la estructuración de esos datos para su análisis y el almacenamiento de esas métricas de manera persistente para poder analizar de una manera adecuada el funcionamiento de nuestro *cluster* de Kubernetes.

4.3. Definición de la arquitectura del sistema

Ante esta situación se debe crear un sistema automático que almacene los datos de manera estructurada de tal manera que su posterior análisis sea lo más sencillo posible. La estructura de los datos que vamos a usar es en forma de *Data Warehouse* (en castellano Almacén de Datos), que es un modelo mejor para el análisis. Por otra parte se van a utilizar dos procesos automáticos que leerán de los ficheros donde se almacenan los logs y las métricas para procesarlos y almacenarlos en el *Data Warehouse.*

Para tener el *Data Warehouse* se necesita una base de datos independiente de la que queremos analizar para no generar bucles de escrituras. Por lo que se crea una segunda instancia para el almacenamiento de los datos.

Este *Data Warehouse* tendrá dos tablas de hechos, una para logs y otra para métricas. En este caso la información de los hechos se puede contener en la propia tabla de hechos, ya que los campos de las tablas de hechos son los siguientes:

- Logs:
    - Fecha
    - Hora
    - Texto del log
    - Flujo (salida estándar o salida de error)
- Métricas:
    - Fecha
    - Hora
    - Memoria
    - CPU

Por otro lado los dos procesos antes mencionados consistirán en dos demonios o servicios que se lanzarán en segundo plano. En este caso los procesos se centrarán solo en los datos generados por los pods que corresponden al despliegue de SQL Server sobre la arquitectura de Kubernetes.

Una vez que todo funcione correctamente y los datos se almacenen, habrá que conectarse con PowerBI al *Data Warehouse* que se ha creado y ya se podrán elaborar los *Dashboard* que permitan hacer un buen análisis del despliegue de SQL Server sobre Kubernetes en cuestión.

4.4. Registro de logs

En primer lugar se define el *script* de Python que permite crear un demonio para después poder importarlo en el *script* principal, donde se define el proceso de almacenamiento de los logs. Este *script*, llamado `daemon.py`, ejecuta dos veces la función `fork()`, que crea un proceso hijo.

El segundo *script* que necesitamos es `logs_storage.py`, que establece el comportamiento del demonio creado al ejecutar la función `daemonize()` del *script* anteriormente detallado. Una vez creado el demonio, este consulta los *pods* activos dentro del *namespace* correspondiente para extraer sus nombres. En nuestro caso únicamente hay un *pod,* por lo tanto nos quedaremos con un único nombre. Después de conseguir el nombre se llama a la función `read_logs()` que se encarga de realizar las siguientes tareas:

1. Consulta los logs del *Pod* mediante el uso de Kubectl.
2. Consulta la fecha y hora del último log insertado en el *Data warehouse*.
3. Si la tabla está vacía inserta en el *Data warehouse* todos los logs, si no lo está comprueba para cada log del *Pod* su fecha y hora e inserta aquellos logs con una fecha y hora posteriores a las de la última inserción.
4. El proceso se suspende durante una hora y se vuelve a ejecutar.

Para lanzar el demonio a ejecución hay que ejecutar la instrucción que se muestra a continuación:

```
$ python3 logs_storage.py
```

Para que esto funcione es necesario que el cluster de Kubernetes esté ejecutándose, de lo contrario recibiremos un error diciendo que el recurso al que se intenta acceder no existe. Una vez ejecutado el comando podremos seguir escribiendo comandos en la consola ya que el proceso padre habrá muerto y se estará ejecutando el hijo, es decir el demonio.

4.5. Registro de métricas

Para el registro de métricas usamos el *script* `daemon.py`, como hacíamos con el registro de logs, para crear el demonio que permita monitorizar las métricas en segundo plano. Aparte de este *script* necesitamos el código que va a ejecutar el proceso demonio, en este caso el nombre del fichero es `metrics_storage.py`.

Al igual que ocurría en el registro de los logs, aquí también se consultan los *Pods* activos dentro del *Namespace* y extrae su nombre. Cuando tiene el nombre del *Pod* se llama a la función `read_metrics()` que realiza los siguientes pasos:

1. Consulta las métricas del *Pod* para ese instante mediante el uso de Kubectl.
2. Carga el archivo JSON (JavaScript Object Notation) que contiene la información de las métricas.
3. Extrae la información de memoria y CPU.
4. Inserta los datos de memoria y CPU con la fecha y hora de la consulta en el *Data Warehouse*.
5. El proceso se suspende durante un minuto y se vuelve a ejecutar.

Como se ve el proceso se ejecuta una vez por minuto, por lo tanto nos permite almacenar muchas observaciones de memoria y CPU por día. De este modo se puede hacer un análisis más fino en el futuro.

Para lanzar el demonio a ejecución hay que ejecutar el siguiente comando:

```
$ python3 metrics_storage.py
```

Igual que ocurría con el registro de los logs, para las métricas también debe estar el *cluster* de Kubernetes ejecutándose antes de lanzar el proceso. En el caso de las métricas debemos esperar unos minutos desde que se empieza a ejecutar el *cluster* para que el servidor de métricas de Kubernetes comience a funcionar. Si lanzamos el proceso antes de tiempo nos dará un error y se matará el registro de métricas. Si todo funciona correctamente tendremos el demonio ejecutándose y el control de la consola de nuevo, como ocurría con el proceso de los logs.

4.6. Análisis con PowerBI

Ahora que ya están listos los procesos para almacenar las métricas y los logs podemos crear los cuadros de mando de PowerBI. Lo primero que hay que hacer es obtener los datos que se van a usar. PowerBi ofrece múltiples opciones para obtener los datos como se muestra en la figura, en este caso elegiremos SQL Server, que es donde tenemos el Data Warehouse.

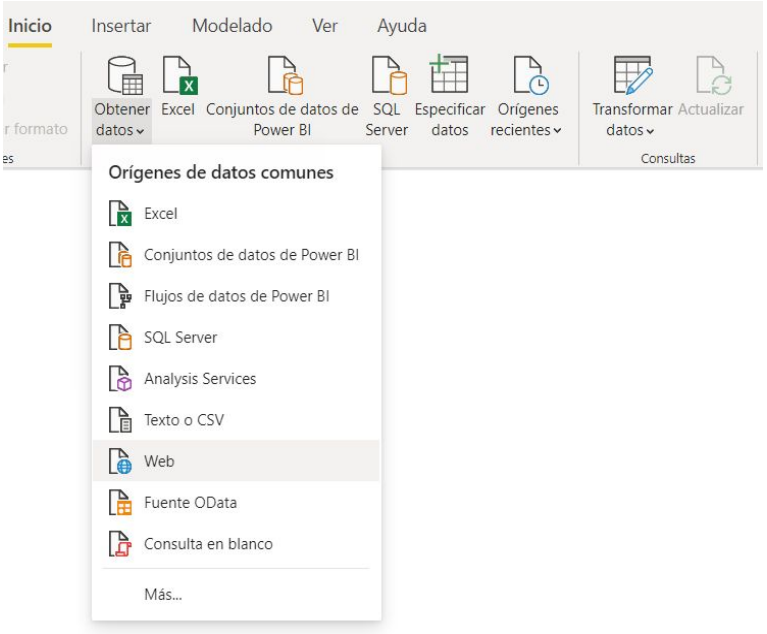

Figura 4.3. Orígenes de datos en PowerBI

Al seleccionar la opción de SQL Server se tendrán que introducir las credenciales para conectarse a la instancia y la base de datos concreta de la que obtener los datos. Una vez que se hayan cargado las tablas de la base de datos ya se pueden crear los cuadros de mando.

