Despliegue de aplicaciones contenerizadas en entornos *cloud*

Araceli García Granados

Master en Ingeniería de Telecomunicación
Telemática

Nombre Tutor/a de TF
Jose Lopez Vicario
Profesor/a responsable de la asignatura
Xavi Vilajosana Guillen

15 de enero de 2023

Universitat Oberta
de Catalunya

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Despliegue de aplicaciones contenerizadas en entornos cloud*
Nombre del autor:	*Araceli García Granados*
Nombre del consultor/a:	*Jose Lopez Vicario*
Nombre del PRA:	*Xavi Vilajosana Guillen*
Fecha de entrega (mm/aaaa):	*01/2023*
Titulación o programa:	Master de Ingeniería de Telecomunicación
Área del Trabajo Final:	*Telemática*
Idioma del trabajo:	*Castellano*
Palabras clave	*Docker, Kubernetes, OpenShift*

Resumen del Trabajo

El sector empresarial está evolucionando a un ritmo tal que los negocios cada vez tienen que responder más rápidamente a las condiciones cambiantes del mercado. Las infraestructuras tradicionales no se adaptan bien a estas velocidades, además de ser caras y al alcance de muy pocos. El *cloud* permite a todo tipo de empresas y particulares disponer de los recursos cuando lo necesiten sin que tengan que realizar grandes inversiones en infraestructura.

La contenerización es una metodología que hace rentable y eficiente el despliegue de las aplicaciones en entornos *cloud* al permitir alojar multitud de cargas de trabajo en una misma infraestructura.

Las tres plataformas de contenerización más utilizadas actualmente son *Docker*, como herramienta líder en la gestión de contenedores e imágenes, *Kubernetes*, estándar de facto para la orquestación de contenedores, y *OpenShift*, plataforma de contenedores para el despliegue de aplicaciones empresariales.

En este TFM se realiza un caso práctico de despliegue de una aplicación *web* en un *cloud* público sobre estas tres plataformas de contenerización. Se comienza exponiendo las bases teóricas de este nuevo paradigma. Posteriormente se desarrolla una aplicación *Nodejs* para desplegarla en un *cluster* bajo estas tres plataformas. Se realizan una serie de pruebas para verificar los beneficios de la orquestación. Finalmente, se hace una comparativa de las tres plataformas.

Abstract

The business sector is evolving at such a rate that businesses increasingly have to respond more quickly to changing market conditions. Traditional infrastructures do not adapt well to these speeds, as well as being expensive and within the reach of very few. The cloud allows all types of companies and users to access to resources when they need them without having to make large investment in infrastructure.

Containerization is a methodology that makes the deployment of applications in cloud environments profitable and efficient by allowing a multitude of workloads to be hosted on the same infrastructure.

The three most widely used containerization platforms today are *Docker*, as the leading container and image management tool, *Kubernetes*, the “de facto standard” for container orchestration, and *OpenShift*, the container platform for enterprise application deployment.

In this TFM, a practical case of deploying a web application in a public cloud on these three containerization platforms is made. It begins by exposing the theoretical bases of this new paradigm. Subsequently, a Nodejs application is developed to deploy it in a cluster under these three platforms. A series of tests are performed to verify the benefits of orchestration. Finally, a comparison of the three platforms is made.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del trabajo

Este TFM es un trabajo de investigación y puesta en práctica de las técnicas que se están utilizando actualmente para el despliegue de aplicaciones *TI* mediante contenerización sobre plataformas *cloud*.

¿Qué se entiende por *despliegue de aplicaciones contenerizadas en entornos cloud*?

- El *despliegue de aplicaciones* corresponde con aquellas actividades que hacen que las aplicaciones informáticas estén disponibles para su uso. No se trata de cómo desarrollar las aplicaciones, sino de las tareas necesarias para que éstas puedan ser consumidas por los usuarios en con unas determinadas condiciones como disponibilidad y escalabilidad del servicio, balanceo de carga, seguridad, etc.

- La palabra *contenerizada* hace referencia a la estanqueidad y la autonomía de una aplicación para que pueda aislarse e independizarse del entorno en el que se está ejecutando. Las aplicaciones alojadas en contenedores evitan interferencias no deseadas entre ellas. Por ejemplo, un cambio de versión de una determinada aplicación no debe influir en el correcto funcionamiento del resto de las aplicaciones alojadas en la misma infraestructura.

- El *cloud* o nube es una infraestructura de terceros que permite el alojamiento de las aplicaciones, y a la que se accede desde cualquier dispositivo conectado a internet. El término opuesto es "*on-premise*", o "*in-house*", que hace referencia a que las aplicaciones de una organización se alojan en sus propias instalaciones.

Se han seleccionado las tres tecnologías de contenerización que tienen una mayor penetración en el mercado, concretamente *Kubernetes* (cuota del 77%), *OpenShift* (9%) y *Docker* (5%) [1].

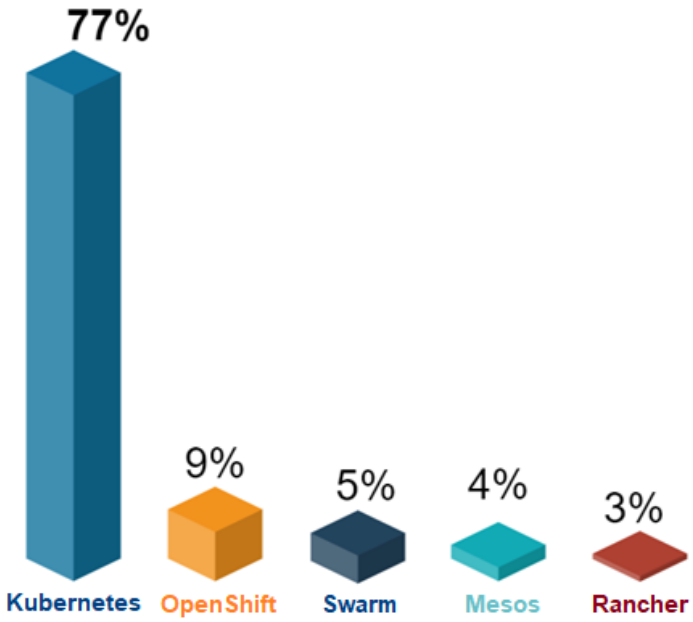


Figura 1. Cuota de mercado herramientas de orquestación [1]

1.2 Objetivos del trabajo

El objetivo que se pretende alcanzar con este trabajo de investigación es la puesta en práctica del despliegue de aplicaciones en varias plataformas de contenerización, bajo un entorno *cloud*, con el fin de evaluar los beneficios de disponer de un sistema orquestado, así como realizar una comparativa de las distintas tecnologías de contenedores que existen actualmente.

Podemos desglosar este objetivo en los siguientes puntos:

1. Investigar sobre los beneficios que aporta el *cloud* y la contenerización.
2. Investigar sobre las técnicas utilizadas actualmente para el despliegue de aplicaciones en entornos *cloud*.
3. Implementar un caso real de despliegue sobre la plataforma *Docker*.
4. Implementar el mismo caso sobre la plataforma *Kubernetes*.
5. Implementar el mismo caso sobre la plataforma *OpenShift*.
6. Comparar estas tres tecnologías de contenerización.

El primer objetivo corresponde con una labor de investigación sobre qué es lo que está motivando a las empresas y particulares a abandonar el tradicional entorno "*on-premise*", donde las aplicaciones se alojan en las propias infraestructuras, a un entorno "*cloud*", en el que la infraestructura es suministrada por un proveedor. También se estudiará qué tipo de virtualización es la que mejor se adapta a este nuevo paradigma: las máquinas virtuales o la contenerización.

Una vez entendido el *por qué* se está evolucionando a este nuevo escenario, se establece como segundo objetivo el investigar sobre el *cómo*, es decir,

cuáles son las herramientas y tecnologías existentes en el mercado que posibilitan esta nueva forma de consumir recursos de *TI*.

Seleccionado ya el *cloud* como entorno para los despliegues, y adquiridos los conocimientos y técnicas que permiten poner en producción las aplicaciones en esta plataforma, es el momento de poner en práctica todos estos conceptos.

El tercer objetivo contemplará el despliegue de una determinada aplicación *web* en una nube pública bajo la plataforma de contenerización líder en la industria: *Docker*.

El cuarto objetivo corresponderá al despliegue de esa misma aplicación, pero en la plataforma de orquestación más utilizada actualmente, *Kubernetes*.

Como quinto objetivo se provisionará esta misma aplicación en la plataforma de contenedores líder en el mercado empresarial, *OpenShift*.

El último, y principal, objetivo que se pretende alcanzar con este trabajo es realizar una comparativa de las tres tecnologías, fundamentalmente respecto a los siguientes aspectos:

- Dificultad de la instalación y configuración de la plataforma.
- Comportamiento de orquestación: disponibilidad, balanceo de carga y escalabilidad de la aplicación.
- Ámbito de aplicación: posibles escenarios de implantación.

1.3 Impacto de sostenibilidad, ético-social y de diversidad

Respecto al impacto en la sostenibilidad, un despliegue de aplicaciones en entornos *cloud* tiene un impacto positivo, fundamentalmente sobre la *ODS 12* (consumo y producción sostenible) de la Agenda 2030, dado que la computación en la nube reduce el consumo global de recursos al compartir la infraestructura por una multitud de usuarios, que además la utilizan únicamente cuando la necesitan. Según [2] las empresas que migran al *Cloud* pueden reducir el consumo energético un 65% y un 84% sus emisiones de carbono, colaborando de esta manera con la sostenibilidad de nuestro planeta.

Respecto al comportamiento ético y de responsabilidad social, este paradigma de despliegue de aplicaciones contribuye al bien común de la sociedad dado que elimina las desigualdades sociales al globalizar el acceso a una infraestructura por parte de particulares y empresas, pagando únicamente por el uso de los recursos que consumen.

En cuanto a la diversidad y derechos humanos, al concentrar los datos de los usuarios en unas plataformas *cloud* cuya propiedad es de unas pocas organizaciones empresariales, se corre el riesgo de sufrir un ataque a la privacidad y seguridad de los ciudadanos, incluso una oportunidad para que

algunos Gobiernos puedan obtener información a través de estos proveedores *cloud*. Por tanto, el impacto sería negativo en este sentido.

1.4 Enfoque y método seguido

Este trabajo está organizado en dos partes. En un primer bloque se realiza una labor de investigación sobre las técnicas que habilitan los despliegues de aplicaciones en un *cloud*. Se recopilan los componentes básicos de una arquitectura en la nube, como los contenedores, máquinas virtuales, *pods*, imágenes, *runtime* de contendores, orquestadores, etc. También se analizan las características de los despliegues en la nube, identificando ventajas e inconvenientes. Se recopilan los distintos tipos de nubles, así como las distintas modalidades se servicio que puede ofrecer. Finalmente, se realiza una introducción a las plataformas de orquestación de contenedores más utilizadas.

En un segundo bloque se pone en práctica todos los conocimientos adquiridos en el primer bloque, desplegando una aplicación desarrollada en *Node* en tres plataformas de contenedores, *Docker*, *Kubernetes* y *OpenShift*. Se detallará todo el proceso partiendo desde cero, es decir, asumiendo que no se dispone de conocimientos profundos en Sistemas y Telemática, y partiendo de una máquina *Windows* sin ningún *software* previamente instalado. Todo el *software* utilizado en este trabajo es "*open source*".

1.5 Planificación

La planificación de este TFM se ha condicionado a las restricciones temporales de las distintas *PEC* de la asignatura. La fecha de inicio corresponde con el 06-10-2022 coincidiendo con la *PEC1* ("*Entrega de la planificación del trabajo*"), y finaliza el 15-01-2023 con la *PEC4* ("*Entrega de la memoria final y presentación*").

El diagrama de Gantt de las tareas principales queda así:

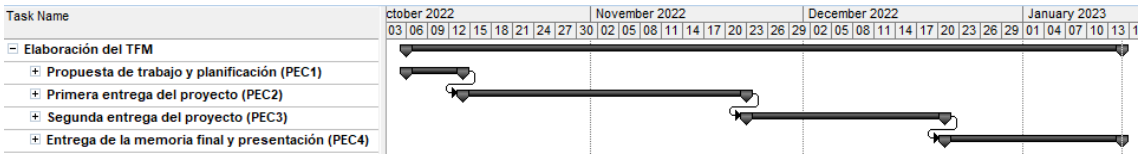


Figura 2. Diagrama de Gantt de las tareas principales

El diagrama de Gantt completo, incluyendo las subtareas, se muestra en la siguiente figura:

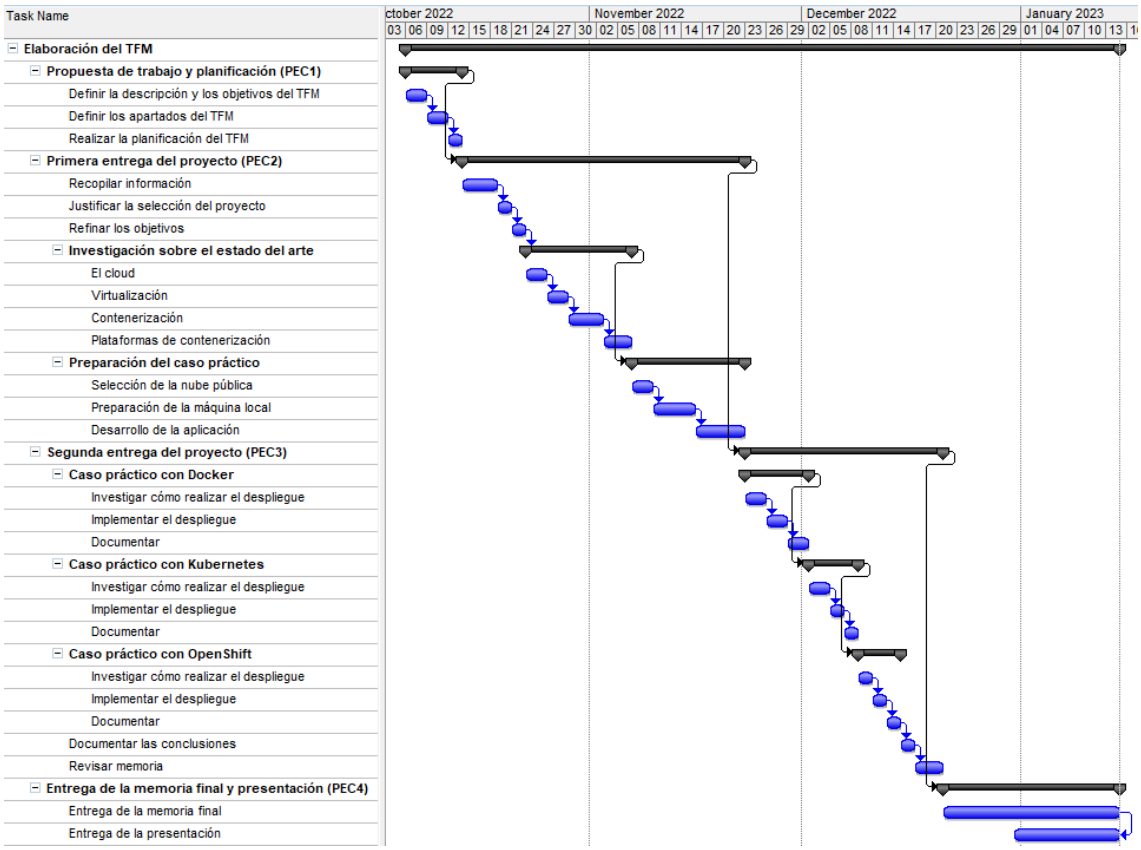


Figura 3. Diagrama de Gantt completo

Como podemos observar, en la *PEC2* ("*Primera entrega del proyecto*") se tendrá que haber elaborado la primera fase del TFM correspondiente a la parte teórica, y con la *PEC3* ("*Segunda entrega del proyecto*") se espera haber concluido la implementación del caso práctico en las tres plataformas de orquestación. Se reservarán los últimos 25 días, correspondientes a la *PEC4* ("*Entrega de la memoria final y presentación*") para dar los últimos retoques al TFM, así como la elaboración de la presentación que se realizará en la defensa de dicho trabajo ante el tribunal evaluador.

