UNIVERSIDAD DE GRANADA

TRABAJO FIN DE MÁSTER

INGENIERÍA DE TELECOMUNICACIÓN

Despliegue automatizado de infraestructura de Microservicios y prueba de concepto de su funcionamiento.

Autor

Manuel Sánchez López

Director

Jorge Navarro Ortiz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Septiembre de 2018

El tribunal constituido para la evaluación del proyecto PFC titulado:

Despliegue automatizado de infraestructura de Microservicios y prueba de concepto de su funcionamiento.

Realizado por el alumno: **Manuel Sánchez López**

Y dirigido por el tutor: **Jorge Navarro Ortiz**

Ha resuelto asignarle la calificación de:

□ SOBRESALIENTE (9 - 10 puntos)

□ NOTABLE (7 - 8.9 puntos)

□ APROBADO (5 - 6.9 puntos)

□ SUSPENSO

Con la nota: puntos.

El Presidente:

El Secretario:

El Vocal:

Granada, a de de 2018

UNIVERSIDAD DE GRANADA

Despliegue automatizado de infraestructura de Microservicios y prueba de concepto de su funcionamiento.

REALIZADO POR:

Manuel Sánchez López

DIRIGIDO POR:

Jorge Navarro Ortiz

DEPARTAMENTO:

Teoría de la Señal, Telemática y Comunicaciones

Granada, Septiembre de 2018

Despliegue automatizado de infraestructura de Microservicios y prueba de concepto de su funcionamiento.

Manuel Sánchez López

PALABRAS CLAVE:

Microservicios, *contenedores, dockers, kubernetes, ansible, KVM, vagrant, Internet of Things, virtualización, automatización.*

RESUMEN:

En la actualidad, las empresas de desarrollo *software* optan por virtualizar sus sistemas de manera que se consigan reducir los costes en *hardware* sin que esto afecte sobremanera al rendimiento de los nuevos despliegues. Además, se pretenden reducir los tiempos empleados para desplegar grandes sistemas debido a configuraciones de grandes equipos imposibles de paralelizar. Es por ello que surgen nuevas tecnologías y herramientas capaces de ofrecer una solución tanto para virtualizarción como para el despliegue eficiente de infraestructuras basadas en *software*.

Concreatemente, las infraestructuras basadas en microservicios comienzan a tomar un peso importante en las decisiones de desarrollo, debido a su potencial en cuanto al despliegue automatizado, escalable, rápido y robusto, así como la reducción de costes gracias a la virtualización en pequeñas partes de los sistemas *hardware*, posibilitando el despliegue disperso de lo que, en el pasado, constituía una única máquina que necesitaba de grandes recursos.

La herramienta que vamos a utilizar en este trabajo de fin de máster para la gestión de un clúster de microservicios es *kubernetes*. Esta herramienta nos permitirá desplegar, controlar, escalar y monitorizar nuestro entorno de Microservicios.

Automatized deployment of a microservice infrastructure and probe of concept about the funcionality.

Manuel Sánchez López

KEYWORDS:

Microservices, containers, dockers, kubernetes, ansible, KVM, vagrant, Internet of Things, virtualization, automatization.

ABSTRACT:

Nowadays, software development companies are opting for virtualizing their systems to reduce hardware costs without affecting their performance. In addition, virtualization allows companies to improve the time to deploy their huge systems because of the configuration complexity in large servers which is impossible to paralelize. This is why new technologies and tools are emerging for both the virtualization and the efficient deployment of software-based infrastructures.

More specifically, microservice-based infrastructures are reaching a relevant role in development decisions due to their potential in automatized deployments, scalability, high performance but also to reduce costs thanks to the virtualization of the hardware systems, allowing us to deploy large systems in small parts by separating the resources between different baremetal hosts.

One of the main tools for the orchestration of a microservice cluster is *Kubernetes.* This tool is designed for the monitorization and the management of microservice deployments. Using this tool, this thesis will create an automatized deployment of a microservice cluster based on Kubernetes to understand these new concepts and evaluate its capabilities.

Yo, Manuel Sánchez López, alumno de la titulación del Máster de Ingeniería de Telecomunicación de la Escuela Técnica Superior de Ingenierías Informácitca y de Telecomunicación de la Universidad de Granada, con DNI XXX, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Máster en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Manuel Sánchez López

Granada a Septiembre de 2018

D. Jorge Navarro Ortiz, Profesor del Área de Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada, como director del Trabajo Fin de Máster de D. Manuel Sánchez López

Informa de que el presente trabajo, titulado:

Despliegue automatizado de infraestructura de Microservicios y prueba de concepto de su funcionamiento

ha sido realizado bajo su supervisión por Manuel Sánchez López, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada Septiembre de 2018. El director:

Agradecimientos

Gracias principalmente a mis padres y a mi tía, ya que ellos han sufrido esta travesía tanto como yo, y se han alegrado de mis aprobados incluso más. Gran parte de esta consecución es suya, y ellos lo saben bien.

Gracias a Cristian y Adri, porque este camino lo hemos hecho juntos, ese camino entre Madrid y Granada que tan largo se nos ha hecho a veces, pero que nos ha dejado momentos de risas también. Gracias también a Carlos, Rafa, Álvaro, Juan, Fran y mi hermano, porque no todo ha sido estudiar, y aquí en Madrid hemos formado nuestra propia familia.

Por otro lado, gracias a Noel y Jorge, ya que sus conocimientos técnicos también me han ayudado a conseguir que este proyecto sea hoy una realidad, y además de compañeros de trabajo son buenos amigos. Gracias también a Eva, por escucharme y darme ánimo para conseguir acabar.

Por último, gracias a Jorge por darme libertad para elegir la temática de este proyecto y hacerme un hueco, pese a tener ya bastante trabajo encima de la mesa. Siempre a disposición cuando lo he requerido, pese a la distancia, y un impulso más para conseguir el objetivo.

Índice general.

Índice de Figuras

Índice de Tablas

Capítulo 1

1. Introducción.

En este capítulo se plasmarán los conceptos básicos de este trabajo para situar al lector y facilitar la comprensión de los puntos que se expondrán en esta memoria.

En primer lugar, se definirá el concepto de microservicios, pues en estos se basa la arquitectura de la que trata este proyecto.

A continuación, se expondrá un apartado de motivaciones con el que se pretende mostrar la relevancia del trabajo realizado y de todas las tecnologías y herramientas involucradas en él.

En el siguiente apartado se incluirán los logros y las aportaciones que se han conseguido con la realización del proyecto. Por último, se expondrá, a modo de esquema didáctico, la estructura que seguirá la memoria, incluyendo un breve resumen de los conceptos que se tratarán en cada uno de los puntos.

Una vez definida la estructua de este capítulo, se incluye el objetivo final de este trabajo de fin de máster antes de comenzar con su desarrollo. El fin último del mismo consistirá en el diseño y despliegue automatizado de un clúster de microservicios basada en un clúster de *Kubernetes* (herramienta desarrollada por *Google* y *open source*) con la intención de ofrecer una plataforma que nos permita testear el código de los diferentes microservicios que se desarrollen a nivel funcional y de sistema. De esta manera se podrá agilizar la integración de nuevos cambios en la arquitectura, dotando al despliegue de una automatización y rapidez para su despliegue que favorecerá en gran medida el desarrollo *software*. Además, se pretende realizar una prueba de concepto a través de una serie de microservicios orientados a ser utilizados dentro del nuevo paradigma de *Internet of Things*, y ver así el funcionamiento en un entorno con la arquitectura antes mencionada.

1.1. Contexto y motivaciones.

En esta sección vamos a tratar de contextualizar el momento en que se encuentran tanto las redes de comunicación como las arquitecturas de los sistemas que la componen. Incidiremos en la importancia del crecimiento del tráfico en la red debido al IoT (*Internet of Things*) y en el empeño por parte de las empresas de abaratar costes en sus sistemas mediante la virtualización de los mismos.

Además, se comentará tanto los beneficios como los inconvenientes que acarrean el hecho de virtualizar los sistemas y veremos en qué entornos pueden encajar las tecnologías del que versa este proyecto.

1.1.1. Virtualización de infraestructuras.

La repercusión de Internet en las últimas décadas se hace patente en el porcentaje de personas del mundo que tienen acceso a la red cada día. En la actualidad, es complicado encontrar a alguien que no haga uso de Internet prácticamente a diario e, incluso, a través de varios dispositivos. Sin embargo, este hecho hace que cada vez sea más complicado ofrecer un servicio robusto, de calidad y escalable. Debido a su naturaleza, adoptar una nueva arquitectura o introducir modificaciones requiere de un gran cambio en los sistemas que hoy en día se han establecido, por lo que la transición para la consecución de los nuevos sistemas provoca un gran impacto en los mismo que las empresas deben asumir para poder dar respuesta a las exigencias de los usuarios en un futuro muy cercano. Uno de estos cambios que provoca un punto de inflexión en la red es precisamente la virtualización de las infraestructuras y las aplicaciones que ofrecen servicios.

1.1.2. ¿Por qué apostar por arquitecturas basadas en microservicios?

Partiendo de lo expuesto en el apartado anterior, podemos extraer que, cada vez con mayor frecuencia, se hace necesaria la posibilidad de que nuestros sistemas sean capaces de actualizar sus requisitos en un corto periodo de tiempo y sin grandes impactos sobre diferentes aspectos del mismo. Esto, unido a la necesidad de sistemas capaces de recuperarse frente a posibles fallos es lo que nos lleva a optar por nuevas arquitecturas que cumplan con las demandas expuestas.

Aquí es donde entran en juego las arquitecturas basadas en microservicios; lejos de seguir optando por sistemas robustos pero muy acoplados, los cuales necesitan horas e incluso días para conseguir su completo despliegue, además de su incapacidad para abordar posibles cambios en “caliente”, se buscan sistemas desacoplados y capaces de abordar posibles pequeños cambios sin que esto afecte al sistema en su totalidad.

Las arquitecturas basadas en microservicios pretenden desgranar los grandes sistemas que se han venido desplegando hasta el momento en diferentes servicios con objetivos concretos y totalmente desacoplados unos de otros, de tal forma que lo único que se exponga en cada uno de ellos sea una interfaz de comunicación bien definida para intercambiar mensajes entre sí. De esta forma, a cada microservicio no le deben afectar los cambios que se produzca en cualquier otro, ya que el único punto de interés es la interfaz que se expone en cada uno de ellos. Dichas interfaces suelen des de tipo REST, pero se pueden implementar libremente utilizando cualquier otro protocolo de comunicación.

Así, se consiguen cuatro objetivos:

1. Los cambios de cada microservicio no afectarán al funcionamiento de los demás.

2. Se facilitan las modificaciones en caliente del sistema, ya que al desacoplar en pequeños servicios sólo habría que modificar una pequeña parte del conjunto.

3. Se reduce el tiempo de tolerancia a fallos, ya que, si uno de los microservicios falla, será cuestión de pocos segundos volver a levantar un semejante que sustituya al que está produciendo el fallo.

4. En caso de requerir escalado, no será necesario levantar un sistema entero nuevo, sino que se escalarán los microservicios que estén más saturados, ahorrando tiempo y recursos.

No obstante, esta arquitectura no es válida para cualquier sistema y dependerá de las características del mismo. Por ello, habrá que tener en cuenta tanto el rendimiento que necesitamos como los tiempos de respuesta requeridos, la necesidad de persistencia, etcétera, ya que estas arquitecturas están diseñadas para desplegar servicios volátiles y sin estado, únicamente orientados a recibir una petición y servirla, sin almacenar ningún tipo de información. Pese a esto, siempre see puede optar por arquitecturas híbridas que permitan persistir información fuera de nuestro sistema, de tal forma que podamos seguir beneficiándonos de este tipo de arquitectura apoyándonos en otras.

1.2. Organización de la memoria.

Este apartado ilustra cómo se va a estructurar la memoria de tal forma que se facilite la ubicación de los diferentes aspectos a tratar al lector. La presente memoria consta de diez capítulos y tres anexos ue se describen a continuación:

Capítulo 1: Introducción

En este capítulo se expondrán los diferentes aspectos que han motivado la realización de este proyecto, así como una descripción introductoria de conceptos básicos de la tecnología implicada en la elaboración del mismo. Finalmente se presenta una recopilación de todos los logros y aportaciones que tratan de plasmar el alcance del proyecto.

Capítulo 2: Estado del arte

El capítulo correspondiente al estado del arte trata de acercar al lector al entorno de los microservicios, presentando el estado actual de las arquitecturas de despliegue, que en su mayoría se basan en la nube, y plasmando las diferentes herramientas del mercado actual que permiten la orquestación de clústeres de microservicios.

Capítulo 3: Análisis de objetivos y requisitos

Este capítulo engloba dos bloques: el análisis de objetivos y la especificación de requisitos.

El primer bloque tratará de exponer los objetivos marcados para el desarrollo del proyecto. En el segundo se desglosarán tanto los requisitos como las decisiones previas, que son necesarios para trabajar.

Capítulo 4: Planificación y estimación de costes

Este capítulo plasma detalladamente todos lo relacionado con la planificación temporal del proyecto, definiendo cada una de las fases en las que éste se divide y aportando una estimación del tiempo invertido en cada una de ellas.

Por otro lado, se analizarán todos los recursos necesarios y se estimarán los costes totales asociados a éstos con el objeto de presentar un presupuesto final en el que se muestre el coste total del proyecto.

Capítulo 5: Herramientas utilizadas

En este capítulo se presentarán las herramientas de más peso en este proyecto: *Kubernetes*, *Ansible* y *Vagrant*. Además, se mencionarán otras herramientas relacionadas con la integración y despliegue continuos, ya que son conceptos muy ligados a esta arquitectura basada en microservicios.

En lo referente a *Kubernetes*, se ofrecerá una descripción de dicha herramienta y se justificará su uso frente a otras posibilidades que ofrece el mercado. Además, se expondrá un pequeño ejemplo para acercar la herramienta al lector. Seguidamente, se justificarán los usos de *Ansible*, como herramienta de automatización de la configuración, y de *Vagrant*, como herramienta de automatización de despliegue.

Capítulo 6: Automatización de despliegue de un clúster de microservicios

Con este capítulo se pretende plasmar la implementación llevada a cabo para conseguir un despliegue de un clúster de microservicios basado en *Kubernetes*, de manera totalmente automatizada, por lo que nos centraremos en el desarrollo con *Ansible* y explicaremos sus conceptos más importantes.

Capítulo 7: Prueba de concepto de despliegue de microservicios

En este capítulo se pasará a explicar unos conceptos previos necesarios para situar al lector y, posteriormente, se detallará paso a paso como se ha llevado a cabo el despliegue de un microservicio basado en un servidor destinado al propósito de *Internet of Things*, tanto en un entorno únicamente de contenedores como dentro un un clúster encargado de gestionarlos, para ver sus diferencias y las posibilidades que cada entorno ofrece.

Capítulo 8: Resultados obtenidos

Llegados a este capítulo trataremos de reflejar los resultados que se han obtenido con el desarrollo de nuestro trabajo, ofreciendo una visión detallada de los puntos fuertes de esta nueva arquitectura de despliegue *software* a partir de la prueba de concepto realizada en el capítulo anterior.

Capitulo 9: Conclusiones y vías futuras

Con este capítulo concluye la memoria y será el que exponga las conclusiones a las que se han llegado una vez finalizado el proyecto. Así mismo, se propondrán una serie de

posibles trabajos futuros a partir de este proyecto que mejorarían sus prestaciones. Por último, se ha incluido una valoración personal sobre todo lo que este trabajo ha acontecido.

Anexo I: Manual de despliegue

Se trata de un manual para el despliegue del clúster de manera automatizada, tanto utilizando *Vagrant* para virtualizar las máquinas de las que constará el clúster como prescindiendo de esta herramienta y automatizar únicamente la configuración de unas máquinas ya desplegadas, es decir, utilizando *Ansible* únicamente.

Anexo II: Manual de usuario para prueba de concepto

Este anexo detalla los pasos que se han seguido para conseguir ejecutar la prueba de concepto relacionada con el despliegue de un microservicio dentro de nuestra arquitectura basada en microservicios.

Anexo III: Complicaciones y errores encontrados

En este anexo se han añadido diferentes puntos que han complicado la labor realizada durante su desarrollo, mostrando casos de error que pueden ser solventados tal y como se explica en el mismo.

Bibliografía

En la bibliografía se exponen las referencias de las que se han hecho uso para adquirir ciertos conocimientos e información sobre las tecnologías utilizadas, herramientas, etcétera.

Capítulo 2

2. Estado del arte

En este capítulo vamos a tratar de acercar al lector a la actualidad referente a las tecnologías de microservicios. Para ello realizaremos un análisis de las arquitecturas para despliegue de *software* actuales, tanto basadas en cloudcomo las que se basan en microservicios y repasaremos también los orquestadores de microservicios presentes en el mercado, las herramientas de automatización de configuración y las herramientas de automatización de despliegue.

2.1. Microservicios.

Los microservicios, tal y como se explica en [1], son una arquitectura y un enfoque sobre la escritura de *software* en componentes más pequeños e independientes entre sí. La idea principal es dejar atrás los despliegues monolíticos, en los que se hace necesario un compromiso estricto entre *hardware* y *software* y en los que los recursos pueden no ser utilizados en su totalidad durante todo el tiempo de vida del sistema, para dar paso a despliegues cuyo desacople entre las diferentes partes del mismo lo hagan dinámico en cuanto a los recursos antes mencionados, pueda ser desplegado en equipos físicos diferentes, se puedan gestionar de manera autónoma e independiente, etcétera.

Estos aspectos derivados de la arquitectura de microservicios nos permiten un menor tiempo de reacción en nuestro sistema, ya que se facilita el desarrollo y la adaptación de las aplicaciones para satisfaces demandas. Otra de las ventajas que nos otorga este nuevo enfoque es la capacidad para que distintos equipos de desarrollo dentro de las empresas puedan trabajar simultáneamente en los productos de un modo ágil, y entregar valor a los clientes de inmediato.

Para acercarnos más al concepto de microservicios vamos a poner el ejemplo de un distribuidor de películas y series. El servidor de este sistema tiene que atender varias demandas:

- Deberá ofrecer un catálogo con las series y películas disponibles.
- Cada película o serie tendrá una sinopsis y una imagen asociadas.
- Dependiendo de la suscripción del usuario, se necesitará de un sistema que identifique si es apto para ver el contenido.
- Deberá reproducir la película o serie en el momento en que el usuario lo solicite en caso de estar autorizado.

En un sistema monolítico, tendríamos un servidor encargado de realizar todas las operaciones y responder al cliente. Sin embargo, en una arquitectura orientada a microservicios, la idea sería desacoplar en diferentes servidores cada pequeña tarea, de tal forma que tendríamos un servidor web encargado única y exclusivamente de mostrar el catálogo con él contenido, otro que, dependiendo del contenido

seleccionado, devuelva la carátula y una sinópsis, otro encargado de verificar si el usuario puede reproducir el contenido y un último que provisione el streaming.

De esta forma, el microservicio encargado del streaming de vídeo sería independiente del que proporciona información sobre la película o serie concreta, e incluso podría darse el caso de que no pudiésemos leer la sinopsis, pero esto no impediría al usuario reproducir el vídeo. Este hecho, en el caso de un sistema monolítico, podría desencadenar una serie de fallos en el sistema que impidieran utilizar cualquier otra funcionalidad.

Y para poder llevar a cabo el despliegue de un microservicio, se cuentan con diferentes herramientas orientadas a este cometido. Entre los diferentes tipos de contenedores, podemos destacar dos:

- Contenedores LXC [2]: es un proyecto de contenedores de Linux cuya idea es separar un conjunto de procesos del resto del sistema, pudiendo ser ejecutados desde una imagen diferenque que proporciona todos los archivos necesarios para dar soporte a los procesos. Al proporcionar una imagen que contiene todas las dependencias de una aplicación, es portátil y consistente en el cambio de la etapa de desarrollo a la de prueba y, finalmente, a la de producción.

- Contenedores Docker [3]: es un proyecto de la comunidad *open source*, y se trata de una tecnología de creación de contenedores. Con Docker, los contenedores se pueden considerar como máquinas virtuales extremadamente livianas y modulares. Dichos contenedores otorgan una gran flexibilidad, pudiendo ser creados, implementadoso copiados de un entorno a otro.

2.1.1. Comparativa entre contenedores LXC y Docker.

Ambos contenedores se fundamentan en una idea común. En un comienzo, la tecnología Docker se desarrolló sobre la base de la tecnología LXC, pero poco a poco comenzó a diverger. LXC era útil como virtualización liviana, pero no tenía una buena experiencia de desarrollador o usuario. Este hecho lo solventó la tecnología Docker, aportando otras características además de la ejecución de contenedores; facilitando el proceso de creación y desarrollo, envío y versionado de las imágenes.

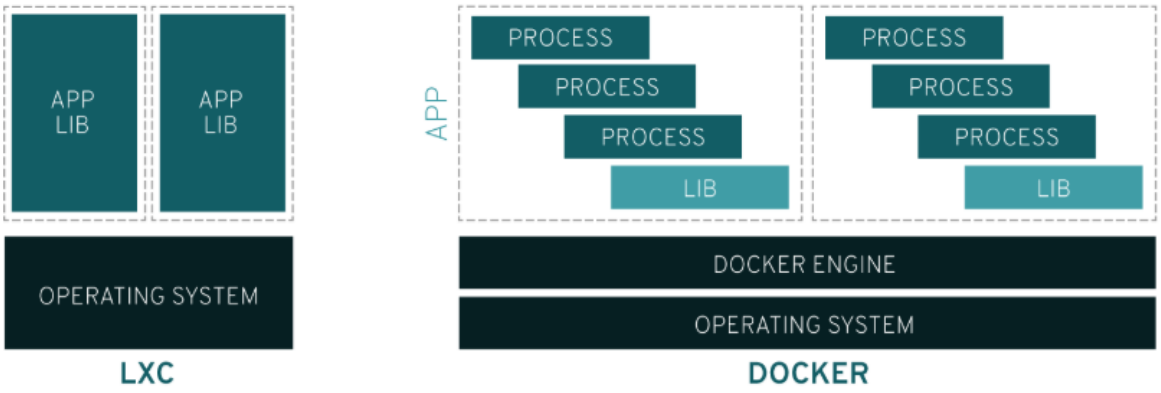

Figura 1. Estructura de LXC vs Docker.

En cuanto a LXC, este corre en un sistema operativo completo, es decir, se inicia con su *init* [4]. En el caso de docker, lo que se pretende es definir lo que se quiere correr, es decir, por el lado de LXC disponemos de todosl os servicios mínimos que tiene un sistema, mientras que Docker está enfocado a correr servicios.

No obstante, en Docker podemos crear imágenes base que permitan correr un *init* simple, pero esto no concierne a la tecnología, sino al usuario y al despliegue deseado.