A la hora de obtener los datos tenemos dos opciones:

- ***DirectQuery***: esta opción permite consultar los datos directamente de la base de datos, y de esta forma si los datos originales se actualizan se actualizan los de nuestros cuadros de mando.
- **Importar**: esta opción importa los datos en el propio archivo de PowerBI y son fijos, es decir no cambian si se actualiza el origen de los datos.

DirectQuery es idóneo si se quiere analizar los datos conforme se van actualizando pero nuestro origen de datos debe estar siempre activo y no es una solución portable. Por otro lado, la opción de importar los datos permite una solución portable sin necesidad de tener acceso al origen de los datos aunque estos datos no se van a actualizar. En nuestro caso se usará la segunda opción para que la solución sea portable y se pueda visualizar sin necesidad de montar todo el despliegue. Para que haya mayor volumen de datos con los que trabajar se dejará nuestro *cluster* funcionando con los servicios monitorizando durante un tiempo.

Para este análisis se ha creado una plantilla con 4 páginas que ayuden al análisis de nuestro sistema. A continuación se enumeran las páginas:

- Memoria y CPU por minuto de monitorización
- Memoria y CPU promedio por horas del día
- Memoria y CPU promedio por días de la semana
- Filtro de logs por fecha y hora

Las unidades de medida de CPU y memoria para Kubernetes son *m* y *mi* respectivamente. La *m* de la CPU se refiere a una división lógica de la CPU en *milicpus*, esto quiere decir que si hay una medida de 21 estamos usando un 2,1% de una CPU. Por otro lado el *mi* de memoria se refiere a Mebibytes, es decir si observamos un valor de 10 hacemos uso de 10 Mebibytes de memoria. Los Mebibytes son similares a los Megabytes, pero en vez de ser potencias de 10 son potencias de 2 (1 Mebibyte = 2^{20} Bytes, 1 Megabyte = 10^6 Bytes)

Para poder crear las gráficas correspondientes a los promedios se crearon tres columnas adicionales en el modelo de datos dentro de Power BI. Dos para guardar el día de la semana con número y nombre, y otra para guardar la hora, todas mediante funciones propias de Power BI.

En las siguientes figuras se muestran las distintas páginas creadas:

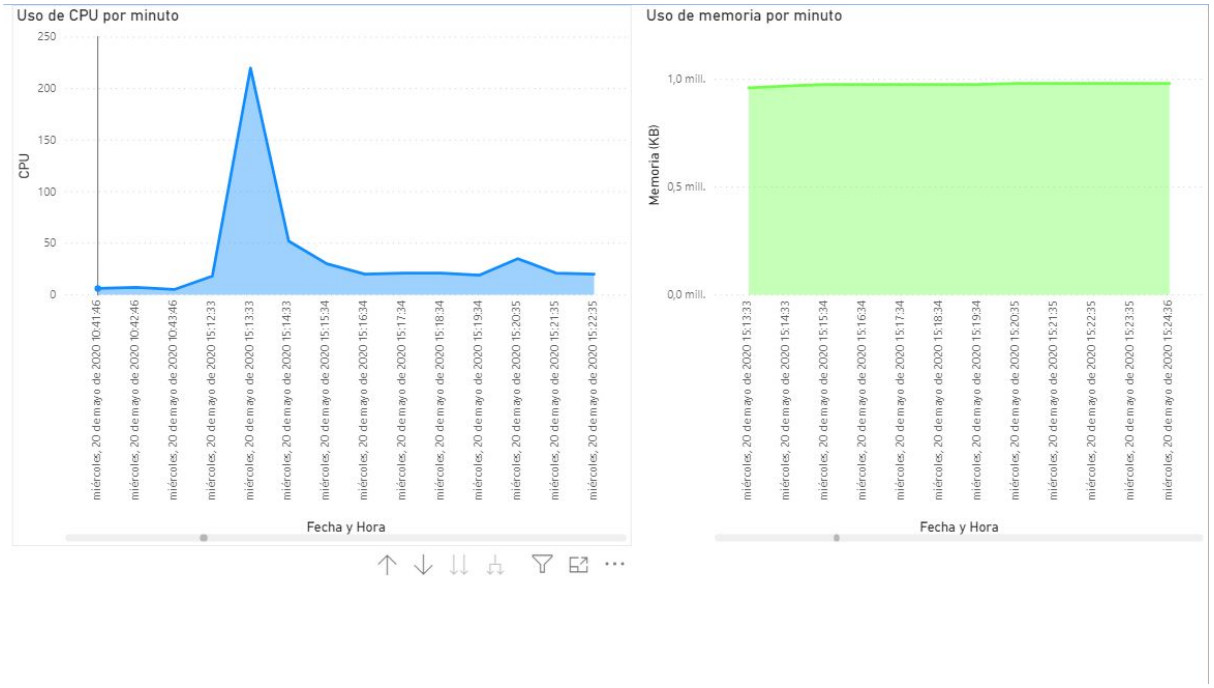

Figura 4.4. Memoria y CPU por minuto de monitorización

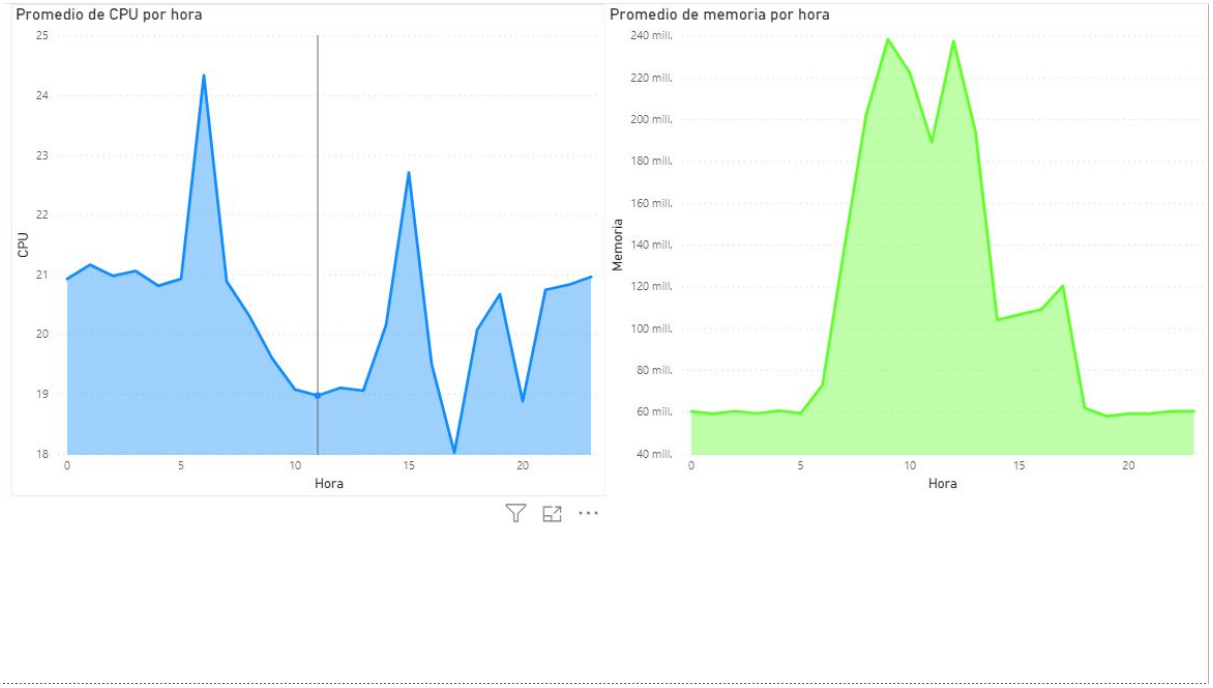

Figura 4.5. Memoria y CPU promedio por horas del día

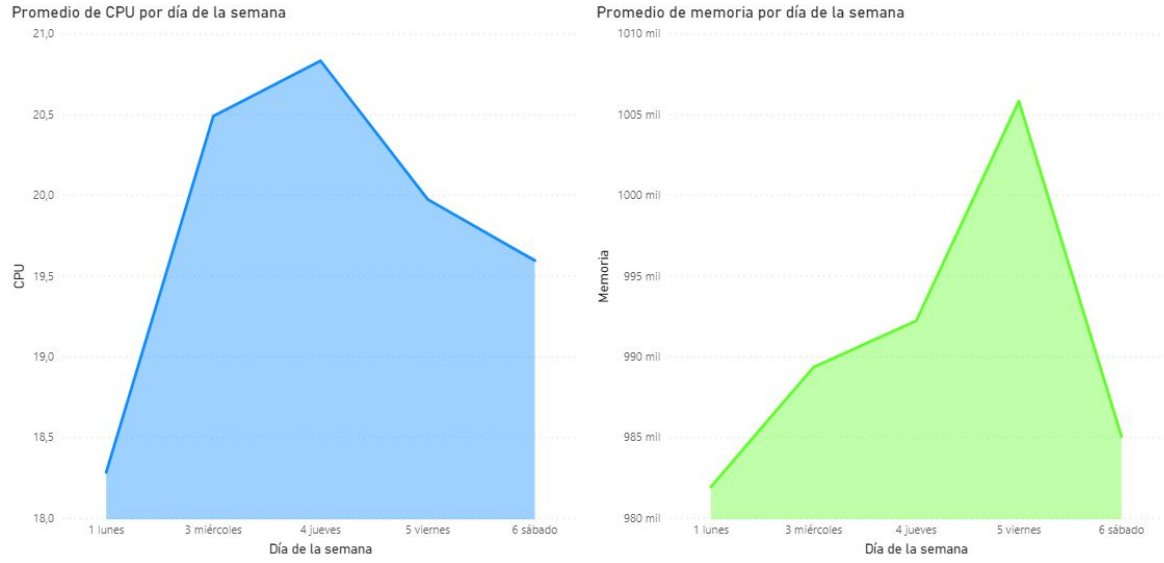

Figura 4.6. Memoria y CPU promedio por días de la semana

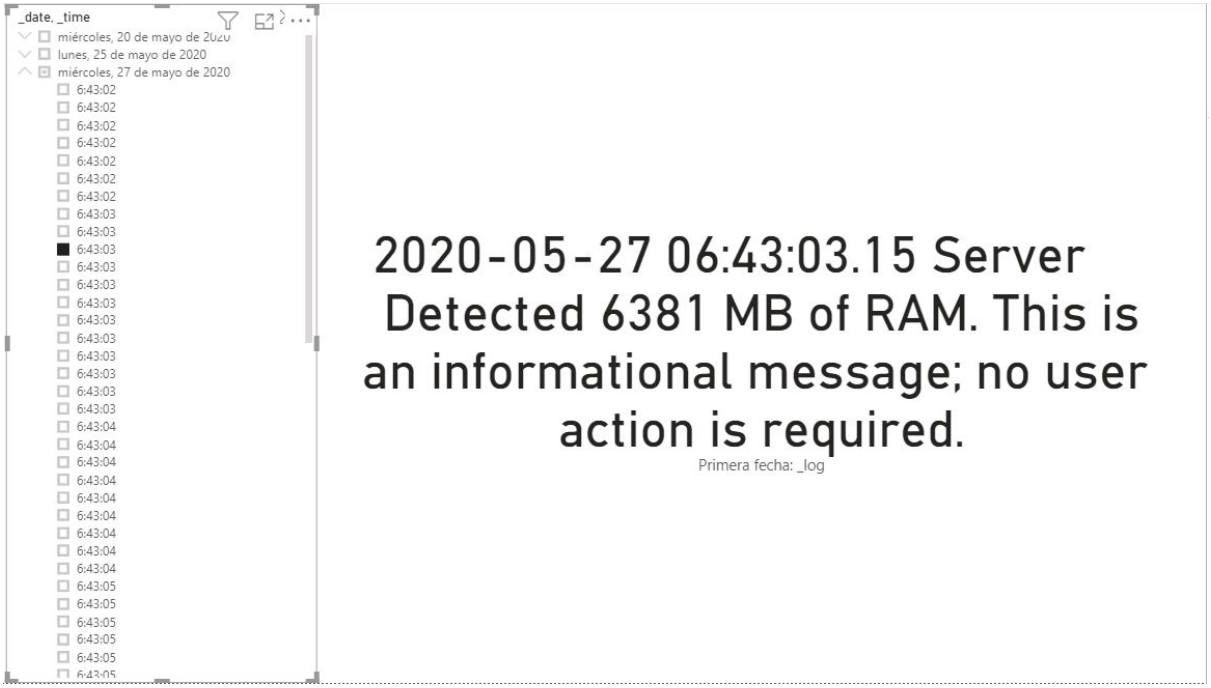

Figura 4.7. Filtro de logs por fecha y hora

Capítulo 5:

Conclusiones del trabajo

Este trabajo empezó como una investigación del estado del arte de Kubernetes y si podía ser una tecnología consistente para la implementación de una plataforma de datos en producción. Tuve que realizar una primera fase de aprendizaje de las tecnologías con las que iba a trabajar antes de poder centrarme en el desarrollo del trabajo. Fue un camino complejo ya que hay muchas tecnologías involucradas en este trabajo (Docker, Kubernetes, SQL Server, Microsoft Azure, Python, Power BI...).

Este proceso de aprendizaje me ha aportado una visión general de algunas de las tecnologías más potentes del momento y una visión más concreta de la tecnología principal de este trabajo, Kubernetes. Han sido meses en los que no solo he adquirido estos conocimientos, sino que he podido poner en práctica otros que ya había adquirido a lo largo de la carrera. Conocimientos como aquellos que tienen que ver con bases de datos, sistemas operativos o redes entre otros. Por eso creo que este trabajo sirve de nexo entre aquello aprendido en la carrera y aquello aprendido de manera autónoma para el desarrollo del mismo.