Actualización de la planificación

Durante la ejecución del proyecto se han producido algunos cambios que ha conllevado la modificación de la planificación. Estos cambios se han debido al haberse estimado inicialmente una disponibilidad de estudio de 3 horas diarias, cuando realmente los meses de octubre y noviembre se han podido destinar casi 4 horas/día. Esto ha generado un adelanto en planificación de unos 9 días, por lo que en el hito de la *PEC2* se ha incluido el caso práctico con *Docker*. La planificación actualizada se aprecia en la siguiente figura:

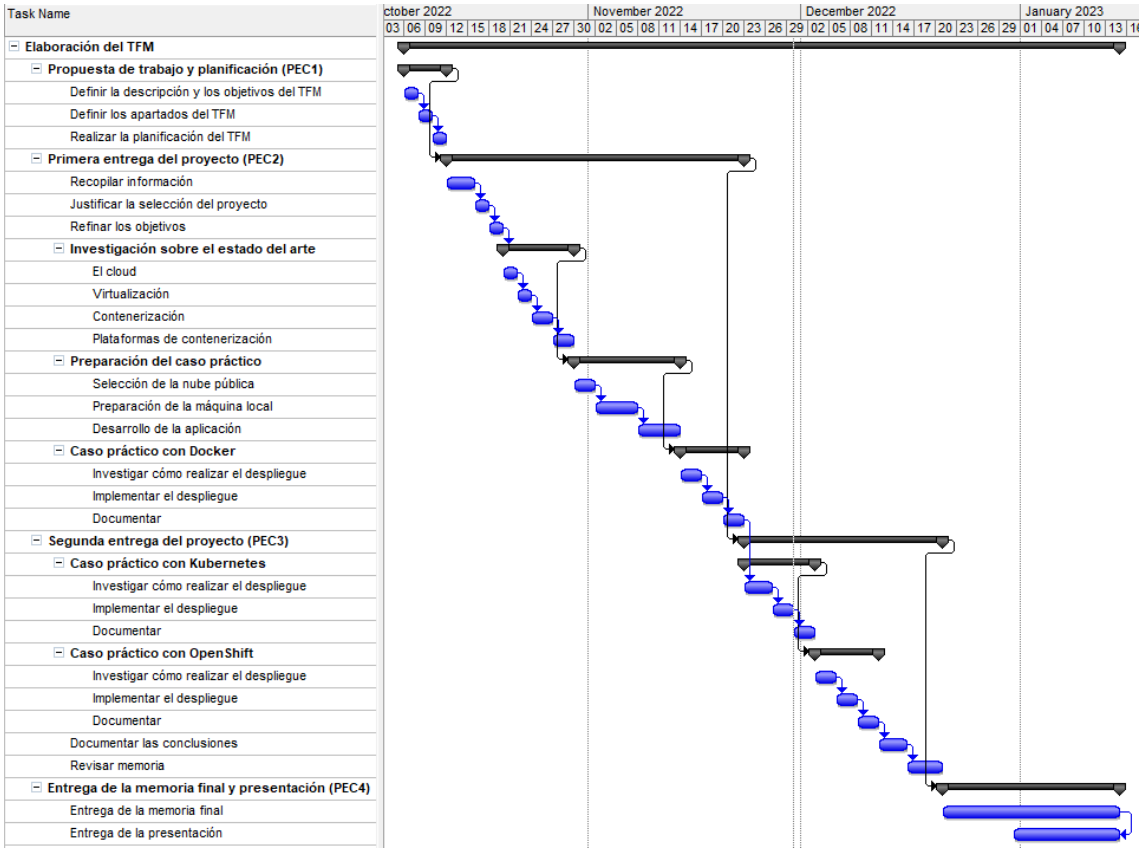


Figura 4. Diagrama de Gantt completo actualizado

1.6 Breve sumario de los productos obtenidos.

Este trabajo proporciona una documentación exhaustiva para instalar y configurar las tres plataformas más importantes del mercado correspondientes al despliegue de aplicaciones *TI* en entornos *cloud*: *Docker*, *Kubernetes* y *OpenShift*. También se documentan los procesos necesarios para la puesta en producción de un desarrollo *web* bajos estas tres tecnologías. Se realizan varios ensayos de orquestación para poner a prueba la disponibilidad, escalabilidad y balanceo de carga del sistema. Finalmente, se realiza un análisis y una comparativa de estas tres plataformas.

1.7 Breve descripción de otros capítulos de la memoria

El trabajo comienza con una introducción que permitirá poner en contexto este nuevo paradigma. En el segundo capítulo se analizan los diferentes métodos de despliegue y se justificará la elección de la contenerización como opción más adecuada en el entorno actual. En el tercer capítulo se presentan los conceptos y tecnologías que actualmente están dando soporte al despliegue de aplicaciones en el *cloud*. En el capítulo cuarto se desarrolla una aplicación *Nodejs*, para posteriormente realizar tanto su provisión en las tres plataformas

de contenerización, como la realización de ciertas pruebas de orquestación que evidencien la disponibilidad, balanceo de carga y escalabilidad que proporcionan. En el apartado quinto se verifica el cumplimiento de los objetivos marcados inicialmente. Finalmente, en el apartado seis se exponen las conclusiones a las que se ha llegado con la elaboración de este trabajo, así como se identifican posibles puntos de mejora que pudieran complementar este trabajo en un futuro.

2. Estado del arte

Tradicionalmente el diseño de las aplicaciones se basaba en una estructura monolítica, es decir, compuesta por multitud de procesos alojados en una misma infraestructura física (o "*bare metal*"), compartiendo tanto el sistema operativo como determinados ficheros binarios y librerías (lo que se denomina "*dependencias*"). En este escenario, cuando se quería realizar una modificación de alguna de las aplicaciones (un cambio de un proceso, una actualización del sistema operativo, un cambio de librería, etc.) había que asegurar que estas modificaciones no afectasen al resto de las aplicaciones y procesos. En cierta medida esto no era un problema cuando se trataba de implementar unas cuantas aplicaciones relativamente sencillas, pero con los años éstas se han ido haciendo cada vez más complejas con multitud de procesos interdependientes, lo que ha llevado a que el mantenimiento del sistema sea cada vez más complejo.

Con el objetivo de encapsular estas aplicaciones y ahorrar infraestructura, sobre el año 2005 se empezó a popularizar una nueva arquitectura llamada *virtualización*. En este escenario se dispone de un *hipervisor* por encima del sistema operativo del equipo físico, después los sistemas operativos huésped (o también llamados "invitados"), más arriba los archivos binarios y librerías, y finalmente las aplicaciones que corren en las máquinas virtuales (*Virtual Machine*, *VM*). La virtualización permite que, en un mismo equipo físico con un sistema operativo anfitrión, podamos disponer de varias máquinas virtuales, cada una con su propio sistema operativo huésped. En este escenario se "exprime" la infraestructura (el "hierro") al compartirla entre las distintas máquinas virtuales. Para que esta arquitectura sea posible se necesita un componente por encima del sistema operativo anfitrión llamado *hipervisor* (*VirtualBox*, *VMware*, etc.), que exponga los recursos hardware (*CPU*, memoria, etc.) utilizados por los sistemas operativos huésped de cada una de las máquinas virtuales. El inconveniente de este modelo es el gran consumo de *CPU* y memoria *RAM* al tener que incluir todo un sistema operativo adicional por cada máquina virtual, lo hace que, por ejemplo, el iniciar una de estas máquinas lleve un tiempo de varios minutos.

Sobre el año 2014 se empezó a popularizar un nuevo modelo de desarrollo de aplicaciones basadas en *microservicios*, con el objetivo de dividir grandes sistemas *software* en piezas más pequeñas. Un microservicio es un pequeño desarrollo autónomo y escalable que se puede desplegar de manera independiente, de forma que las aplicaciones se puedan desarrollar como un conjunto de microservicios interrelacionados. Si un sistema se compone de unas pocas aplicaciones, una solución es alojar cada aplicación en una máquina virtual, pero en un sistema compuesto por una gran cantidad de microservicios es inviable alojar cada microservicio en una *VM* si queremos mantener un bajo coste del hardware y no desperdiciar recursos, además del volumen de mano de obra técnica que se necesita para configurar y mantener esta cantidad de máquinas virtuales.

En los últimos años ha aparecido una nueva arquitectura llamada *contenerización* con el objetivo de aislar las aplicaciones (o microservicios) con un consumo mínimo de recursos. Un contenedor empaqueta, además de la propia aplicación, sus dependencias, es decir, los recursos que exclusivamente necesite para funcionar (binarios, librerías, etc.), sin requerir un sistema operativo completo, por lo que es mucho más ligero que una máquina virtual. Con los contenedores se exprime aún más "el hierro" pues, en lugar de disponer de un sistema operativo huésped para cada máquina virtual, se comparte únicamente el *kernel* del sistema operativo del *host*. El *kernel* es el núcleo del sistema operativo y, entre otras funciones, se encarga de otorgar los permisos a los procesos (contenedores) para usar el *hardware* de la máquina. Con la contenerización se puede alojar en una misma infraestructura un mayor número de aplicaciones (microservicios), además, el tiempo de lanzamiento de las aplicaciones es prácticamente instantáneo (en segundos frente a los varios minutos que se necesita con las *VMs*), debido a que no hay que levantar ningún sistema operativo para arrancar el contenedor. Por tanto, los contenedores son claramente una mejor solución para alojar aplicaciones desarrolladas bajo el paradigma de microservicios. También tiene una desventaja, pues al compartir el *kernel* del sistema operativo del *host*, el nivel de aislamiento de los contenedores es algo inferior que el de las máquinas virtuales y, por otro lado, si este *kernel* tiene cualquier tipo de problema (fallo, ataque malicioso, etc.), se verán afectados todos los contenedores de la máquina (física o virtual) del *host*.

En la siguiente imagen se puede apreciar una comparativa de estos tres tipos de arquitecturas: tradicional, virtualizada y contenerizada:

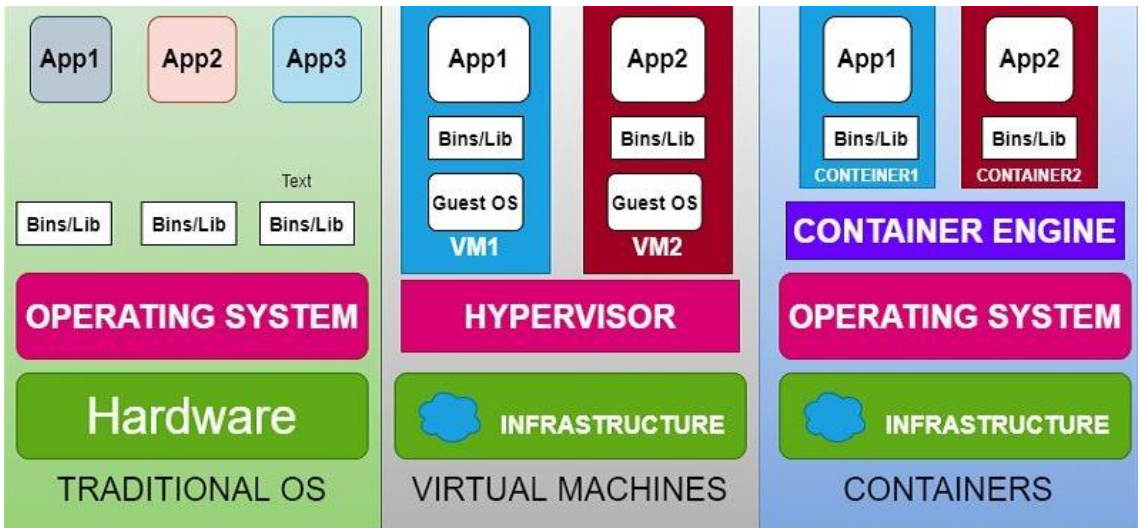


Figura 5. Despliegue tradicional/máquinas virtuales/contendores [4]

Los trabajos de contenerización que se encuentran en la literatura se centran en el desarrollo de aplicaciones que, o no están alojados en una nube pública, o no hacen uso de la orquestación de contenedores. No se ha encontrado ningún trabajo que justifique la elección de una determinada tecnología de orquestación. Tampoco se ha encontrado ningún estudio comparativo de las

distintas tecnologías actuales de contenerización. La siguiente lista de trabajos representa un ejemplo de ello:

- “Hacia un red eficiente y virtualizada” de Parejo Jimenez, Jesús. Universitat Oberta de Catalunya (UOC). En este trabajo se realiza el despliegue de una aplicación de telefonía con *Asterisk* sobre contenedores *Docker*, pero sin orquestar.
- “Estudio de sistemas cluster de Kubernetes y su monitorización e implantación en entorno productivo” de Piña Cubas, Luis. Universitat Oberta de Catalunya (UOC). Este trabajo trata el tema de la orquestación de *Kubernetes*, pero se aplica sobre un cluster *on-premise* y, por tanto, sin hacer uso del *cloud*. Tampoco se pone en práctica la orquestación de los *pods*.
- “Análisis técnico para la modernización y migración y su implementación a servicios cloud con AWS” de Nadal Castell, Juan Carlos. Universitat Oberta de Catalunya (UOC). Este trabajo trata cómo implementar una aplicación de reserva hotelera haciendo uso de un *cluster* de *Kubernetes* en la nube pública de *Amazon*, pero sin detalle de su implementación ni pruebas de orquestación.
- “Implementación de un SIEM para la auditoría de eventos de seguridad sobre cluster de Kubernetes en un entorno multicloud” de Fernández Ameijeiras, José Ángel. Universitat Oberta de Catalunya (UOC). Como se indica en el propio trabajo, no se entra en detalle ni en el despliegue de contenedores ni en su orquestación.

A diferencia de estos trabajos, este TFM pone el foco en el despliegue de las aplicaciones en distintas plataformas de contenerización, en lugar del diseño de un determinado sistema alojado en el *cloud*. Se pone a prueba los beneficios de la orquestación de contenedores, como la disponibilidad, escalabilidad y balanceo de carga de las aplicaciones mediante diferentes ensayos. Además, se realiza una comparativa de diferentes plataformas de contenerización, lo que puede ser relevante para que los implantadores de sistemas puedan elegir la tecnología que mejor se adapte a sus diseños.

3. Conceptos previos

3.1 El *Cloud*

Este apartado describe qué es el *cloud* y cómo es la infraestructura que permite desplegar aplicaciones en este entorno. También se van a exponer las ventajas de este nuevo paradigma, así como los distintos tipos de *cloud* y modelos de servicio.

3.1.1 Introducción

El *Cloud Computing*, o simplemente *cloud*, ha representado una gran revolución tecnológica en los últimos años debido a que las empresas han encontrado en este nuevo paradigma la forma de optimizar sus procesos, incrementar la productividad y, por tanto, reducir sus costes. El concepto de "*nube*" se usa habitualmente para describir una infraestructura remota, con capacidad de ofrecer servicios de *TI* bajo demanda, cuyo acceso se realiza a través de internet, y sin que los usuarios, particulares y empresas, requieran prácticamente de infraestructura propia para desplegar sus aplicaciones, además, éstos pagan únicamente por los recursos que consumen. Los servicios que se ofrecen en el *cloud* se asignan y liberan de una forma muy rápida con una mínima intervención del proveedor de la nube.

La infraestructura de la nube se compone de una serie de sistemas *hardware* físicos repartidos en distintas áreas geográficas, llamadas Centros de Datos o "*Data Centers*", y que incluyen equipos tales como servidores, *routers*, *switches*, *firewalls*, balanceadores de carga, discos de almacenamiento, etc. Los clientes despliegan sus aplicaciones contenerizadas en una serie de nodos agrupados en un *cluster*. Estos nodos son máquinas virtuales con su sistema operativo propio que se alojan en la infraestructura del proveedor del *cloud*, es decir, en los *Data Centers*. A su vez, estos nodos dan cabida a unas entidades similares a las máquinas virtuales, más pequeñas y ligeras, llamadas *contenedores*, donde los clientes ubican sus aplicaciones, pero a diferencia de las máquinas virtuales, los contenedores no incorporan un sistema operativo completo, sino exclusivamente lo que necesita la aplicación para poder ejecutarse en un determinado *kernel* del sistema operativo del *host*, lo que los hace muy ligeros y rápidos de arrancar.

En la siguiente figura se puede apreciar un "zoom" de esta infraestructura:

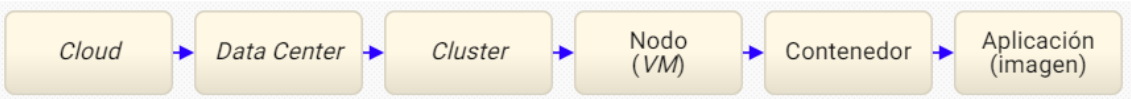


Figura 6. Zoom de la infraestructura *cloud* (elaboración propia)

Los principales proveedores de servicios en la nube a nivel mundial son los denominados "*the big 3 cloud providers*", es decir, *Amazon Web Service* (*AWS*), *Microsoft Azure* y *Google Cloud Platform* (*GCP*) [3].

3.1.2 Ventajas e inconvenientes

Los principales beneficios que se obtiene de un despliegue en el *cloud* son los siguientes:

Reducción de costes. No se necesita realizar una gran inversión inicial en infraestructura, ni asumir el coste de su mantenimiento, ni el pago de licencias correspondientes a sus actualizaciones hardware y software.

Agilidad. Con una infraestructura tradicional se requiere estar semanas instalando y levantando un servidor. Con el *cloud*, es cuestión de segundos.

Seguridad/disponibilidad. En una arquitectura en la nube la disponibilidad de la infraestructura recae en el proveedor y éste, sabiendo que su negocio depende de la confianza de sus clientes, utiliza sofisticados mecanismos de seguridad para que los datos se mantengan privados y almacenados de forma segura. El proveedor protege los entornos informáticos, las aplicaciones que se ejecutan en la nube, así como los datos que se almacenan en ella.

Escalabilidad. Los usuarios de la nube pueden redimensionar los recursos contratados en función de sus necesidades, de una manera rápida, en cualquier momento y sin intervención manual del proveedor.

Pago por uso. El cliente del *cloud* paga únicamente por el consumo real de los recursos que tenga contratados, es decir, no se paga por adquirir los recursos, sino por el uso de los mismos.

Autoservicio a demanda. El cliente del *cloud* utiliza los servicios en función de sus necesidades sin intervención del proveedor de servicios.

Recursos compartidos. Los servicios que ofrece el proveedor de la nube (pública) se comparten por todos sus clientes. El usuario, en general, no tiene control de la ubicación de los recursos del proveedor, aunque en ciertos casos sí que los puede especificar (país, centro de datos, etc.).

Métricas de los servicios. El proveedor de la nube controla y optimiza los recursos mediante la supervisión y medición de los servicios ofrecidos por la nube, de forma transparente para el usuario. La capacidad de almacenamiento, el procesamiento, el ancho de banda, el tiempo de utilización, etc., se monitorizan continuamente para que el proveedor pueda optimizarlos y facturar a los usuarios.

Amplio acceso a la red. Dado que la nube está conectada a internet, el acceso a los recursos se realiza desde cualquier lugar, y desde cualquier dispositivo con conexión a internet, como teléfonos móviles, portátiles, etc.

Capacidad (casi) ilimitada de almacenamiento y proceso. Los usuarios tienen la impresión de que los recursos a los que tiene acceso son ilimitados y se pueden adquirir en cualquier momento.