Además, como ventajas de Docker podemos destacar las siguientes [3]:

- **Modularidad**: el enfoque de Docker se centra en la capacidad de tomar una parte de una aplicación, actualizarla o repararla, sin ser necesaria la implicación de toda la aplicación. Además, se pueden compartir procesos entre varias aplicaciones de la misma forma que en una arquitectura orientada a servicios (SOA).

- **Capas y control de versión de imagen**: cada archivo de Docker está conformado por una serie de capas. Estas capas se combinan en una única imagen. Una capa se crea cuando la imagen se actualiza. Estas capas hacen referencia a una etiqueta del contenedor en cuestión. El control de versiones es inherente a la creación de capas. Cada vez que hay un nuevo cambio se registra para tener un control completo de las imágenes de un contenedor.

- **Restauración**: el hecho de que esté presente el control de versiones permite la fácil restauración de versiones anteriores en caso de ser necesario. Esto agiliza el desarrollo y ayuda a hacer realidad la integración e implementación continuas (CI/CD) desde una perspectiva de herramientas útiles para ello.

- **Implementación rápida**: con los contenedores basados en Docker se pueden reducir el tiempo de despliegue a segundos ya que, al crear un contenedor para cada proceso, se pueden compartir rápidamente los procesos similares con nuevas aplicaciones. Esto, sumado a que un sistema operativo no necesita iniciarse para agregar o mover un contenedor, los tiempos de implementación son sustancialmente inferiores. Asimismo, con la velocidad de implementación de dichos contenedores, se pueden crear y destruit sin preocupación de manera fácil.

2.2. Arquitecturas de despliegue de *software*.

Con el devenir del tiempo, han ido surgiendo nuevos requisitos y funcionalidades que han obligado a adaptar las arquitecturas de despliegue tanto de *software* como de *hardware* en busca de la solución que mejor se adapte a cada momento y cada caso particular.

Es evidente que las arquitecturas basadas en instalaciones nativas o baremetal han ido perdiendo peso a lo largo de los años, debido a su deficitaria gestión de los recursos de las maquinas en muchos intervalos de tiempo, lo que provoca una pérdida económica importante para las empresas proveedoras. Además, hay que sumar la dificultad de

escalado que esto supone, lo cual limita bastante las capacidades de servicio en unos tiempos en que cada vez fluye más tráfico en la red y se hace más necesario dar respuestas rápidas y de calidad a los usuarios.

Es por ello que nacen arquitecturas basadas en la nube, arquitecturas cloud, las cuales se basan en la optimización de los recursos y su distribución entre diversas máquinas físicas de manera virtualizada.

Además de las arquitecturas en la nube, surgen alternativas a los despliegues baremetal clásicos añadiendo connotaciones que permiten mejorar el ecosistema y orientarlo a un nuevo paradigma basado en microservicios. Estas nuevas arquitecturas basadas en microservicios utilizan los recursos, que pueden ser tanto *hadware* como virtualizados, para optimizar el despliegue de *software*, y no está en disonancia con despliegues en la nube, ya que podrían incluso llevarse a cabo dentro de la misma. Su objetivo principal es el de permitir desplegar pequeñas porciones de una gran aplicación en diferentes nodos y que, mediante su orquestación, se permite su funcionamiento conjunto.

No obstante, tanto si nuestro sistema se basa en *cloud* como si se basa en *baremetal*, necesitará de una plataforma capaz de gestionar dicho sistema. A continuación, se describen plataformas y orquestadores capaces de llevar a cabo la gestión de las diferentes arquitecturas presentadas.

2.2.1. Plataformas basadas en *cloud*.

Openstack

Openstack [5] es un *software* de código abierto que permite crear clouds tanto públicas como privadas. Este *software* controla grandes grupos de computación, almacenamiento y recursos de red a través de datacenters, y nos permite gestionarlo a partir de un panel de control con su interfaz web o mediante a su API. Esta herramienta ha sido adoptada por una gran cantidad de importantes empresas entre las que destacan Ericsson o Amazon, las cuáles han optado por adaptar su solución comercial a partir de la solución de código abierto propuesta por la comunidad.

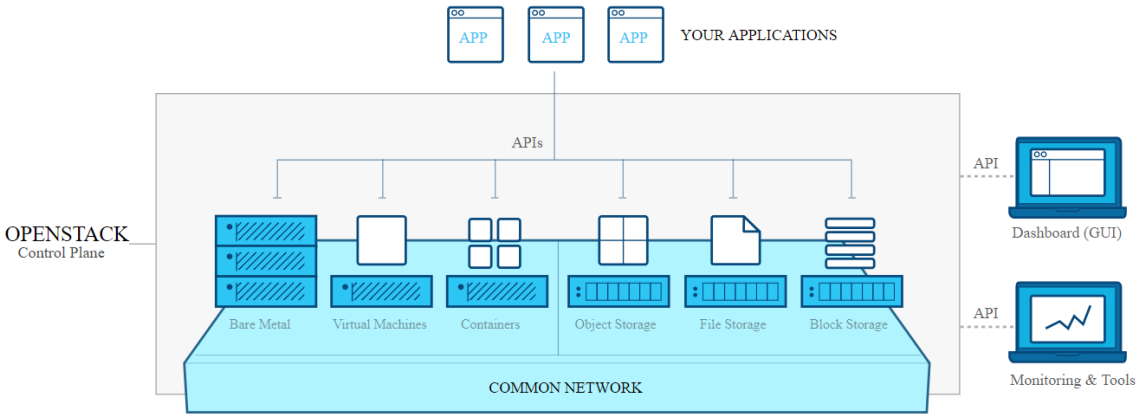

Figura 2. Arquitectura de Openstack.

Como vemos en la figura anterior, *Openstack* utiliza la infraestructura *hardware* para desplegar toda una infraestructua virtualizada o, incluso, baremetal si es necesario. Además, en ella pueden coexistir tanto máquinas virtuales como contenedores, por lo que podemos afirmar que estas arquitecturas no son excluyentes con las orientadas exclusivamente a microservicios, sino que pueden complementarse. Además, tiene tres tipos diferentes de almacenaje; almacenamiento de objetos, ficheros o bloques.

Por otra parte, está dotada de herramientas para su gestión y monitorización, además de constar de una red común y que puede ser incluso delegada en otras herramientas externas, como podría ser el controlador de SDN *Opendaylight*.

VMware cloud

VMware cloud [6] es un *software* dedicado también a la gestión de entornos cloud, gracias a la cual se pueden crear nubes tanto públicas como privadas, como pasa en el caso de *Openstack*. VMware vCloud Suite es una solución integrada que combina el hipervisor *VMware vSphere* con la plataforma de gestión de clouds híbridas multiproveedor *VMware vRealize Suite*, ambas pertenecientes a dicha organización. Sus características principales son [7]:

- Virtualización de centros de datos.
- Operaciones inteligentes: capaz de detectar y corregir problemas de forma proactiva.
- Automatización del entorno de TI: automatiza la distribución de una infraestructura preparada para entornos de producción en entornos multicloud mediante la automatización y el control basado en políticas, reduciendo el tiempo necesario para responder a las solicitudes de servicios de TI.
- Cloud orientada a desarrolladores: acelera la distribución de aplicaciones tradicionales y basadas en contenedores al permitir a los desarrolladores utilizar libremente las herramientas que les ayudan a ser más productivos, al tiempo que garantiza la transferencia fluida de aplicaciones del portátil del desarrollador al entorno de producción.

No obstante, esta herramienta no es *open source*, siendo este su punto de contrapartida más importante.

2.2.2. Plataformas basadas en microservicios.

Como comentamos en la introducción a esta sección, en la actualidad surge un nuevo paradigma de arquitectura de despliegue *software*, la cual se basa en la gestión y despliegue de microservicios, y no está en disonancia con las infraestructuras *cloud*, ya que podríamos acometer dicha arquitectura en un entorno basado en la nube. Por ello, en este subapartado se presentan una serie de herramientas enfocadas al

manejo y despliegue de microservicios. Dichos microservicios se basarán en un conjunto de contenedores que el orquestador de este tipo de plataformas gestionará para su ejecución a corde a como sería en caso de que dicha aplicación no estuviese dividida.

La tecnología en boga para levantar contenedores que, a posteriori, compondrán los diferentes microservicios, suele ser *Docker,* debido a las ventajas que presenta frente a sus competidores, como describimos en el primer apartado de este capítulo. No obstante, esta herramienta necesita de una capa de abstracción superior que le permita gestionar y monitorizar de manera conjunta a un grupo de contenedores, por lo que se hace necesario un orquestador de microservicios.

Entre las plataformas de orquestación de microservicios más importantes se encuentran las siguientes [8]:

Docker Swarm

Docker Swarm permite agrupar y programar contenedores de *Docker*. De hecho, es el módulo que ofrece *Docker* para gestionar clusters y orquestar servicios. Esta herramienta se compone de múltiples *hosts Docker* que se ejecutan con uno de estos dos roles [9]:

- *Swarm Manager*: gestión y administración.
- *Swarm Worker*: ejecución de los servicios.

Un *host* puede ser *manager, worker* o tener ambos roles.

Una de las características importantes de este orquestador es el *networking*, que hace transparente la comunicación en red distribuida que se hace entre los nodos y los contenedores de los mismos. Además, permite crear redes por *software* para que los contenedores conectados a esas redes se comuniquen de forma privada. También es capaz de monitorizar el estado de los servicios recreando contenedores si alguno de ellos deja de funcionar. Por otra parte, los servicios en *Docker Swarm* permiten realizar un balanceo de carga entre las instancias del mismo, asignando dicho servicio a más de una instancia y mediante la asignación de un DNS y dirección IP por el que identificarlos.

Mesosphere DC/OS

Mesosphere Enterprise DC/OS (basado en Apache Mesos) es una plataforma para entornos de producción que permite ejecutar contenedores y aplicaciones distribuidas.

DC/OS funciona mediante la abstrancción de una colección de los recursos disponibles en el clúster y poniendo dichos recursos a disposición de los componentes creados sobre él. Además, se suele utilizar *Marathon* como un programador integrado en esta plataforma.

Este sistema incluye una interfaz web intuitiva [10], aunque permite también interactual por la CLI.

Figura 3. Interfaz web de Mesosphere.

Gracias a *Mesosphere* se puede simplificar la instalación de los servicios distribuidos, como pueden ser *Cassandra, Jenkins, Chronos, Kafka, Spark,* etcétera, por lo que no es una herramienta orientada únicamente a la gestión de contenedores, cuya característica se incluye al añadir *Marathon*, como si comentó anteriormente, componente que ya viene incluido y que nos permitirá programar infinitas tareas en miles de nodos paralelamente.

También ofrece la posibilidad de escalado, a través de la configuración de modelos basados en el número de sesiones a través del balanceador de *Marathon*, además de altadisponibilidad, gracias a la posibilidad de gestionar un gran número de instancias y servicios. Para ello, *Marathon* monitoriza todos los servicios y, en caso de fallo, los reinicia.

Por otro lado, esta herramienta se puede integrar con *Amazon Web Services*, poniendo a disposición de los usuarios un *template* para la creación y configuración automática de la infraestructura. El inconveniente es que no es una configuración personalizada y únicamente se puede elegir entre desplegar 1 o 3 masters y la región en la que se desea desplegarlos. Esta herramienta, además, se encuentra más desarrollada en Linux que en Windows.

Azure Service Fabric

Service Fabric es una plataforma de microservicios de *Microsoft* para crear aplicaciones. Es un orquestador de servicios y crea clústeres de máquinas. Esta herramienta permite implementar servicios como contenedores o procesos estándar. Puede, incluso, combinar servicios en procesos con servicios en contenedores dentro de la misma aplicación y el mismo clúster.

Además, esta herramienta ofrece compatibilidad con los contenedores de *Docker* y también con sus orquestadores mediante el servicio de *Azure Container Service* (ACS), es decir, que podríamos utilizar *Azure* como un "orquestador de orquestadores" como podemos ver en la figura siguiente.

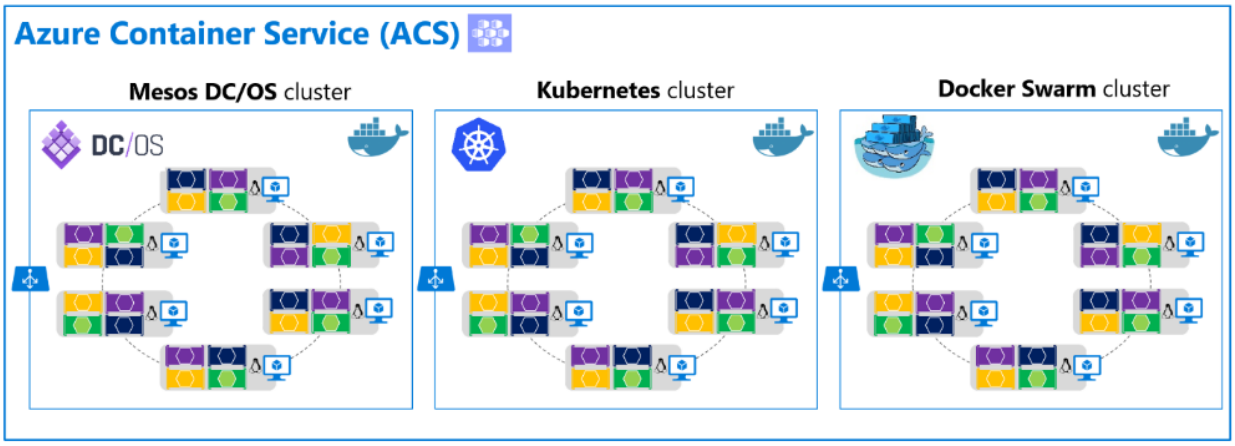

Figura 4. Ejemplo gráfico de Azure Container Service.

ACS proporciona una manera de simplificar la creación, configuración y administración de un clúster de máquinas virtuales que están preconfiguradas para ejecutar aplicaciones en contenedores.

Kubernetes

Kubernetes es un producto de código abierto cuya funcionalidad abarca desde la infraestructura de clúster y la programación de contenedores a las capacidades de orquestación. Permite automatizar la implementación, el escalado y las operaciones de contenedores de aplicaciones en varios clústeres de hosts. Además, ofrece una agrupación de contenedores de una aplicación en unidades lógicas, lo cual facilita la administración y detección de las mismas.

Esta herramienta ha sido desarrollada por *Google* y es la que utilizan innumerables empresas para llevar a cabo su automatización de despliegue, debido a que se trata de una herramienta muy madura y que ofrece una gran facilidad para llevar a cabo un *continuous integration/continuous deployment* de las aplicaciones y se puede integrar con otro tipo de abstracciones por encima para facilitar su manejo por parte del desarrollador.

Kubernetes, al ser la elegida para llevar a cabo nuestro proyecto, se explicará con más detalle en el apartado 5.1. Kubernetes.

A la hora de seleccionar donde vamos a lanzar nuestras aplicaciones basadas en microservicios, conviene estudiar aspectos como el tiempo que tardarán estos en ser levantados, la complejidad del sistema, tolerancia frente a fallos, seguridad, etcétera.

Teniendo en cuenta todos estos aspectos, decidimos no utilizar un entorno basado en la nube, ya que esto implicaría añadir un nivel más de virtualización a nuestros sistemas, teniendo en cuenta lo que ello supone en cuanto a complejidad y tiempos de respuesta, además de que, utilizando directamente máquinas donde hospedar nuestros microservicios, podemos tener un control más exhaustivo de los mismos,

evitando añadir capas de abstracción que puedan suponer una pérdida de gestión y monitorización de nuestro entorno.

Por otro lado, cabe destacar el tema de los recursos, ya que instalar un *vMware Cloud* o un *OpenStack* requiere de unas capacidades *hardware* de las que no disponemos, y además los resultados de *performance* obtenidos posiblemente sean peores debido a esa doble virtualización.

Debido a esto, nos decidimos por un entorno con máquinas virtuales que actuarán como nodos esclavos de un clúster de microservicios por encima de la propia máquina física.

La justificación de por qué hemos optado por utilizar *Kubernetes* como piedra angular del mismo, se puede ver en el capítulo 6, Herramientas utilizadas, donde se realiza una exhaustiva comparativa entre nuestra elección y las herramientas del mismo cometido.

2.3. Herramientas de automatización de configuración.

El foco principal del presente proyecto es la nueva arquitectura de microservicios, pero se han utilizado una serie de herramientas que giran en torno a dicho foco para consolidar esta idea, entre las cuáles se encuentran las herramientas de automatización de configuración.

Estas herramientas permiten llevar a cabo la configuración de máquinas de manera sencilla, automatizada y fácilmente reconfigurable, pudiendo escalar dicha configuración o incluso añadir nuevas propiedades modificando simplemente una serie de ficheros.

Básicamente, estas herramientas se basan en realizar conexiones “ssh” y ejecutar comandos para copiar, modificar archivos, instalar paquetes, etcétera. Gracias a una serie de etiquetas, es sencillo distinguir las instalaciones entre múltiples sistemas operativos, e incluso existen etiquetas destinadas a ejecutar comandos recurrentes en las diferentes máquinas.

Algunos ejemplos de herramientas de automatización son los siguientes [11]:

- **Puppet:** herramienta de código abierto basada en ruby. Mediante esta herramienta, el usuariodescribe los recursos del sistema y sus estados utilizando el lenguaje declarativo que proporciona Puppet. Dicha información se almacena en archivos denominados “manifiestos”. Además, Puppet descubre la información del sistema a través de una utilizad llamada “*Facter*”, y compila los manifiestos en un catálogo específico del sistema que contiene los recursos y la dependencia de los mismos. Estos catálogos son ejecutados en los sistemas de destino. Esta herramienta puede ser utilizada en diferentes plataformas además de Linux y Windows, como pueden ser servidores de IBM, switches Cisco o inclusoen servidores con Mac OS. Por otro lado, Puppet está basada en la configuración de agentes en las máquinas a ser configuradas.

- **Ansible:** es una herramienta de automatización IT (*Information Technologies*) [12]. Puede configurar sistemas, desplegar *software* y orquestar tareas más avanzadas de IT. El objetivo principal de esta herramienta es la simplicidad y facilidad de uso. Además, centra sus esfuerzos en la seguridad y la fiabilidad, mediante el uso de OpenSSH para transporte, y un lenguaje diseñado para ser legible y entendible por el humano, a pesar de no estar familiarizado con el programa. En cuanto al modo de gestión de los servidores, Ansible gestiona las máquinas sin necesidad de instalar agentes en las máquinas a provisionar.

- **Chef:** escrita en *Ruby* y *Erlang*, chef es una herramienta de código abierto que abraza totalmente la metodología de trabajo *DevOps* (*Development-Operations*). Una de sus ventajas fundamentales es que es totalmente integrable con plataformas *cloud* como AWS, Azure, GCP o Rackspace [13]. Esta herramienta también necesita de la instalación de los agentes en los servidores que van a ser configurados. La arquitectura de la misma puede verse en la Figura 2. Chef se basa en el desarrollo de módulos en los que se describen las configuraciones a realizar en los distintos nodos.

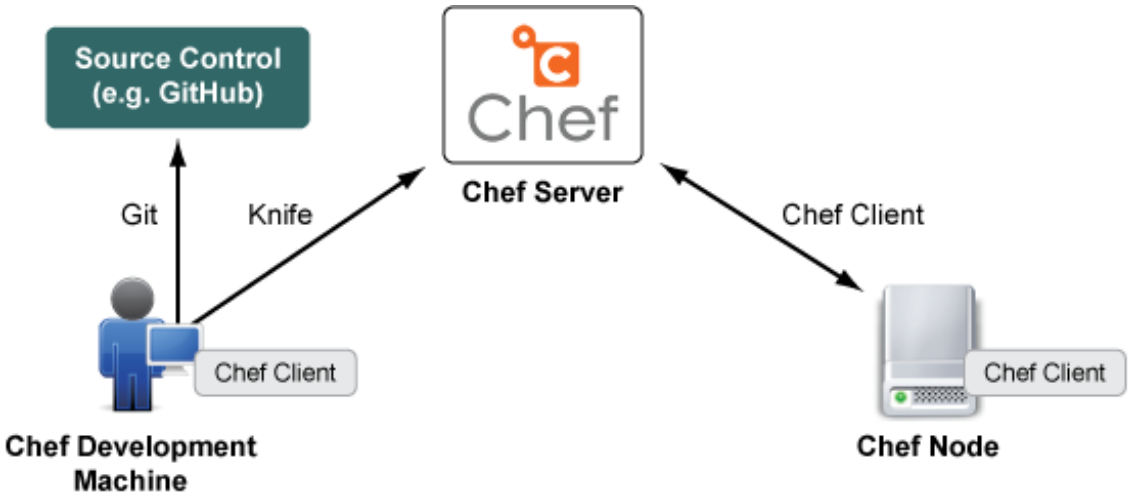

Figura 5. Arquitectura de Chef.

El motivo principal por el que nos hemos decantado por *Ansible* es su curva de aprendizaje, la cuál es notablemente inferior a las de *Chef* o *Puppet*, además de su funcionamiento secuencial, lo cuál facilita mucho nuestra tarea lógica, ya que muchas acciones han de realizarse estrictamente en un orden. Para más detalle del por qué de esta elección nos podemos referir al punto 6.2. Ansible.

2.4. Herramientas de automatización de despliegue.

Antes de la configuración automatizada de las máquinas con las que se va a trabajar, es necesario su despliegue. Para un sistema como el de microservicios, cuyo objetivo principal es agilizar las arquitecturas de CI/CD, el hecho de contar con una herramienta que automatice el despliegue de infraestructuras puede ser de una gran utilidad de cara a la integración continua. Además, dichas herramientas pueden ser integradas con las referentes a la automatización de provisionamiento y configuración.

La herramienta más extendida para automatización de despliegues virtualizados es ***Vagrant***.

Vagrant [14] es una herramienta capaz de desplegar de máquinas de manera virtualizada a partir de un fichero de configuración en el que se describen todos los requisitos del *software*, paquetes, configuraciones de sistema operativo, usuarios y mucho más.

Como comentamos antes, se puede integrar con herramientas de gestión de la configuración como Chef, Puppet, Ansible o Salt, pudiendo configurar los scripts para configurar Vagrant en un entorno de producción.

Además, es una herramienta multiplataforma, pudiendo ser instalada y utilizada en los sistemas operativos más comunes como son los de Windows, Mac o Linux. Por otro lado, los hipervisores con los que trabaja también son diversos, soportando virtualización en VirtualBox, KVM y VMware, además de entornos de servidores como Amazon EC2.

Dicha herramienta está escrita en Ruby, pero puede ser utilizada en proyectos escritos en otros lenguajes como pueden ser PHP, Pythom, Java, C#, y JavaScript. Incluso, es capaz de soportar la tecnología de contenedores Docker, por lo que podemos afirmar que es una herramienta que se encuentra en continua evolución y se adapta a las tecnologías emergentes.