En cuanto a la parte de análisis de este trabajo nos hacíamos una serie de preguntas al principio del mismo. Las características fundamentales para poder responder a estas preguntas han sido respondidas a lo largo de los capítulos anteriores y ahora toca dar una respuesta en base a aquello que ya hemos explicado.

¿Para qué puede ser usado Kubernetes?

El Capítulo 2 sirvió para responder a esta pregunta mostrando sus características más importantes, ya sabemos para qué sirve. Entonces vayamos un poco más allá y analicemos el estado de la tecnología como tal.

Kubernetes es una tecnología que ha crecido de manera exponencial en los últimos años a pesar de su juventud, apenas tiene 6 años. Su capacidad para facilitar los procesos de lanzamiento y mantenimiento de aplicaciones han hecho que sea una tecnología que haya llamado la atención, pero como toda tecnología ha tenido que seguir un proceso de maduración para llegar a ser una tecnología usada en el mundo real. El hecho de tener a un gigante tecnológico como Google detrás ha hecho que ese proceso de maduración se acorte considerablemente.

Hoy en día ya es una tecnología referente con muchos casos de éxito como los de Booking, Adidas, IBM, Huawei y otras grandes empresas que utilizan esta tecnología para sus aplicaciones. Aun así es una tecnología que, a buen seguro, irá creciendo y mejorando con los años para ofrecer más utilidades a sus usuarios.

Personalmente creo que es una tecnología potente que hoy ya es referente, pero en un futuro próximo se convertirá en un estándar ya que es una herramienta que agiliza unos procesos de puesta en producción que resultan tediosos.

¿Cloud u On Premise?

En el Capítulo 3 vimos las principales características de ambas opciones y vimos las ventajas y desventajas que ofrecían, pero no se dio una respuesta definitiva. En la pregunta anterior vimos que Kubernetes es una tecnología consolidada que despierta mucho interés y debido a ello, los principales proveedores de plataformas *Cloud* han decidido crear soluciones que tengan Kubernetes ya configurado y, además, ofrecer características propias que los diferencien. Google, Microsoft, Amazon o IBM son algunos de los proveedores que han lanzado ya varias soluciones *Cloud* que implementan Kubernetes con características propias de cada uno de ellos.

Recordemos que Google es quien está detrás de Kubernetes y es uno de los proveedores que más soluciones *Cloud* distintas ofrece a sus usuarios. Debido a esto pienso que Kubernetes es una tecnología con una clara orientación *Cloud*, lo cual no quiere decir que no se pueda usar en entornos *On Premise*, pero se ofrecen muchas más facilidades en la primera opción.

No obstante en el Capítulo 3 ya se habló de una solución híbrida que combinaba ambas opciones y creo que es una opción bastante buena también. No debemos olvidar que hay entornos que no pueden ser desplegados en *Cloud* debido a las normativas vigentes en ciertos sectores, por lo tanto un entorno híbrido sería una solución al problema.

En este caso no hay una respuesta definitiva y única, hay que analizar costes de ambas opciones y decidir en función de ellos qué opción se adapta mejor a cada escenario.

¿Cómo se puede saber si va a responder a nuestras necesidades?

Para esta pregunta se implementó un sistema de análisis para el despliegue de SQL Server en Kubernetes en el Capítulo 4. Este sistema fue una sencilla solución que finalmente nos permitía ver un histórico de observaciones sobre nuestro despliegue y sería una herramienta útil para poder analizar una instancia de SQL Server en producción.

No podemos replicar un entorno de producción con todas las transacciones realizadas propias de este tipo de entornos así que resulta difícil responder a esta cuestión de manera definitiva. En su defecto la solución más aproximada sería lanzar este sistema de análisis en un entorno de prueba que replicase al de producción y hacer el correspondiente análisis a posteriori viendo el número de caídas, si las hay, con el consumo de recursos. Esta solución nos aportaría conocimiento a la hora de tomar una decisión sobre si queremos lanzar nuestra plataforma de datos SQL Server sobre Kubernetes o no.

¿Es recomendable su uso para la gestión de bases de datos?

La respuesta a esta pregunta sigue en la línea de lo anterior, es posible el despliegue de SQL Server sobre Kubernetes, pero no podemos dar una respuesta segura sobre si es fiable en un entorno de producción. No es una cuestión fácil de responder ya que no he encontrado casos de uso de SQL Server sobre Kubernetes. Realmente creo que el camino a seguir sería lo establecido en el punto anterior, analizando resultados en una arquitectura de test para la posterior toma de la decisión.

Chapter 5:

Project conclusions

This project began as a Kubernetes state of art investigation and whether it could be a consistent technology for a production data platform implementation. I had to do a first step of the project technologies learning before to be able to focus on the project development. It was a complex path because there are many technologies involved in this project (Docker, Kubernetes, SQL Server, Microsoft Azure, Python, Power BI...).

This learning process has provided me with a global view of some of the most powerful technologies at this moment and a more specific view of this project main technology, Kubernetes. They've been months wherein I have not only acquired this knowledge, but I have been able to put into practice others that I got along the university degree. Knowledge like those related to databases, operating systems and networks among other things. That's the reason why I think this work acts as a link between that degree learning and the autonomous learning for the development of it.

In respect of the part of the analysis of this work we had some questions at the beginning of it. The main features to be able to answer to these questions have been answered along the previous chapters and now is the moment to give an answer on the basis of what we already explained.

What can be used Kubernetes for?

The Chapter 2 was used to answer this question showing its more important features, we already know what it is used for. Then let's go further and let's analyze the state of the technology itself.

Kubernetes is a technology that has grown exponentially in the last years despite its youthness, it is just 6 years old. Its ability to make easy the processes of releasing and maintenance of applications has made this a technology that attracts attention, but like every technology had to follow a process of maturation to become a used technology in the real world. The fact of having a technology giant like Google behind made that process of maturation gets reduced.

Nowadays it's already a referent technology with a lot of successful cases like Booking, Adidas, IBM, Huawei and other big companies that use it for their applications. Despite of this, it is a technology that will grow and improving along the years for sure to offer more utilities to their users.

Personally, I think it is a powerful technology that is a reference today, but in the near future it will become a standard as it is a tool that streamline the processes of production readiness that are tedious.

¿Cloud or On Premise?

In the Chapter 3 we saw the main features of both options and the advantages and disadvantages they offered, but it wasn't given a final answer. In the previous question we saw that Kubernetes is an established technology that wakes much interest up and related to that, the main Cloud providers decided to make solutions that have Kubernetes already configured and, in addition, to offer unique features that differentiate them. Google, Microsoft, Amazon or IBM are some of the providers that already released several Cloud solutions that implement Kubernetes with unique features of each one of them.

Let's remember that Google is who is behind of Kubernetes and it is one of the providers who more different Cloud solutions offers to their users. Because of this I think Kubernetes is a technology with a clear Cloud orientation, this doesn't mean that it can't be used On Premise, but there are much more facilities in the first option.