Alta elasticidad. Las necesidades de los consumidores se pueden provisionar de forma muy rápida al ritmo que las demanden, haciendo que sea ágil tanto el escalado como la liberación de recursos.

Actualización automática. Se libera al usuario del *cloud* de las tareas asociadas a instalación de nuevas versiones en las plataformas *hardware* y *software*.

Optimización del uso de los recursos. Al tratarse de una plataforma fácilmente escalable, los usuarios solo consumen los servicios que verdaderamente necesitan sin tener que disponer de un exceso de capacidad como ocurre en el modelo *on-premise*.

Reducción de las barreras a la innovación como consecuencia de una reducción de los costes, disminución de los tiempos de comercialización y una mayor seguridad en las aplicaciones y los datos.

Igualdad. Las pequeñas empresas y usuarios se encuentran en las mismas condiciones que las grandes corporaciones con respecto al acceso a la tecnología, lo que les permite competir en igualdad de oportunidades.

Respeto con el medio ambiente. El proveedor de la nube comparte la infraestructura entre todos sus clientes disminuyendo el consumo eléctrico global y la huella de carbono, generando un menor impacto ambiental agregado.

Como inconvenientes se destacan los siguientes:

Pérdida de control. El usuario no tiene control sobre la infraestructura de la nube por lo que sus datos y aplicaciones quedan en manos del proveedor.

Seguridad de los datos. En caso de que la nube sufra un ataque de seguridad, la información de los clientes puede verse comprometida. No obstante, hay que destacar que los mecanismos de seguridad de estas infraestructuras son muchos más avanzados que los que pueden tener los clientes en sus infraestructuras *on-premise*.

Acceso remoto. Puesto que el acceso se soporta sobre una conexión a internet, en caso de no disponer de conexión por cualquier motivo (caída de la red, fuera cobertura, etc.) no se dispondrá de acceso a las aplicaciones.

3.1.3 Modalidades de *Cloud*

Los distintos tipos de nubes son los siguientes:

Nube privada. Este tipo de nube es utilizada por una única empresa siendo gestionada por la propia organización o delegada a un tercero. Normalmente el cliente es el propietario de la infraestructura de la nube con un control total de las aplicaciones que se despliegan en ella. Tiene el inconveniente de que es el propio cliente el que debe soportar el coste de la infraestructura, pero la ventaja de que los datos no salen de sus instalaciones. Puede ser *on-premise*, si se ubica en las mismas instalaciones del proveedor, u *off-premise*.

Nube pública. Los servicios que ofrece la nube están disponibles para uso público. El proveedor de la nube es el responsable de su mantenimiento y seguridad. La mayor ventaja de este tipo de *cloud* es que el cliente dispone de capacidad de proceso y almacenamiento sin tener que acometer una inversión inicial en infraestructura ni asumir su mantenimiento. Otra ventaja es que el cliente paga por el uso de los recursos. Tiene como inconveniente la exposición de la información a una tercera empresa.

Nube híbrida. Es la combinación de una nube privada y una nube pública, administradas como si se tratara de una única entidad, existiendo conectividad entre ellas, y pudiendo decidir en cada momento qué cargas críticas se manejarán en la nube privada y qué información se migrará a la nube pública. Un ejemplo típico es aquella empresa que transformó hace tiempo su infraestructura a una nube privada, y ahora se encuentra en la necesidad de respaldar ciertas aplicaciones críticas en una nube pública, o incluso que requiera de ciertas aplicaciones propietarias que deban ser alojadas en la nube de un determinador proveedor.

Multicloud. Se refiere a la utilización de al menos dos implementaciones de nube del mismo tipo (pública o privada) de distintos proveedores y todo gestionado de forma integral. La tendencia actual de las empresas es utilizar nubes híbridas o *multicloud* ya que con ellas se incrementa la seguridad y el desempeño del sistema. Con el *multicloud* se evita el inconveniente de depender de un único proveedor pues tenemos la posibilidad de trasladar cargas de un proveedor a otro.

Nube comunitaria. En este caso la infraestructura de la nube se comparte entre diversas organizaciones y empresas con objetivos e intereses similares.

3.1.4 Modelos de servicio

Los servicios en la nube se pueden ofrecer básicamente bajos tres tipos de modelos:

Software como Servicio (*SaaS*).

En este modelo de servicio, el proveedor de la nube pone sus aplicaciones e infraestructura a disposición de los consumidores. El proveedor se encarga de mantener y actualizar el hardware, sistemas operativos, aplicaciones, etc., mientras que los clientes simplemente usan las aplicaciones del proveedor.

Ejemplos de este modelo de servicio son los que ofrece *Gmail, Microsoft Office 365, WordPress*.

Plataforma como Servicio (*PaaS*)

En este caso, el proveedor pone tanto la infraestructura como las herramientas necesarias para que los usuarios únicamente se enfoquen en implementar y desarrollar sus aplicaciones (sistema operativo, gestor de bases de datos, etc.). Este es el entorno preferido por los desarrolladores pues se desentienden completamente de los servidores, del sistema operativo, de sus actualizaciones, las dependencias, etc. Ejemplo de servicios *PaaS* son los que ofrece *Google* con *Google Cloud*, *Red Hat* con *OpenShift*, etc.

Este TFM se enfoca en el modelo *PaaS*, en el cual los usuarios desarrollan sus aplicaciones (directamente o a través de terceros) y la plataforma *cloud* proporciona el entorno de implantación, desde el motor de contenedores hasta la propia infraestructura. Veremos como, por ejemplo, *Google Cloud* proporciona un *cluster* de *Kubernetes* con un motor de contenedores como *Docker* o *Containerd*, mientras que las aplicaciones que se cargan en los contenedores las desarrollan los propios usuarios.

Infraestructura como Servicio (*IaaS*)

En esta modalidad de servicio, el proveedor de la nube suministra la infraestructura, el “hierro”, que pone a disposición de los clientes a cambio de una cuota o alquiler, es decir, el proveedor ofrece capacidad de proceso (*CPU*), espacio de almacenamiento (memoria), máquinas virtuales, cortafuegos, balanceadores, etc.

En la siguiente figura se aprecia una comparativa entre los diferentes modelos de servicios, incluyendo una solución *on-premise* (primera columna). El color azul se marcan las tareas que son responsabilidad del usuario, y en verde las que corresponden al proveedor de la nube.

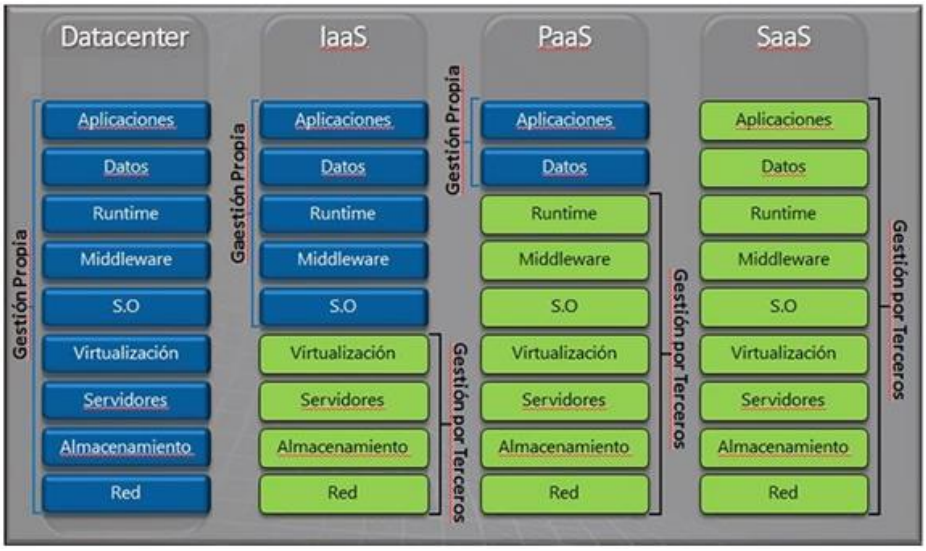


Figura 7. Comparación de los modelos de servicios [5]

3.2 Componentes básicos

Como ya se dijo previamente, la infraestructura de la nube consiste en una serie de sistemas *hardware* físicos repartidos en distintas áreas geográficas llamados *Data Centers*, que alojan la infraestructura virtualizada de los usuarios. En este apartado se hará una breve descripción de más elementos básicos de un entorno *cloud* relacionados con el despliegue de aplicaciones.

3.2.1 Nodo

Un nodo es un servidor o máquina de trabajo, física o virtual, con un sistema operativo y con suficiente capacidad de procesamiento (*CPU*) y memoria como para que los clientes puedan ejecutar sus aplicaciones. Los nodos se agrupan en un *cluster* y cada uno de ellos se identifica con una dirección IP privada (interna) y pública (externa).

Los nodos de un *cluster* pueden ser de dos tipos:

- Nodos *master*. Son los nodos que deciden qué se ejecuta en cada nodo *worker* del *cluster*. Planifican las cargas (*workloads*) de trabajo como, por ejemplo, las aplicaciones alojadas en los contenedores, y administran el ciclo de vida de dichas cargas, su escalado y las actualizaciones. Puede existir, por redundancia, más de un nodo *master* en los *cluster* de alta disponibilidad. Los nodos *master* forman "*plano de control*" del *cluster*.

- Nodos *worker*. Son las máquinas que ejecutan las aplicaciones alojadas en contenedores, así como otras cargas de trabajo. Disponen de un motor o *runtime* para gestionar estos contenedores. Estos nodos forman el "*plano de carga*".

3.2.2 *Cluster*

Un *cluster* es una entidad constituida por un conjunto de nodos que trabajan coordinadamente, y que se conectan a través de una red de alta velocidad, compartiendo recursos hardware y software. Como se ha dicho, se compone de nodos *master*, que controlan el *cluster*, y nodos *worker*, que ejecutan las aplicaciones.

Una vez configurado el *cluster* es posible abstraerse de su infraestructura, es decir, todo el *cluster* se convierte en el espacio de trabajo donde se ejecutan las aplicaciones sin importar en qué nodo se han alojado ni las características del mismo. El nodo *master* es quién elige el mejor nodo para alojar las cargas según los requisitos de la aplicación y de los recursos disponibles en cada nodo.

Las ventajas que se obtiene con la implementación de un *cluster* son:

- Disponibilidad muy alta. Si un nodo falla el resto mantendrá la funcionalidad.
- Alta velocidad, pues la carga se reparte entre distintos nodos.
- Balanceo de carga, lo que reduce la sobrecarga en los nodos.
- Escalabilidad horizontal, permitiendo añadir nodos adicionales en caso de un incremento de la demanda.
- Limitación del ataque por denegación del servicio (*DDoS*), pues al disponer de nodos escalables es más difícil inundar el servicio con peticiones fraudulentas.

3.2.3 Contenedor

Un contenedor es la unidad más pequeña de computación en un *cluster* donde se puede desplegar una aplicación. Además, aloja todos los recursos necesarios para poderse implementar en cualquier *hardware* con un determinado *kernel* de sistema operativo (*SO*). En lugar de incluir un *SO* completo, como en las máquinas virtuales, los contenedores incorporan un *SO* sin el *kernel*, que es el componente más pesado del *SO*. Al ser más livianos que las máquinas virtuales son muy rápidos y, además, evitan incompatibilidades tanto del *hardware* como del *software* del *host*. Los contenedores son efímeros, es decir, al detenerlos se pierden todos los datos que se hubieran guardado en ellos, por tanto, los contendores no se pueden reiniciar.

Aparte de ser más ligeros que las *VM*, otra característica importante de los contenedores es que son portables, es decir, una vez creado un contenedor en una máquina con un determinado *kernel* del sistema operativo, éste puede correr en cualquier otra máquina con cualquier otro sistema operativo que use ese mismo *kernel*, y puesto que el *Kernel* de *Linux* es el mismo para cualquier distribución de este *SO*, la portabilidad entre máquinas *Linux* es total, es decir, podemos crear un contenedor en una máquina *Ubuntu* y luego ejecutarla en otra máquina *Fedora*.

Respecto al número de procesos que es aconsejable correr en un contenedor, lo ideal es uno, de forma que podamos administrar cada uno de estos procesos independientemente, como el escalado, uso de *CPU*, memoria, etc. Los contenedores suelen tener unos recursos mínimos y máximos previamente asignados en el *host* donde se alojan como *CPU*, *RAM*, etc.

En un contenedor podemos instalar, por ejemplo, un servidor *Apache*, una máquina con una base de datos *MySQL*, una máquina con un sistema operativo *Ubuntu*, etc., o como se verá en este caso práctico, un servidor *Nodejs*.

3.2.4 Imagen

Una imagen es una plantilla de solo lectura que empaqueta todo el *software* necesario para poder ejecutarse en un contenedor, tanto el propio código de la aplicación como todas las dependencias necesarias (librerías, etc.). Podemos considerar que es el *software* que va a correr en el propio contenedor, de hecho, se dice que un contenedor es una *instancia* de la imagen con las que se ha creado. Existen imágenes de un servidor *Apache*, un servidor *Nginx*, una máquina *Ubuntu*, un servidor con una base de datos *Redis*, etc.

Para crear una imagen de una determinada aplicación se dispone básicamente de tres opciones:

- Crearla con *Docker*, a partir de un fichero de texto plano denominado *Dockerfile*, donde se indican las instrucciones (comandos) necesarias para generar dicha imagen. El proceso se indica en la siguiente figura:

Figura 8. Proceso de creación de una imagen con *Docker* (elaboración propia)

- Crearla sin *Docker,* a partir de una imagen base (”*Image Builder*”) usando herramientas del tipo *S2I* (“*Source to Image*”), como se hace en *OpenShift*.

Figura 9. Proceso de creación de una imagen sin *Docker* (elaboración propia)

- Descargándolas de un registro de imágenes o repositorio. De esta manera se fomenta el intercambio de imágenes entre los diferentes desarrolladores. El repositorio oficial de *Docker* es *Docker Hub*. En su página oficial [19] se encuentran todas estas imágenes, cada una con distintas versiones o *tags*, por lo que se recomienda siempre especificar una versión al descargarlas. A la última versión de una imagen en un repositorio también se le da el nombre de “*latest*”.

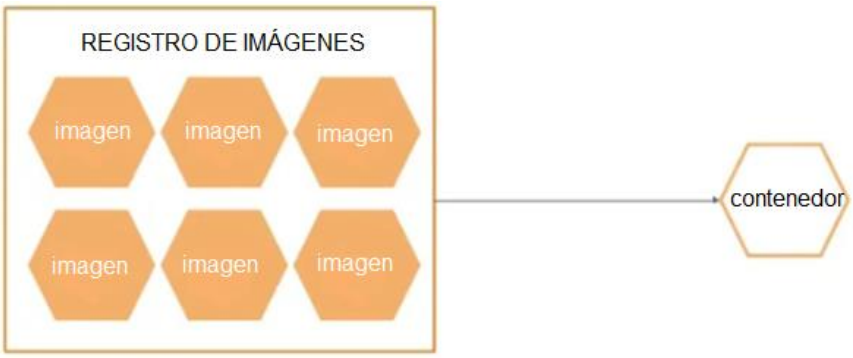


Figura 10. Registro de imágenes [24]

Las imágenes de los contenedores se estructuran en unidades más pequeñas denominadas "*capas*", o *layers*, que pueden compartirse y reutilizarse internamente para crear otras imágenes. De esta forma, al generar una imagen solo se descargarán ciertas capas si el resto ya se descargaron con otras imágenes que contenían esas mismas capas. Esto hace que el sistema sea muy eficiente ya que reduce el tiempo de descarga de las imágenes y el espacio de almacenamiento de las mismas. En la figura de abajo se observa cómo los tres contenedores comparten la capa 2, mientras que los contenedores A y B comparten la capa 1. Si la aplicación del contenedor A, por ejemplo, necesita modificar un fichero de una de las capas, realiza previamente una copia de dicho fichero en el propio contenedor, modificando posteriormente esta copia. De la misma forma, cuando un contenedor elimina un fichero compartido en alguna capa, lo registra localmente, sin que el resto de los contenedores se vean afectados. De esta forma se garantiza el aislamiento entre contendores. Este procedimiento se denomina *CoW* (*Copy On Write*).

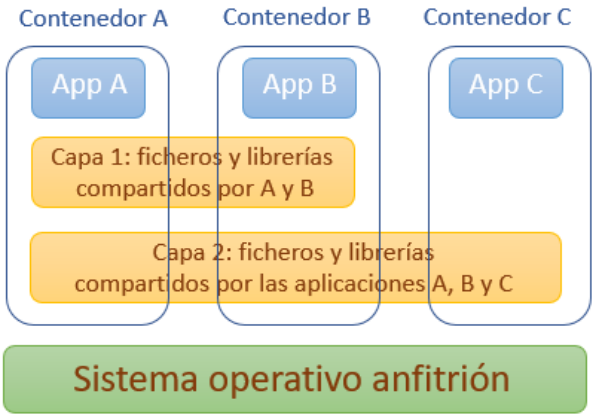


Figura 11. Contenedores compartiendo capas de imágenes (elaboración propia)

Por último, hay que destacar que, si se construye una imagen en un *host* que tiene un determinado *Kernel* del sistema operativo, puede que se generen problemas al levantar un contenedor con dicha imagen en otra máquina que tenga un *kernel* distinto (desactualizado, etc.). Evidentemente no es posible iniciar un contenedor en un *host* de *Windows* con una imagen creada en un *host* de *Linux* ya que estos *kernel* son incompatibles. Todos los sistemas

operativos *Linux* tienen el mismo *kernel* por lo que no deben darse problemas de este tipo, aunque se usen distribuciones de *Linux* distintas, siempre que el *kernel* esté actualizado.