Capítulo 3

3. Análisis de objetivos y requisitos.

El tema a tratar en este capítulo versa sobre las cuestiones previas que se han abordado como antesala al diseño e implementación del presente proyecto.

En primer lugar, se expondrán los diferentes objetivos que serán fundamentales para dicho diseño e implementación. A continuación, se identificarán todos los requisitos y decisiones previas que se deben cumplir de un modo genérico en el proyecto. Dichos requisitos emergen a raíz de los objetivos previamente marcados.

Se incluye una valoración en cuanto a la consecución de esos objetivos previamente marcados, así como la inclusión de nuevos objetivos surgidos a lo largo del diseño e implementación del proyecto.

3.1. Objetivos.

El objetivo principal de este proyecto es el despliegue automatizado de un clúster de microservicios basado en *Kubernetes*. Para llevar a cabo este despliegue, se pretenden desgranar los requisitos y decisiones previas tomadas de cara a administrar un clúster de *Kubernetes* y analizar las herramientas del mercado que nos permitan llevar a cabo una configuración de las máquinas y desplegarlas sin la necesidad de grandes esfuerzos.

Una vez conseguido este primer hito, el objetivo será realizar pruebas en base a las posibilidades que ofrece este orquestador de microservicios, tratando de desarrollar un microservicio propio o reutilizar alguno existente. Una vez tengamos el microservicio en cuestión, se pasará a desplegarlo en el clúster e investigar las características que tiene el entorno, buscando tanto los puntos fuertes como los puntos débiles que se desprendan de dichas pruebas.

Para ello, se realizarán pruebas de distintos tipos, forzando al sistema a dar respuesta a situaciones como fallos de un nodo dentro del clúster, caída de uno de los contenedores que compongan un microsericio, etcétera.

3.2. Especificación de requisitos.

En este apartado se exponen los requisitos y las decisiones previas que se han tenido en cuenta para el diseño de este proyecto. Este proceso es importante para conceptualizar los requisitos que son necesarios en la implementación posterior.

Por lo tanto, con esta sección se facilitará la comprensión del diseño, reduciendo el tiempo empleado en la puesta en marcha de todos los elementos implicados, y se obviarán los problemas encontrados durante su desarrollo, los cuales serán expuestos en el Anexo III. Complicaciones y errores encontrados.

3.2.1. Requisitos.

Pasamos ahora a exponer los requisitos mínimos necesarios para alcanzar los objetivos expuestos en la sección 4.1 de este documento. Fundamentalmente se trata de requisitos que deben cumplir tanto nuestro clúster de microservicios, a través del despliegue automatizado con *Ansible*, como nuestro servidor de *IoT* alojado en el mismo.

Las características principales que deberemos abordar en nuestro despliegue automatizado del clúster de microservicios son:

· Infraestructura bien definida y flexible.

Nuestro despliegue ha de permitir modificaciones sin que esto provoque grandes impactos en el mismo. Dichas modificaciones pueden ser tanto a nivel de máquina, pudiendo asignar diferentes IPs, nombres de dominio, número de interfaces, etcétera, como a nivel de inclusión de nuevos servicios o configuraciones que nos puedan ser de interés para futuros despliegues.

· Capacidad de escalado de los nodos.

Otra de las características que debe ofrecer nuestro clúster automatizado es la capacidad para realizar escalados, tanto en el caso de querer añadir nuevas máquinas como en el caso de querer eliminarlas. Esto, además, debe poder llevarse a cabo sin que nuestro entorno sufra cualquier tipo de impacto.

· Tolerancia del sistema frente a fallos.

Dicha característica viene ya incluida en el caso de un despliegue correcto de un clúster de *Kubernetes*, por lo que, en caso de sufrir cualquier tipo de problema relacionado con un nodo, el sistema será capaz de reestructurarse automáticamente y seguir funcionando sin ningún tipo de impacto.

Por otro lado, los microservicios contenidos en dicho clúster deberán también cumplir los siguientes requisitos:

· Microservicios separados y bien definidos.

Cada uno de los microservicios deberá estar segregado según su funcionalidad. Así, el servicio que se encargue de recibir las peticiones no podrá compartir contenedor con el servicio encargado de persistir información en base de datos, o el servicio encargado de reenviar las peticiones al servidor principal. Esta es una premisa clave para que un entorno de microservicios funcione correctamente sin posibilidad de

colapsar en caso de necesitar escalado o de que las peticiones aumenten en un volumen considerable.

· Capacidad de recuperación frente a fallos.

En caso de que nuestro microservicio sufra cualquier tipo de fallo, el sistema ha de ser capaz de levantar un nuevo microservicio que sustituya al corrupto, minimizando los posibles errores que se puedan producir.

· Tiempos de recuperación asumibles.

Será necesario que, tras detectar un fallo en el sistema, éste sea capaz de recuperarse en la mayor brevedad posible, minimizando los fallos que se puedan desencadenar. Para esto, *Kubernetes* también ofrece ya una solución, por lo que en principio no habría que preocuparse, pero sí que deberemos de definir correctamente nuestro microservicio para que, en caso de fallo, no sufra pérdidas de información o incluso que se notifique de alguna manera dicho fallo.

3.2.2. Decisiones previas.

Dentro de las decisiones previas, vamos a repasar las herramientas de las que se ha hecho uso, tanto el entorno de trabajo como el software que ha servido de apoyo.

· Entornos de trabajo.

Para llevar a cabo este proyecto se ha trabajado sobre el sistema operativo Ubuntu 16.04. El hecho de escoger este sistema viene condicionado por *Kubernetes*, ya que su uso está orientado a sistemas Linux. Además, Linux nos permite utilizar KVM como hipervisor, lo cual nos facilita el despliegue de máquinas virtualizadas para llevar a cabo nuestra prueba de concepto.

Por otro lado, dichas máquinas virtuales fueron desplegadas con Centos como sistema operativo, ya que la referencia utilizada [15] para desplegar los diferentes servicios de *Kubernetes* se basaba en dicho sistema operativo.

· Software y compatibilidad.

Como hipervisor se ha utilizado KVM. Este hipervisor nos permite asignar a cada máquina virtual un *hardware* específico como [16] pueden ser tarjetas de red, disco, adaptadores gráficos, etcétera.

Para el despliegue sobre KVM de forma automatizada se ha utilizado Vagrant, ya que permite con flexibilidad y a través de ficheros JSON definir las máquinas que se van a desplegar.

Como herramienta de automatización de la configuración se ha utilizado ansible debido a la facilidad con la que se pueden definir las diferentes “recetas” y la curva de aprendizaje de la herramienta, la cual no es muy grande.

En cuanto al clúster de kubernetes, los principales servicios y sus versiones son:

- kubernetes: 1.5.2
- etcd: 3.2.15
- flanneld: 0.7.1
- docker: 1.13.1

Todas las herramientas de *software* serán ampliamente descritas en el capítulo 5 de la presente memoria.

Capítulo 4

4. Planificación y estimación de costes

Este capítulo describe los aspectos relacionados con la planificación del diseño y la estimación de costes prevista.

En primer lugar, Pse describirá cuáles son las previosiones de desarrollo de cara a conseguir los resultados finales de manera satisfactoria. Se tratará de describir a grandes rasgos cada una de las fases y sus aspectos más destacados.

Seguidamente, nos centraremos en los diferentes paquetes de trabajo que han ido dando forma a este proyecto desde que se comenzó a elaborar. También se añade una estimación del tiempo empleado en cada una de estas etapas, que culminará con la elaboración de un diagrama de Gantt que sirva como representación gráfica del desarrollo.

Una vez clara la planificación, pasaremos a exponer los recursos involucrados en este trabajo. Dentro de éstos haremos una distinción entre recursos humanos, *software* y *hardware*.

Por último, se mostrará una estimación de costes necesarios para abordar el presente proyecto en el ámbito económico.

4.1. Previsiones de desarrollo.

A modo de resumen de este trabajo del que ha surgido el presente proyecto vamos a desgranar las perspectivas para su desarrollo. Para ello, vamos a describir cronológicamente los pasos que se deberían dar para su elaboración, comenzando por las fases previas en las que será necesario recabar información acerca de los diferentes aspectos que aquí se exponen.

En primer lugar, se realizará un estudio bibliográfico para obtener los conocimientos básicos y esenciales acerca de todo lo relacionado con los microservicios. En ella recopilaremos herramientas y artículos claves posteriormente para la fase de diseño, profundizando en el conocimiento de la herramienta más madura enfocada a este ámbito: *Kubernetes*.

Una vez claros los conceptos básicos nos lanzaremos a trabajar con las herramientas básicas e indispensables del proyecto como son *Vagrant, Ansible* y el orquestador de microservicios *Kubernetes*. Comenzaremos creando “recetas” básicas con *Ansible* para adaptarnos a este lenguaje, y poco a poco ampliaremos el conocimiento y añadiremos estructuras y funcionalidades más complejas. Al mismo tiempo, buscaremos la vía para incluir en *Vagrant* las sentencias que llevaran a cabo la automatización de configuración con *Ansible*.

El siguiente paso será comenzar a programar las “recetas” orientadas a ejecutar *Kubernetes* y todos los servicios y ficheros de configuración necesarios para su correcto

funcionamiento. Para ello, habrá que apoyarse en distintas fuentes de referencia y filtrar la información realmente válida, ya que es posible que nos encontramos con algunas referencias que no se ajuste a lo que realmente se necesita, bien por tratarse de versiones demasiado antiguas de la herramienta o bien por problemas técnicos que se pueden encontrar al realizar pruebas. Muchos de estos servicios, que en un principio parecen algo abstractos, se estudiarán con el objetivo de entender el papel de todos y cada uno de ellos y ver su influencia en el sistema. Este será uno de los procesos más largos y complicados, ya que el desarrollo en este campo es reciente y no hay mucha información al respecto, además de que son conceptos totalmente nuevos y diferentes a los que venímos tratando, por lo que hubrá que realizar una gran cantidad de pruebas y buscar soluciones a los problemas de implementación (*bugs*) que se irán encontrando dentro de las distintas herramientas.

Una vez desplegado el clúster de *Kubernetes*, se dispondrá a comprobar el correcto funcionamiento del sistema, desplegando un microservicio de prueba y comprobando que se inicie correctamente en las distintas máquinas destinadas a levantar los diferentes contenedores. Para ello, ejecutaremos un microservicio con un servidor REST o similar, y se comprobará que responde a peticiones desde fuera del mismo.

Una vez conseguido esto, se pasará a estudiar el servidor *LoraServer*, destinado a aplicaciones de *Internet of Things*. El objetivo fundamental será adaptar dicho servidor para conseguir incluirlo en una infraestructura orientada a microservicios, por lo que el primer paso será conseguir añadir distintos contenedores para cada una de las funcionalidades de este servidor y, posteriormente, crear un despliegue en *Kubernetes* que abarque todos los contenedores y permita la interacción entre ellos. Este punto de acción se prevee que también nos lleve bastante tiempo, ya que no es valadí adaptar el servidor a los requisitos de microservicios, por lo que hubrá que realizar gran cantidad de pruebas y leer bastante documentación para conseguir el correcto funcionamiento.

Como conclusión final, se evaluará dicho servidor dentro de nuestro clúster de microservicios para confirmar si realmente las virtudes que ofrece a priori esta nueva arquitectura se corresponden con un caso práctico.

4.2. Planificación.

En este apartado se realizará la descripción de cada uno de los paquetes de trabajo en los que se divide este proyecto. La idea es acercar lo máximo posible a la realidad el proceso de desarrollo de las diferentes partes de manera detallada y precisa.

Los paquetes de trabajo en los que se divide este proyecto son los siguientes:

PT1: Búsqueda de información

Este primer paquete consiste en recabar el máximo de información posible que facilite tanto la comprensión como la posterior puesta en marcha del presente proyecto. Por lo tanto, será importante aclarar todos los aspectos ligados a las

tecnologías implicadas con nuestro trabajo para conocer lo que ya existe y los avances que se pueden realizar en este campo.

PT2: Familiarización con *Ansible*.

El siguiente paso consiste en la toma de contacto con la herramienta de automatización de configuración *software Ansible*. Esta herramienta es fundamental para el funcionamiento de todas las partes, por lo que es importante descubrir todas y cada una de las posibilidades que nos ofrece. Es importante también conocer su integración con la herramienta de automatización de despliegue de máquinas virtuales *Vagrant*.

PT3: Toma de contacto con *Kubernetes*.

Con este paquete pasamos a identificar todos los servicios y posibilidades que nos ofrece esta herramienta una vez desplegada con *Ansible* en nuestro clúster. Para ello será necesario tener conocimiento de redes, comunicación REST, microservicios y YAML, utilizado para los descriptores de los microservicios a desplegar en el clúster.

PT4: Desarrollo con *Ansible* de la configuración de máquinas para clúster de *Kubernetes*.

Esta etapa es fundamental para poder desarrollar el resto del proyecto. En ella, se van a configurar distintas recetas con *Ansible* de cara a ser capaces de provisionar la configuración necesaria a las diferentes máquinas con el objetivo último de levantar un clúster de microservicios gestionado con *Kubernetes*. Por lo tanto, será necesario traducir los conocimientos de *Kubernetes* y de *Ánsible* en recetas de esta última herramienta para, con la ayuda de *Vagrant*, levantar las máquinas y proveerlas automáticamente de su configuración.

PT5: Estudio de servidor de *IoT LoraServer*.

Antes de comenzar la adaptación del servidor, es importante conocer el funcionamiento del mismo para poder testearlo posteriormente dentro de nuestra infraestructura. En esta etapa se estudia dicho servidor y cada uno de los componentes por separado que permiten su interacción con las motas que envían la información. Además, se estudian las posibilidades de desarrollo de contenedores aislados de cada uno de estos componentes para adaptar así el servidor a una arquitectura basada en microservicios.

PT6: Desarrollo de contenedores y *deployment* de *LoraServer* para *Kubernetes*.

En este paquete se consigue el despliegue con un solo contenedor de dicho servidor en el clúster en forma de *deployment* de *Kubernetes*.

Una vez comprobado su funcionamiento se pasa a complicar más el escenario, dividiendo en pequeños contenedores cada uno de los servicios más relevantes

de la aplicación de tal forma que el *deployment* esté formado por varios contenedores interactuando entre sí gracias a *Kubernetes*.

Este paquete de trabajo será fundamental para adquirir los conocimientos básicos del funcionamiento de *Kubernetes* para sacar conclusiones a cerca de su rendimiento y las posibilidades que ofrece.

PT7: Fase de pruebas

Una vez implementadas con éxito todas las partes del proyecto se establece un periodo de pruebas en el que se examina el sistema en diferentes escenarios. En primer lugar, se prueba su correcto funcionamiento del clúster de *Kubernetes*. Más tarde se testea el funcionamiento del servidor *LoraServer* sin introducirlo en el clúster, a partir de la configuración de distintos contenedores utilizando *Docker*. Una vez comprobado el correcto funcionamiento, se procede a adaptar dichos contenedores a una arquitectura de microservicios gestionada por *Kubernetes*. Además, se realizan pruebas destinadas a comprobar la reacción frente a fallos tanto en el clúster como en el propio microservicio para corroborar un correcto funcionamiento de todos los sistemas implicados.

PT8: Elaboración de la memoria técnica del proyecto

Con la elaboración de la memoria se pretende documentar todo lo relativo al trabajo realizado, recogiendo tanto los aspectos teóricos como las implementaciones prácticas realizadas de manera detallada. También se pretende recoger todos los resultados obtenidos en el análisis del sistema una vez puesto en marcha. Su elaboración se realiza paralelamente al desarrollo del proyecto, pese a que finalmente se recoja todo en un documento final.

Identificados y descritos todos los paquetes de trabajo que componen nuestro proyecto es necesario realizar una planificación. La planificación que se expone a continuación tiene un carácter meramente orientativo, ya que son varios los factores que influyen a la hora de desarrollar el proyecto, pero la intención es acercarlo lo máximo posible a la realidad de los tiempos marcados en cada plazo. En la siguiente tabla se muestra todo el proceso desglosado en un diagrama de Gantt.

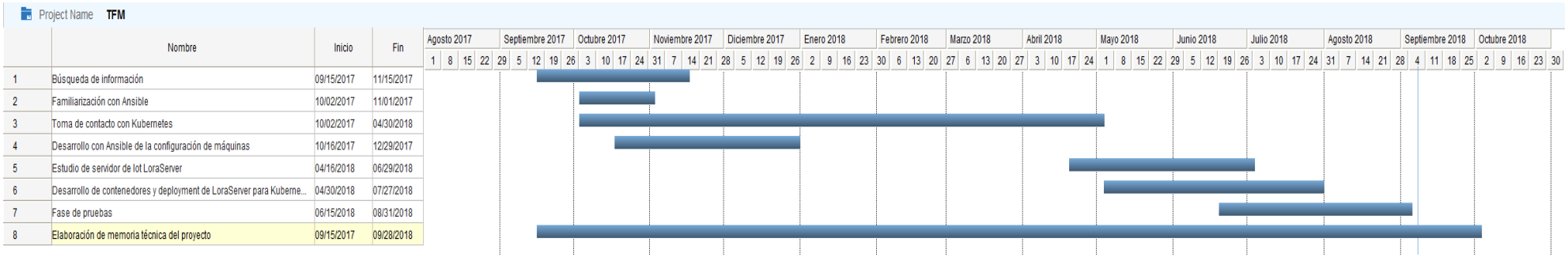

Figura 6. Diagrama de Gantt estimado del proyecto.

En cuanto a horas de trabajo se refiere, en la siguiente tabla hemos desglosado cada uno de los paquetes de trabajo con una aproximación del tiempo a emplear en cada uno de ellos.

Paquete de trabajo	Descripción	Tiempo estimado
PT1	Búsqueda de información	50 horas
PT2	Familiarización con *Ansible*	60 horas
PT3	Toma de contacto con *Kubernetes*	40 horas
PT4	Desarrollo con *Ansible* de la configuración de máquinas para clúster de *Kubernetes*	60 horas
PT5	Estudio de servidor de IoT *LoraServer*	20 horas
PT6	Desarrollo de contenedores y *deployment* de *LoraServer* para *Kubernetes*	100 horas
PT7	Fase de pruebas	20 horas
PT8	Elaboración de la memoria técnica del proyecto	100 horas
TOTAL		**450 horas**

Tabla 1. Distribución temporal del proyecto.

De esta manera queda conformada la planificación de nuestro proyecto, desglosada tanto en las acciones a realizar con en el tiempo estimado que deberá emplearse en cada una de las diferentes etapas.

4.3. Recursos utilizados.

A continuación, se van a tratar de identificar todos los recursos involucrados en el proyecto. Se realizará una clasificación diferenciando entre los recursos de tipo humano, *hardware* y software.

5.3.1. Recursos humanos.

· D. Jorge Navarro Ortiz, Profesor Contratado Doctor de la Universidad de Granada en el Departamento de Teoría de la Señal, Telemática y Comunicaciones, en calidad de tutor del proyecto.

· Manuel Sánchez López, alumno del Máster de Ingeniería de tecnologías de telecomunicación y autor del presente proyecto.

4.3.2. Recursos *hardware*.

· Ordenador portátil Dell, con procesador Intel Core i5-430M a 2.26 GHz, memoria RAM de 4 GB y disco duro de 500 GB de capacidad. Este ordenador se utiliza tanto para el despliegue de las diferentes máquinas que componen el clúster de *Kubernetes* de manera virtualizada, sobre el hipervisor de KVM como para el desarrollo de código en *Ansible* y plantillas para creación de contenedores y despliegues en *Kubernetes*.

· Ordenador HP ELITEBOOK con procesador Intel Core i5, memoria RAM de 16 GB y disco duro SSD de 200 GB de capacidad. Este ordenador se utiliza tanto para pruebas de concepto con *Docker* como para búsqueda de información y redacción de la memoria.

· Línea de acceso a Internet. Necesaria para la realización de pruebas. Imprescindible para la actualización de versiones de los contenedores y del código en sí.

4.3.3. Recursos *software*.

· Sistema operativo *Linux Ubuntu 16.04* (64 bits), utilizado en el ordenador portátil, sobre el que se desarrollará el código de automatización de configuración de máquinas, la programación de plantillas para el despliegue de microservicios y se trabajará con las herramientas *Ansible*, *Vagrant* y *Kubernetes*.

· *Ansible, software* esencial para la automatización de la configuración de las diferentes máquinas que compondrán el clúster.

· *Kubernetes*, orquestador de clúster de microservicios clave para un funcionamiento coordinado y supervisado del desarrollo *software* de las aplicaciones incluidas.

· Hipervisor *KVM*, esencial para el despliegue de máquinas virtuales que compondrán nuestro clúster de *Kubernetes*.

· *Vagrant*, herramienta utilizada para automatizar los despliegues de máquinas virtuales en *KVM*.

· *Docker,* herramienta utlizada para el despliegue de contenedores previa a su inclusión en el clúster *Kubernetes* (el cuál trabaja internamente con esta herramienta) para la realización de pruebas de concepto.

· *Microsoft Word 2016*, para la elaboración de la memoria del presente proyecto. Facilita el seguimiento del tutor del proyecto permitiendo añadir comentarios y sugerencias a los documentos.

· *Gantter*, herramienta online para la elaboración de diagramas de Gantt.

· *Vim,* editor de textos utilizado para programar tanto las plantillas de contenedores y despliegues como el código de *Ansible*.

· Navegador web *Google Chrome* para comprobar el correcto funcionamiento de la interfaz web de nuestro servidor de *IoT* y el provisionamiento de las motas en el mismo.

· Servidor de *IoT LoraServer*, utilizado para realizar la prueba de concepto de un microservicio dentro del clúster.

4.4. Estimación de costes.

En este punto trataremos de plasmar la estimación de costes prevista al abordar este proyecto.

Es notable que los costes no son elevados ya que, como puede comprobarse en el punto anterior, el *software* es prácticamente en su totalidad gratuito.

Recursos humanos

Los costes relacionados con los recursos humanos se van a calcular tomando como base el tiempo invertido en cada uno de los paquetes de trabajo. Es por ello que tomaremos como referencia la *Tabla 5.1.1* en la que se detalla de manera aproximada el tiempo empleado en cada una de las diferentes etapas. Adicionalmente estableceremos las siguientes premisas:

· El sueldo medio de un graduado en Ingeniería en Tecnologías de Telecomunicación es de 20 euros / hora.

· El sueldo de un Profesor Contratado Doctor de la Universidad de Granada se estima en torno a 50 euros / hora. Se considera que en total ha podido invertir entre tutorías para orientar al alumno y la posterior revisión del trabajo unas 15 horas.

· Teniendo en cuenta la embergadura del proyecto, los tiempos necesarios para su realización y las tecnologías sobre las que versa, las cuáles son punteras y necesitan de un esfuerzo de aprendizaje previo, el coste relativo a los recursos humanos se establecerá conforme a lo antes mencionado y las horas empleadas en cada actividad.