However in the Chapter 3 we already talked about a hybrid solution that combined both options and I think it is a quite good option too. We must not forget that there are environments that can't be deployed on Cloud because of the current rules in some sectors, therefore a hybrid environment would be a solution to this problem.

In this case there's not a definitive and unique answer, the costs of both options would have to be analyzed and choose according to them which option is better for each scenario.

How can you know if it is going to meet our requirements?

For this question it was implemented an analysis system in the Chapter 4 for SQL Server deployment on Kubernetes. This system was a simple solution that finally allowed us to see an observations history about our deployment and it would be an useful tool to be able to analyze a SQL Server instance in production.

We can not replicate a production environment with all the kind of transactions made in this types of environments, so it results difficult to answer this question in a definitive way. In the absence of it, the most appropriate solution would be to launch this analysis system in a testing environment that replicated the one of production and to make the correspondent analysis watching the number of interruptions, if there are anyone, with the resources consumption. This solution would give us knowledge at the moment to make a choice about whether we want launch our SQL Server Data Platform on Kubernetes or not.

Is its use advisable for databases management?

The answer to this question keeps the way of the one above; it is possible the SQL Server deployment on Kubernetes, but we cannot give a safe answer about if it is reliable in a production environment. It is not an easy question since i did not find use cases of SQL Server on Kubernetes. I really think the way to follow would be the one established in the previous point, analyzing results in a testing environment for the afterwards decision making.

Capítulo 6:

Trabajos futuros

Como última parte de este trabajo se quiere mostrar cuáles serían algunos de los posible trabajos futuros a realizar a continuación de este. Esto es algo que puede tomar distintos caminos por eso lo divido en los siguientes puntos:

- **Continuación del análisis**: en este sentido se podrían crear procesos que extrajeran más información del despliegue o de distintos despliegues, también podrían crearse cuadros de mando más elaborados que contasen una historia como se suele hacer en proyectos de *Business Intelligence*.
- **Ampliación del despliegue**: se podría hacer un despliegue completo en el cual se conectasen varios servicios de una aplicación a la base de datos o crear procesos completos de ciencia de datos con despliegues de Kubernetes.
- **Uso de herramientas existentes**: una última opción sería el estudio y utilización de herramientas que se han creado a partir de Kubernetes para implementar plataformas de datos versátiles y de gran complejidad dentro del paradigma *Big Data*. Un ejemplo de este tipo de herramientas sería Big Data Cluster que creó Microsoft para el lanzamiento de SQL Server 2019.

Estas son algunas de las ideas que se proponen pero pueden existir más puesto que Kubernetes ofrece muchas posibilidades y el mundo de los datos es un mundo cada vez más complejo que exige soluciones cada vez mejores.

Bibliografía y referencias

Aspin, A. *Pro Power Bi Desktop*, Second.; Apress: United States, 2017.

Ward, B.; Oks, S. *Pro Sql Server on Linux : Including Container-Based Deployment with Docker and Kubernetes*; Apress: New York, 2018.

Buchanan, S.; Rangama, J.; Bellavance, N. *Introducing Azure Kubernetes Service : A Practical Guide to Container Orchestration*; Apress L.P: Berkeley, CA, 2020.

[1]https://aws.amazon.com/es/microservices/
[2]https://www.altia.es/es/altia
[3]https://www.solidq.com/es/
[4]https://kubernetes.io/es/
[5]https://palletsprojects.com/p/flask/
[6]https://www.docker.com/resources/what-container
[7]https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/
[8]https://www.docker.com/blog/designing-your-first-application-kubernetes-communication-services-part3/
[9]https://medium.com/@KevinHoffman/building-a-kubernetes-cluster-in-virtualbox-with-ubuntu-22cd338846dd
[10]https://vitux.com/install-and-deploy-kubernetes-on-ubuntu/
[11]https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/
[12]https://github.com/enriquecatala/mssql-server-samplesdb
[13]https://github.com/kubernetes/kubernetes/tree/master/pkg/kubelet/apis

Apéndice A:

Preparación del entorno On Premise

A.1. Preparación de la máquina virtual

Para realizar las pruebas que se detallan a lo largo del primer capítulo del documento se ha configurado una máquina virtual en Oracle VirtualBox. Las características de la máquina son las siguientes:

- Sistema operativo: Ubuntu Server 18.04 LTS
- Memoria: 8GB
- Nº de procesadores: 2
- Almacenamiento: VDI reservado dinámicamente

Para configurar la máquina y que podamos usarla de la mejor manera es necesario llevar a cabo una serie de pasos. Los pasos a seguir son los siguientes:

1. Creamos la máquina siguiendo los pasos mostrados en las Figuras A.1 y A.2. Posteriormente accedemos a la configuración de la máquina virtual y modificamos el nº de procesadores como se muestra en la Figura A.3:

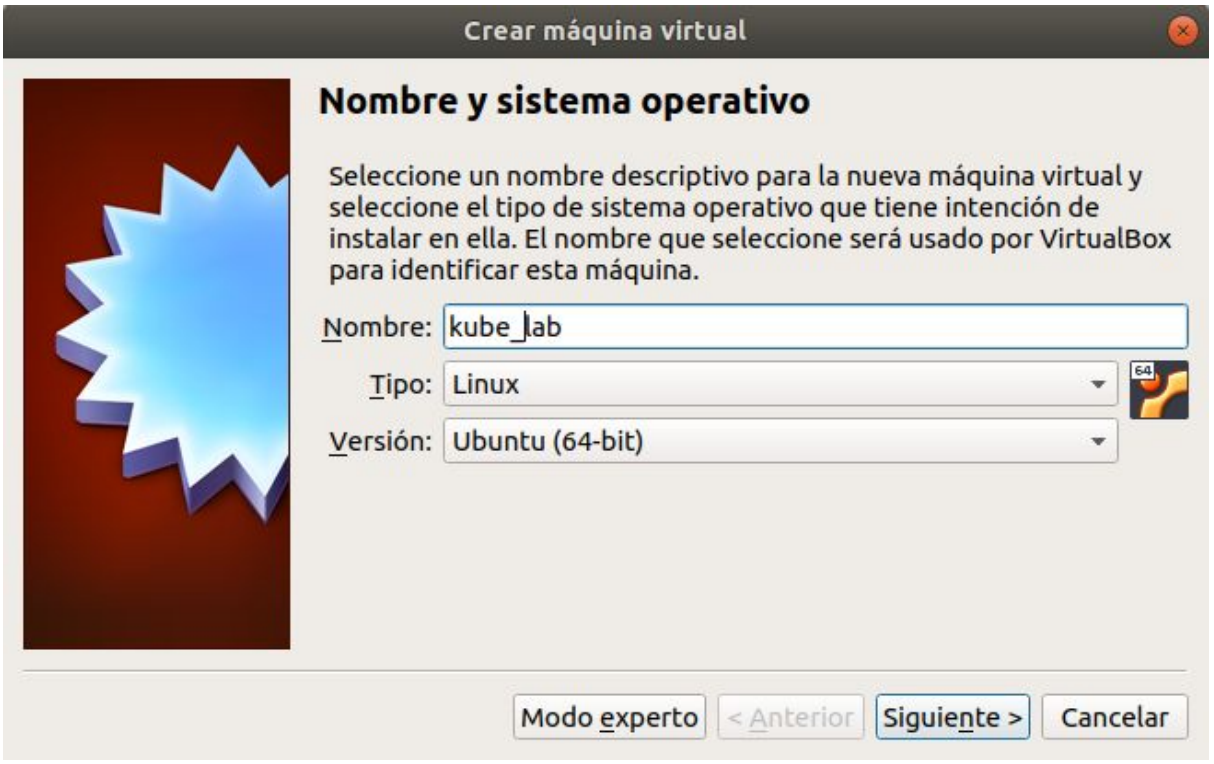

Figura A.1. Crear máquina virtual.