3.2.5 *Pod*

Un *pod* es un recurso lógico de *Kubernetes* que agrupa una serie de contenedores que intervienen en la ejecución de una determinada aplicación. En *Kubernetes* no se pueden arrancar, parar o escalar contenedores de forma directa como ocurre en *Docker*, en su lugar se opera con los *pods*. Un *pod* puede agrupar a un único contenedor o, en escenarios más complejos, a varios contenedores alojados en un mismo nodo, pero con la restricción de que los contenedores de un mismo *pod* deben alojarse en el mismo nodo, y de que todo contenedor debe alojarse en un *pod*. Todos los contenedores de un *pod* comparten la misma la dirección *IP*, y aunque cada contenedor tiene sus propios puertos, éstos son accesibles desde cualquier contenedor del *pod*, incluso desde el propio *pod*. Se debe prestar atención de no asignar un mismo puerto a dos contenedores del mismo *pod*.

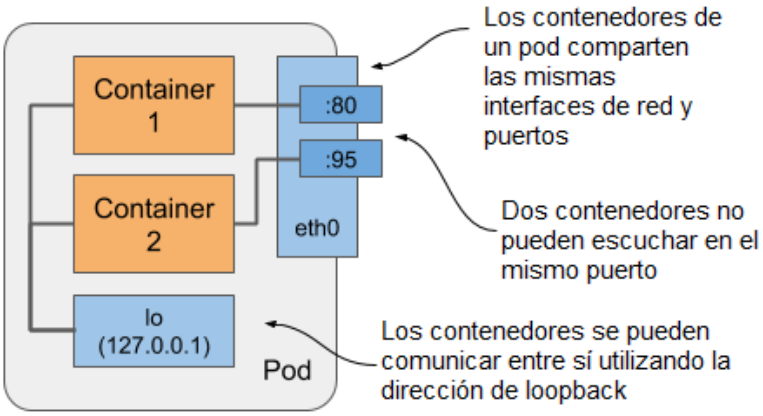


Figura 12. Contenedores de un *pod* compartiendo interfaces de red [6]

Los *pods* de un *cluster* se pueden comunicar entre ellos incluso aunque estén en nodos diferentes. Esto se realiza ejecutando en cada nodo un agente llamado *CNI* (*Container Network Interface*) que crea una *VPN* (*Virtual Private Network*) entre los pods. Este agente se encarga de crear rutas IP para que desde cualquier IP de un *pod* se pueda llegar a la *IP* de cualquier otro *pod*.

Los contenedores están diseñados para ejecutar un solo proceso, por tanto, cuando una aplicación consta de varios procesos es conveniente que cada uno de ellos se aloje en un contenedor distinto, agrupándolos en un *pod*. Tampoco es conveniente alojar todas las aplicaciones de un sistema en un mismo *pod*, siendo preferible incluir en un *pod* únicamente los procesos de las aplicaciones que estén relacionados. Por ejemplo, si en una aplicación tanto el *front-end* como el *back-end* lo ejecutamos en el mismo *pod*, entonces ambos estarán obligados a alojarse en el mismo nodo, desequilibrando las cargas en los nodos del *cluster*. Otra ventaja de separarlos en *pods* es que si la carga del *front-end* es distinta al del *back-end* podemos replicarlos por separado, mientas que si están en el mismo *pod* se replicarán a la vez. Esto se debe a que un *pod*

es una unidad básica de escalado, es decir, *Kubernetes* no replica contenedores dentro de un *pod*, sino todo el *pod*.

Los *pods* tienen un ciclo de vida definido, que comienza con la fase "*Pending*", progresa a la fase "*Running*" si al menos uno de sus contenedores principales se inicia normalmente, y luego a las fases "*Succeeded*" o "*Failed*", dependiendo de si algún contenedor en el *pod* terminó en fallo. El ciclo de vida de un *pod* se puede apreciar en la siguiente figura:

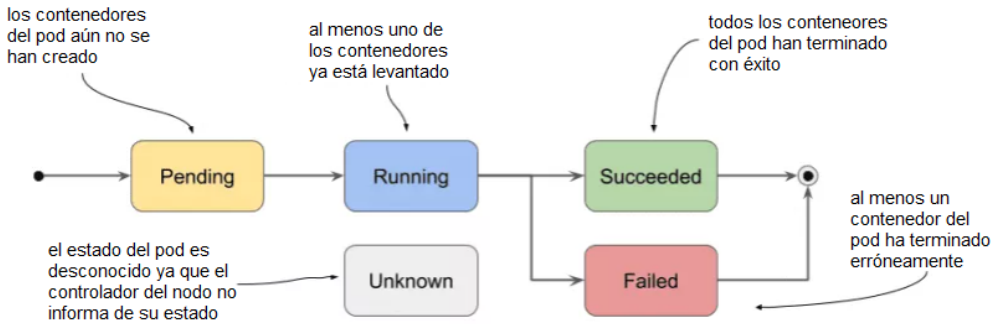


Figura 13. Ciclo de vida de un *pod* [31]

Los estados principales de un pod son los siguientes:

- *Pending*: el *pod* ya se ha creado, pero aún no se han levantado sus contenedores ni se ha decidido en qué nodos se alojarán.
- *Running*: los contenedores del *pod* ya se han creado y el *pod* ya se está ejecutando en un nodo. Al menos un contenedor del *pod* se está ejecutando o está en proceso de creación. El pod continua en "*running*" hasta que se completa con éxito o se termina.
- *Succeeded*: todos los contenedores del *pod* han finalizado su ejecución de forma satisfactoria.
- *Failed*: todos los contenedores del *pod* han finalizado su ejecución y al menos uno de ellos lo ha hecho con error.
- *Unknown*: el estado del *pod* es desconocido.

Por último, cuando se crea un *pod* se puede solicitar una demanda de recursos como memoria o *CPU*, más concretamente, tanto una garantía mínima de recursos del *pod*, como un límite máximo del uso de dichos recursos.

3.2.6 Runtime

Un *runtime*, rutina o motor de contenedores es el *software* que permite ejecutar contenedores en un nodo, cargar las imágenes desde un repositorio, controlar el ciclo de vida de los contenedores, etc.

El motor de *Docker* se denomina *Docker Engine* y se compone de un *runtime* de bajo nivel, llamado *runC*, que se encarga de crear y ejecutar contenedores, y de un *runtime* de alto nivel, llamado *Containerd*, que ofrece un conjunto completo de funciones adicionales.

Kubernetes utiliza por defecto el motor *Containerd* desde la versión 1.20, aunque puede trabajar con cualquiera que cumpla el estándar *CRI* (*Container Runtime Interface*) como, por ejemplo, *CRI-O*. El cliente de comandos de *Kubernetes*, llamado *kubectl*, se comunica con el *runtime* a través del *CRI*.

OpenShift, en versión v4, utiliza el motor de contenedores *CRI-O* y *Kubernetes* como orquestador. *OpenShift* versión v3, que es la que actualmente utiliza *MiniShift*, usa el motor de *Docker*.

3.2.7 Orquestador

Los *runtime* funcionan junto a los orquestadores de contendores. El orquestador es la herramienta que administra el *cluster*, siendo responsable, por ejemplo, de escalar las aplicaciones cuando es necesario. Mientras que el *runtime* se encarga de la gestión de los contenedores, el orquestador gestiona todo el *cluster*. Los motivos por los que se ha de utilizar un orquestador quedan muy bien reflejados en [29].

Las funciones principales de un orquestador son las siguientes:

- Reparto o balanceo de carga entre contenedores, asegurando que cada contenedor se ejecute en un nodo con recursos suficientes.
- Redistribución de procesos tras la caída de un nodo trasladando la carga (contenedores o *pods*) al resto de los nodos.
- Gestión de una red interna para la comunicación entre contenedores independientemente de en qué nodo se encuentren.
- Monitorización de la carga de los contenedores comprobando su estado de salud.
- Escalado y desescalado de contenedores, adaptando los recursos al nivel de carga exigido.
- Descubrimiento de servicios, asignando los *pods* y contenedores que deben atender cada servicio.
- Almacenamiento de datos en volúmenes persistentes.

Las plataformas de orquestación de contenedores más populares son *Kubernetes* (también llamado *k8s* o *Kube*), *Docker Swarm* y *OpenShift*. *Docker Swarm* es la herramienta de *Docker* con la que podemos crear y orquestar un *cluster*, *Kubernetes* es una plataforma de orquestación siendo actualmente un estándar de facto [8], además de ser líder del mercado para la implementación de contenedores, y *OpenShift* es una plataforma de orquestación orientada al sector empresarial en entornos de nube híbrida.

3.3 Plataformas de contenerización

En este apartado se hará una breve introducción de las tres de las plataformas de contenerización más utilizadas actualmente.

3.3.1 *Docker*

Aunque las bases de la contenerización se iniciaron en el año 1979 con el sistema *chroot*, no fue hasta el año 2008 cuando se popularizó con la aparición de *Docker*. Se trata de una plataforma de código abierto que simplificó la automatización de los despliegues de aplicaciones alojadas en contenedores, así como la portabilidad de los mismos. Con *Docker* se pueden crear las imágenes de las aplicaciones, y compartirlas con la comunidad de desarrolladores de una manera muy sencilla mediante un repositorio público denominado *Docker Hub*.

Docker es una plataforma que dispone, además de un motor de contenedores (o *runtime*) llamado *Docker Engine*, muchas otras funcionalidades como, por ejemplo, *Dockerfile* para la generación de imágenes, *Docker Compose* para la creación y gestión compartida de múltiples contenedores, *Docker Machine* para la creación de nodos (tanto en local o en la nube), y un orquestador llamado *Docker Swarm*.

Docker proporciona tanto un terminal de comandos (*CLI*) denominado *Docker CLI* o simplemente "*docker*", como el propio proceso de *Docker* llamado "*dockerd*" (a este proceso también se le llama "demonio" o "*daemon*"). Aunque se puede trabajar directamente con el demonio de *Docker* a través de una *API*, es más sencillo hacerlo por medio del terminal de comandos "*docker*", como se realizará en este trabajo.

Por defecto, *Docker* ejecuta los contenedores con el usuario "*root*" (*UID*=0) lo que representa una brecha de seguridad. En efecto, un usuario malicioso con permisos para correr contenedores *Docker* (añadiendo su usuario al grupo de usuario "*docker*" de *Linux*) podrá levantar contenedores, como "*root*". Si en uno de estos contenedores monta un volumen de parte del sistema de archivos del *host*, podremos acceder a estos ficheros del *host*, aunque estén protegidos a usuarios que no sean "*root*". Este es uno de los motivos por los que *Kubernetes* ha dejado de usar el motor de contenedores de *Docker*. Existen ya soluciones para evitar este problema, como es "remapear" los identificadores de los usuarios (*UID*) de forma que éstos corran los contenedores con un *UID* distinto al 0 (es decir, distinto a "*root*") por lo que no se tienen estos privilegios en el *host*.

3.3.2 *Kubernetes*

Kubernetes (o también llamado *k8s* o *Kube*) es una plataforma de código abierto para la implementación y gestión de aplicaciones contenerizadas a gran escala. Esta tecnología apareció en el año 2014 de la mano de *Google* para solucionar los problemas de *Docker* con la seguridad y la administración de contenedores a gran escala, siendo actualmente un estándar de facto para la orquestación de contenedores, de hecho, está considerado como un sistema operativo de contenedores. Una desventaja de *Kubernetes*, frente a *Docker* y *OpenShift*, es que no dispone de un sistema propio para la generación de imágenes.

Kubernetes puede utilizar cualquier *runtime* de contenedores que cumpla la especificación *CRI* (*Container Runtime Interface*), como *CRI-O* y *Containerd*.

Inicialmente *Kubernetes* utilizaba el *runtime* de *Docker*, pero a partir de la versión v1.20 dejó de hacerlo debido, entre otros motivos, a los complementos adicionales que trae *Docker* y que ocasionan ciertos problemas de seguridad a *Kubernetes*.

En la siguiente figura podemos ver la arquitectura de *Kubernetes*:

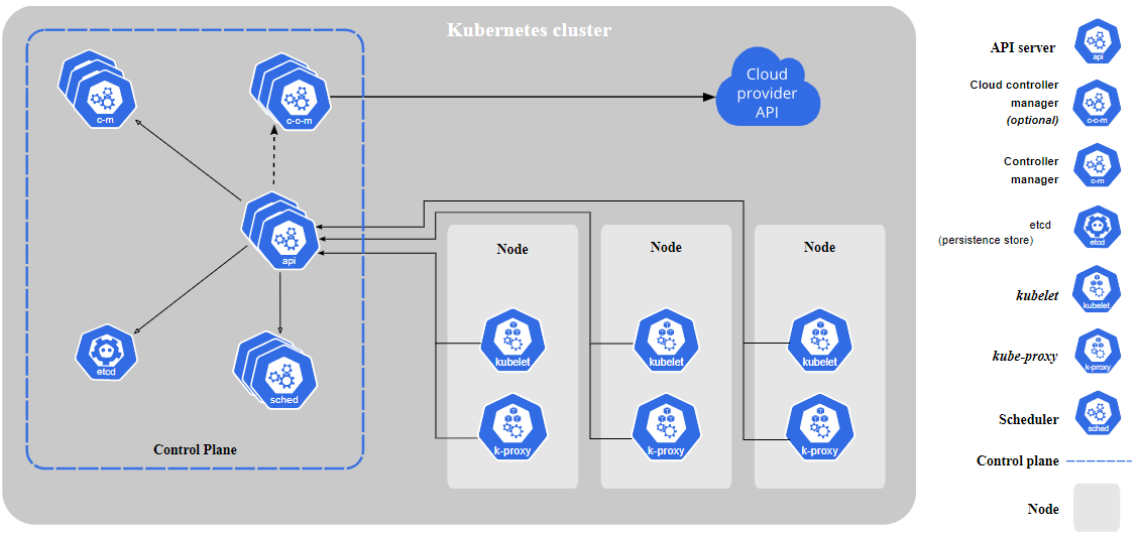


Figura 14. Componentes de *Kubernetes* [9]

Los componentes básicos de *Kubernetes* son los siguientes [10]:

- Un *Control Plane*. Hace las funciones del nodo *master* y se compone de:
    - Un módulo *API server*. Se encarga de atender las peticiones de los usuarios, por ejemplo, a través de comandos *kubectl*.
    - Un *Scheduler*. Se encarga de asignar los *pods* a los nodos según los requerimientos de la aplicación y los recursos disponibles.
    - Un *Controller*. Se encarga de controlar el estado del *cluster* y de realizar tareas rutinarias, como mantener el número de réplicas de los *pods*, monitorizar el estado de nodos, etc.
    - Un *Etcd*. Es una base de datos, de tipo clave-valor, en la que se guarda el estado del *cluster*, los nodos y contenedores donde se ejecutan las aplicaciones, resolución *DNS*, etc. El resto de los componentes del *cluster* son “sin estado” por lo que pueden reiniciarse o sustituirse en caso de fallo sin afectación del servicio.
    - Un *Cloud Controller Manager*. Se encarga de conectarse a la *API* del proveedor cloud para crear un balanceador, añadir un disco, etc.

- Los nodos *worker*: Forman el plano de carga y se componen de:
    - Un *Kube-proxy*. Se encarga de recibir el tráfico y enviarlo a los *pods* adecuados.
    - Un *Kubelet*. Es el controlador del nodo. Se encarga de la comunicación con el nodo *master* para recibir las órdenes que

debe ejecutar. También monitoriza sus contenedores y los reinicia en caso necesario.

Kubernetes basa el despliegue de aplicaciones en un conjunto de objetos definidos en ficheros de texto "*yaml*" o "*yml*", que se envían a la *API* a través de un terminal de comandos llamado *kubectl*.

3.3.3 *OpenShift*

OpenShift es una plataforma empresarial de orquestación creada por la empresa *Red Hat*, basada en *Kubernetes*, pero con una mejor experiencia de usuario. Permite desarrollar, construir, implementar y administrar grandes aplicaciones en el *cloud* de manera ágil. *OpenShift* utiliza internamente *Kubernetes*, pero introduce muchas mejoras adicionales como, por ejemplo, la utilización de *S2I* (*Source to Image*) para la creación automática de imágenes a partir del código fuente de la *aplicación*. Ofrece un mayor nivel de seguridad y mejor rendimiento que *Kubernetes* por lo que las grandes organizaciones confían en *OpenShift* para alojar sus aplicaciones.

Existen dos formas básicas de comunicarse con el *cluster* de *OpenShift*: mediante una consola *web* muy funcional y amigable, y a través de una interfaz de comandos "*oc*", más orientada para desarrolladores.

Los productos *OpenShift* más destacados son los siguientes:

- *OKD* (antes denominado *OpenShift Origin*): es un *software* libre que permite montar un *cluster* de *OpenShift* en una infraestructura propietaria (*on-premise*). Actualmente está en versión 4, es decir, *OKD4*. Es gratuito y, por tanto, no se tiene soporte de *Red Hat* ni se pueden usar las imágenes oficiales de *Red Hat*. La instalación se realiza sobre los sistemas operativos *Red Hat Linux* (*RHEL*) o *Centos*.

- *OpenShift Online*: este producto permite montar un *cluster* de *OpenShift* en un *cloud* público de manera autogestionada. De pago.

- *OpenShift Dedicated*: permite montar un *cluster* de *OpenShift* en un *cloud* de *AWS* o *Google Cloud*, pero gestionado y mantenido por *Red Hat*, es decir, es esta última empresa quien construye el *cluster*. De pago.

- *OpenShift Container Platform*: permite montar un *cluster* de *OpenShift* en una infraestructura propietaria (*on-premise*) con soporte de *Red Hat*. De pago.