De este modo se estipula el presupuesto que presentamos en la siguiente tabla:

Paquete de trabajo	Descripción	Coste estimado
PT1	Búsqueda de información	1000 euros
PT2	Familiarización con *Ansible*	1200 euros

PT3	Toma de contacto con *Kubernetes*	800 euros
PT4	Desarrollo con *Ansible* de la configuración de máquinas para clúster de *Kubernetes*	1200 euros
PT5	Estudio de servidor de IoT *LoraServer*	400 euros
PT6	Desarrollo de contenedores y *deployment* de *LoraServer* para *Kubernetes*	2000 euros
PT7	Fase de pruebas	400 euros
PT8	Elaboración de la memoria técnica del proyecto	2000 euros
Tutorización por Profesor Contratado Doctor		750 euros
TOTAL		**9750 euros**

Tabla 2. Coste estimado de recursos humanos.

Para una comparativa más visual vamos a representar el coste asociado a cada paquete en el siguiente gráfico de barras:

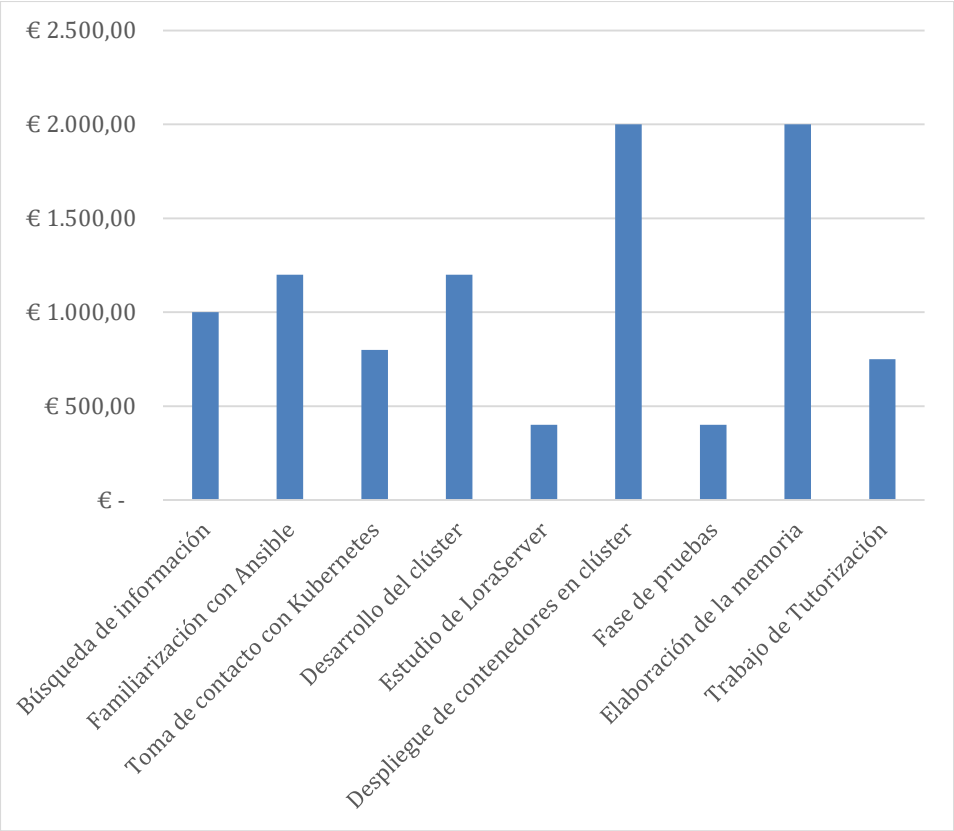

Figura 7. Gráfico de barras del coste de recursos humanos para cada paquete de trabajo.

Recursos *hardware*

Los costes asociados a los recursos *hardware* están asociados al material previamente mencionado en el apartado 4.3.2. A continuación se exponen con detalle:

Recurso	Coste estimado	Vida media
Ordenador portátil Dell	300 €	36 meses
Ordenador portátil HP	1350 €	36 meses
Línea acceso a Internet	30 euros / mes	-

Tabla 3. Costes estimados de recursos hardware.

Recursos *software*

Como ya se comentó con anterioridad, los recursos *software* empleados para la realización de este proyecto son gratuitos, por lo que no supondrán ningún tipo de coste adicional a nuestro proyecto. Por lo tanto, ésto supondrá un importante abaratamiento en los costes finales.

4.5. Presupuesto final.

Como resumen de los costes surgidos de los diferentes recursos utilizados vamos a exponer este apartado, con el objetivo de presentar una estimación final del presupuesto necesario para abordar este proyecto en su totalidad. En la tabla 5.4.1. se recogen los costes asociados a cada recurso considerando una amortización de 10 meses para los recursos *hardware*; periodo empleado para elaborar el proyecto.

Concepto	Coste
Recursos humanos	**9750 euros**
Ordenador portátil DELL	**83,33 euros** *(300 euros x 10 meses / 36 meses)*
Ordenador portátil HP	**375 euros** *(1350 euros x 10 meses / 36 meses)*
Línea de acceso a Internet	**300 euros** *(30 euros / mes x 10 meses)*
TOTAL	**10.508,33 euros**

Tabla 4. Presupuesto final estimado.

Si echamos un vistazo al presupuesto final estimado, podemos comprobar que la mayor parte está ligada a los recursos humanos con un 93% de los costes totales. Por lo tanto, podemos concluir que este proyecto no requiere de un gran esfuerzo económico en cuanto a recursos *hardware* y *software*, sino que centra su importancia en la capacidad de trabajo de los ingenieros contratados.

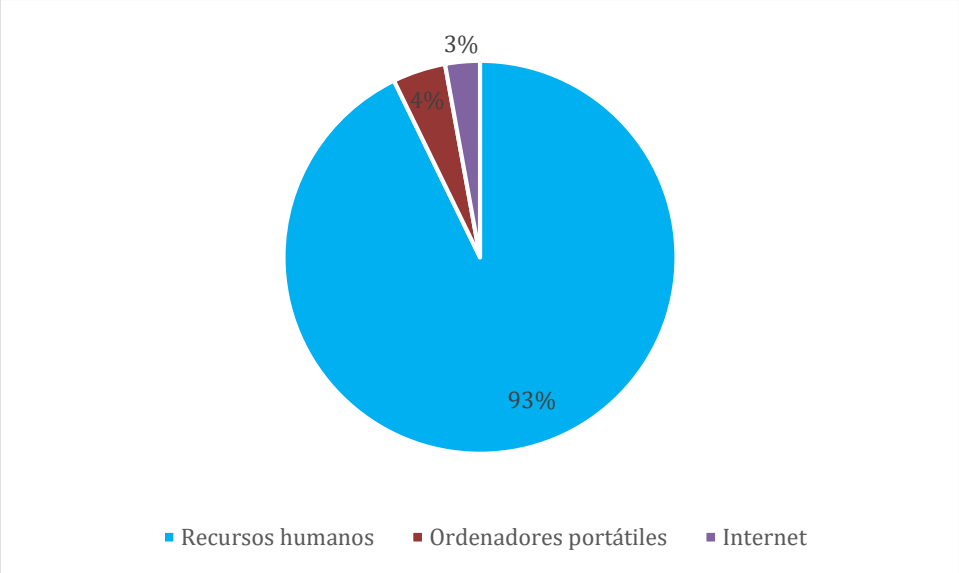

Figura 8. Porcentaje de costes final.

4.6. Consideraciones finales sobre la planificación.

Una vez finalizado el proyecto podemos concluir que se han cumplido prácticamente acorde con los plazos marcados desde un principio para la consecución de los objetivos de este trabajo. No obstante, es cierto que el desarrollo del paquete de trabajo referente a la familiarización y desarrollo de despliegues con *Kubernetes* no se desarrolló de forma continuada, ya que hubi que estudiar en paralelo otros aspectos como el servidor de *IoT LoraServer*.

Capítulo 5

5. Herramientas utilizadas

Este capítulo versa sobre las herramientas utilizadas en la elaboración del clúster automatizado de *Kubernetes*. En él, definiremos con detalle dichas herramientas y se expondrán las funcionalidades que han ayudado a la consecución del proyecto.

En primer lugar, introduciremos *Kubernetes*, una herramienta de orquestación de microservicios que nos facilitará tanto su despliegue cómo su gestión y monitorización. Posteriormente describiremos *Ansible*; la herramienta seleccionada para la automatización de configuración de máquinas. Para finalizar este capítulo, también se añade un apartado para presentar la herramienta que nos ayuda a automatizar los despliegues de máquinas virtuales, que en nuestro caso es *Vagrant.*

5.1. Kubernetes.

Pasamos a describir la herramienta de orquestación de clústers de microservicios, *Kubernetes*. Esta herramienta es el núcleo sobre el que gira este proyecto, por lo que su importancia es vital en el engranado. En los siguientes apartados analizaremos sus características principales y el potencial que ofrece en la gestión y monitorización de los microservicios desplegados.

5.1.1. ¿Qué es Kubernetes?

Kubernetes [17] no es más que un sistema de código abierto que permite gestionar un clúster de microservicios mediante la implementación, escalado y administración de aplicaciones en contenedores de manera automatizada o manual. Esta herramienta fue desarrollada por *Google*.

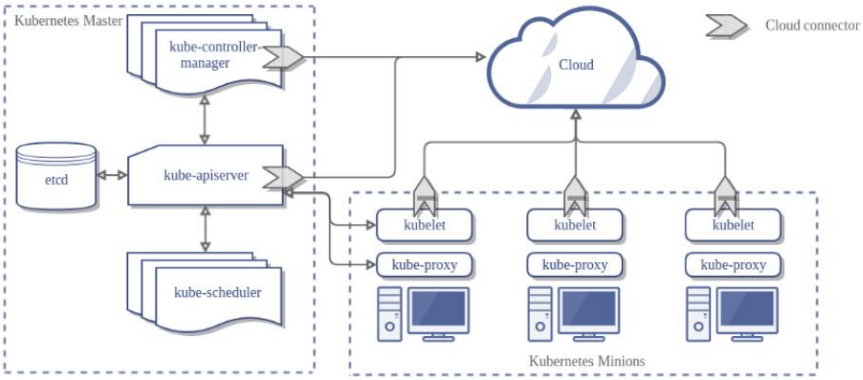

Figura 9. Ejemplo de arquitectura de clúster de Kubernetes.

Si observamos la Figura 6, podemos identificar los diferentes elementos que componen la arquitectura de este sistema y que pasamos a definir:

· *Etcd*: encargado de almacenar los datos de configuración de nuestro clúster.

· *Kube-Apiserver*: es el núcleo del nodo que gestionará el sistema (*Master*), encargado de comunicarse con los diversos componentes del mismo y monitorizando el estado del clúster.

· *Kube-Controller-Manager*: su cometido es el de comprobar que el estado deseado del clúster y el estado actual se mantienen en concordancia.

· *Kube-Scheduler*: es el encargado de repartir la carga entre los diferentes nodos esclavos (*Minions*).

· *Kubelet*: es el encargado de administrar los despliegues realizados en los nodos esclavos, recibiendo las especificaciones del servidor API.

Por otro lado, conviene especificar detalladamente algunos términos utilizados en *Kubernetes* de especial relevancia:

· *Pods*: es una agrupación de contenedores que se ejecutan en un mismo nodo y comparten los recursos como sistemas de archivos, espacio de nombres en el kernel y una dirección IP.

· *Deployments*: estos bloques de construcción se pueden usar para crear y administrar un grupo de *pods*. Los *deployments* pueden ser utilizados a nivel de servicio para un escalado horizontal que garantice la disponibilidad.

· Servicios: son los puntos finales que se pueden direccionar por nombre y se pueden conectar a los *pods* utilizandos los selectores de etiquetas. El servicio automáticamente enviará solicitudes por turnos entre los diferentes *pods* desplegados. *Kubernetes* configurará un servidor DNS para el clúster que busca nuevos servicios y les permite ser direccionados por su nombre. Los servicios son la parte frontal de las cargas de trabajo de su contenedor.

· Etiquetas: son pares clave-valor unidos a objetos y se pueden usar para buscar y actualizar múltiples objetos como un conjunto.

Una vez definidos los conceptos básicos de *Kubernetes*, vamos a justificar su uso en el presente proyecto.

5.1.2. ¿Por qué utilizar Kubernetes?

Previamente a la inclusión en esta herramienta, se realizó un estudio de las diferentes opciones que permitían la gestión de un clúster de microservicios, llegándose a probar otras altenativas antes de tomar la decisión. Entre dichas alternativas se encuentran *Docker Swarm* [18] o *Apache Mesos* [19].

Dependiendo de cuál sea el propósito del proyecto, cada uno de ellos tiene sus ventajas e inconvenientes. No obstante, podemos hacer una división clara entre *Docker Swarm*, el cuál es un sistema más sencillo y con menos funcionalidades, y *Kubernetes* y *Apache Mesos*, los cuales son sistemas bastante más complejos, pero que ofrecen mayores posibilidades.

Si comparamos arquitecturas, todos siguen el mismo paradigma de utilizar nodos controladores que gestionan nodos esclavos. No obstante, si entramos más en detalle podremos ver algunos aspectos en los que cada cuál tiene una solución concreta y unos alcances determinados.

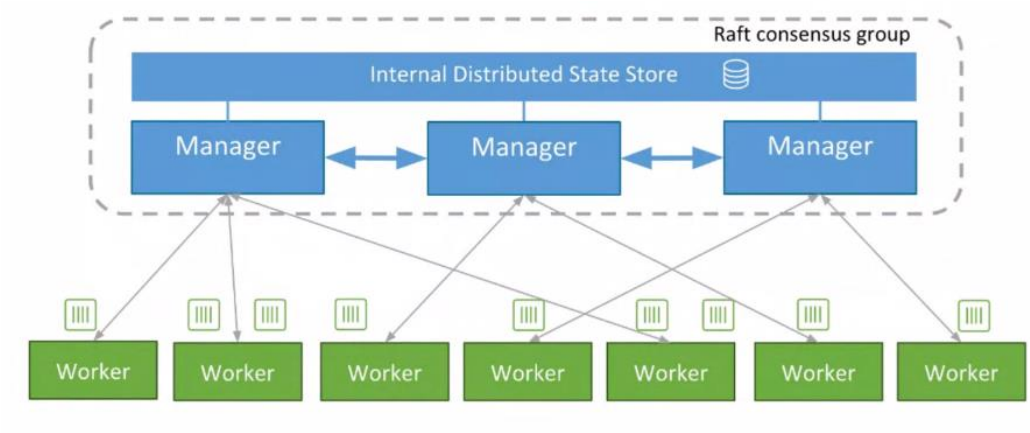

Figura 10. Arquitectura de Docker Swarm.

Si nos centramos en la comparativa entre *Kubernetes* y *Docker Swarm* [17], podemos extraer las siguientes conclusiones:

· **Alta disponibilidad:** ambos son capaces de ofrecer alta disponibilidad, permitiendo *Kubernetes* un escalado tanto manual como automatizado según la definición en la implementación especificada en YAML, al igual que *Docker Swarm*, que define plantillas para este mismo cometido.

· **Balanceo de carga:** en *Kubernetes*, gracias a que los *pods* se exponen a través de un servicio, el balanceo se gestiona dentro del cúster. En *Docker Swarm* existe un componente DNS que se puede utilizar para distribuir las solicitudes a un nombre de servicio.

Escalado automático para la aplicación: el escalado automático se define de forma declarativa en *Kubernetes* al describir la implementación. Además, está disponible el objetivo de utilización de CPU por *pod*, al igual que otros muchos. En *Docker Swarm*, por contro, no está disponible directamente. Para cada servicio se puede declarar el número de tareas que se desean ejecutar, pero el escalado se realiza manualmente y, de estaforma, se adaptará automáticamente el administrador *Swarm* para agregar o eliminar tareas.

· **Actualizaciones de aplicaciones continuas y reversión:** *Kubernetes* permite, a través de su controlador, estrategias de actualización gradual y recreación. Las

actualizaciones continuas pueden especificar el número máximo de *pods* no disponibles o el número máximo que se ejecuta durante el proceso. En cuanto a *Docker Swarm*, en el momento de despliegue, puede aplicar actualizaciones continuas a los servicios. Su administrador permite controlar la demora entre la implementación del servicio a diferentes conjuntos de nodos, siendo capaz de actualizar en cada turno.

· **Chequeo de estado:** en el caso de *Kubernetes* se permiten dos comprobaciones; tanto la vitalidad (es sensible a la aplicación) como la preparación (responde a la aplicación, pero está ocupado preparando y aún no puede servir peticiones). En cuanto a Docker Swarm, se limita solo al chequeo de los servicios; si un contenedor que respalda el servicio no aparece, se inicia un nuevo contenedor.

· **Almacenamiento:** *Kubernetes* consta de dos APIs para este propósito; una que proporciona abstracciones para *backends* de almacenamiento individual y otra que está centrada en la abstracción para una solicitud de recursos de almacenamiento, que puede cumplirse con diferentes *backends* destinados a este propósito. Además, permite distintos tipos de vólumenes persistentes con soporte de bloque o archivo. Por su parte, *Docker Swarm* admite volúmenes de montaje en un contenedor, y tiene un catálogo parecido al de *Kubernetes* para este propósito.

· **Redes:** el modelo de red de *Kubernetes* es plano, lo cual permite que todos los *pods* se comuniquen entre sí. Este modelo requiere de dos CIDR, uno destinado a la dirección IP de los *pods* y otro para los servicios. En cuanto a *Docker Swarm*, se crea una red de superposición para servicios que abarcan todos los hosts además de una red de puente para los contenedores.

· **Descubrimiento del servicio:** en *Kubernetes* se pueden encontrar utilizando variables de entorno o DNS, mientras que en *Docker Swarm* se asigna a cada servicio un nombre DNS único y se lleva a cabo un balanceo de carga entre contenedores. También se puede utilizar un archivo estático o una lista de nodos *backend* de descubrimiento, el cuál debería almacenarse en un *host* al que pueda acceder *Swarm Manager*.

· **Rendimiento y escalabilidad:** a partir de la versión 1.6, *Kubernetes* es capaz de escalar el clúster a 5000 nodos. Sus objetivos de nivel de servicio son una capacidad de respuesta del 99% de las llamadas y un tiempo de inicio de *pod* de 5 segundos del 99% de los mismos (con imágenes previamente lanzadas).

En cuanto a la popularidad y uso actual, *Kubernetes* supera a *Docker Swarm* en todas las métricas, ostentando un 80% del interés en artículos de noticias, popularidad en herramientas como *Github* y búsqedas en la web.

No obstante, si que es cierto que su complejidad es mayor que la de *Docker Swarm*, sobre todo a la hora de su implementación, pese a que *Kubernetes* ha intentado mitigar dicho inconveniente incorporando opciones como *Minikube* [20] o *Kubeadm* [21].

La razón por la que nos decantamos por *Kubernetes* y no por *Apache Mesos* se fundamenta en que, en nuestro caso, el objetivo era trabajar exclusivamente con la orquestación de contenedores *Docker* y ser capaces de personalizar nuestra infraestructura de clúster para levantarlo de manera automatizada, mientras que el uso *Apache Mesos* está más orientado a un entorno de mayor embergadura, en el que se incluyen además servicios de sistemas de datos distribuidos, portabilidad de plataformas, etcétera. Esto podría integrarse en nuestra infraestructura de *Kubernetes*, pero no se hace necesario para empezar a trabajar con él.

5.1.3. Despliegue de microservicios con Kubernetes.

En este apartado vamos a adentrarnos en el uso de *Kubernetes* más básico para ofrecer los conceptos y la metodología clave de la herramienta al lector, facilitando su incursión en el uso de la misma.

El primer punto a tener presente es el uso de plantillas YAML [22] para llevar a cabo el desarrollo en esta plataforma. Así, podemos crear un despliegue de un *pod* sencillo como el que mostramos a continuación y el cuál desgranaremos poco a poco en las siguientes líneas para comprender su contenido:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

Este es un ejemplo sencillo que despliega un *Pod*, el cuál contiene un único contenedor y que ejecuta un comando en *bash* mostrando un mensaje cada 3600 segundos.

La primera etiqueta que se utiliza, "*apiVersion*", identifica la versión de la api que vamos a utilizar, mientras que la etiqueta "*kind*" marca el tipo de despliegue que vamos a realizar (*pod, deployment, system-controller,* etcétera), en nuestro caso se trara de un *Pod*. Seguidamente podemos ver campos de metadatos, mediante los cuales asignamos un nombre a nuestro *pod*, etiquetas que podrán ser utilizadas por el sistema para encontrar nuestra aplicación, etcétera. Por último, mediante la etiqueta de "*spec*" se especifican las características y en engranaje de nuestro *Pod*, que en este caso contiene un container al que asignamos el nombre de "*myapp-container*" y que se basa en la imagen "*busybox*". Dicha imagen se descargará de los repositorios de *Docker*, por lo que debe existir allí, y al iniciarse lanzará el comando que estamos especificando en la línea correspondiente a "*command*".

Si queremos ahora lanzar este *Pod* y que se despliegue en uno de los "*Minions*" de nuestro clúster, bastará con ejecutar:

```
# kubectl create –f <nombre_plantilla>
```

Una vez hecho esto, podemos testear que se ha acometido la acción consultando los *Pods* desplegados en nuestra infraestructura, o incluso solicitando nuestro *pod* en cuestión por su nombre mediante:

```
# kubectl get pod myapp-pod
```

Como podemos ver en la siguiente figura, se muestra el *pod* que hemos instanciado, su estado., el número de reintentos para su creación o el tiempo que lleva activo.

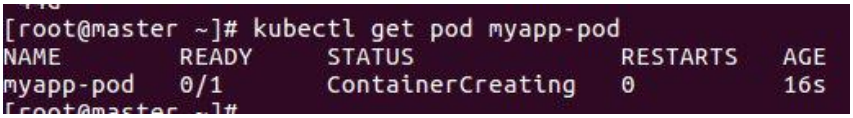

Figura 11. Petición de un pod con kubernetes recién creado.

Una vez esté activo y operativo, podemos ver que así se refleja en el sistema:

Figura 12. Petición de un pod con kubernetes activo y corriendo.

Otra opción es crear "*Deployments*" [23], el cual representa un conjunto de *pods* idénticos. Un *deployment* ejecuta múltiples réplicas de una aplicación y es capaz de reemplazar las instancias automáticamente en caso de fallo o de no ser accesibles. En este sentido, los *deployments* ayudan a asegurar que una o más instancias de la aplicación está disponible para atender a las peticiones de los usuarios.

Un ejemplo de *deployment* de *Kubernetes* se muestra a continuación:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
```

```
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Como podemos aprecias, en este caso, la etiqueta "*kind*" ahora se rellena con el tipo *Deployment* y además podemos ver algunas otras etiquetas como la de "*app*", que sirve para identificar a qué *pod* corresponde cada especificación. Por otro lado, en este despliegue se indican cuántas réplicas queremos mediante su etiqueta correspondiente, por lo que, en este caso, tendríamos tres *nginx* corriendo de manera simultánea.