Figura A.2, Crear disco duro virtual.

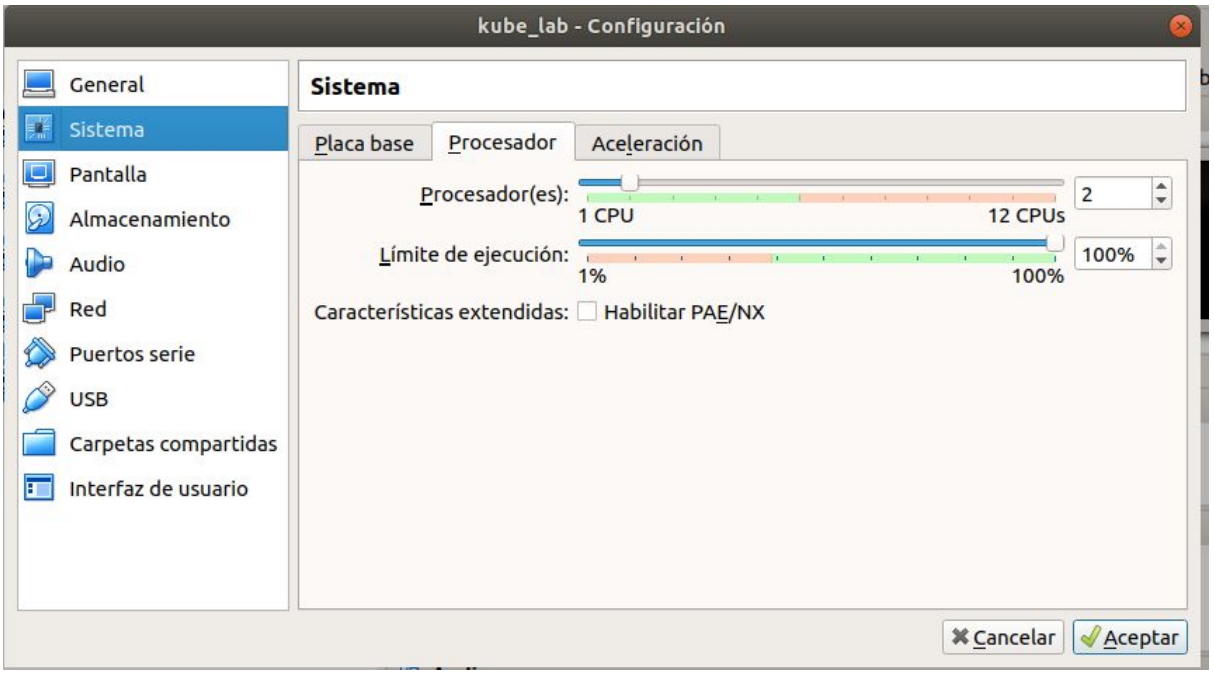

Figura A.3. Configuración de Sistema->Procesador.

2. Arrancamos la máquina y configuramos el sistema operativo. Al reiniciar y hacer *login* debería aparecer algo similar a lo mostrado en la Figura A.4:

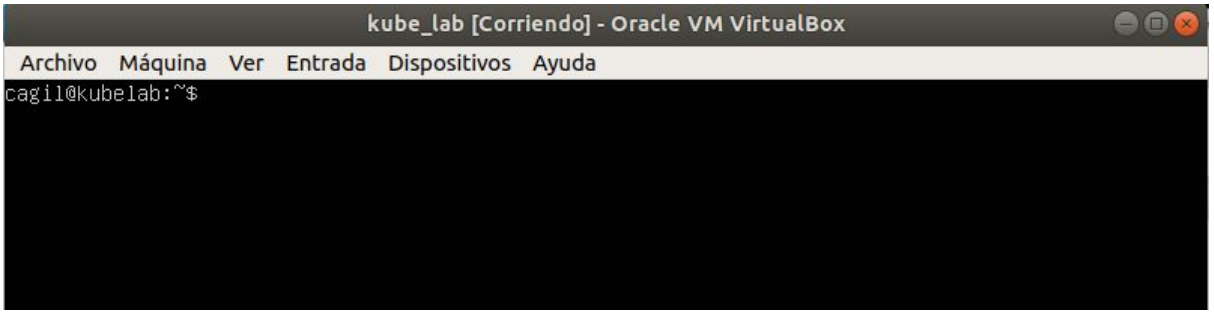

Figura A.4. Vista de la máquina virtual tras haber iniciado sesión.

3. Para evitar posibles problemas a la hora de instalar ciertas características debemos abrir el fichero `/etc/modprobe.d/blacklist.conf` y añadir `blacklist floppy`. Debería quedar como se muestra en la Figura A.5:

La inclusión de este módulo (`floppy`) en el archivo `blacklist.conf` evita que los scripts de hotplug lo carguen y así evitar que se produzcan problemas de sobreescritura de drivers o controladores. Este tipo de errores ocurre en algunas ocasiones al instalar Docker y Kubernetes sobre Ubuntu.

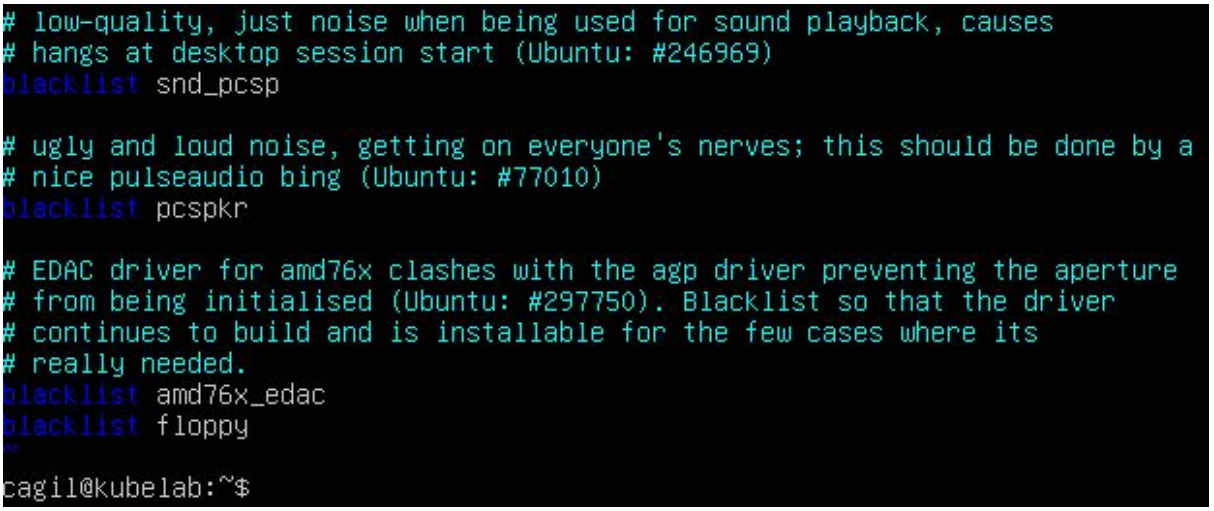

Figura A.5. Edición del fichero blacklist.conf

4. Llegados a este punto debemos actualizar Ubuntu ejecutando los siguientes comandos:

```
sudo apt-get update
sudo apt-get upgrade -y
sudo apt-get autoremove -y
sudo apt-get autoclean -y
sudo shutdown -r now
```