- *MiniShift*: es un *OpenShift* en versión 3 y proporciona un *cluster* de un solo nodo en una máquina local. Está destinada a fines educativos y de laboratorio. Se puede instalar sobre una máquina con sistema operativo *Windows* y los hipervisores *VirtualBox* o *Hyper-V*. Es gratuito.

- *CRC* (*CodeReady Containers*): es un *OpenShift* en versión 4 y proporciona un *cluster* de un solo nodo en una máquina local. Destinada a fines educativos y de laboratorio. No es compatible con *Windows 10 Home Edition*. Es gratuito.

En este trabajo se utilizará la versión *MiniShift* por los siguientes motivos:

- *OKD*, aunque es un software libre, requiere de una infraestructura *on-premise* de la que no se dispone.

- *OpenShift Online*, *OpenShift Dedicated* y *OpenShift Container Platform* se soportan únicamente sobre un sistema operativo de *Red Hat* (*RHEL*) no permitiendo otras distribuciones de *Linux* como *Debian* o *Ubuntu*, además de no ser gratuitos.

- *CRC* no es compatible con *Windows 10 Home Edition*, sistema operativo del equipo con el que se está realizando este TFM. También se ha descartado ya que, *CRC* virtualiza el nodo del *cluster* en la máquina local donde se instala, que en este caso también está virtualizada, por lo que se produciría una virtualización anidada, que no es recomendable.

3.4 Resumen

El paradigma *cloud* es el entorno ideal para que las empresas optimicen sus recursos y puedan abordar sus proyectos de transformación digital. A medida que el diseño del *software* de las aplicaciones se ha vuelto cada vez más complejo, se ha impuesto una arquitectura enfocada en microservicios, basada en la descomposición del sistema en pequeñas piezas *software* que se comunican mediante unas determinadas *APIs*. La contenerización es la solución óptima para alojar aplicaciones desarrolladas bajo la arquitectura de microservicios al garantizar la autonomía de cada uno de ellos con un consumo mínimo de recursos, tanto de *CPU* como de memoria.

De las distintas modalidades de *cloud*, la nube privada está destinada a aquellas organizaciones que desean tener un control total de sus datos y aplicaciones, a cambio de tener que hacer frente al coste de su infraestructura. La nube pública es la mejor opción para las organizaciones que necesitan capacidad de proceso y almacenamiento sin tener que asumir inversión inicial en infraestructura, pagando únicamente por los recursos consumidos. La nube híbrida es adecuada para las organizaciones que deseen, por un lado, un control de ciertas funcionalidades que alojarán en una infraestructura propietaria (nube privada), y por otro, un respaldo de ciertas aplicaciones que las desplegarán en la infraestructura de un proveedor *cloud* (nube pública).

El modelo ideal para el despliegue de aplicaciones en entornos *cloud* es el de plataforma como servicio, *PaaS*. En este modelo el proveedor de la nube ofrece una plataforma donde sus clientes consumen servicios y funciones que necesitan para poner en producción las aplicaciones, delegando en la propia

plataforma el lugar donde se ejecutarán las aplicaciones (nodos, contenedor, etc.), así como su mantenimiento y escalado.

La infraestructura del *cloud* se compone de un determinado número de *Data Centers* donde los clientes definen unas entidades llamadas *clusters*, compuestas por unos nodos virtualizados que dan cabida a los contenedores. Son estos contenedores donde finalmente se instalan los distintos microservicios que componen las aplicaciones. Dependiendo de la plataforma de contenedores, éstos a su vez se pueden agrupar en unas entidades denominadas *pods* permitiendo de esta forma el escalado simultáneo de contenedores.

Una imagen es una plantilla que contiene todo lo necesario para que una aplicación pueda ejecutarse de forma autónoma en un contenedor, tanto su propio código como sus dependencias (librerías, etc.). Las imágenes son portables en el sentido de que se puede ejecutar en cualquier *host* siempre que éste tenga el mismo *kernel* de sistema operativo que la máquina donde se han creado dichas imágenes.

Al *software* que permite crear y ejecutar contenedores en un *host* se le denomina motor (o *runtime*) de contenedores. Cuando el número de contenedores de una aplicación es alto, su administración se hace muy compleja pues requiere coordinarlos, supervisarlos, escalarlos, etc. El orquestador es el *software* que automatiza la administración de los despliegues, velando por la salud de los contenedores, monitorizándolos, y escalándolos para adaptarse la carga que deben soportar las aplicaciones, en definitiva, administrando los microservicios de esta arquitectura distribuida.

El paradigma de la contenerización se popularizó sobre el año 2008 con la plataforma *Docker* debido a la sencillez de uso de su interfaz de comandos para el manejo de contenedores, de su diseño de imágenes organizadas en capas, y del éxito de su biblioteca de imágenes (repositorio).

Al ritmo que las aplicaciones se han ido haciendo cada vez más complejas, el número de contenedores del sistema ha ido creciendo, y con ello, la necesidad de la orquestación del sistema. *Docker* dispone de un orquestador denominado *Docker Swarm* pero con unas funcionalidades muy limitadas, de hecho, actualmente su desarrollo está prácticamente congelado. *Kubernetes*, sin embargo, permite la orquestación de un número muy elevado de contenedores con un mayor nivel de seguridad, lo que le ha valido para ser actualmente el orquestador estándar de facto en la actualidad. *OpenShift* es una distribución de *Kubernetes* en el entorno empresarial, que permite una administración de contenedores muy ágil y amigable, mayores funcionalidades que *Kubernetes*, y unos niveles de seguridad superiores a éste. *OpenShift* es la plataforma más adecuada para despliegues sobre una nube híbrida, entorno considerado por muchos autores [38] como el ideal para que las grandes organizaciones desplieguen sus aplicaciones alojadas en el *cloud*.

4. Puesta en práctica

En este capítulo se describirán todos los trabajos que son necesarios para desplegar una aplicación en las tres plataformas de contenerización: *Docker*, *Kubernetes* y *OpenShift*, así como la realización de distintas pruebas de orquestación.

El despliegue de una aplicación en una plataforma de contenedores alojada en *Google Cloud* incluye una serie de procesos relacionados, tal como se puede observar en la siguiente figura. Las flechas indican el orden en las que han de llevarse a cabo.

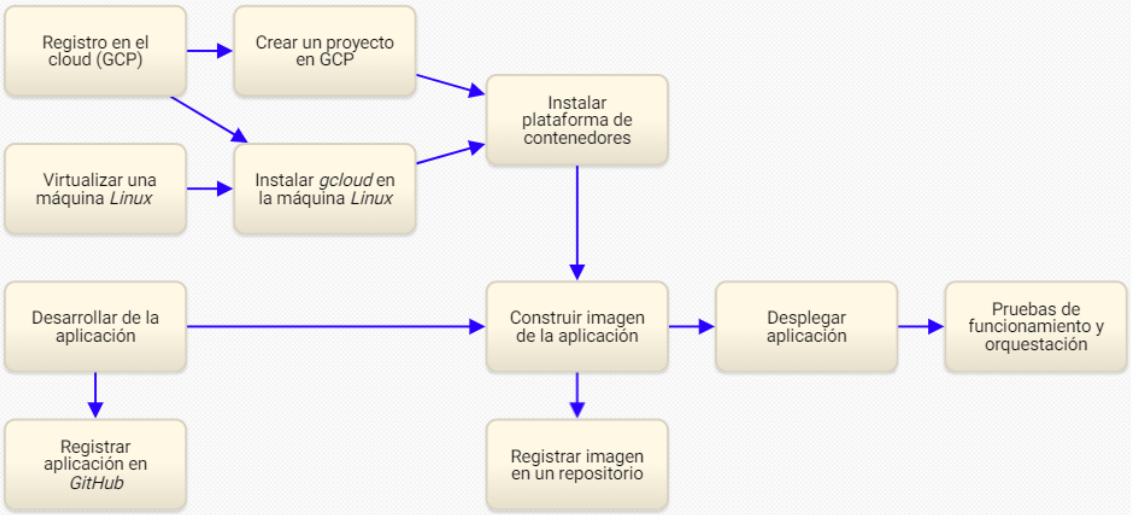


Figura 15. Procesos despliegue en un entorno *cloud* (elaboración propia)

En los siguientes apartados se detallarán estos procesos, aunque aquí se realiza un breve resumen:

Registro en una nube pública. Se realiza un registro en la plataforma del proveedor *cloud* (*Google Cloud*, *Azure*, *AWS*, etc.).

Crear un proyecto en la plataforma *cloud*. Este proyecto englobará todos los recursos que se utilizarán en el *cluster* como, por ejemplo, máquinas virtuales, *firewall*, etc.

Virtualizar una máquina *Linux*. Se crea una máquina virtual *Linux* en la máquina local *Windows* con *VirtualBox* debido a que *Windows* es un sistema operativo que no está muy bien soportado por las actuales plataformas de contenerización.

Instalar *gcloud* en la máquina *Linux*. Esta herramienta permite la conexión de la máquina local *Linux* con el *cluster* de *Google Cloud*.

Instalar plataforma de contenedores. Este proceso conlleva la instalación y configuración de cualquiera de las plataformas de contenerización: *Docker*, *Kubernetes* u *OpenShift*.

Desarrollar la aplicación. Aquí se codifica la aplicación para posteriormente alojarla en los contenedores de las distintas plataformas.

Registrar aplicaciones en *GitHub*. Este es el repositorio donde se almacenará el código de la aplicación.

Construir la imagen de la aplicación. Este proceso tiene como objetivo crear una imagen a partir del código de la aplicación y de sus dependencias.

Registrar imagen en un repositorio. Una vez creada la imagen de la aplicación se ha de registrar en un repositorio como *Docker Hub*.

Desplegar la aplicación. Son los trabajos necesarios para poner en producción la aplicación en una plataforma de contenedores.

Pruebas de funcionamiento y orquestación. Aquí se realizarán las pertinentes pruebas de funcionamiento y orquestación para evidenciar tanto la simplicidad de administración de las aplicaciones, como la disponibilidad, escalabilidad, y balanceo de carga que ofrecen estas tecnologías.

4.1 Trabajos previos

En este apartado se describen las tareas previas necesarias para poder iniciar la instalación y configuración de una plataforma de contenedores en *Google Cloud*. También incluye el desarrollo de la aplicación que se utilizará posteriormente tanto para su despliegue como para las pertinentes pruebas de orquestación.

Usando como base la figura 15, en la siguiente imagen se pueden apreciar, en fondo resaltado, los procesos correspondientes a estos trabajos previos:

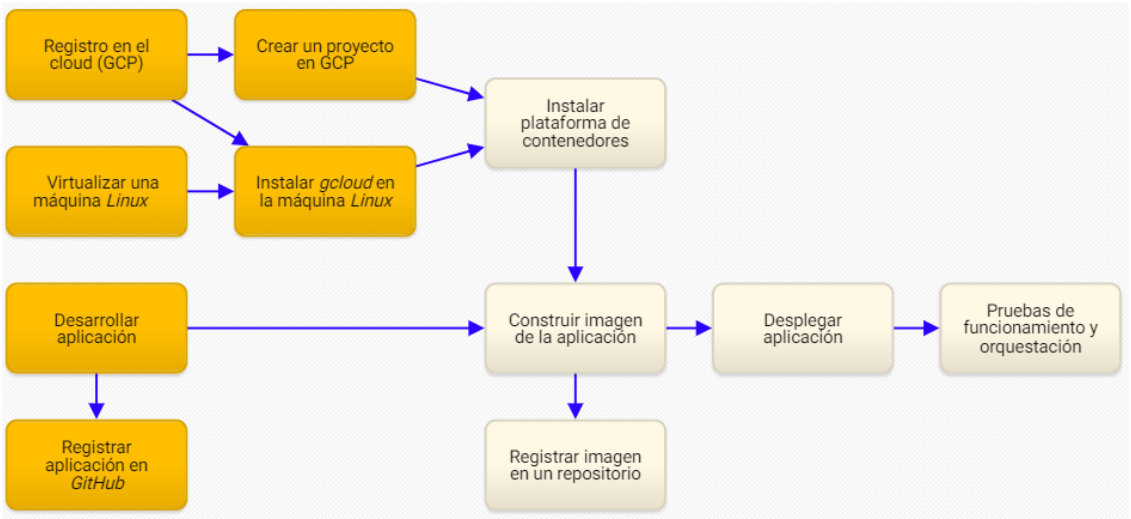


Figura 16. Procesos previos al despliegue (elaboración propia)

4.1.1 Registro en la nube pública *Google Cloud*

De los distintos proveedores *cloud* del mercado (*Microsoft Azure, Amazon Web Service, Google Cloud, Alibaba Cloud, IBM, Salesforce, Oracle*, etc.) se ha elegido a *Google* por su alto nivel de seguridad y su gran experiencia. *Google Cloud* se compone de un conjunto de centros de datos (*Data Centers*) repartidos por Asia, Australia, Europa, América del Norte y América del Sur. *GCP* (*Google Cloud Platform*) es la plataforma o el portal *web* sobre el que *Google* ofrece directamente a sus usuarios los servicios en la nube.

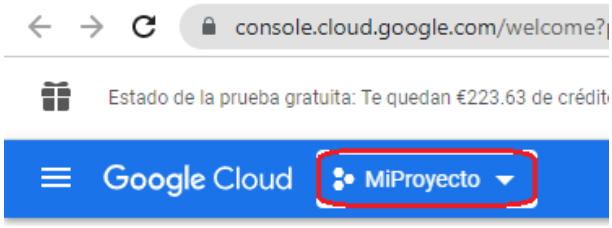
Los pasos que se deben seguir para el registro en *Google Cloud* son los siguientes:

1. Acceder a su portal web [11].
2. Seleccionar una cuenta de correo.
3. Introducir los siguientes datos personales:
    - País de procedencia
    - Tipo de proyecto (por ejemplo, "*Personal Project*")
    - Número de teléfono
    - Tipo de cuenta (por ejemplo, "*individual*")
    - Dirección postal
    - Número de una tarjeta de crédito, con el propósito para verificar la identidad. No se producirán cargos en el periodo de prueba.
4. Pulsar en "*START MY FREE TRIAL*".

Este registro "*trial*" proporciona acceso a *Google Cloud* durante un periodo de prueba gratis de 90 días con un límite de facturación de 300€. En caso de superar este límite el portal emitirá un aviso para que se autorice el cargo. En cualquier momento se puede visualizar tanto el crédito disponible como la fecha de finalización del periodo de prueba, seleccionando "*Facturación*" en el menú de navegación.

4.1.2 Creación del proyecto de *Google Cloud*

Un proyecto de *Google Cloud* es la entidad sobre la que se habilitan los servicios de *Google Cloud* como, por ejemplo, una máquina virtual, un *clúster*, un *firewall*, etc., y que se consumen en las aplicaciones. Por defecto, *Google Cloud Platform* ya proporciona un proyecto denominado "*My First Project*", aunque se pueden crear nuevos proyectos pulsando sobre el enlace "*PROYECTO NUEVO*" de la consola. Para trabajar sobre un determinado proyecto previamente hay que seleccionarlo del desplegable de proyectos, tal como se aprecia en la siguiente imagen:



4.1.3 Preparación de la máquina local *Windows*

En este apartado se va a configurar la máquina con la que se va a trabajar, un *PC* de *Windows* de 64 bits, con el fin de poder interactuar con *Google Cloud*. Se ha decidido virtualizar una máquina de *Linux* en el *host* de *Windows*, concretamente *Ubuntu*, dado que es un sistema operativo muy bien soportado por las plataformas de contenerización. Para acceder a los recursos de *Google Cloud* se debe instalar la herramienta "*gcloud*".

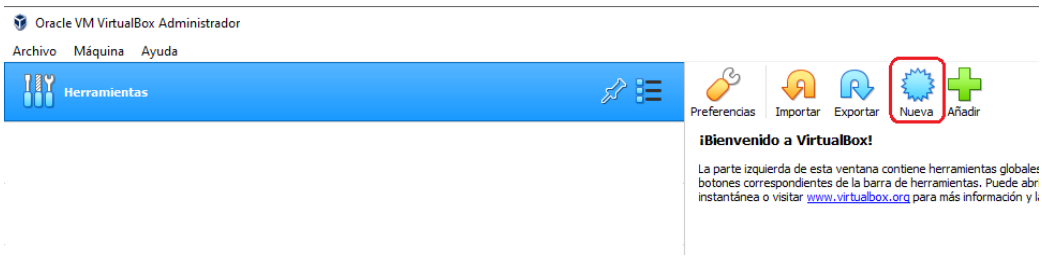
4.1.3.1 Instalación del hipervisor y virtualización de una máquina *Linux*

De las distintas versiones de *Ubuntu*, se ha optado por *Lubuntu* dado que es muy ligera y, por tanto, adecuada para utilizarla en contenedores. La herramienta de virtualización elegida es *VirtualBox* al ser uno de hipervisores más utilizados actualmente.

La imagen "*iso*" del sistema operativo, por ejemplo, *Lubuntu 22.04.1 LTS* se puede obtener de [12] seleccionando el apartado "*Desktop 64-bits*".

La descarga de *VirtualBox* se puede realizar desde la página oficial [13] en el apartado "*Windows hosts*". Se descargará un fichero .exe" que posteriormente se debe ejecutar para iniciar la instalación de *VirtualBox*.

Una vez instalado *VirtualBox* se debe abrir la aplicación y crear una máquina virtual pulsando en "*Nueva*", como puede apreciarse en la siguiente imagen:



Seguidamente hay que introducir un nombre para la máquina virtual como, por ejemplo "*Lubuntu 22.04*", un tamaño de memoria de memoria RAM de al menos 4 *GBytes*, seleccionar "*Crear un disco duro virtual ahora*", tipo "*VDI (VirtualBox Disk Image)*", "*Reservado dinámicamente*", y con una capacidad de al menos 20 *GBytes*.