5.2. Ansible.

Una vez claros los conceptos de *Kubernetes* pasamos ahora a la herramienta que nos ayudará a automatizar el despliegue de un clúster del mismo, *Ansible* [12]. Basándonos en todos los requisitos necesarios por parte de *Kubernetes*, el objetivo será extrapolarlos al desarrollo de recetas en plantillas YAML de tal forma que consigamos disponer de la infraestructura necesaria para trabajar con ello.

Antes de meternos de lleno con el desarrollo de las recetas de *Ansible* vamos a introducirlo y a comentar algunos aspectos de especial relevancia.

Un ejemplo sencillo de *playbook* de *ansible* definido en YAML podría ser el siguiente:

```
---
- hosts: cluster
  tasks:
    - name: Crea directorio
      file: path=/tmp/my_dir state=directory mode=0755


    - name: Descarga web
      get_url: url=https://google.com
      dest=/tmp/my_dir/index.html
```

Básicamente, estamos instando a que se ejecute en todos los *hosts* que tenemos configurados mediante el valor "*cluster*" y le hemos definidos dos tareas: la primera es crear un directorio con unos permisos determinados y la segunda descargar una web e insertar su contenido en un fichero en el directorio que creamos en la anterior tarea. Al ser un lenguaje secuencial, no habrá problema de que dicho directorio no esté creado previamente. Para lanzar ahora nuestro *playbook* basta con ejecutar:

```
# ansible-playbook -i hosts <nombre_playbook>
```

Donde la opción "-i" sirve para indicar cual es el fichero que contiene la información sobre los hosts que vamos a provisionar, y cuyo contenido podría ser el siguiente:

```
[master]
10.10.10.51
[minions]
10.10.10.52
10.10.10.53
[cluster]
10.10.10.51 cluster_host=master
10.10.10.52 cluster_host=minion-1
10.10.10.53 cluster_host=minion-2
```

Al añadir el valor "*cluster*", la configución se realizará en las tres máquinas que vemos añadidas bajo dicha etiqueta.

5.2.1. ¿Por qué utilizar Ansible?

Dentro del campo de las herramientas encargadas de la automatización de la configuración de máquinas podemos destacar, como las más relevantes, las tres siguientes: *Ansible, Puppet y Chef.* Cada una de estas herramientas utiliza diferentes metodologías para conseguir un objetivo común [24].

Estas tres herramientas de gestión se han desarrollado para reducir la complejidad a la hora de configurar infraestructuras con recursos distribuidos, posibilitando una mayor rapidez, y asegurando fiabilidad y elasticidad.

Comenzaremos describiendo *Puppet* [25]. Esta herramienta de código abierto está desarrollada en *Ruby* [26] y se basa en diferentes *templates* que describen de manera declarativa lo que se desea realizar. *Puppet* se basa en una arquitectura agente/máster, donde los agentes gestionan los nodos y solicitan información relevante a los masters que controlan la información de configuración. El agente lleva a cabo reportes de estado y peticiones que envía a la máquina donde reside el servidor master de *Puppet*, el cuál luego comunica su respuesta y los comandos requeridos.

Entre sus puntos fuertes destacan su elasticidad en la automatización y como herramienta de reporte, el soporte activo de la comunidad, una interfaz web muy intuitiva para llevar un seguimiento de las tareas, incluyendo reportes y gestión en tiempo real de los nodos, robustez, soporte de varios sistemas operativos, y aspectos como su madurez, estabilidad y capacidad para gestionar infraestructuras heterogéneas.

No obstante, la curva de aprendizaje de *Puppet* es bastante pronunciada y puede ser tediosa su inicialización para nuevos usuarios. Por otra parte, el proceso de instalación carece de un informe de errores suficientemente detallado y no es la mejor opción para soluciones que impliquen despliegues escalados, como es nuestro caso.

Chef, por su parte, es una herramienta que utiliza también una arquitectura basada en cliente-servidor. Es flexible en infraestructuras automatizadas en cloud, permitiendo instalar aplicaciones a máquinas virtuales en baremetal o en contenedores en la nube. Su arquitectura por tanto es muy parecida a la de *Puppet* en tanto en cuanto se basa en las peticiones/respuestas intercambiadas entre los masters y los agentes.

Entre sus puntos fuertes destacan la flexibilidad en soluciones para gestión de sistemas operativos y middleware, el hecho de estar diseñada para programadores, su documentación amplia, gran soporte y comunidad activa, su posibilidad de uso en sistemas híbridos, estadísticos y reportes y su ejecución de órdenes secuencial, al contrario que *Puppet*.

En contraposición, *Chef* necesita también de un tiempo importante para su aprendizaje, está supeditado a las peticiones de los agentes para realizar los cambios o acciones, lo cual hace que no sea inmediato y que se siga una planificación concreta, y la documentación es bastante amplia y dispersa, lo cual puede dificultar la búsqueda de contenido concreto. Además, esta herramienta requiere de una invesión para poder ser utilizada con todas sus características.

Llegamos por último a *Ansible*, la herramienta seleccionada para llevar a cabo este proyecto. Esta herramienta fue desarrollada para simplificar la complejidad de orquestar y configurar las tareas de gestión. La herramienta está escrita en *Python* [27], y permite a los usuarios mediante scripts en YAML definir las tareas a realizar. En el caso de *Ansible* no se requiere de la instalación de agentes en los nodos para llevar a cabo el provisionamiento; el despliegue se realiza desde un punto centralizado y únicamente se indica en un script las máquinas que se quieren provisionar, hacia las cuáles realizará una comunicación mediante un túnel *ssh* y ejecutará los comandos pertinentes.

Entre sus puntos fuertes podemos destacar su fácil ejecución remota, la capacidad de adaptarse a diseños escalables, fácil instalación, baja dificultad para empezar a trabajar con esta herramienta, ejecución en orden secuencial, soporta tanto modelos basados en pull como en push, la falta de un nodo master elimina posibles puntos de fallo, el hecho de una comunicación sin agente agiliza la comunicación frente al modelo master-agente, alta seguridad con *ssh*, etcétera.

Pese a todo, *Ansible* también tiene algunos contras como pueden ser el hecho de que se requeran permisos de *root* en el acceso *ssh* y que las máquinas donde se ejecute tengan instalado un intérprete de *Python* (en los nodos a provisionar no es necesario), la falta de una interfaz web suficientemente desarrollada y que no es una herramienta tan madura comparada con las anteriores.

No obstante, para el cometido de nuestro proyecto es la herramienta idónea, ya que, como se comentó, lo que nos interesa es que nos permita poder escalar fácilmente y que su curva de aprendizaje no sea demasiado grande.

5.3. Vagrant.

Vagrant [28] es una herramienta utilizada para gestionar y construir entornos de máquinas virtuales en un flujo de trabajo único. De una forma sencilla, *Vagrant* está focalizado en automatizar despliegues facilitando la definición de dichas máquinas y reduciendo tiempos para su puesta en marcha.

Un ejemplo sencillo de un *Vagrantfile*, utilizado para levantar máquinas, podría ser el siguiente:

```
Vagrant.configure("2") do |config|
    config.vm.box = "ubuntu/trusty64"
end
```

Mediante este fichero, lo que vamos a crear una máquina Ubuntu sencilla. Lo único que necesitamos hacer para levantarla será, en la misma carpeta donde se encuentre este fichero, ejecutar:

```
# vagrant up --provider <hipervisor>
```

Con este sencillo comando, indicando el hipervisor que tengamos instalado en nuestra máquina (*libvirt* o *virtualbox*, por ejemplo), podremos tener corriendo una máquina Ubuntu. Este ejemplo deja en evidencia la sencillez de esta herramienta, aunque podemos añadir mucha más funcionalidad a estos ficheros como se puede ver en el Anexo I. Manual de despliegue.

5.3.1. ¿Por qué utilizar Vagrant?

Vagrant es fácil de configurar, reproducible y los entornos que se desarrollan con esta herramienta son portables, por lo que una vez hecha la definición podremos desplegarla en cualquier equipo que disponga de dicha herramienta y de los hipervisores donde deseemos levantar las máquinas. Esto permite maximizar la productividad y flexibilidad a la hora de trabajar.

Las máquinas pueden ser definidas para hipervisores como *VirtualBox*, *KVM,* o entornos cloud como *VMWare*, *AWS*, etcétera. Además, esta herramienta permite adicionar y ejecutar los scripts de provisionamiento de *Ansible*, por lo que al lanzar las máquinas podemos llevar a cabo simultáneamente su configuración.

Además, cabe destacar que la definición de máquinas no es excesivamente compleja y que trabaja en los sistemas operativos más importantes como son Linux, MAC OSX o Windows.

5.4. Continuous Integration / Continuous Development.

En este apartado se va a tratar el concepto de CI/CD debido a su gran relevancia y su nexo con la tecnología de microservicios.

En la actualidad, ha ganado gran peso el hecho de ser capaces de integrar cualquier tipo de cambio de nuestro código de manera rápida y sencilla. A partir de esta premisa nacen estos nuevos paradigmas, orientados a facilitar dicha integración e, incluso, ser capaz de realizar un despliegue en caliente de dichos cambios sin que nuestro sistema se resienta.

En cuanto a las herramientas involucradas para llevar a cabo la integración continua, se encuentran las referentes al control de versiones, las encargadas de testear el código o las que organizan los flujos de trabajo para la prueba del código o el sistema.

5.4.1. Herramientas de control de versiones.

El control de versiones [29] es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que se pueden recuperar versiones específicas más adelante.

Entre las diferentes capacidades que debe abordar una herramienta de control de versiones están, por tanto, la de almacenar los elementos que debe gestionar, la posibilidad de realizar cambios sobre dichos elementos o el registro histórico de las acciones realizadas con cada elemento o conjunto de elementos.

Adicionalmente, dichas herramientas suelen ofrecer [30] una generación de informes con los cambios introducidos entre versiones, informes de estado, marcado con nombre identificativo de la versión de un conjunto de ficheros, etcétera.

Lo que estas herramientas nos van a permitir es trabajar colectivamente sobre el mismo archivo facilitando la integración de los cambios y permitiéndonos gestionar cualquier conflicto posible. En nuestro caso, podríamos dividir la integración continua tanto en la imagen de un contenedor (microservicio), la cual puede ser actualizada en un repositorio como “Docker Hub”, y el código que irá integrado dentro del microservicio posteriormente, el cual podrá ser gestionado en plataformas como “Gerrit”.

5.4.2. Herramientas de automatización de tareas.

Por otro lado, tenemos herramientas capaces de automatizar tareas que testeen nuestros cambios y confirmen su integridad, de tal forma que los cambios puedan ser verificados y añadidos en una actualización.

La herramienta más conocida en este ámbito es Jenkins [31], un software de integración continua de código abierto escrito en Java. Con Jenkins, se pueden acometer una serie de tareas llamadas "jobs", los cuáles testean el código en tiempo y forma indicado en el código de los mismos. Además, se acopla con facilidad con herramientas de control de versiones como Git, Subversion, CVS, etcétera.

De esta forma, podemos utilizar Jenkins como complemento en la integración de código, pudiendo comunicarse con otras herramientas como Gerrit e incluso programar su pipeline con otras herramientas como Zuul [32], capaz de programar los pasos a seguir en la integración, entrega y despliegue continuos.

Incluso, podríamos utilizar Jenkins para llevar a cabo el despliegue de infraestructuras de microservicios donde posteriormente testear el código a nivel de componentes e integrar directamente con producción el código o las imágenes, en caso de obtener una respuesta positiva de Jenkins, de manera automatizada.

Capítulo 6

6. Automatización de despliegue de un clúster de microservicios.

En este capítulo vamos a abordar uno de los puntos esenciales de este trabajo; el desarrollo en *Ansible* de la configuración de un clúster de microservicios gestionado por *Kubernetes*.

El código desarrollado está disponible en [33], e incluye tanto el desarrollo en *ansible* como todo lo referente a los despliegues de microservicios que se han realizado para este proyecto.

6.1. Desarrollo en *Ansible*.

Como explicamos con anterioridad en el capítulo 5 de esta memoria, *Ansible* es una herramienta que nos permite desarrollar mediante "recetas" una serie de scripts que sirvan para provisionar la configuración de las máquinas. Dentro de la carpeta "k8s-cluster/ansible" de [33], podemos encontrar el proyecto de *ansible* completo.

Básicamente, lo que buscamos con este trabajo es lo que escenifica el siguiente diagrama:

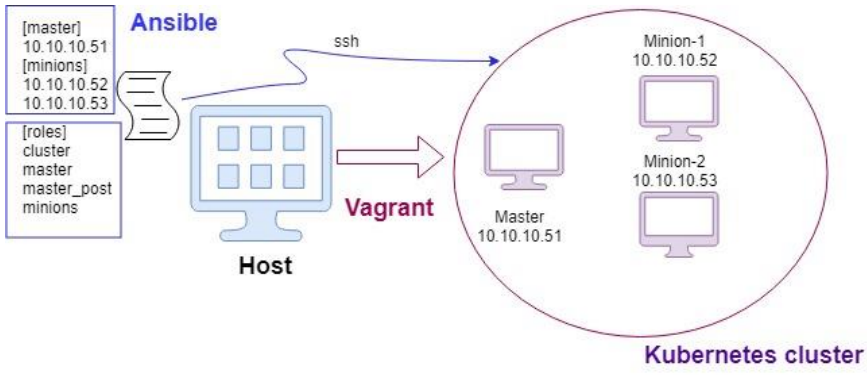

Figura 13. Diagrama del sistema con Ansible y Vagrant.

Como vemos, una por una parte tendremos el despliegue con *Vagrant* de las máquinas y por otro la provisión de su configuración con *Ansible*. En este capítulo nos vamos a centrar en el desarrollo de la primera parte, concerniente a los scripts de *Ansible*.

El fichero fundamental sobre el que gira toda la configuración es el llamado "kubernetes.yml", y su contenido es el siguiente:

```
---
        - hosts: cluster
          become: true
          roles:
```

```
      - cluster
  - hosts: master
    become: true
    roles:
      - master
  - hosts: minions
    become: true
    roles:
      - minion
  - hosts: master
    become: true
    roles:
      - master_post
```

Como se puede apreciar, este fichero no es ni excesivamente amplio ni excesivamente complejo. Si lo analizamos detenidamente, podemos ver tres etiquetas bien diferenciadas; la de *hosts*, la de *become* y la de *roles*.

La etiqueta de *hosts* hace referencia a los *hosts* donde se va a llevar a cabo el despliegue y la gestión de la configuración en cada caso.

Por su parte, la etiqueta de *become* nos permitirá ejercer dichas acciones como usuario *root* en caso de estar a *root*.

En cuanto a la etiqueta de *roles*, ésta es la que define lo que se va a llevar a cabo en cada paso. Como comentamos anteriormente, *ansible* ejecuta las órdenes de manera secuencial, por lo que primero se ejecutará el provisionamiento del bloque que está más arriba en este fichero. Además, es importante comentar que el valor de esta etiqueta se corresponde con las carpetas situadas dentro de la subcarpeta "roles", por lo que *ansible* buscará dentro de cada una un fichero llamado "*main.yml*", el cuál contiene las instrucciones de cada uno de los roles.

Al mismo nivel que el archivo descrito en estas líneas podemos encontrar otro llamado "hosts". En él se define la relación entre el nombre asignado al host en el "kubernetes.yml" y las ips a las que hace referencia. Decimos que son ips porque cada nombre puede hacer referencia tanto a una como a varias máquinas como vemos en nuestro fichero:

```
[master]
10.10.10.51
[minions]
10.10.10.52
10.10.10.53
[cluster]
10.10.10.51 cluster_host=master
```

```
    10.10.10.52 cluster_host=minion-1
    10.10.10.53 cluster_host=minion-2
```

Por lo tanto, si el host seleccionado es “cluster”, la configuración dentro de ese rol será la misma y afectará a las tres máquinas indicadas en este archivo.

Pasamos al contenido de la carpeta “group_vars”, la cual contiene el fichero “all”. Este fichero únicamente contendrá variables que utilizaremos dentro de nuestros scripts, de tal forma que flexibilicemos al máximo los posibles cambios que queramos acometer. A continuación, se muestra el contenido:

```
---
        # Variables listed here are applicable to all host groups
        #General Cluster Configuration
        master_hostname: master
        master_ip: 10.10.10.51
        minions_hostnames: minion-1, minion-2


        #Kubernetes API Configuration
        kube_api_address: 0.0.0.0
        kube_api_port: 8080
        kubelet_port: 10250
        etcd_port_servers: 2379
        kube_service_address: 10.254.0.0/16
        args_cluster_dns: 10.254.254.254
        args_cluster_domain: cluster.local


        #Flannel Configuration
        SubnetLen: 24
        SubnetMin: 10.254.50.0
        SubnetMax: 10.254.199.0
        Backend_Type: vxlan
        Backend_VNI : 1
```

Llegados a este punto, vamos a analizar los scripts de provisionamiento de la configuración, los cuáles componen la parte más importante de la herramienta. Como comentamos, en nuestro caso se ha dividido cada una de las taréas en roles, y en cada rol se llevarán a cabo unas acciones determinadas. Dentro de la carpeta “roles” vamos a echar un vistazo al rol de “cluster”, por lo que nos adentramos en dicha carpeta. Cada una de las carpetas de roles suele tener las subcarpetas de “tasks” (donde se encuentran los scripts con las sentencias a ejecutar) y “files”

(donde se añaden lso ficheros que queremos copiar a las máquinas), y a veces podemos encontrar también la carpeta de "templates" (que contiene plantillas que también podemos estar interesados en copiar).

De todas estas carpetas, la que contendrá los archivos con las instrucciones a ejecutarse será la carpeta de "tasks", la cual contiene el archivo "main.yaml" que será el que contenga dichas sentencias que deberán ejecutarse en las máquinas de cada rol. A modo de ejemplo, se presenta el archivo "main.yaml" del rol de clúster, en el que podemos ver algunas de las acciones que se llevarán en las máquinas desplegadas:

```
---
- name: Containers | Installing dockers
  action: yum pkg={{ item }} state=installed
  with_items:
    - docker
    - docker-logrotate

- name: Move docker-storage default file of docker to a backup
  shell: mv /etc/sysconfig/docker-storage /etc/sysconfig/docker-
storage.backup

- name: Insert docker-storage file with devicewrapper driver
  copy: src=docker-storage dest=/etc/sysconfig/docker-storage mode=0644

- name: Start docker service
  service: name=docker state=started enabled=yes

- name: Create docker group
  shell: groupadd docker
  ignore_errors: yes

- name: Add your vagrant user to the docker group
  shell: gpasswd -a vagrant docker

- name: Restart docker service
  service: name=docker state=restarted

- name: Cluster Orchestrator | Installing kubernetes
  action: yum pkg={{ item }} state=installed
  with_items:
    - kubernetes
    - etcd
    - flannel
…
```

Si nos paramos a analizar un poco su contenido, podemos ver que el lenguaje es descriptivo y que se insta a realizar una serie de tareas. Por ejemplo, en primer lugar, se realiza la instalación mediante un blucle de dos paquetes; *docker* y *docker-logrotate*, los cuales se utilizarán para levantar los contenedores en los *minions* y llevar un registro de logs de los mismos. Seguidamente, lo que se hace es mover un

fichero y, posteriormente, se copia un fichero, que tendremos ubicado en la carpeta "files" a la máquina, indicando el *path* donde queremos que esto se lleve a cabo.

También podemos ver la inicialización de servicios, como en la línea donde se indica que queremos que el servicio de *docker* comience en la máquina, o la copia de una plantilla a partir de la etiqueta "template".

Como se puede observar, es un lenguaje que permite adivinar lo que se está realizando a simple vista, por lo que facilita mucho el desarrollo y comprensión del mismo.

Capítulo 7

7. Prueba de concepto de despliegue de microservicios.

Este capítulo explica de forma detallada la prueba de concepto realizada para comprobar que nuestro clúster de microservicios funciona en perfectas condiciones y que sus características en un entorno a pequeña escala se cumplen.

Tras haber desgranado la herramienta de *Ansible* y los scripts referentes a la automatización del despliegue de un clúster de *Kubernetes* a partir de ella, vamos ahora a sumergirnos en su uso, desde los despliegues más sencillos hasta conseguir el despliegue de un microservicio completo, que en nuestro caso se trata de un servidor relacionado con *IoT* llamado *LoraServer* [34].

7.1. Conceptos previos.

Antes de empezar, vamos a presentar una serie de conceptos previos que faciliten la comprensión de este PoC. En primer lugar, vamos a presentar la herramienta *Docker*. Es importante entender bien su funcionamiento antes de trabajar con *Kubernetes*, ya que el orquestador utiliza por debajo dicha herramienta para llevar a cabo el despliegue de sus contenedores enforma de microservicios.

Docker es una tecnología *software* para crear contenedores de aplicaciones. Esta herramienta puede instalarse tanto en sistemas operativos Linux como en MAC y Windows, y ofrece un repositorio [35] para obtener imágenes prediseñadas, en el cuál podemos incluso crear una cuenta y añadir las nuestras propias.

Se basa en ficheros que describen nuestros contenedores, llamados *Dockerfile*. En ellos, definimos la imagen base de la que vamos a partir y podemos añadir una serie de sentencias para copiar ficheros, ejecutar scripts o comandos, exponer puertos y mucho más. Un ejemplo de fichero es el siguiente:

```
FROM loraserver/loraserver:1
MAINTAINER Manuel Sánchez López "manusl.teleco@gmail.com"

COPY ./configuration/loraserver /etc/loraserver
WORKDIR /root/

ENTRYPOINT ["./loraserver"]
```

En primer lugar, con la etiqueta *FROM*, indicamos cual es la imagen que queremos utilizar como base. Esta imagen debe estar, como comentamos anteriormente, en el repositorio de *Docker Hub*, a no ser que indiquemos otro repositorio y nos registremos

en él previamente o que tengamos dicha imagen en local. La etiqueta *MANTAINER* sirve para especificar quién es el responsable de esta imagen.

Por otro lado, tenemos etiquetas encargadas de llevar acabo acciones como copiar archivos de local a nuestro contenedor (*COPY*), indicar el directorio de trabajo del contenedor (*WORKDIR*) o insertar el punto de entrada de nuestra aplicación dentro del contenedor (*ENTRYPOINT*). También existen otras etiquetas, como la que nos permite ejecutar un comando (*CMD*), o para exponer puertos (*EXPOSE*).

En este caso, el punto de entrada de nuestro fichero de *Docker* será un script llamado "loraserver", por lo que la vida de nuestro contenedor dependerá de la ejecución de ese script; si ese script finaliza, el contenedor se apagará, ya que habrá terminado de realizar su trabajo.