5. Una vez reiniciada la máquina instalamos `nmap`:

```
sudo apt install nmap -y
```

6. Instalamos también `openssh-server` (si no viene instalado por defecto) para poder acceder a la máquina mediante `ssh`:

```
sudo apt-get install openssh-server -y
sudo service ssh status
```

   El segundo comando sirve para comprobar que todo funcione correctamente.

7. Una vez hecho todo lo anterior accedemos desde VirtualBox, con botón derecho, a la configuración de la máquina. Una vez dentro vamos a la pestaña de red y nos aseguramos de que el adaptador es de tipo NAT (*Network Address Translation)*. Después desplegamos la parte avanzada y pulsamos sobre Reenvío de puertos como se muestra en la Figura A.6. Una vez dentro creamos una regla como se muestra en la Figura A.7 para poder acceder por `ssh` a la máquina poniendo como IP invitado la IP obtenida de la máquina virtual.

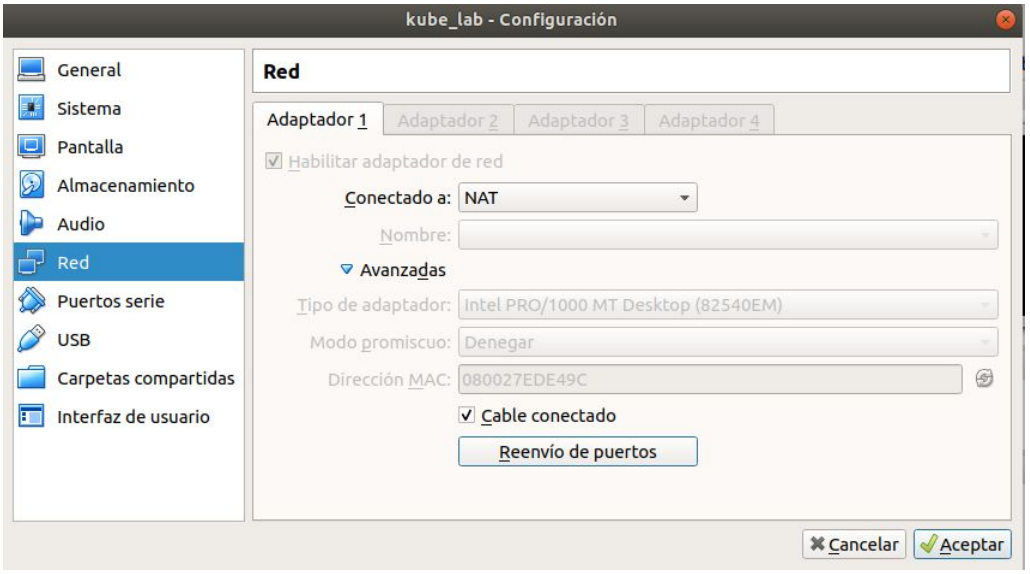

Figura A.6. Configuración de red de la máquina virtual

Figura A.7. Reglas de reenvío de puertos

8. Probamos la conexión ssh desde un terminal del host. En mi caso muestro cómo sería desde una máquina Ubuntu en la Figura A.8. En el caso de Windows se puede usar un cliente como `PuTTY` e introducir los valores correspondientes.

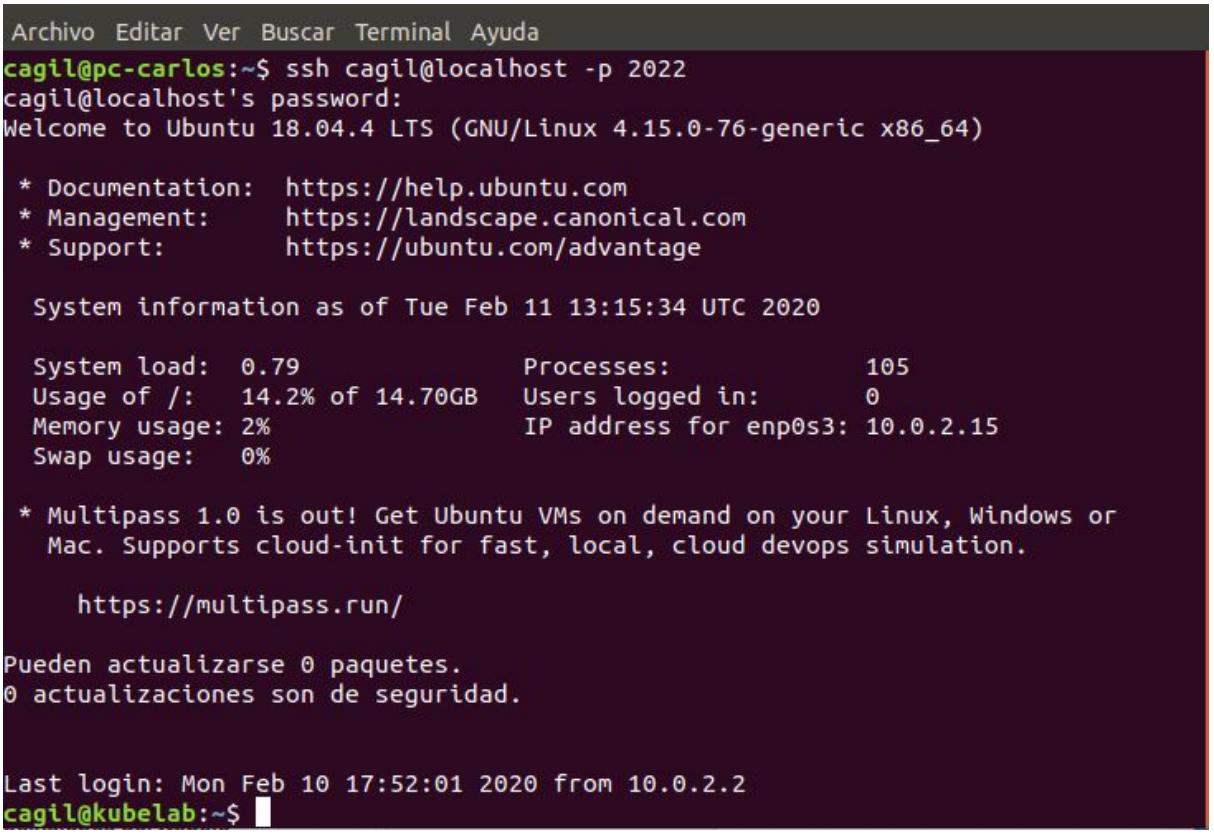

Figura A.8. Conexión por ssh a la máquina virtual.