Se debe configurar el controlador gráfico de la *VM* de *VirtualBox* para que el monitor virtual se ajuste al tamaño de la pantalla. Para ello, se pulsa con el botón derecho del ratón sobre la máquina, se selecciona "*Configuración*", y en el apartado "*Pantalla*" se elige el controlador gráfico "*VBoxVGA*".

Para arrancar la máquina virtual se pulsa en el botón "*Iniciar*" del menú. Al ser la primera vez que se inicia, hay que seleccionar la imagen "*iso*" descargada previamente. Aparecerá un escritorio de *Lubuntu* con un icono con el nombre "*Install Lubuntu 22.04 LTS*" que se debe ejecutar para iniciar la instalación de *Lubuntu*. La configuración solicitará el idioma del sistema operativo, la ubicación (por ejemplo, la región de "*Europa*" y la zona de "*Madrid*"), y el tipo de teclado (por ejemplo, "*Spanish*" y "*Default*"). En el apartado de particiones se debe seleccionar "*Borrar disco*". Posteriormente hay que configurar un usuario como, por ejemplo:

- Nombre: *Araceli*
- ¿Qué nombre desea usar para ingresar?: *araceli*
- Nombre del equipo: *ubuntu*
- Contraseña: *xxx*

4.1.3.2 Instalación de *gcloud*

Tras crear la máquina virtual de *Lubuntu* hay que instalar en ella la aplicación "*gcloud*" para poder acceder a los recursos de *Google Cloud* siguiendo los pasos indicados en [5].

Una vez instalada la aplicación *gcloud* hay que proceder a la validación en la plataforma de *Google Cloud* con el comando *gcloud auth login*.

4.1.4 Desarrollo de la aplicación

En este apartado se diseñará una aplicación *Nodejs* que se utilizará como ejemplo de servicio *TI* a desplegar en las distintas plataformas *cloud*. Como ya se indicó, este TFM no se enfoca en el diseño de servicios de TI sino en el despliegue de éstos, por lo que no se tratará de una aplicación compleja.

Como entorno de programación en el lado del servidor se ha elegido *Nodejs* al ser de código abierto y que utiliza un lenguaje de programación muy conocido, como es *JavaScript*.

La aplicación arrancará un servidor con el objetivo de probar la conectividad a *google.com* desde el *host* donde se aloje dicho servidor, mediante el envío de un único comando *ping*. El servidor escuchará en el puerto 8080, y el resultado de la conectividad se deberá mostrar en el navegador del usuario.

El desarrollo de la aplicación comienza creando una carpeta en la máquina *Ubuntu* (o *Lubuntu*) por ejemplo, con el nombre, "*mi-app*", donde se deben incluir los dos ficheros típicos de una aplicación *Nodejs*:

- Un fichero llamado, por ejemplo, “*servidor.js*”, y que contiene el código *JavaScript* de la propia aplicación:

```
const express = require('express');
const exec = require('child_process').execSync;
const os = require('os');
const mi_servidor = express();
mi_servidor.get('/', function (req, res) {
  var texto=exec("ping -c 1 google.com")+"";
  while (texto.indexOf("\n")!=-1)
    texto=texto.replace("\n","<br/>");
  texto='<h1>Ping a Google desde el servidor '+os.hostname()+'</h1>'+texto;
  res.send(texto);
});
mi_servidor.listen(8080);
console.log('Servidor escuchando en el puerto 8080');
```

  La explicación del código anterior es la siguiente. Con las tres primeras sentencias se importan los módulos “*express*” (es un *framework* de *Nodejs* que se utiliza para crear servidores), “*child_process*” (es un *framework* de *Nodejs* que se utiliza para ejecutar procesos en el sistema operativo) y “*os*” (módulo de *Nodejs* para poder utilizar ciertas funciones del sistema operativo). Se instancia el servidor como una constante de nombre “*mi_servidor*” mediante la instrucción “*const mi_servidor = express();*”. Luego se indica qué respuesta (“*res*”) debe tener el servidor cuando reciba una petición (“*req*”). El servidor enviará un *ping* a *google.com* y la respuesta a dicho *ping* se guardará en la variable “*texto*”. Después se sustituye (“*replace*”) todos los retornos de carro (‘*\n*’) de “*texto*” por etiquetas *HTML*, “*
*”, para visualizar la respuesta correctamente en un navegador. La función “*texto.indexOf(“|n”)*” devuelve la posición del primer retorno de carro que encuentre en “*texto*”, y “*-1*” en caso de no encontrar ninguno. Luego se antepone a la variable “*texto*” la cadena “*<h1> Ping a Google desde el servidor* ” seguido del nombre del *host* (función *hostname* del módulo “*os*”), de “</h1> y de “*texto*”. Con “*res.send(texto);*” se envía la variable “texto” como respuesta del servidor. La instrucción “*mi_servidor.listen(8080);*” hará que el servidor quede a la escucha en el puerto 8080. Finalmente, con la última instrucción se enviará a la consola de comandos del servidor el texto “*Servidor escuchando en el puerto 8080*”.

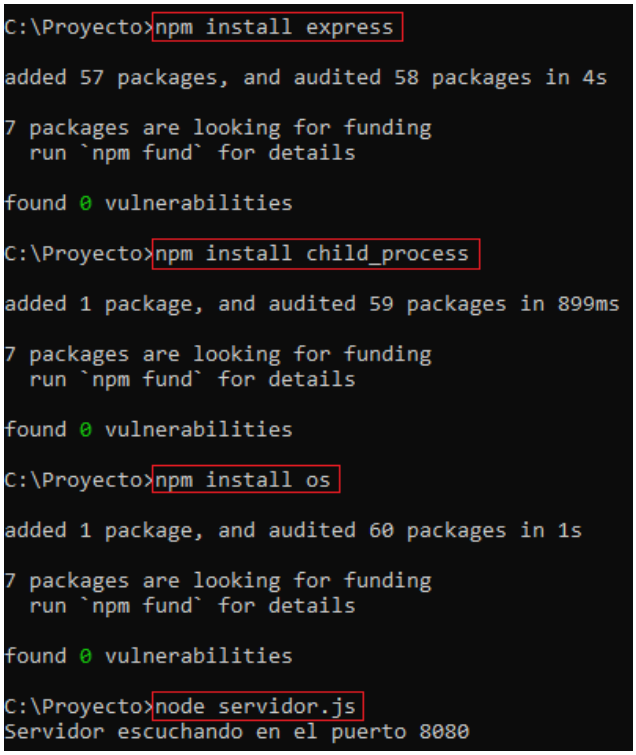
- Un fichero *JSON*, con el nombre de “*package.json*”, que contiene información del proyecto *Nodejs*. Este fichero se utiliza principalmente para que el servidor identifique qué dependencias tiene la aplicación *Nodejs* así como el fichero principal con el que debe arrancar la aplicación:

```
{
  "name": "ping_a_Google",
  "version": "1.0.0",
  "description": "hace un ping a un servidor de Google",
  "author": "Araceli",
```

```
  "main": "servidor.js",
  "dependencies": {
    "child_process": "^1.0.2",
    "express": "^4.18.2",
    "os": [ "linux" ]
  },
  "scripts": {
    "start": "node servidor.js"
  }
}
```

La explicación del código anterior es la siguiente. Se le asigna un nombre al proyecto *Nodejs* ("*ping_a_Google*"), una versión al mismo ("*1.0.0*"), una descripción ("*hace un ping a un servidor de Google*"), un autor ("*Araceli*"). También se define el fichero principal de la aplicación (*"servidor.js"*), así como los módulos de los que depende la aplicación *Nodejs* (*"child_process"*, "*express*" y "*os*"). La aplicación arrancará ("*start*") con el comando "*node servidor.js*".

Antes de arrancar la aplicación en un contenedor es recomendable probarla en la máquina local de *Windows*. Para ello se descargará e instalará *Nodejs*, por ejemplo, desde la *URL* [16]. Una vez instalado *Nodejs*, con un editor, como el "*block de notas*", se crean los dos ficheros de la aplicación ("*servidor.js*" y "*package.js*") y luego se inicia, como administrador, un terminal de *Windows* (*CMD*), y se ejecutan los comandos "*npm install express*", "*npm install chil_process*" y "*npm Install os*" para instalar dichos módulos. Finalmente, se arranca la aplicación con el comando "*node servidor.js*" desde el mismo directorio donde se alojan los dos ficheros de la aplicación *Nodejs*, tal como aparece en la siguiente imagen:



Con el servidor ya arrancado, para probar la aplicación, se introduce la *URL* "*localhost:8080*" en un navegador del mismo *Windows*, y si todo ha ido bien, aparecerá la respuesta del *ping* que el servidor a enviado a *Google*:

Hay que destacar que en *Windows* si se quiere enviar un único *ping* a *Google*, el comando correcto es "*ping -n 1 google*", por lo que al introducir la opción "*-c 1*" (en lugar de "*-n 1*") la obvia y renvía cuatro solicitudes y no una, como aparecía en la imagen anterior. En un entorno *Linux*, como se utiliza en los contenedores, si se reconoce la opción *"-c 1*" del comando *ping* y, por tanto, se enviará un único *ping*.

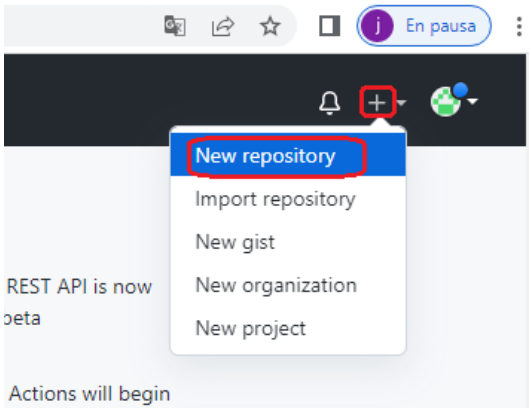
Una vez diseñada la aplicación es conveniente almacenarla en un repositorio como, por ejemplo, *GitHub*, como se indica en el siguiente apartado.

4.1.5 Repositorio *GitHub*

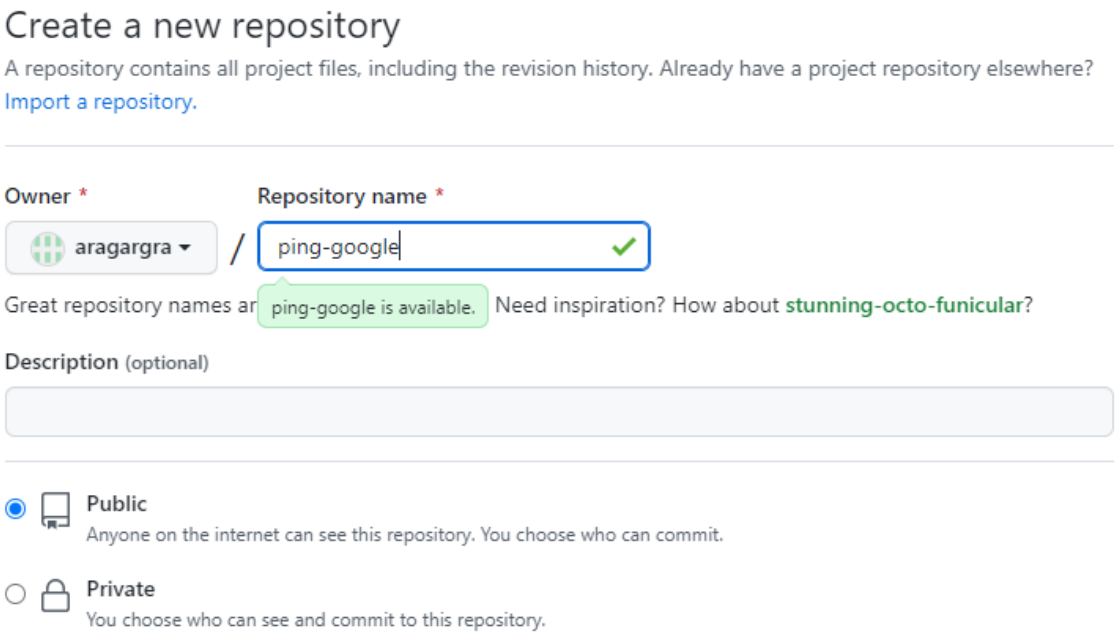
GitHub es un portal *web* en el que los desarrolladores pueden guardar el código de sus aplicaciones en repositorios. El coste del alojamiento en *GitHub* es gratuito siempre que el repositorio creado sea público. La plataforma funciona como una red social en la que los desarrolladores pueden bajarse las aplicaciones de otros, incluso reportar errores, crear nuevas versiones mejoradas, etc. El registro se realiza desde la página web [17].

4.1.5.1 Creación del repositorio en *GitHub*

Una vez que realizado el registro en la plataforma ya se puede crear el repositorio. En la parte superior derecha de la pantalla se pulsa sobre el botón "+" y se selecciona "*New repository*", tal como aparece en la siguiente imagen:



Se asigna un nombre al repositorio y se configura como público:



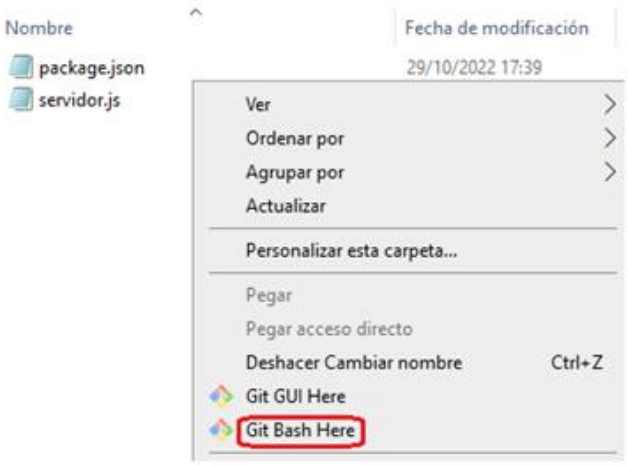
Finalmente, se pulsa en "*Create repository*":

4.1.5.2 Almacenar la aplicación en *GitHub* con *Git*

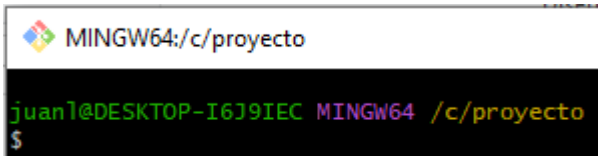
En este apartado se subirá la aplicación al repositorio de *GitHub* utilizando *Git*. La herramienta *Git* proporciona un sistema de control de versiones de aplicaciones y, además, es de código abierto. La descarga de *Git* se puede realizar desde la página [18].

Una vez descargado el fichero "*exe*" (por ejemplo, "*Git-2.38.1-64-bit.exe*") se procede a su instalación aceptando por defecto todas las opciones que aparecen.

Para subir el proyecto con *Git* es necesario que todos los ficheros de la aplicación estén alojados en una misma carpeta de la máquina *Windows*, por ejemplo, "*C:/proyecto*". Luego se arranca *Git* desde esa misma carpeta pulsando en ella con el botón derecho del ratón y seleccionando "*Git Bash Here*":



Se abre un terminal de comandos *Git* similar al de *Linux* tal como aparece en la siguiente imagen:



Una vez arrancado *Git*, el proceso de subir la aplicación a *GitHub* se realiza siguiendo los siguientes seis pasos:

1. Se crea el proyecto *Git* en la carpeta de trabajo con el siguiente comando (aparecerá una carpeta oculta con el nombre de "*.git*" en el directorio de trabajo):

   git init

2. Se agregan todos los archivos a nuestro proyecto *Git*:

   git add .

3. Se copian los ficheros agregados a un repositorio local de *Git* con un comentario, por ejemplo, "*Versión inicial de la aplicación*". Este proceso se denomina *"commit"*:

   git commit -m "Versión inicial de la aplicación"

4. Se asigna una rama del repositorio donde se subirán los ficheros, por ejemplo, la rama "*master*". Las ramas son las líneas de desarrollo de una aplicación.

   git branch -M master

5. Se conecta el repositorio local con el repositorio remoto de *GitHub*, además de asignar un nombre al repositorio remoto, por ejemplo "*origin*":

   git remote add origin https://github.com/aragargra/ping-google.git

6. Finalmente se suben los ficheros a la rama "*master*" del repositorio remoto que anteriormente se le ha llamado "*origin*":

   git push -u origin master

Se puede comprobar que ya aparecen los ficheros de la aplicación en la rama "*master*" del repositorio de *GitHub*, tal como aparece en la siguiente imagen:

4.2 Despliegue en la plataforma *Docker*

Una vez construida la aplicación, en este apartado va a proceder a su despliegue en un *cluster* de *Docker* alojado en el *cloud* de *Google*.

Los contenedores de *Docker* que se van a gestionar en este TFM utilizan el *kernel* de *Linux*, por tanto, deben correr en una máquina con alguna distribución de *Linux* como, por ejemplo, *Ubuntu* (concretamente *Lubuntu*). Actualmente existen ya soluciones para correr ficticiamente contenedores de *Linux* en máquinas *Windows*, de una manera indirecta, es decir, la propia instalación de *Docker* sobre *Windows* crea una pequeña máquina virtual de *Linux* que es donde realmente están corriendo, lo que es transparente para el usuario de *Windows*. También han aparecido soluciones para crear contenedores de *Windows*, que obviamente deben correr en máquinas *Windows*. Lo que se debe tener presente es que no se pueden correr contenedores de *Linux* directamente en una máquina *Windows*.