Si queremos registrarnos en otro repositorio de imágenes distinto al que nos ofrece *Docker*, podemos ejecutar el siguiente comando y añadir nuestro usuario y contraseña cuando se nos solicite:

```
# docker login <nombre_repositorio>
```

Una vez explicados los conceptos básicos de un fichero de *Docker*, vamos a ver cómo desplegar un contenedor a partir de uno de ellos. El comando para crear un contenedor es:

```
# docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

De esta forma, crearemos una imagen en local con el nombre que le indiquemos con la opción "--name".

El comando para ejecutar y levantar un contenedor en la máquina donde tengamos instalada la herramienta será tal que así:

```
# docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Para tener una visión global de las opciones que permite esta herramienta se puede consultar la referencia [36].

Un ejemplo podría ser el siguiente:

```
# docker run --name mongodb_slave -d mongodb_slave:latest
```

Como vemos, indicamos el nombre del contenedor con la etiqueta "--name" y con la opción "-d" arrancaremos el contenedor en *background*, y por último se indica la imagen y el *tag* con la versión de la misma que queremos arrancar.

Si queremos borrar un contenedor podemos ejecutar:

```
# docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

Mientras que si lo que queremos es borrar una imagen de nuestro local podemos ejecutar:

```
# docker rmi [OPTIONS] IMAGE [IMAGE...]
```

Para más información a cerca de los distintos comandos que ofrece *Docker* se puede consultar la referencia [37].

Además, para visualizar los contenedores que se encuentran desplegados, basta con ejecutar:

```
# docker ps
```

Figura 14. Ejemplo de despliegue y consulta de un contenedor.

Si lo que deseamos es ver las imágenes descargadas en nuestro entorno local:

```
# docker images
```

Figura 15. Consulta de imágenes descargadas en local.

7.2. Despliegue de microservicios orientado a IoT.

Una vez claros los conceptos básicos de *Docker*, vamos a pasar a explicar el despliegue de de un microservicio concreto. En este caso se trata del servidor de *Internet of Things LoraServer*, el cuál hemos adaptado para llevar a cabo un despliegue del mismo en nuestra arquitectura.

Básicamente, el servidor *LoraServer* se compone de los siguientes servicios:

- *LoRa App Server* [38]: es una aplicación del proyecto de *LoRa Server* responsable de la parte de inventario de los distintos dispositivos que forman la infraestructura LoRaWAN. Ofrece una interfaz web donde los usuarios, organizaciones, aplicaciones y dispositivos son gestionados. Además, ofrece una API RESTful y gRPC. Podemos ver una captura de dicha interfaz en la Figura 13.

- *LoRa Server* [39]: es la responsable de los componentes del seridor de red. Gestiona las peticiones que recibe de los *gateways*, gestión de la capa mac y encolado de las transmisiones de datos en sentido descendente.

- *LoRa Gateway Bridge* [40]: es un servicio que abstrae el reenvío de paquetes del protocolo UDP y lo transforma en JSON a través de *MQTT*. Este *software* tiene como propósito adecuar la comunicación con todo el entramado del servidor. Con este servicio, se obtiene visibilidad del servidor, un enrutado entre las diferentes máquinas y una capa de seguridad, ya que *MQTT* va sobre *TLS*, por lo que la comunicación entre el *Gateway* y el servidor es segura.

- *Postgresql*: adicionalmente a estos componentes principales, se hace necesario un nivel de persistencia para la información que se registre. Por ello, es necesaria la implementación de una base de datos *postgresql* a la que se conectará nuestro servidor para persistir dicha información.

- *Redis*: es una base de datos en memoria utilizada para el almacenamiento de datos de tránsito relativo.

- *MQTT bróker*: es un protocolo de publicador/suscriptor, que permite a los usuarios publicar información bajo unas etiquetas a las que otros podrán suscribirse si desean recibirla. Su implementación más popular es a través del servicio de *Mosquitto*.

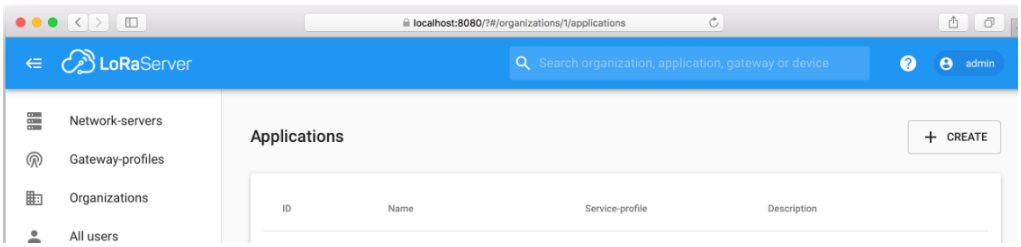

Figura 16. Captura de la interfaz web de Lora App Server.

Cada uno de estos componentes tendrá su implementación en su propio contenedor, como veremos en los siguientes apartados.

7.2.1. Despliegue con *Docker*.

Comenzamos con un despliegue en *Docker*, sin la interveción de ningún tipo de orquestador que lo gestione por arriba. Para ello, se han desarrollado una serie de *Dockerfiles* correspondientes a los diferentes servicios por separado que se pueden consultar en la referencia [41].

Vamos a proceder a levantar cada uno de ellos por separado, configurando su conectividad de red y comprobando su correcto funcionamiento.

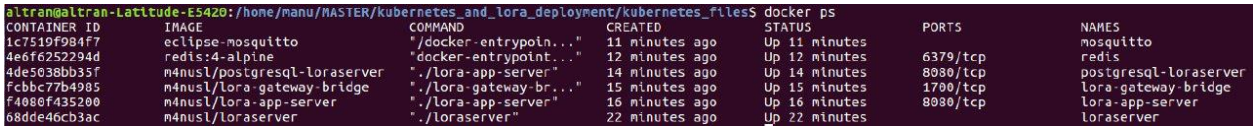

Figura 17. Despliegue de todos los contenedores que componen el LoRa Server.

Como vemos, están todos los contenedores levantados y con los puertos que utilizan abiertos. Vamos ahora a conectarnos al contenedor de lora-app-server para ver su estado. Para ello vamos a atacharnos al mismo:

```
# docker attach lora-app-server
```

Con ello, vamos a entrar en el contenedor, en el punto que esté ejecutando. Como en este caso se está ejecutando el script "lora-app-server", lo que veremos serán los *logs* que se desprenden del mismo tal y como se muestra a continuación.

Figura 18. Logs del script lora-app-server ejecutado dentro del contenedor.

Como vemos, estos *logs* muestran cómo nuestro contenedor se conecta con otros contenedores que componen este microservicio, estableciendo la comunicación y sincronizándose.

En la siguiente imagen mostramos la interfaz *web* que nos ofrece este contenedor, cuya dirección IP es la 172.17.0.3, y a la que podemos acceder mediante *https* por el puerto 8080:

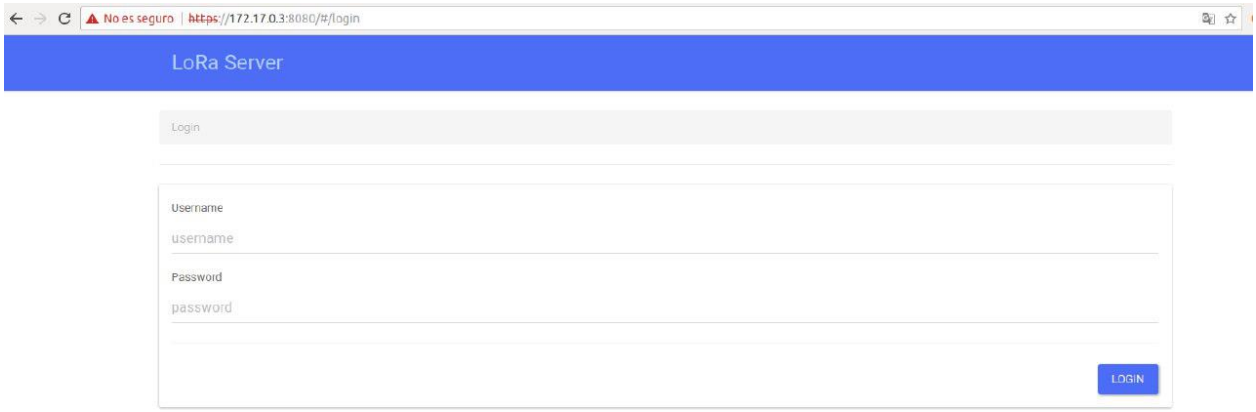

Figura 19. Interfaz web del contenedore lora-app-server.

Además, podremos registrarnos con el usuario "admin" y contraseña "admin", comprobando así que la interacción es correcta:

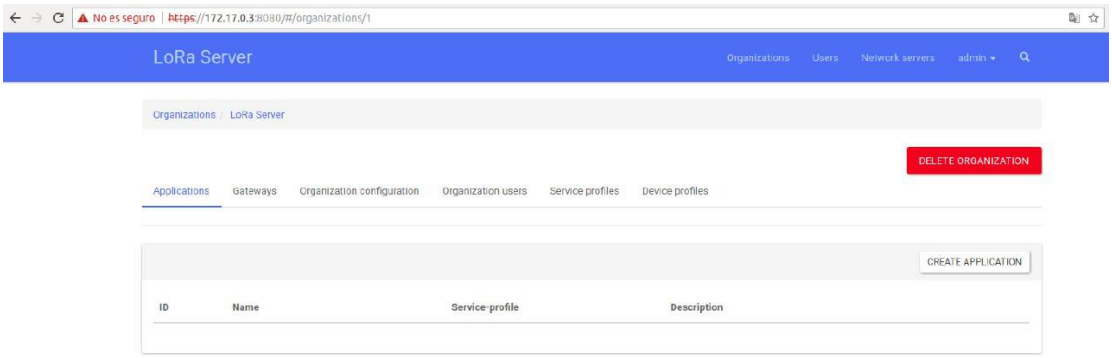

Figura 20. Interfaz de lora-app-server una vez registrados con usuario admin.

El principal problema de trabajar únicamente con *Docker* es que, para poder llevar a cabo la comunicación entre contenedores, tendremos que averiguar o asignar de manera manual direcciones IP, para luego indicarlas en los distintos archivos de configuración de cada uno de los contenedores, lo cuál no es efectivo. Por otro lado, si se cae un contenedor o hay algún tipo de problema relativo a la carga de trabajo, conectividad, etcétera, el sistema no será capaz de reaccionar.

A continuación, se muestra parte del fichero de configuración del contenedor que ofrece la interfaz web, que pone en relieve el problema comentado anteriormente.

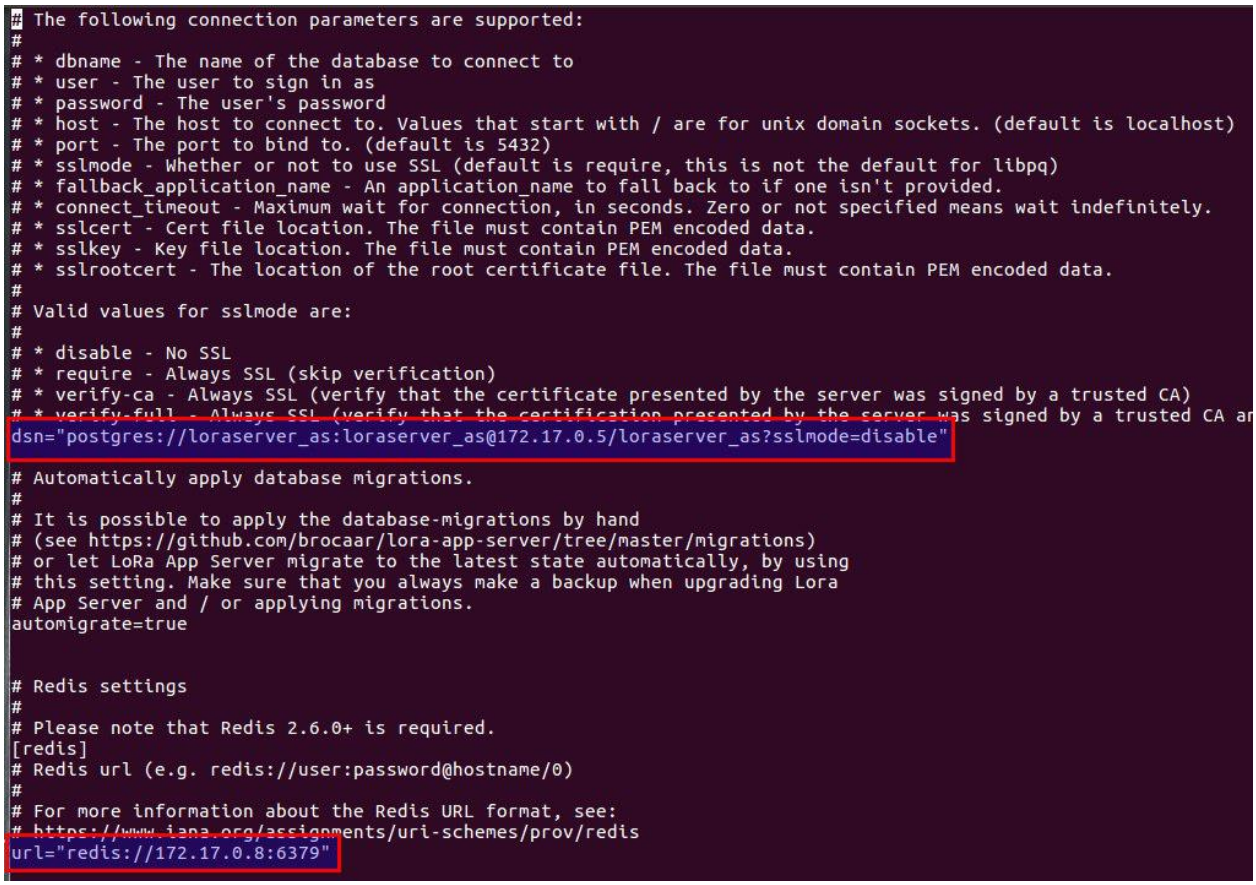

Figura 21. Parte de fichero de configuración del contenedor lora-app-server.

Es por ello que se hce necesario la existencia de una herramienta que sea capaz de añadir una capa de abstracción y manejar todo este tipo de contingentes.

7.2.2. Despliegue en un clúster de *Kubernetes*.

Una vez hemos comprobado que el despliegue de los contenedores es correcto y que el sistema se ejecuta correctamente, pasamos a adaptarlo a un despliegue para un clúster de *Kubernetes*, el cuál solventará el tema del direccionamiento IP y añadirá propiedades importantes al sistema, como son la monitorización, respuesta frente a posibles errores, etcétera. Para ello, hemos definido el siguiente *deployment*:

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: lora-full-deployment
  labels:
    app: lora
spec:
  replicas: 1
  minReadySeconds: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  progressDeadlineSeconds: 60
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: lora
    spec:
      containers:
      - image: m4nusl/lora-app-server
        name: lora-app-server
        ports:
        - containerPort: 8080
        - containerPort: 8000
        resources: {}
      - image: m4nusl/postgresql-loraserver
        name: postgresql
        ports:
        - containerPort: 5432
        resources: {}
      - image: m4nusl/loraserver
        name: loraserver
        ports:
        - containerPort: 8000
        - containerPort: 8001
        resources: {}
      - image: m4nusl/lora-gateway-bridge
        name: lora-gateway-bridge
        ports:
        - containerPort: 1700
```

```
        resources: {}
      - image: eclipse-mosquitto
        name: mosquitto
        ports:
        - containerPort: 1883
        resources: {}
      - image: redis:4-alpine
        name: redis
        ports:
        - containerPort: 6379
        resources: {}
      restartPolicy: Always
status: {}
```

Como se puede apreciar, en ella hemos incluido un contenedor por cada uno de los componentes previamente descritos, además de exponer, según cada caso, el puerto correspondiente para ser accesibles. Se puede apreciar que, salvo los contenedores de "eclipse-mosquitto" y "redis", el resto han sido modificados y almacenados en un repositorio personal, el cual es público.

Hemos desarrollado un *Deployment* y, para poder realizar una las pruebas pertinentes, en este caso solo se va a levantar un servidor; en caso de querer levantar una réplica deberíamos añadir otro valor en la etiqueta "replicas".

Una vez tenemos la estructura de nuestro microservicio, necesitaremos definir una forma para acceder al servicio desde fuera de nuestro clúster. Para ello, será necesario desarrollar una plantilla de un *Servicio*.

Gracias a éste, podremos mapear los puertos y ofrecer nuestra aplicación accediendo directamente a las IPs de los *minions*.

Esta plantilla se muestra a continuación:

```
apiVersion: v1
```

```
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: lora-service
  name: lora-full-services
spec:
  type: NodePort
  ports:
    - port: 8080
      name: app-port
    - port: 1700
      protocol: UDP
      name: gateway-port
    - port: 1883
      name: mosquitto-port
    - port: 6379
      name: redis-port
  selector:
    app: lora
status:
  loadBalancer: {}
```

Como vemos, el tipo de mapeo de puertos que vamos a utilizar es el que nos permite la ifnraestructura que estamos utilizando, *NodePort*, aunque para más información a cerca de este tema se puede consultar la siguiente referencia [42]. En concreto, este tipo de mapeo nos ofrecerá un puerto externo por cada uno de los puertos que estamos indicando en esta plantilla, de tal forma que, aunque algún contenedor se redespliegue o, aunque tengamos varias replicas de un contenedor, *Kubernetes* se encargará de redirigir automáticamente las peticiones al contenedor que esté activo de forma balanceada gracias a la asociación creada en este servicio.

Pasamos a mostrar el *deployment* activo en nuestro clúster:

```
[root@master ~]# kubectl get deployments
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
lora-full-deployment   1         1         1            1
```

Figura 22. Muestra del deployment realizado en Kubernetes.

Podemos entrar más en detalle el mismo ejecutando el comando “describe”, como se muestra en la siguiente figura. Esto nos da detalles más exhaustivos de lo que estamos desplegando y de su estado actual.

```
altran@altran-Latitude-E5420:~$ ssh root@10.10.10.51
Last login: Fri Sep  7 07:20:33 2018 from 10.10.10.1
[root@master ~]# ip^C
[root@master ~]# iptables-save | grep FORWARD
[root@master ~]# kubectl describe deployment lora-full-deployment
Name:                   lora-full-deployment
Namespace:              default
CreationTimestamp:      Tue, 24 Jul 2018 07:38:35 +0000
Labels:                 app=lora
Selector:               app=lora
Replicas:               1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        5
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet:  lora-full-deployment-1701056508 (1/1 replicas created)
No events.
[root@master ~]#
```

Figura 23. Descripción detallada del deployment de LoRa Server.

Por otra parte, vamos a mostrar también el servicio.

```
[root@master ~]# kubectl get service lora-full-services
NAME                 CLUSTER-IP      EXTERNAL-IP   PORT(S)
lora-full-services   10.254.69.181   <nodes>       8080:32250/TCP,1700:31002/UDP,1883:31731/TCP
```

Figura 24. Servicio desplegado en el clúster de Kubernetes.

Como se aprecia en la Figura 22, se ha realizado un mapeo de puertos tal y como se había solicitado en la plantilla que describía los servicios, por lo que si queremos acceder, por ejemplo, a la interfaz web del contenedor de aplicación, tendremos que apuntar al puerto 32250.

Vamos a realizar la prueba y a confirmar que nuestro sistema se ha desplegado correctamente:

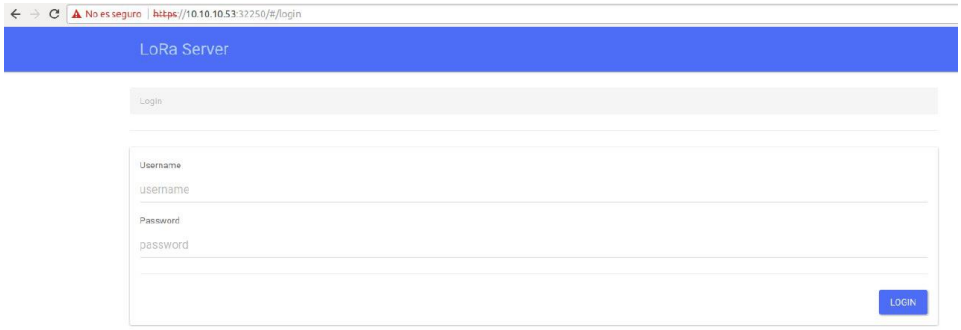

Figura 25. Captura de conexión a interfaz web al puerto mapeado por Kubernetes.

El siguiente paso será provisionar un equipo para ver que todo está correcto y que, al conectar un dispositivo, este envía los datos correctamente al servidor.

Una vez añadidos tanto el servidor de red, como el *Gateway*, el perfil de usuario, el dispositivo, etcétera, podremos ver cómo se han registrado los sensores a neustro servidor, por lo que, gracias a la asociación con la MAC de los mismos, una vez se detecten datos de cada uno de ellos se registrarán en nuestro servidor.

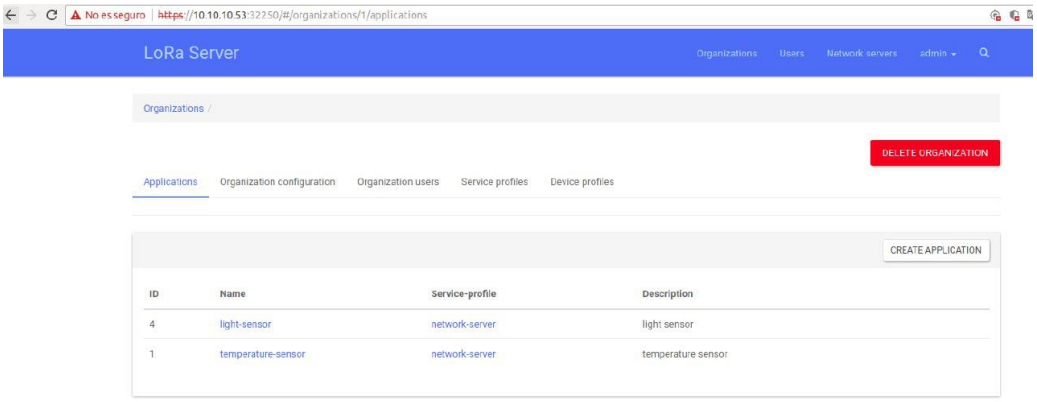

Figura 26. Registro en el servidor de los sensores.

A continuación, vamos a mostrar algunos de los pasos realizados para conseguir asociar una de estas motas a nuestro sistema y, así, comprobar su operatividad. En la siguiente figura se aprecia el registro del *gateway* de estos dispositivos.