A.2. Instalación de Docker

Una vez que tenemos configurada la máquina podemos empezar a instalar las tecnologías que vamos a usar. Antes de instalar las herramientas de Kubernetes necesitamos instalar y configurar Docker en la máquina virtual que anteriormente hemos creado.

Los pasos a seguir son los siguientes:

1. Incorporamos el repositorio de Docker usando los siguientes comandos:

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

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

sudo apt-get update
```

2. Instalamos Docker con:

```
sudo apt-get install -y docker-ce
```

Y comprobamos que se está ejecutando de manera correcta. Vista de ejemplo en la Figura A.9 de cómo debería verse:

```
sudo systemctl status docker
```

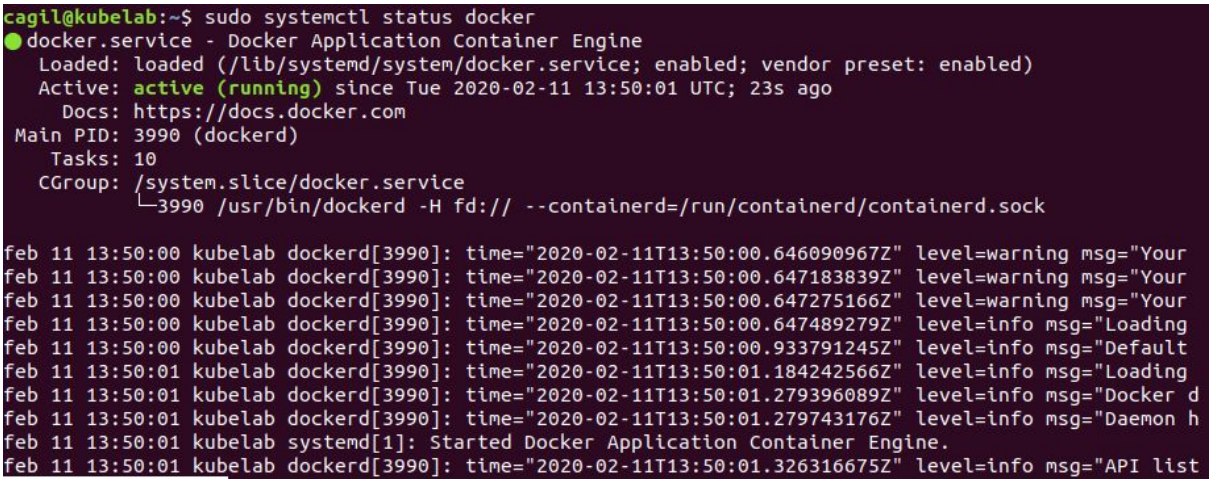

Figura A.9. Estado de Docker

3. Creamos nuestro registro privado de Docker que nos será útil para cargar imágenes Docker desde nuestro repositorio privado:

```
mkdir -p certs
```

```
openssl req -newkey rsa:4096 -nodes -sha256 -keyout certs/domain.key -x509 -days 365 -out certs/domain.crt
```

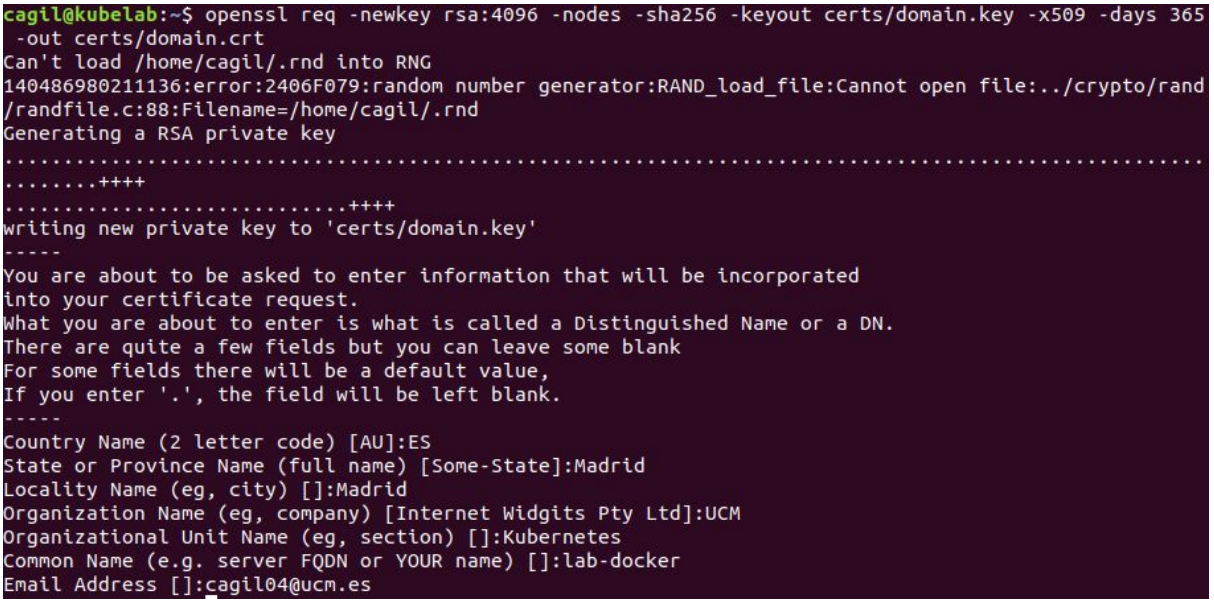

Figura A.10, Proceso de generación de las claves necesarias

Después de generar la clave debemos asegurarnos de que el contenedor se inicia con la opción `restart=always` y en el puerto 444, ya que más adelante veremos que el 443 lo usaremos con Kubernetes:

```
sudo docker run -d -p 444:444 --restart=always --name
registry \
-v "$(pwd)"/certs:/certs \
-e REGISTRY_HTTP_ADDR=0.0.0.0:444 \
-e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
-e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
-e REGISTRY_STORAGE_DELETE_ENABLED=true \
registry:2
```

A.3. Instalación de Minikube y Kubectl

Una vez instalado Docker ya podemos proceder finalmente a instalar las herramientas necesarias para manejar los recursos de Kubernetes. Estas no son otras que Minikube y Kubectl.

El proceso para instalar estas herramientas en la máquina anterior es el siguiente:

1. Primero instalamos Minikube, que sirve para manejar el *cluster* (de un solo nodo):

```
curl -Lo minikube
https://storage.googleapis.com/minikube/releases/latest/minik
ube-
linux-amd64

chmod +x minikube

sudo mv minikube /usr/local/bin/
```

2. Después instalamos Kubectl, que es la herramienta de línea de comandos para gestionar los recursos internos de Kubernetes:

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

chmod +x kubectl

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

3. Creamos los ficheros necesarios para guardar la configuración de las herramientas anteriormente instaladas:

```
mkdir $HOME/.kube || true touch $HOME/.kube/config
```