Los procesos que se van a describir en este apartado aparecen destacados (fondo anaranjado) en la siguiente imagen:

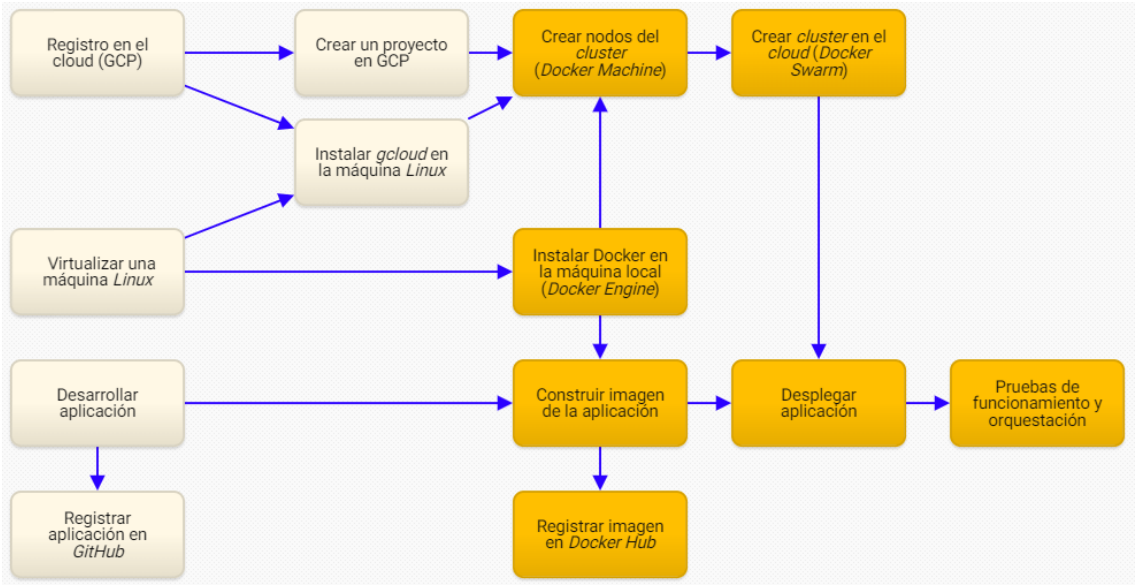


Figura 17. Proceso de despliegue en un *cluster* de *Docker* (elaboración propia)

4.2.1 Instalación de *Docker* en la máquina local

En este apartado se instalará el motor *Docker Engine* en la máquina virtual *Ubuntu* que se creó en el apartado 4.1.3.1 con el objetivo de introducir los comandos *Docker* que permitan crear y ejecutar contenedores. Tras seguir los pasos indicados en [23] se puede visualizar las versiones instaladas de *Docker*, tanto en la parte cliente (*CLI*) como en la parte del servidor (*demonio*):

```
araceli@ubuntu:~$ sudo docker version
Client: Docker Engine - Community
 Version:           20.10.18
 API version:       1.41
Server: Docker Engine - Community
 Engine:
  Version:          20.10.18
```

Si se desea ejecutar comandos de *Docker* con el usuario de *Ubuntu* sin disponer de permisos de "*root*", es necesario añadirlo al grupo de usuario "*docker*".

4.2.2 Creación de la imagen de la aplicación

Como se ha dicho anteriormente, los contenedores no se arrancan con el código de las aplicaciones sino con las imágenes de éstas. Se puede considerar que los contenedores son "instancias" de las imágenes con las que se han creado.

Para generar una imagen *Docker* es necesario crear un fichero de texto con el nombre *Dockerfile* (sin extensión) y en que se deben incluir las instrucciones necesarias para que *Docker* pueda construir la imagen. Las imágenes se estructuran en "capas", generándose una capa por cada instrucción del fichero *Dockerfile*. Esto permite a *Docker* reutilizar las capas de otras imágenes, haciendo muy eficiente la construcción de las imágenes.

En este apartado se construirá una imagen a partir de la aplicación *Nodejs* desarrollada en el apartado 4.1.4. Esta aplicación se compone de dos ficheros, "*servidor.js*" y "*package.json*", alojados en una determinada carpeta de la máquina *Ubuntu* (como por ejemplo "*/home/araceli/mi-app*"). En esta misma carpeta se debe crear un fichero con el nombre *Dockerfile*, sin ninguna extensión, con el siguiente contenido:

```
FROM node:10.16.0-alpine
WORKDIR /app
COPY package.json /app
COPY servidor.js /app
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node","servidor"]
```

La explicación de este código es la siguiente. Todo fichero *Dockerfile* debe comenzar con una instrucción *"FROM*" que identifica la imagen "padre", es decir, la imagen de la que se parte y de la que se heredará el entorno de ejecución (sistema operativo sin el *kernel*, etc.). En este caso se partirá, por ejemplo, de una imagen de *Nodejs* llamada "*node*" con la versión (*tag*) "*10.16.0-alpine*" que contiene *Nodejs*, *npm*, etc. La palabra "*alpine*" indica que esta imagen de *Nodejs* se creó partiendo a su vez de otra imagen con la distribución "*alpine*" de *Linux*, distribución muy ligera y especialmente diseñada para instalarse en contenedores.

Con el comando "*WORKDIR*" se establecerá el directorio "*/app*" como el directorio de trabajo de la imagen. Esta carpeta no existe en la imagen de *Nodejs* de la que se ha partido, pero la creará automáticamente con este comando. A partir de este momento cualquier comando se ejecutará en esta carpeta, si no se indica lo contrario.

Con las dos instrucciones "*COPY*" se copiarán los ficheros "*package.json*" y "*servidor.js*", desde la carpeta de trabajo de *Linux* al directorio de trabajo de la imagen, "*/app*". Posteriormente se comprobará que, al arrancar un contenedor con esta imagen, se creará en dicho contenedor un directorio con el nombre "*/app*" donde estarán alojados estos dos ficheros.

Con el comando "*RUN*" se da la orden de ejecutar el comando "*npm install*" con el fin de instalar en la imagen todas las dependencias que se han descrito en el fichero "*package.json*". En este caso "*express*", "*child_pocess*" y "*os*".

Con el comando "*EXPOSE*" se expone el puerto 8080 del contenedor internamente en el *cluster*.

Finalmente, con el comando "*ENTRYPOINT*" se ejecutará la instrucción *Nodejs* "*node servidor.js*" en el momento que se arranque el contenedor.

Con estos tres ficheros ("*servidor.js*", "*package.json*" y "*Dockerfile*") ya se puede construir la imagen de *Docker*. Para ello es necesario introducir el siguiente comando "*build*" desde la misma carpeta donde se encuentran estos tres ficheros (el punto final del comando indica que los debe buscar en la carpeta de trabajo, es decir, en el directorio "*/home/araceli/mi-app*"). La opción "*-t*" (de "*tag*") permite dar un nombre a la imagen, por ejemplo, "*ping-google*", versión "*latest*" (si no se indica la versión se le asigna por defecto ésta):

sudo docker build -t ping-google:latest **.**

```
araceli@ubuntu:~/mi-app$ sudo docker build -t ping-google:latest .
Sending build context to Docker daemon   5.12kB
Step 1/6 : FROM node:10.16.0-alpine
10.16.0-alpine: Pulling from library/node
e7c96db7181b: Pull complete
bbec46749066: Pull complete
89e5cf82282d: Pull complete
5de6895db72f: Pull complete
Digest: sha256:07897ec27318d8e43cfc6b1762e7a28ed01479ba4927aca0cdff53c1de9ea6fd
Status: Downloaded newer image for node:10.16.0-alpine
 ---> 9dfa73010b19
Step 2/6 : WORKDIR /app
```

Como puede apreciarse en la imagen anterior (no está completa), la instrucción anterior se ejecuta en tantos pasos como comandos tenga el fichero *Dockerfile*, creándose una capa de la imagen en cada paso. *Docker* informa de los siguientes "*warnings*" que se pueden obviar:

```
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN ping_a_Google@1.0.0 No repository field.
npm WARN ping_a_Google@1.0.0 No license field.
```

El siguiente comando permite listar las imágenes disponibles en el repositorio local:

sudo docker images

```
araceli@ubuntu:~/mi-app$ sudo docker images
REPOSITORY          TAG                 IMAGE ID
ping-google         latest              bd9220a3219d
node                10.16.0-alpine      9dfa73010b19
araceli@ubuntu:~/mi-app$
```

Como se puede observar, se han creado dos imágenes, una con el nombre "*ping-google:latest*", y que corresponde con la que *Docker* ha construido, y otra con el nombre "*node:10.16.0-alpine*", que es de la que se ha partido (y que previamente se ha descargado de *Docker Hub*).

En este momento ya se puede probar la imagen "*ping-google*" arrancando un contenedor en la máquina local, poniéndolo a la escucha en el puerto 8080,

más concretamente, mapeando el puerto 8080 de la máquina local con el puerto 8080 del contenedor. Esto se realiza con el siguiente comando *Docker*:

sudo docker run -p 8080:8080 ping-google:latest

```
araceli@ubuntu:~/mi-app$ sudo docker run -p 8080:8080 ping-google:latest

> ping_a_Google@1.0.0 start /usr/src/app
> node servidor.js

Servidor escuchando en el puerto 8080
```

Como se aprecia, el servidor se ejecuta en primer plano ("*foreground*") monopolizando el terminal y, por tanto, sin poder ejecutar ningún otro comando. Se puede arrancar el contenedor en segundo plano ("*background*") con la opción "*-d*" (de "*detach*"), o uniendo las dos opciones, "*-d*" y "*-p*", tal como se indica en el siguiente comando:

*sudo docker run -**d**p 8080:8080 ping-google:latest*

Ya se puede probar la aplicación introduciendo en un navegador de la máquina local la *URL* "*localhost:8080*". Esto generará una petición *HTTP* al contenedor, y éste arrancará un subproceso para ejecutar un *ping* a *google.com*, y una respuesta al navegador con el resultado del *ping*. En efecto:

Como se observa, la respuesta del servidor informa del *ID* del contenedor, en este caso, "*dc18c9ef141f*".

4.2.3 Registro de la imagen en el repositorio de *Docker*

Una vez creada y probada la imagen de la aplicación es el momento de guardarla en el registro oficial de *Docker*, denominado *Docker Hub.* Los repositorios en *Docker Hub* son gratuitos siempre que no sean de acceso privado. Lo primero es entrar en su página web [19] y crear un usuario asignándole un "*username*" (por ejemplo "*araceli96*"), un "*email*" y un "*password*".

Para registrar la imagen se deben seguir los siguiente tres pasos:

1. Se procede a la validación del usuario en *Docker Hub* con el comando "*docker login"*:

sudo docker login --username=araceli96

2. Se asigna un nuevo nombre (nuevo "*tag*") a la imagen con el nombre del repositorio de *Docker Hub*, en este caso, "*araceli96/ping_google:latest*":

   sudo docker tag ping-google:latest araceli96/ping-google:latest

   Si se listan todas las imágenes del *cluster* se puede comprobar que la imagen (*IMAGE ID*) tiene ahora dos nombres distintos, en efecto:

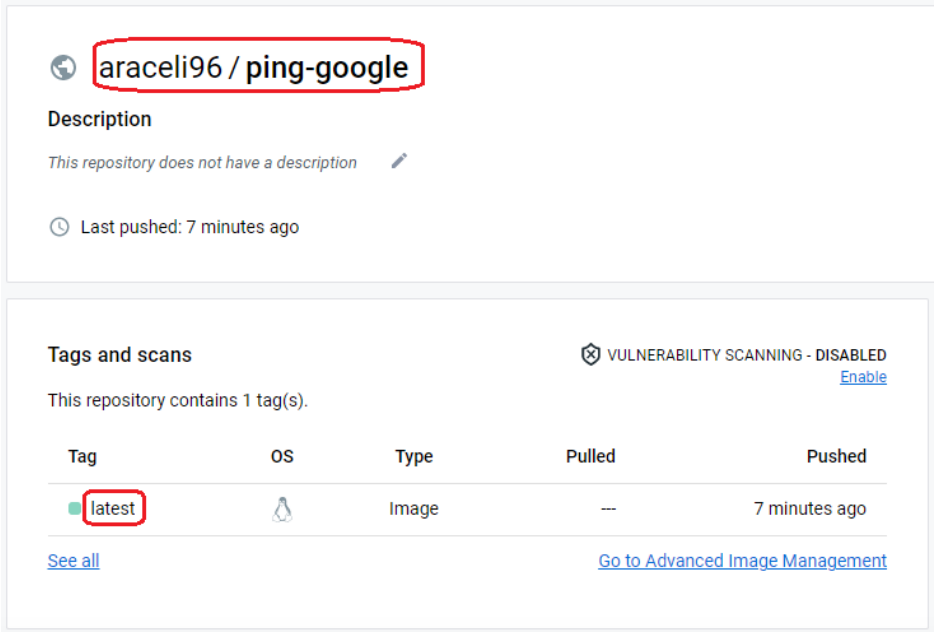
   sudo docker images

```
araceli@ubuntu:~$ sudo docker images
REPOSITORY              TAG       IMAGE ID       CREATED          SIZE
araceli96/ping-google   latest    79828ff0f61e   3 minutes ago    661MB
ping-google             latest    79828ff0f61e   3 minutes ago    661MB
node                    argon     ef4b194d8fcf   4 years ago      653MB
araceli@ubuntu:~$
```

3. Finalmente, con el comando "*docker push*" se sube la imagen al repositorio de *Docker Hub*:

   sudo docker push araceli96/ping-google:latest

En la página web de *Docker Hub* deberá ya aparecer en el repositorio "*araceli96*" la imagen "*ping-google*" con la etiqueta "*latest*", como se muestra aquí:



Al igual que se hizo en el apartado anterior, ya se puede arrancar un contenedor con la imagen del repositorio remoto, escuchando en el puerto 8080 (mismo mapeo de puertos que antes):

sudo docker run -p 8080:8080 araceli96/ping-google:latest

```
araceli@ubuntu:~/mi-app$ sudo docker run -p 8080:8080 araceli96/ping-google:latest

> ping_a_Google@1.0.0 start /usr/src/app
> node servidor.js

Servidor escuchando en el puerto 8080
```

Si ahora se entra en un navegador con la dirección del *host* ("*localhost*") y el puerto 8080, se accederá al servicio ofrecido por la aplicación:

Hasta ahora se ha provisionado la aplicación en un contenedor alojado en el propio *host* (máquina local de *Ubuntu*). En los siguientes apartados prepararemos la infraestructura para poder provisionarla en un *cluster* de *Docker* en *Google Cloud*.

4.2.4 Creación de los nodos del *cluster*. *Docker-Machine*

En este apartado se crearán tres nodos (máquinas virtuales) en *Google Cloud* y se instalarán en ellos *Docker Engine*. La herramienta *Docker Machine* permite instalar *Docker Engine* en *host* virtuales.

La instalación de *Docker Machine* en la máquina local *Ubuntu* se realiza con la instrucción, tal como se indica en [41]:

```
base=https://github.com/docker/machine/releases/download/v0.14.0 &&
curl -L $base/docker-machine-$(uname -s)-$(uname -m) >/tmp/docker-machine &&
sudo install /tmp/docker-machine /usr/local/bin/docker-machine
```

Ahora hay que validarse en *Google Cloud* con el siguiente comando *gcloud*:

```
gcloud auth application-default login
```

Se crea el primer nodo del *cluster*, asignándolo al proyecto anterior (en este caso, *ID* "*miproyecto-364518*") de *Google Cloud* ("*driver google*"), con un determinado tipo de máquina (por ejemplo, "*f1-micro*"), en una determinada zona (por ejemplo, "*us-central1-c*"), y al que se le dará el nombre de, por ejemplo, "*worker1*". Hay que tener en cuenta que se debe instalar una versión de *Docker* anterior a la 20.10.0 debido a la incompatibilidad con *Docker Machine* versión 0.14. En este caso se instalará, por ejemplo, la versión *Docker* 19.03.9:

docker-machine create --driver ***google*** *--google-project* ***miproyecto-364518*** *--google-zone* ***us-central1-c*** *--google-machine-type* ***f1-micro*** *--engine-install-url "https://releases.rancher.com/install-docker/****19.03.9****.sh"* ***worker1***

De la misma forma, se crea un segundo nodo del *cluster* con el nombre de, por ejemplo, "*worker2*":

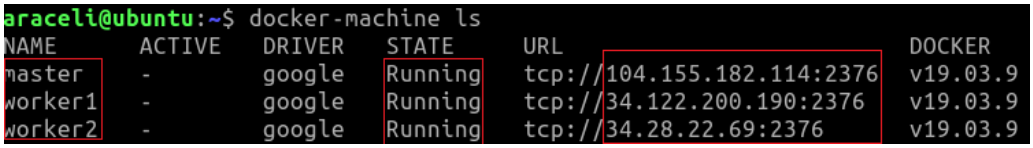
docker-machine create --driver google --google-project miproyecto-364518 --google-zone us-central1-c --google-machine-type f1-micro --engine-install-url "https://releases.rancher.com/install-docker/19.03.9.sh" ***worker2***

Igualmente se crea un tercer nodo del *cluster* con el nombre, por ejemplo, "*master*":

docker-machine create --driver google --google-project miproyecto-364518 --google-zone us-central1-c --google-machine-type f1-micro --engine-install-url "https://releases.rancher.com/install-docker/19.03.9.sh" ***master***

Los nodos están ya operativos en el *cloud* (campo "*STATE*" en estado "*Running*"). Con el siguiente comando se pueden visualizar estos nodos con sus direcciones *IP*, que más tarde se utilizarán:

docker-machine ls



Ahora hay que abrir el puerto 2377 en estos tres nodos pues es el puerto con el que trabaja *Docker Swarm* (en un *cluster* sería suficiente con abrirlo en el "*master*"). Esto se puede realizar desde la *web* de *Google Cloud*, apartado *Red de VPC/Firewall*, pulsando en "*CREAR REGLA FIREWALL*" y seleccionando lo siguiente:

- Nombre: el nombre de la regla *firewall*, por ejemplo, "*puerto-2377*".
- Descripción: por ejemplo, "*Abre el puerto 2377 para trabajar con Swarm*".
- Destinos: "*Todas las instancias de la red*", es decir, cualquier elemento del proyecto de *Google Cloud*, ya sean *cluster*, nodos, etc.
- Rango de *IPv4* de origen: "*0.0.0.0/0*"., es decir, desde cualquier origen.
- Marcar "*Protocolos y puertos especificados*".
- Marcar *TCP*, y en el campo "*Puertos*" introducir el valor 2377.
- Marcar *UDP*, y en el campo "*Puertos*" introducir el valor 2377.