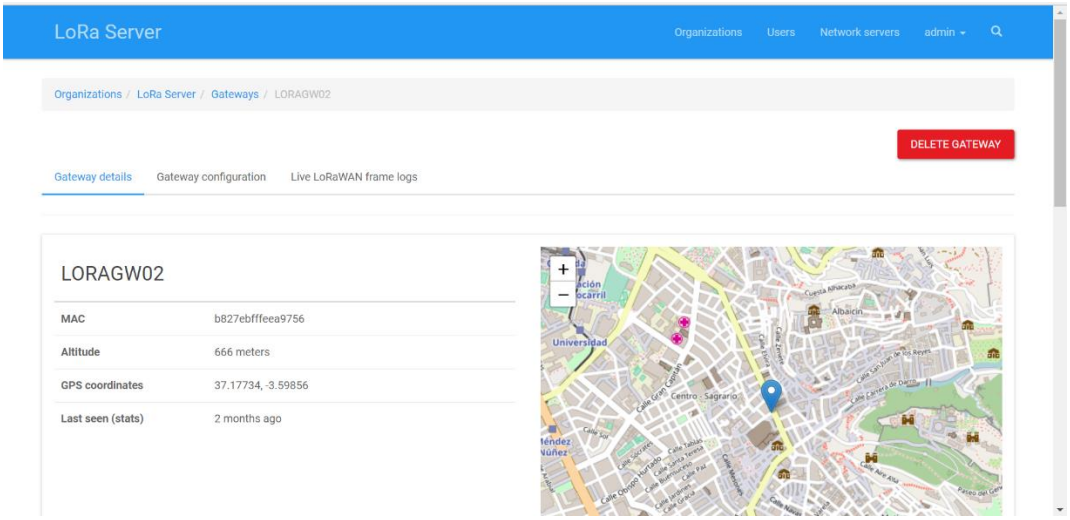

Figura 27. Registro de gateway en servidor LoRa.

También podemos ver la asociación de un dispositivo concreto:

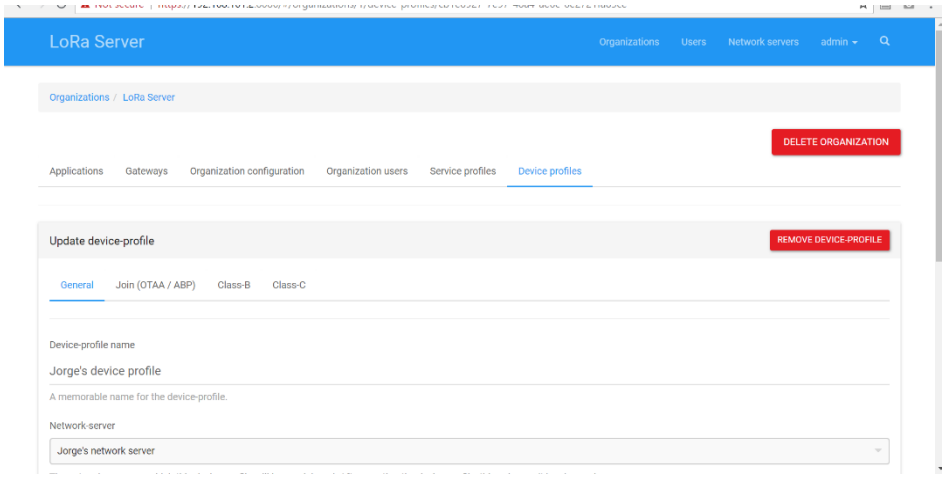

Figura 28. Registro de un dispositivo en el servidor LoRa.

Una vez llevado a cabo los registros de *Gateway*, usuario, perfiles, organización y aplicación, en caso de una correcta configuración, podremos comenzar a recibir datos de la mota. Para ello, conectamos el *gateway* a nuestro PC donde estaba alojado el clúster de *Kubernetes*, y configuramos el mismo para que apuntase a nuestro microservicio de servidor LoRa alojado dentro (configuración de IP y puerto). Una vez hecho esto, gracias a la asociación de la MAC de este dispositivo, comenzamos a recibir los datos que mostramos a continuación:

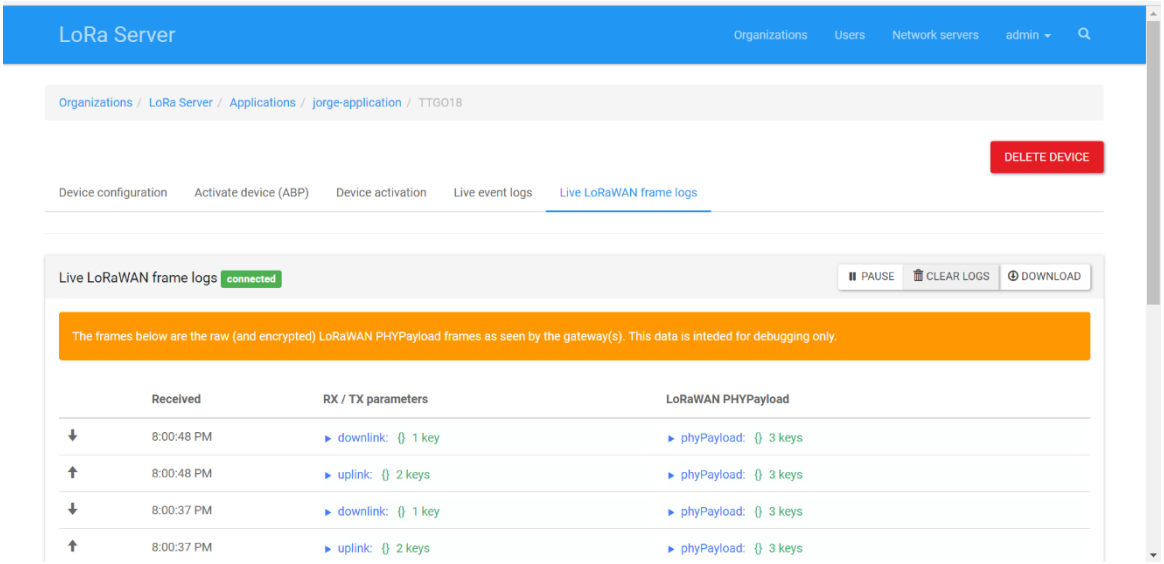

Figura 29. Recepción de datos en el servidor loRa.

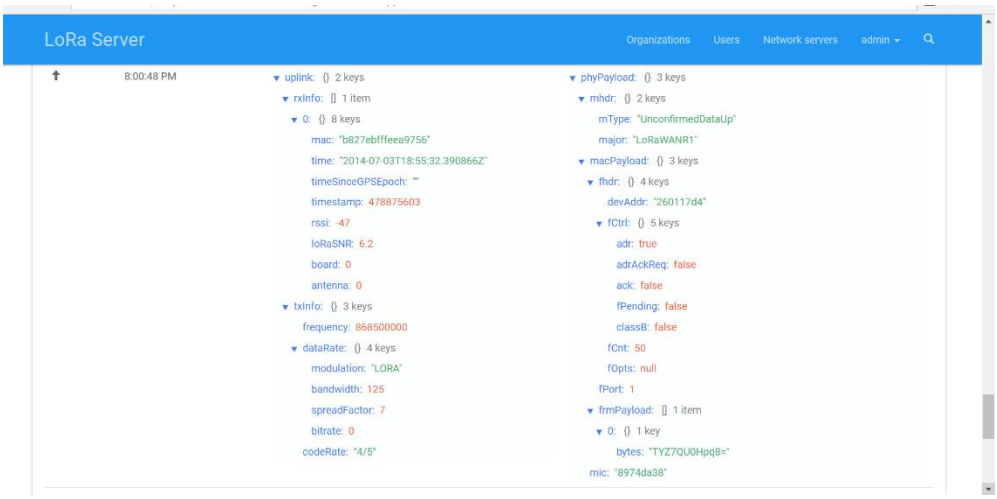

Figura 30. Análisis de datos recibidos en el servidor LoRa.

Por lo tanto, podemos confirmar que nuestro sistema está activo y esperando solicitudes para trabajar.

Capítulo 8

8. Resultados obtenidos.

En este capítulo vamos a realizar algunas pruebas de rendimiento de nuestro sistema. En primer lugar, vamos a explicar la prueba de concepto que vamos a llevar a cabo en nuestro clúster de microservicios, para poner en situación al lector.

Una vez hecho esto, pondremos a prueba nuestro sistema, simulando la caída de un nodo y observando como el sistema, gracias al orquestador *Kubernetes*, se recupera automáticamente y vuelve a levantar nuestro servidor en el otro nodo.

Por último, ofreceremos una reflexión a cerca de los logros y aportaciones que se han conseguido con el presente proyecto.

8.1. Evaluación de resultados.

Nuestro sistema consiste en la arquitectura presentada en la Figura 26.

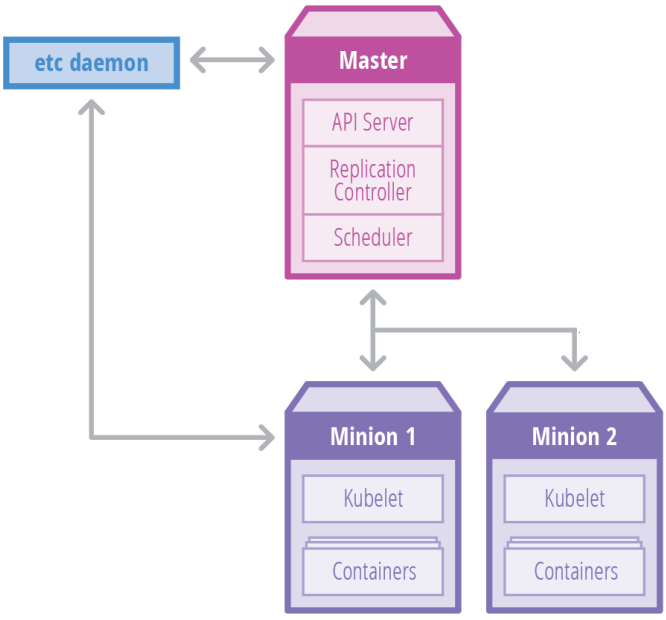

Figura 31. Arquitectura del clúster de Kubernetes desplegado.

En el, se ha desplegado el microservicio con el servidor de *Internet of Things LoRa Server*, y en este primer caso de uso se ha desplegado sin replicas.

Lo que vamos a acometer a continuación es la simulación de la caída de un nodo *minion*, en este caso, el nodo en el que tenemos el único despliegue.

En este caso, como podemos ver aquí, hemos desplegado nuestro *Pod* utilizando un tipo de despliegue *Deployment* en el *minion-2*.

```
[root@master ~]# kubectl create -f full-lora-deployment.yaml
deployment "lora-full-deployment" created
[root@master ~]# kubectl get deployments
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
lora-full-deployment   1         1         1            0           9s
```

Figura 32. Creación de deployment de nuestro microservicio de LoRa Server.

Como vemos en la Figura 27, el *deployment* se ha lanzado, pero aún no está disponible. Por otro lado, podemos ver la descripción del mismo:

```
[root@master ~]# kubectl describe deployment lora-full-deployment
Name:                   lora-full-deployment
Namespace:              default
CreationTimestamp:      Fri, 07 Sep 2018 11:08:39 +0000
Labels:                 app=lora
Selector:               app=lora
Replicas:               1 updated | 1 total | 0 available | 1 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        5
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet:  lora-full-deployment-1701056508 (1/1 replicas created)
Events:
  FirstSeen     LastSeen        Count   From                            SubObjectPath   Type          Reason              Message
  ---------     --------        -----   ----                            -------------   --------      ------              -------
  44s           44s             1       {deployment-controller }                        Normal        ScalingReplicaSet   Scaled up replica set lora-full-deploymen
t-1701056508 to 1
```

Figura 33. Descripción del deployment de LoRa Server.

En ella, se aprecia que nuestro despliegue aún no está activo, pues se encuentra levantando los distintos contenedores.

Una vez se ha llevado a cabo el despliegue de todos los contenedores, el estado se actualizará y nos indicará que está listo para su uso.

```
[root@master ~]# kubectl get deployments
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
lora-full-deployment   1         1         1            1           1m
```

Figura 34. Captura de estado habilitado del despliegue.

También podemos ver que el *pod* tiene los seis contenedores corriendo:

```
[root@master ~]# kubectl get pod lora-full-deployment-1701056508-zt9jh
NAME                                    READY     STATUS    RESTARTS   AGE
lora-full-deployment-1701056508-zt9jh   6/6       Running   0          2m
```

Figura 35. Captura de estado arrancado del pod.

Para una información más extensa de nuestro despliegue, podemos ejecutar el comando *describe* y recibiremos algo parecido a lo mostrado en la Figura 31.

```
[root@master ~]# kubectl describe pod lora-full-deployment-1701056508-zt9jh
Name:           lora-full-deployment-1701056508-zt9jh
Namespace:      default
Node:           minion-2/10.10.10.53
Start Time:     Fri, 07 Sep 2018 11:08:39 +0000
Labels:         app=lora
                pod-template-hash=1701056508
Status:         Running
IP:             10.254.57.2
Controllers:    ReplicaSet/lora-full-deployment-1701056508
Containers:
  lora-app-server:
    Container ID:               docker://465b5613d560c55737d0ca6c764ecd7bfeaa9f8073
    Image:                      m4nusl/lora-app-server
    Image ID:                   docker-pullable://docker.io/m4nusl/lora-app-server@
    Ports:                      8080/TCP, 8000/TCP
    State:                      Running
      Started:                  Fri, 07 Sep 2018 11:08:47 +0000
    Ready:                      True
    Restart Count:              0
    Volume Mounts:              <none>
    Environment Variables:      <none>
  postgresql:
    Container ID:               docker://bec02dd6b6f7b768f3b0bb0e82d780156f3a36c068
    Image:                      m4nusl/postgresql-loraserver
    Image ID:                   docker-pullable://docker.io/m4nusl/postgresql-loras
    Port:                       5432/TCP
    State:                      Running
      Started:                  Fri, 07 Sep 2018 11:08:55 +0000
    Ready:                      True
    Restart Count:              0
    Volume Mounts:              <none>
    Environment Variables:      <none>
  loraserver:
```

Figura 36. Descripción detallada del despliegue de LoRa Server.

El siguiente paso será simular una caída del nodo que alberga a nuestros contenedores. En la Figura 32 podemos ver la IP del nodo donde se han desplegado, la 10.10.10.53, perteneciente al *minion-2*. Por lo tanto, podemos proceder a apagar dicha máquina y ver lo que acontece.

Para ello, desde la máquina nativa, ejecutamos:

```
# virsh shutdown minion-2
```

El estado del despliegue pasa a ser el siguiente:

```
[root@master ~]# kubectl get deployments
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
lora-full-deployment   1         1         1            0           50m
```

Figura 37. Estado del despliegue tras caída del minion-2.

Donde se puede apreciar que el *lora-full-deployment* se encuentra en un estado inoperativo.

Para corroborar que *Kubernetes* detectó la caída del nodo, mostramos la Figura 33. En ella se muestra como el estado del *minion-2* es *NotReady*, es decir, no está preparado para levantar ningún despliegue en ella.

```
[root@master ~]# kubectl get nodes
NAME       STATUS     AGE
minion-1   Ready      158d
minion-2   NotReady   158d
```

Figura 38. Estado de los nodos esclavos tras apagar la máquina del minion-2.

Automáticamente, *Kubernetes* pasará a redesplegar el *pod* con los contenedores de dicho despliegue:

```
[root@master ~]# kubectl describe pod lora-full-deployment-1701056508-l72l9
Name:           lora-full-deployment-1701056508-l72l9
Namespace:      default
Node:           minion-1/10.10.10.52
Start Time:     Fri, 07 Sep 2018 11:59:49 +0000
Labels:         app=lora
                pod-template-hash=1701056508
Status:         Running
IP:             10.254.53.5
Controllers:    ReplicaSet/lora-full-deployment-1701056508
Containers:
  lora-app-server:
    Container ID:               docker://a80af9c16ad3b1369e1442736f16641d209bcf5d70196e76103072281c9c7
    Image:                      m4nusl/lora-app-server
    Image ID:                   docker-pullable://docker.io/m4nusl/lora-app-server@sha256:255abc2acaaa
    Ports:                      8080/TCP, 8000/TCP
    State:                      Running
      Started:                  Fri, 07 Sep 2018 12:00:05 +0000
    Ready:                      True
    Restart Count:              0
    Volume Mounts:              <none>
    Environment Variables:      <none>
  postgresql:
    Container ID:               docker://49cdf74c738cb976ce7db51fe86aaf21f6bf9dee826cbc0a0446d86390d0b
    Image:                      m4nusl/postgresql-loraserver
    Image ID:                   docker-pullable://docker.io/m4nusl/postgresql-loraserver@sha256:1418eb
    Port:                       5432/TCP
    State:                      Running
      Started:                  Fri, 07 Sep 2018 12:00:11 +0000
    Ready:                      True
    Restart Count:              0
```

Figura 39. Descripción del nuevo Pod desplegado en el minion-1.

Como se aprecia en esta imagen, el nuevo *pod* referente a nuestro *deployment* se ha lanzado en el *minion-1*, por lo que en cuestión de segundos volverá a estar funcionando. Como podemos comprobar en la siguiente figura, ahora la ip ha

cambiado, pero el puerto se mantiene gracias al servicio que vinculamos al *deployment*, y nuestra aplicación vuelve a funcionar correctamente en cuestión de segundos:

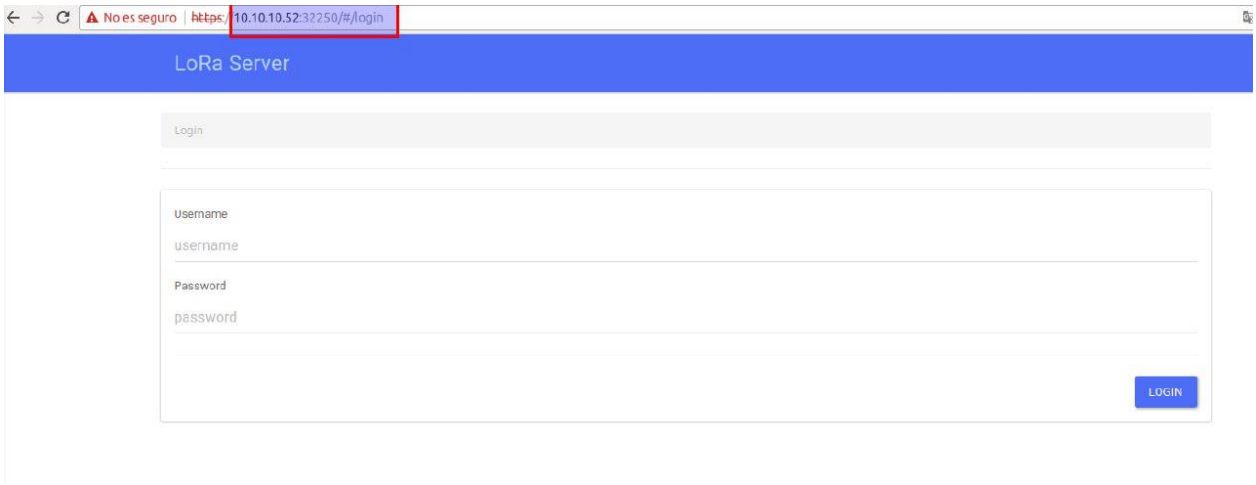

Figura 40. Captura de la interfaz web ahora lanzada desde el minion-1.

8.2. Logros y aportaciones.

El principal objetivo al comienzo de este proyecto consistía en la familiarización con las arquitecturas basadas en microservicios, más concretamente en Kubernetes, un orquestador capaz de controlar y gestionar un clúster de microservicios de forma automatizada. De este modo, la aportación fundamental sería el desarrollo de un despliegue automatizado de una arquitectura basada en microservicios haciendo uso de dicho orquestador, de tal forma que podamos obtener en un tiempo reducido un sistema completo para el despliegue de los mismos.

De forma intrínseca, se pretendió también realizar una prueba de concepto en la que se consiguiesen desplegar una serie de microservicios para ver los beneficios de este tipo de arquitecturas en un caso de uso orientado a las tecnologías de 5G. Más concretamente, la idea principal fue desplegar un microservicio basado en un servidor capaz de recoger información sobre motas orientadas a IoT, con información relevante para ser tratada posteriormente.

A partir de este punto, se podrían exponer diferentes escenarios para comprobar las capacidades de robustez, escalado y recuperación frente a fallos de nuestros sistemas en este tipo de arquitectura.

Capítulo 9

9. Conclusiones y vías futuras

En este capítulo se recogerán las conclusiones una vez finalizado el presente Trabajo de Fin de Máster en todo lo referente al mismo, así como una serie de vías futuras posibles.

Por lo tanto, se expondrán algunas cuestiones relevantes surgidas a raíz de la elaboración, así como las contribuciones aportadas al campo en que hemos trabajado. También se comentarán las posibilidades que surgen para el desarrollo de nuevas líneas futuras a partir del trabajo realizado en este proyecto.

Por último, se ofrecerá una valoración personal del autor de este proyecto para dar por finalizado el mismo.

9.1. Conclusiones.

En el presente Trabajo de Fin de Máster se ha diseñado un despliegue automatizado de un clúster de microservicios basado en el orquestador de *Kubernetes*, haciendo uso de otras herramientas como *Vagrant* o *Ansible* que se han ido acoplando en diferentes partes del proyecto.De esta forma, se ha conseguido automatizar la creación de dicho clúster, permitiendo además escalarlo también de manera automatizada, y se han llevado a cabo pruebas de concepto desplegando microservicios en dicha plataforma para corroborar su correcto funcionamiento y el rendimiento óptimo del sistema desarrollado.

Las principales contribuciones que nuestro proyecto ofrece son:

- Ofrecer una gran recopilación de información en lo referente a este tipo de arquitecturas basadas en microservicios. Con el trabajo realizado se ofrece una idea de las posibilidades que este nuevo paradigma ofrece tanto a las empresas como a los usuarios en cuanto a diferentes aspectos como son el desarrollo de aplicaciones, la integración continua del código, la actualización en caliente del código y, por tanto, de las aplicaciones, etcétera.
- Se ha experimentado con *Docker* y *Kubernetes* de manera que se aporta una gran fuente de información para comenzar a trabajar con estas herramientas. Esto es una ventaja para los desarrolladores que deseen introducirse en estas nuevas tecnologías, ya que la investigación en este campo aún es incipiente.
- El código desarrollado en *ansible* para la automatización de la configuración de máquinas también es una contribución importante ya que, como se ha comentado varias veces durante la redacción de esta memoria, el código es exportable y reutilizable. Además, la información con respecto a esta herramienta nos permite introducirnos con facilidad a este tipo de desarrollos, ofreciendo una visión global de su funcionamiento para usuarios con poca experiencia en este ámbito.

- Otra aportación importante son la adaptación de una aplicación alojada completamente en una máquina virtual a una arquitectura de microservicios, dividiendo en contenedores cada uno de los servicios que la componen.
- Por último, se ha conseguido automatizar la recuperación del sistema cuando se cae uno de los servidores, lo que permite una recuperación ante fallos bastante eficiente y una fiabilidad del sistema notable.

9.2. Valoración del alcance de los objetivos.

En este punto vamos a proceder a valorar el alcance de los objetivos propuestos una vez se ha completado la realización del presente proyecto.

Finalizado éste, podemos afirmar que se han alcanzado todos los objetivos que se plantearon en el apartado 3.1 por completo. Hemos conseguido automatizar el despliegue y configuración de un clúster de *Kubernetes* posibilitando su escalado en caso de ser necesario y facilitando cualquier modificación en el mismo gracias a las herramientas utilizadas. Esto permite añadir nuevas funcionalidades o configuraciones sin impactar negativamente en el proyecto actual.

Adicionalmente, se han conseguido adaptar una serie de contenedores en *Docker* relacionados con un servidor de *IoT* de cara a su despliegue en *Kubernetes* en forma de microservicios. Para conseguir dicho despliegue se desarrolló la plantilla correspondiente, ofreciendo dicho servicio desde el cúster con acceso al exterior, de tal forma que se consiguió establecer la comunicación entre las motas y servidor en la prueba de concepto.

Por lo tanto, hemos logrado crear un producto independiente, flexible y adaptable que incluye el clúster automatizado para despliegue de microservicios y un microservicio para corroborar su funcionamiento y comprobar las virtudes de esta nueva arquitectura *software*.

9.3. Vías futuras.

Pese a que los objetivos propuestos para este proyecto se han conseguido, a partir del mismo se pueden realizar algunas mejoras que podrían ser de gran utilidad para la consecución de un producto más completo y competitivo en el mercado.

Una de estas mejoras adicionales podría ser desplegar el clúster en máquinas dentro de dos clouds diferentes situados en diferentes data centers y comprobar que el sistema funciona correctamente sin necesidad de realizar ningún tipo de modificación. Esto se puede conseguir modificando únicamente las direcciones ip de los hosts configurables en ansible y obviando la parte de Vagrant. De esta forma, podría ser posible ver la respuesta del clúster cuando se despliega dentro de un cloud y realizar pruebas de carga, robustez, alta disponibilidad, etcétera. Además, en estos entornos podremos

probar a incluir volúmenes persistentes para los microservicios, lo cual es un aspecto que permite Kubernetes y que en nuestro caso no hemos podido testear.

Otra de las mejoras que se pueden realizar es generalizar las configuraciones para máquinas con otros sistemas operativos, ya que nuestro proyecto se basa en que las máquinas tendrán instalado un Centos.

Por último, sería interesante añadir un proxy capaz de balancear el tráfico entre los diferentes minions, lo cual es posible utilizando el servicio de pcs. De esta forma, en caso de tener un problema con un minion, esta herramienta permitirá detectarlo y, automáticamente, actualizar su estado a inactivo y mandar todas las peticiones por otra vía.

9.4. Valoración personal.

Para finalizar, vamos a realizar una serie de apreciaciones y reflexiones personales acerca del desarrollo de este Trabajo de Fin de Máster.

Este trabajo se postula como el punto final para la consecución de las aptitudes necesarias que acreditan la obtención del título Ingeniería de Telecomunicación. Por ello, comprende una serie de capacidades que hay que poner en práctica para demostrar todo lo aprendido durante el transcurso del Máster. El objetivo final es poner de manifiesto la comprensión de conocimientos referentes a un área de estudio, aplicar dichos conocimientos al propio trabajo, obtener la capacidad de analizar, reunir e interpretar diferentes datos relevantes, transmitir información, ideas, problemas y soluciones a un público (tanto si es especializado en ese tema como si no) y, como punto final, la adquisición de las competencias de aprendizaje que permitan emprender posteriores estudios con un alto grado de autonomía.

Todos estos puntos se intentan plasmar en este trabajo con la ayuda de esta memoria en la que se aportan una gran cantidad de conocimientos adquiridos acerca del tema tratado, como son las arquitecturas basadas en microservicios en general y la automatización de un clúster y despliegue de microservicios dentro de él en particular. Se ha intentado ofrecer una visión global de lo que representa el nuevo paradigma de microservicios y de lo que éste puede aportar al despliegue de *software*, al mismo tiempo, se ha propuesto una solución para desplegar un microservicio y analizar su comportamiento dentro de dicho clúster.

Además, estos análisis y comparativas se han intentado justificar de modo que el lector pudiese comprender el porqué de las diferentes decisiones tomadas a lo largo del desarrollo.

Por otro lado, con este proyecto nos acercamos bastante a lo que podría ser un escenario de trabajo real, debido a que este tipo de arquitecturas están empezando a ser adoptadas en gran cantidad de empresas, de cara a ser capaces de agilizar la integración y el despliegue continuos, además de experimentar lo que podría ser un trabajo propio del sector ya que, dentro del objetivo final, se marcan ciertas etapas en las que dividir el trabajo para conseguir cumplir con los plazos y presentar el producto

solicitado. Finalizado este se evalúa si se ha cumplido con lo previamente establecido y en qué medida se han cumplido los requisitos marcados.

Centrándonos ahora en la valoración sobre este tipo de arquitecturas, podemos decir que sus perspectivas son bastante alentadoras, ya que la idea en la que se fundamentan abre muchas puertas de cara a ofrecer un servicio al consumidor escalable, de alta disponibilidad, que reduce los costes y, además, agiliza el trabajo de desarrolladores e integradores, ofreciendo vías para actualizar ciertas partes del sistema en caliente sin que esto conlleve un impacto en el resto del mismo. Empresas de la talla de *Google, Amazon o Netflix* ya trabajan con estas arquitecturas con sus propios sistemas avanzados, por lo que parece ser un camino interesante por el que seguir investigando.

Como punto final, cabe destacar la satisfacción que supone tanto a nivel personal como profesional la consecución de todo un proyecto por parte del autor. Este trabajo se presentaba como un gran reto debido a su complejidad y dimensiones, ya que se trata de un proyecto de investigación en el que, a día de hoy, existen pocas fuentes de información. Por ello se ha realizado un enorme trabajo para la obtención de los conocimientos previos a abordar su implementación. Finalmente hemos conseguido alcanzar los objetivos propuestos e incluso se han logrado algunos otros de forma adicional.

Anexo I. Manual de despliegue.

· Despliegue con *Vagrant*.

Para un despliegue completo, el cual contendría tanto la creación de las máquinas virtuales como el provisionamiento de la configuración, partiremos de *Vagrant*. En nuestro proyecto, hemos incluido dos ficheros fundamentales:

- *infraestructura.json*: este fichero contiene una descripción de las máquinas que vamos a desplegar, incluyendo información detallada de las mismas que será utilizada por *vagrant*. Una pequeña parte del mismo se muestra a continuación:

```
[{
                    "name": "minion-1",
                    "box": "centos/7",
                    "box-url":
      "https://atlas.hashicorp.com/centos/boxes/7/versions/1702.01/provid
      ers/libvirt.box",
                    "ram": 1024,
                    "cpu": 1,
                    "hostname_resolver": "on",
                    "apic": "on",
                    "insert_key": false,
                    "disable_synched_folder": true,
                    "ip_addr": "10.10.10.52"
            }, {
                    "name": "minion-2",
                    "box": "centos/7",
      ……
```

- *Vagrantfile*: fichero principal para la creación de nuestras máquinas virtuales. En él se indican tanto cuál es el fichero de infraestructura en el que se tiene que basar como la orden de que ejecute nuestro proyecto de *ansible* tras levantar las máquinas. Parte de este fichero se muestra a continuación:

```
# -*-
mode:
ruby
-*-
        # vi: set ft=ruby :


        require 'json'
```

```
          require 'fileutils'


          flag = 0


          Vagrant.configure(2) do |config|
              servers =
          JSON.parse(File.read(File.join(File.dirname(__FILE__),
          'infraestructure.json')))
                  #Iteramos una a una las máquinas de la lista
                  servers.each do |server|
                      config.vm.define server['name'] do |srv|
                          srv.vm.box_url = server['box_url']
                         srv.vm.box = server['box']
                          srv.ssh.insert_key = server['insert_key']
                          srv.ssh.forward_x11 = true
                          srv.vm.hostname = server['name']

                          srv.vm.network 'private_network', :ip =>
          server['ip_addr']
.....


          if     flag == servers.length
              srv.vm.provision :ansible do |ansible|
              ansible.playbook = "ansible/kubernetes.yml"
              ansible.inventory_path = "ansible/hosts"
              ansible.limit = "all"
          end
```

Como vemos, en las últimas líneas se aprecia la forma de llamar a *ansible* para llevar a cabo el provisionamiento de la configuración.

Con estos datos, el proceso para levantar el clúster en este caso sería el siguiente:

1. Descargar el proyecto de *GitHub*:

```
# git pull https://github.com/pahharo/kubernetes_and_lora_deployment.git
```

2. Adentrarnos en la carpeta “k8s-cluster”:

```
# cd kubernetes_and_lora_deployment/k8s-cluster
```

Antes de levantar el clúster, es necesario modificar el script ubicado en "*scripts/prepare_cluster.py*" y añadir la clave pública *ssh* al mismo, la cual estará ubicada en "*~/.ssh/id_rsa.pub*". En caso de no ser así, tendremos que generarla mediante el comando "*ssh-keygen*".

3. Ejecutar el siguiente comando para que *vagrant* comience a ejecutarse:

```
# cd vagrant up --provider libvirt
```

Automáticamente, debe iniciarse el proceso para que se despliegue y se provisione la configuración de un clúster formado por un nodo "*master*" y dos "*minions*".

Cabe destacar que, para que esto pueda realizarse correctamente, se debe tener instalado y funcionando el hipervisor de KVM en la máquina nativa.

· Despliegue de configuración con *Ansible*.

En caso de no querer hacer el despliegue de las máquinas virtuales, podemos realizar el provisionamiento de la configuración de máquinas Centos que ya tengamos operativas. Por lo tanto, lo único que tendríamos que hacer sería apuntar a dichas máquinas y ejecutar nuestro *playbook* de *ansible*.

El primer paso, por tanto, será modificar el fichero de *hosts* correspondiente a *ansible*, así que nos ubicamos en la carpeta correspondiente:

```
# cd kubernetes_and_lora_deployment/k8s-cluster/ansible
```

Aquí, en el fichero "*hosts*", modificaremos las direcciones ip para que cuadren con las de nuestras máquinas a provisionar.

Tras realizar esto, lo único que debemos hacer es ejecutar el siguiente comando en *ansible*:

```
# ansible-playbook kubernetes.yml –i hosts
```

Con esto, se lanzará la configuración automática de las diferentes máquinas contenidas en el fichero de *"hosts"* antes mencionado.

Anexo II Manual de usuario para prueba de concepto.

Para llevar a cabo la prueba de concepto cuyos resultados se mostraron en el capítulo 8, será necesario seguir las instrucciones que se indican a continuación.

Como paso previo al despliegue, será necesario modificar el script de Python situado en "*k8s-cluster/scripts/prepare_cluster.py*" y añadir la clave pública *ssh* en la línea 5, sustituyendo la indicación que se muestra:

```
public_ssh_key = "put-your-id_rsa-key here"
```

Una vez llevada a cabo la tarea anterior, habrá que desplegar el entorno, en nuestro caso un clúster de *Kubernetes* formado por tres máquinas *Centos*. Para ello, habrá que situarse en la carpeta "*/k8s-cluster*" del proyecto [33], donde se ubica el *Vagrantfile*, y ejecutar:

```
# vagrant up –provider libvirt
```

Con este comando estaremos arrancando tres máquinas sobre el hipervisor KVM de nuestra máquina y provisionando la configuración necesaria vía *Ansible*. Una vez finalizada esta tarea, tendremos nuestro clúster de *Kubernetes* operativo.

Si mantenemos la configuración de la infraestructura del clúster contenida en el fichero "*infraestructure. json*", podremos conectarnos a nuestro nodo máster vía ssh de la siguiente forma:

```
# ssh root@10.10.10.51
```

Para comprobar que tenemos a los dos nodos *minions* preparados, podemos ejecutar desde nuestro nodo *master*:

```
# kubectl get nodes
```

La respuesta a este comando debe ser que ambos están preparados (*status: Ready*), en caso contrario, significará que algo salió mal.

El siguiente paso será llevar a cabo un despliegue, crear un *Pod*, o acometer cualquier otra opción que deseemos en el clúster. En la carpeta "*kubernetes_files*" hemos añadido dos subcarpetas que contienen ejemplos tanto de *deployments* (en la carpeta "*deployments*") y servicios (en la carpeta "*services*") con los que se pueden llevar a cabo algunas pruebas en

nuestro clúster. No obstante, podemos realizar cualquier otro tipo de acción permitida para *Kubernetes*.

Un ejemplo será desplegar un *deployment* y asociarlo a un servicio. Para llevarlo a cabo a partir de los ficheros que hemos indicado, bastará con ejecutar:

```
# kubectl créate –f <ruta_del_fichero>
```

Como nota importante, si queremos acceder desde fuera de nuestras máquinas utilizando la etiqueta de *Nodeport*, tendremos que habilitar el reenvío de tráfico de entrada en nuestros *minions* con *iptables* ejecutando:

```
# iptables -P FORWARD ACCEPT
```

Por último, exponemos algunos comandos para llevar a cabo un seguimiento del estado de nuestros *deployments* o *Pods*.

1. Para comprobar el estado de los *deployments* podemos ejecutar:

   ```
   # kubectl get deployments
   ```

2. Para describir los *deployments* podemos ejecutar:

   ```
   # kubectl describe deployment <nombre_deployment>
   ```

3. Para comprobar el estado de los *pods* podemos ejecutar:

   ```
   # kubekubectl get pods
   ```

4. Para describir los *pods* podemos ejecutar:

   ```
   # kubectl describe pod <nombre_pod>
   ```

Siguiendo esta nomenclatura, podremos hacer lo propio para los servicios, *ReplicationControllers,* etcétera.

Anexo III. Complicaciones y errores encontrados.

En este anexo indicaremos una serie de errores que se nos han presentado durante la elaboración del proyecto y que resaltamos para informar tanto del error en sí como de su posible resolución si es que se ha encontrado.

· Error de *ansible* al intentar provisionar la configuración de máquinas lanzadas con *Vagrant.*

Este error se solventó añadiendo el *script* de "prepare_cluster.py", y era debido a que la clave pública *ssh* que se añadía no era la del usuario con la que lanzábamos el *playbook*, si no con el usuario de *vagrant*. Es importante añadir la clave pública si vamos a lanzar el *playbook* con un usuario distinto, tal y como se explica en el Anexo I. Manual de despliegue.

· Errores con nombres de máquinas en ficheros de configuración.

Encontramos un error relacionado con las versiones de *Kubernetes* referente al fichero de configuración de *apiserver*, el cuál espera recibir ips, no nombres de dominio.

· Errores relacionados con el direccionamiento en un mismo *pod*.

Al adaptar el servidor de *Internet of Things LoraServer*, fue un engorro entender la comunicación entre las distintas máquinas, ya que cada una recibe una ip. No obstante, pudimos comprobar que, al formar todos los contenedores parte del mismo *pod*, el direccionamiento entre ellos se puede realizar via "localhost", por lo que esa dirección es la que debemos añadir en los ficheros de configuración de cada uno respectivamente.

· Error en el registro de dispositivos en el servidor via *LoRa Server App*.

Al intentar registrar un dispositivo, el cuál era el último paso para tener nuestro servidor corriendo y bien configurado, pudimos apreciar un error relacionado con su provisionamiento. Este error se debía a que la base de datos de *redis* no estaba bien configurada. Una vez solventado el error, el registro se pudo acometer sin problemas.

· Errores debidos a pérdida de conexión de red.

Es importante tener conectividad a internet en caso de querer lanzar un despliegue en *Kubernetes* cuya plantilla requiera descargarse las imágenes de los contenedores de un repositorio externo. En caso de no hacerlo, podremos ver como, pese a que *Kubernetes* intenta levantar nuestro *Pod* o *Deployment*, este no llega a levantarse debido a que no es capaz de obtener los recursos.

· Error al volver a arrancar las máquinas con KVM.

Si hemos levantado las máquinas con *Vagrant* y, por algún casual, estas se han apagado, debemos asegurarnos de que las volvemos a levantar con esta herramienta, ya que es posible que, si intentamos hacerlo mediante la herramienta de KVM, con un "virsh start <nombre_de_la_máquina>", obtengamos un error. Para volver a levantar las máquinas con

su configuración bastará con ejecutar en la carpeta donde tenemos el *Vagrantfile* el comando "vagrant up".

· Error debido al servicio de *iptables*.

La primera vez que intentamos acceder desde fuera a uno de nuestros microservicios utilizando un mapeo de puertos con la etiqueta *Nodeport* tuvimos complicaciones debidas al servicio de *iptables*. Para posibilitar la entrada y salida de tráfico sin problema, permitimos el reenvío de tráfico de las máquinas *minions* ejecutando:

```
# iptables -P FORWARD ACCEPT
```

Bibliografía.

[1] RedHat, «redhat.com,» [En línea]. Available: https://www.redhat.com/es/topics/microservices.

[2] RedHat, «¿Qué es un contenedor de Linux?,» [En línea]. Available: https://www.redhat.com/es/topics/containers/whats-a-linux-container.

[3] RedHat, «¿Qué es Docker?,» [En línea]. Available: https://www.redhat.com/es/topics/containers/what-is-docker.

[4] A. M. Vizcaino, «Diferencias entre linux containers y docker,» [En línea]. Available: https://krahser.github.io/ops/containerization/.

[5] Openstack, «Openstack,» [En línea]. Available: https://www.openstack.org/.

[6] VMware, «VMware,» [En línea]. Available: https://cloud.vmware.com/.

[7] VMware, «Descripción de vCloud Suite,» [En línea]. Available: https://www.vmware.com/es/products/vcloud-suite.html.

[8] Microsoft, «Orquestación de microservicios y aplicaciones de varios contenedores para una alta escalabilidad y disponibilidad,» [En línea]. Available: https://docs.microsoft.com/es-es/dotnet/standard/microservices-architecture/architect-microservice-container-applications/scalable-available-multi-container-microservice-applications.

[9] R. Orayen, «Docker Swarm,» [En línea]. Available: https://robertoorayen.eu/2018/03/18/docker-swarm/.

[10] Nubersia, «Mesosphere DC/OS,» [En línea]. Available: https://www.nubersia.com/es/blog/mesosphere-dcos/.

[11] CLE formación, «Puppet, Chef o Ansible. Comparando herramientas de aprovisionamiento,» [En línea]. Available: http://www.cleformacion.com/tic-tek/-/blogs/puppet-chef-o-ansible-comparando-herramientas-de-provisionamiento.

[12] Ansible, «Ansible Documentation,» 01 September 2017. [En línea]. Available: https://docs.ansible.com/ansible/2.3/index.html.

[13] Ncora, «Chef, una herramienta básica de automatización,» [En línea]. Available: https://www.ncora.com/blog/chef-una-herramienta-basica-de-automatizacion/.

[14] HashiCorp Vagrant, [En línea]. Available: https://www.vagrantup.com/.

[15] J. M. T. Díaz, «Kubernetes simple (1 Master y 1 Minion),» [En línea]. Available: http://tedezed.github.io/Celtic-Kubernetes/HTML/2-Kube_simple.html.

[16] Linux-KVM, «KVM,» [En línea]. Available: https://www.linux-kvm.org/page/Main_Page.

[17] Nubersia, «Kubernetes vs Docker Swarm! cuál es mejor opción?,» [En línea]. Available: https://www.nubersia.com/es/blog/kubernetes-vs-docker-swarm/.

[18] Docker, «Docker Swarm,» [En línea]. Available: https://docs.docker.com/engine/swarm/.

[19] Apache, «Apache Mesos,» [En línea]. Available: http://mesos.apache.org/.

[20] Google, «Running Kubernetes Locally via Minikube,» [En línea]. Available: https://kubernetes.io/docs/setup/minikube/.

[21] Google, «Creating a single master cluster with kubeadm,» [En línea]. Available: https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/.

[22] YAML, «YAML,» [En línea]. Available: http://yaml.org/.

[23] Google, «What is a Deployment?,» [En línea]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/deployment.

[24] Intigua, «Puppet vs Chef vs Ansible vs Saltstack,» [En línea]. Available: https://www.intigua.com/blog/puppet-vs.-chef-vs.-ansible-vs.-saltstack.

[25] Puppet, «Puppet,» [En línea]. Available: https://puppet.com/.

[26] Ruby, «Ruby,» [En línea]. Available: https://www.ruby-lang.org/es/.

[27] Python, «Python,» [En línea]. Available: https://www.python.org/.

[28] HashiCorp, «Vagrant,» [En línea]. Available: https://www.vagrantup.com/.

[29] GIT, «A cerca del control de versiones,» [En línea]. Available: https://git-scm.com/book/es/v1/Empezando-Acerca-del-control-de-versiones.

[30] Wikipedia, «Control de versiones,» [En línea]. Available: https://es.wikipedia.org/wiki/Control_de_versiones.

[31] Jenkins, «Jenkins,» [En línea]. Available: https://jenkins.io/.

[32] Openstack, «Zuul - A project Gating System,» [En línea]. Available: https://zuul-ci.org/docs/zuul/.

[33] M. S. López, «GitHub: Kubernetes in a box - Three nodes deployment + Scaling,» [En línea]. Available: https://github.com/pahharo/kubernetes_and_lora_deployment.

[34] LoraServer, «LoraServer, open-source LoRaWAN network-server,» [En línea]. Available: https://www.loraserver.io/.

[35] Docker, «Docker Hub,» [En línea]. Available: https://hub.docker.com/.

[36] Docker, «Docker run usage,» [En línea]. Available: https://docs.docker.com/engine/reference/commandline/run/.

[37] Docker, «Child commands,» [En línea]. Available: https://docs.docker.com/engine/reference/commandline/.

[38] LoraServer, «LoRa App Server,» [En línea]. Available: https://www.loraserver.io/lora-app-server/overview/.

[39] LoraServer, «LoRa Server,» [En línea]. Available: https://www.loraserver.io/loraserver/overview/.

[40] LoraServer, «LoRa Gateway Bridge,» [En línea]. Available: https://www.loraserver.io/lora-gateway-bridge/overview/.

[41] M. S. López, «Github: kubernetes_and_lora_deployment/containers-manu/,» [En línea]. Available: https://github.com/pahharo/kubernetes_and_lora_deployment/tree/master/containers-manu.

[42] Google, «Kubernetes Services,» [En línea]. Available:

https://kubernetes.io/docs/concepts/services-networking/service/.

[43] Microsoft, «Orquestación de microservicios y aplicaciones de varios contenedores para una alta escalabilidad y disponibilidad,» [En línea]. Available: https://docs.microsoft.com/es-es/dotnet/standard/microservices-architecture/architect-microservice-container-applications/scalable-available-multi-container-microservice-applications.