4.2.5 Creación del *cluster*. *Docker-Swarm*

Una vez creados los tres nodos hay que agruparlos para formar un *cluster*. Para ello hay que instalar la herramienta *Docker Swarm* en el nodo que va a hacer funciones de "*master*". Lo primero es conectarse a este nodo vía *SSH* desde la máquina local de *Ubuntu*, con el siguiente comando *Docker*:

```
docker-machine ssh master
```

El *prompt* habrá cambiado indicando que ahora los comandos introducidos en el terminal se ejecutarán en el nodo *master*.

Se inicializa *Docker Swarm* en el nodo *master*, identificando a éste con su dirección *IP* pública:

```
sudo docker swarm init --advertise-addr 104.155.182.114
```

La salida del comando anterior mostrará una instrucción "*join*", que deberá ejecutarse en los nodos *worker* para agruparlo en un *cluster*

```
docker swarm join --token SWMTKN-1-
3twf68mk5qvg5czflv6pt6441gr1q1err7yz487znt8xzkm6ue-
cqxnmlhe1awgw1i2ot88znb5w 104.155.182.114:2377
```

Se abre un terminal *SSH* en el nodo *worker1*:

```
docker-machine ssh worker1
```

, y se introduce, como *root*, el comando anteriormente indicado por el nodo *master*. De igual forma, se agrupa el nodo *worker2* al *cluster*:

Se conecta el *CLI* de *Docker* de la máquina local *Ubuntu* con el *daemon* del *master* para que los comandos introducidos desde esta máquina se ejecuten en el nodo *master*. Esto se realiza con el siguiente comando:

```
eval $(docker-machine env master)
```

En cualquier momento se puede volver al estado anterior, es decir, que el *CLI* de la máquina local se conecte al *daemon* de la máquina local, con el comando *eval $(docker-machine env -unset)*.

Si ahora se listan los nodos del *cluster* se observará que el nodo *master* es el activo (aparece con un asterisco), lo que significa que es este nodo el que recibirá los comandos *Docker* que se introduzcan en la máquina local *Ubuntu*:

```
docker-machine ls
```

```
araceli@ubuntu:~$ docker-machine ls
NAME      ACTIVE   DRIVER   STATE     URL                         DOCKER
master    *        google   Running   tcp://104.155.182.114:2376  v19.03.9
worker1   -        google   Running   tcp://34.122.200.190:2376   v19.03.9
worker2   -        google   Running   tcp://34.28.22.69:2376      v19.03.9
```

4.2.6 Despliegue de la aplicación

El despliegue de una aplicación orquestada en un *cluster* de *Docker* se realiza mediante un objeto "*service*". Un *service* es un conjunto de tareas (*task*) que se crean en los nodos para poder orquestar los contenedores. A diferencia del comando "*run*", que ya se vio en los apartados 4.2 y 4.3 para levantar un contenedor en un nodo, el comando "*service*" levanta en el *cluster* un conjunto de contenedores que estarán replicados y orquestados.

Con el siguiente comando (y único) se descargará de *Docker Hub* la imagen de la aplicación y arrancará un contenedor en *cluster*. Se puede utilizar la opción "*replicas*" en caso de requerir más contenedores. El contenedor abrirá el puerto 8080 por donde escuchará el servicio.

docker service create -p 8080:8080 araceli96/ping-google:latest

El proceso que sigue *Docker* para crear el contenedor es el siguiente:

1. El comando es recibido por el *CLI* de *Docker* de la máquina local.
2. El *CLI* envía una petición *API* al *daemon* del nodo *master*.
3. El *master* comprueba si el *cluster* dispone de la imagen requerida.
4. Si la imagen no la tiene en local, la baja al repositorio local desde *Docker Hub*.
5. Con la imagen en el repositorio local se crea una tarea y un contenedor en el nodo que haya elegido el orquestador.

Se puede comprobar que, en efecto, se ha creado el servicio en el *cluster* con el comando *docker service ls*:

```
araceli@ubuntu:~$ docker service ls
ID             NAME           MODE         REPLICAS   IMAGE                          PORTS
rtybhwc6mbab   crazy_panini   replicated   1/1        araceli96/ping-google:latest   *:8080->8080/tcp
```

Se observa que se ha creado un servicio con el nombre "*crazy_panini*", con la imagen del repositorio "*araceli96/ping-google*", con una sola réplica, y en el que se ha mapeado el puerto 8080 de *host* con el puerto 8080 del contenedor.

El nodo *master* asigna una tarea (*task*) en los nodos *worker* por cada réplica (contenedor) que se cree, para poder orquestarlo, pues ahora los contenedores no son independientes entre sí.

Las tareas asociadas a un determinado servicio, así como los nodos que las tienen asignadas, se puede visualizar con el comando *docker service ps crazy_panini*:

```
araceli@ubuntu:~$ docker service ps crazy_panini
ID             NAME              IMAGE                          NODE
osalr3ec1l9c   crazy_panini.1    araceli96/ping-google:latest   master
```

Observamos que para el servicio "*crazy_panini*" existe una única tarea con el nombre "*crazy_panini.1*", y cuyo responsable es el nodo *master*.

Entrando en este nodo (*docker-machine ssh master*) se puede visualizar el contenedor que está asociado a esa tarea (*sudo docker ps*):

```
araceli@ubuntu:~$ docker-machine ssh master
docker-user@master:~$ sudo docker ps
CONTAINER ID        IMAGE                          NAMES
d919175b71d5        araceli96/ping-google:latest   crazy_panini.1.osalr3ec1l9c3ta6qtkrha24f
```

En efecto, el nodo *master* tiene un único contenedor con *ID* "*d919175b71d5*" con el nombre "*crazy_panini.1.osalr3ec1l9c3ta6qtkrha24f*".

Ahora se puede entrar en este contenedor y comprobar que, en efecto, existe una carpeta llamada "*/app*", que es el directorio donde se ha alojado la aplicación tal como se indicó en el *Dockerfile*. Esto se realiza introduciendo la instrucción "*exec*" para ejecutar un comando en un contenedor, por ejemplo, el comando "*sh*" con el que abrir un terminal de comandos (o "*shell*") en dicho contenedor, con la opción "*-it*" ("*-i*" para crear una sesión interactiva, y "*-t*" para abrir una terminal de comandos):

```
docker-user@master:~$ sudo docker exec -it d919175b71d5 /bin/sh
/app # ls
node_modules       package-lock.json  package.json       servidor.js
/app # exit
docker-user@master:~$ exit
logout
araceli@ubuntu:~$
```

Finalmente, se puede probar el servicio abriendo un navegador e introduciendo la *IP* de cualquiera de los nodos (por ejemplo, la del *master*) y el puerto 8080:

Observamos que el *host* que ha contestado es el propio contenedor.

En cualquier momento se puede eliminar el servicio con el comando:

docker service rm crazy_panini

4.2.7 Pruebas de orquestación

En este apartado se van a realizar varias pruebas de escalabilidad, balanceo de carga y disponibilidad, para verificar los beneficios de desplegar aplicaciones en un *cluster* orquestado.

Prueba 1. Escalado horizontal de la aplicación

El escalado es una de las características más deseables de las aplicaciones. Se refiere a la capacidad de crecimiento de la aplicación para poder atender un mayor número de peticiones de usuario. El escalado horizontal es la agregación de contenedores adicionales (llamadas réplicas), mientras que el escalado vertical supone ampliar los recursos en los contenedores existentes.

El escalado horizontal en un *cluster* de *Docker* se realiza introduciendo el comando "*docker service scale*". Por ejemplo, siguiendo con el caso práctico, si se quiere atender el servicio ahora con tres contenedores, se haría con el comando:

*docker service **scale** crazy_panini**=3***

```
araceli@ubuntu:~$ docker service scale crazy_panini=3
crazy_panini scaled to 3
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
```

En efecto, el servicio "*crazy_panini*" ahora tiene tres tareas que corresponden a cada uno de los contenedores que se han creado:

docker service ls

```
araceli@ubuntu:~$ docker service ls
ID             NAME           MODE         REPLICAS
rtybhwc6mbab   crazy_panini   replicated   3/3
```

Docker Swarm no soporta el escalado automático, es decir, no se puede adaptar automáticamente el número de contenedores del servicio en función de la carga, por lo que no se realizará ninguna prueba al respecto.

Prueba 2. Balanceo de carga entre contenedores

En esta prueba se verificará que el *cluster* (concretamente, el orquestador) balancea en tráfico uniformemente entre los distintos contenedores que atienden el servicio, según el algoritmo denominado *round-robin*. En este caso,

el tráfico entrante al servicio ha de repartirse entre los tres contenedores. Se van a realizar solicitudes consecutivas para comprobar qué contenedor lo atiende:

Ping a Google desde el servidor 51ef6cf832c2

PING google.com (142.250.148.102): 56 data bytes
64 bytes from 142.250.148.102: seq=0 ttl=114 time=1.197 ms

--- google.com ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 1.197/1.197/1.197 ms

ID del contenedor que atiende la petición

Ping a Google desde el servidor 3ead39b53595

PING google.com (74.125.201.139): 56 data bytes
64 bytes from 74.125.201.139: seq=0 ttl=114 time=1.019 ms

--- google.com ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 1.019/1.019/1.019 ms

Ping a Google desde el servidor d919175b71d5

PING google.com (108.177.112.102): 56 data bytes
64 bytes from 108.177.112.102: seq=0 ttl=114 time=1.141 ms

--- google.com ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 1.141/1.141/1.141 ms

Se observa como, en efecto, las distintas solicitudes al servicio se van atendiendo en reparto de carga entre los distintos contenedores.

Prueba 3. Reparto uniforme de contenedores entre los nodos

El orquestador reparte los contendores entre los distintos nodos del *cluster* de forma que se produzca un balanceo la carga entre todos los nodos.

Con el siguiente comando se visualiza en qué nodos se han alojado los distintos contenedores:

docker service ps crazy_panini

```
araceli@ubuntu:~$ docker service ps crazy_panini
ID             NAME              IMAGE                              NODE
osalr3ec1l9c   crazy_panini.1    araceli96/ping-google:latest       master
jfgkpyxc1srb   crazy_panini.2    araceli96/ping-google:latest       worker1
zvqzbhqnaomr   crazy_panini.3    araceli96/ping-google:latest       worker2
```

En efecto, los contenedores se han repartido uniformemente entre los tres nodos del *cluster*.

Prueba 4. Desescalado de la aplicación

El desescalado es la reducción de réplicas que atienden el servicio, y se realiza con el mismo comando anterior indicando el número de réplicas deseado. Siguiendo con el caso, para desescalar la aplicación dejándola con únicamente dos réplicas, se realiza así:

*docker service **scale** crazy_panini**=2***

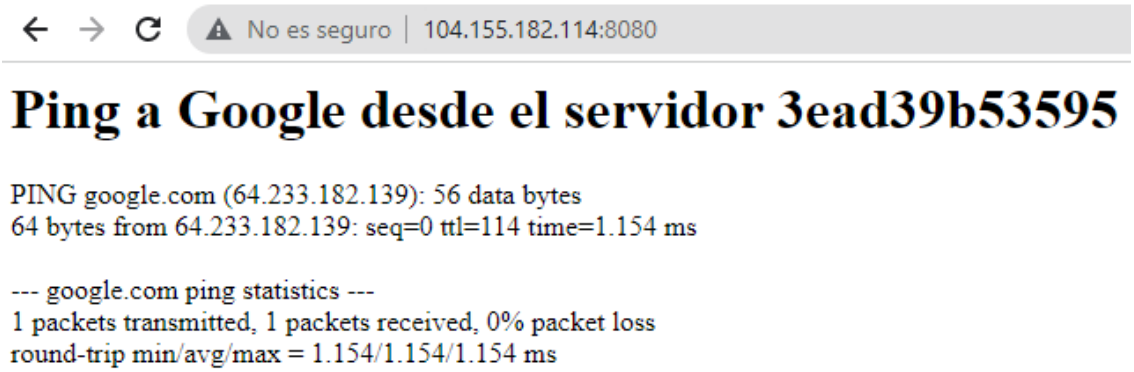
```
araceli@ubuntu:~$ docker service scale crazy_panini=2
crazy_panini scaled to 2
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
```

A igualdad de volumen de carga de los nodos, el orquestador no debería asignar las dos tareas del servicio en el mismo nodo. En efecto:

docker service ps crazy_panini

```
araceli@ubuntu:~$ docker service ps crazy_panini
ID             NAME              IMAGE                              NODE
osalr3ec1l9c   crazy_panini.1    araceli96/ping-google:latest       master
jfgkpyxc1srb   crazy_panini.2    araceli96/ping-google:latest       worker1
```

Se puede comprobar que el servicio sigue funcionando perfectamente:



Prueba 5. Mantenimiento de réplicas o fallo de un contenedor

Ahora se pondrá a prueba la disponibilidad de la aplicación eliminando uno de los contenedores del servicio, y se comprobará como el orquestador del *cluster* levantará un nuevo contenedor con el objetivo de mantener el número de réplicas establecido. Se eliminará, por ejemplo, el contenedor alojado en el *master*:

```
araceli@ubuntu:~$ docker-machine ssh master
docker-user@master:~$ sudo docker ps
CONTAINER ID        IMAGE                          COMMAND             CREATED
d919175b71d5        araceli96/ping-google:latest   "node servidor"     37 minutes ago
docker-user@master:~$ sudo docker rm -f d919175b71d5
d919175b71d5
docker-user@master:~$ exit
logout
araceli@ubuntu:~$ docker service ps crazy_panini
ID             NAME                IMAGE                          NODE      DESIRED STATE
7uv06pfh5qfw   crazy_panini.1      araceli96/ping-google:latest   worker2   Running
osalr3ec1l9c    \_ crazy_panini.1  araceli96/ping-google:latest   master    Shutdown
jfgkpyxc1srb   crazy_panini.2      araceli96/ping-google:latest   worker1   Running
```

En efecto, al eliminar un contenedor en el *master* ("*docker rm*"), el orquestador *Docker Swarm* ha levantado automáticamente uno nuevo manteniendo dos réplicas para el servicio. En la imagen se observa que el contenedor del *master* está caído y que se ha levantado un nuevo contenedor en el nodo *worker2*.

4.3 Despliegue en la plataforma *Kubernetes*

En este apartado se desplegará la aplicación desarrollada en el punto 4.1.4 en un *cluster* de *Kubernetes* alojado en *Google Cloud*.

Los procesos que se van a describir en este apartado aparecen destacados (fondo anaranjado) en la siguiente imagen:

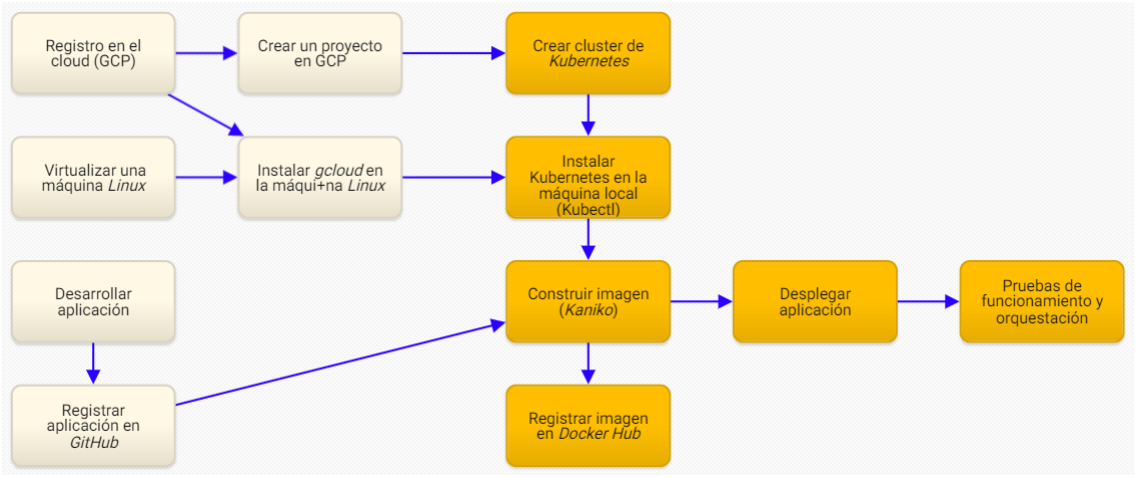


Figura 18. Procesos de despliegue en un *cluster* de *Kubernetes* (elaboración propia)

4.3.1 Creación del *cluster* de *Kubernetes*

En este apartado se construirá un *cluster* de *Kubernetes* en la nube pública de *Google*.

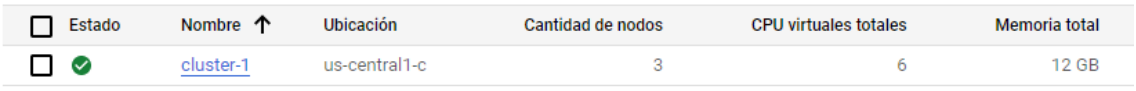
Para crear el *cluster* hay entrar en el proyecto de *Google Cloud Platform* que se generó en el apartado 4.1.2, desplegar el menú de navegación, seleccionar "*Kubernetes Engine*" y luego ir a "*Clústeres*". La primera vez que se cree un *cluster* en el proyecto, aparecerá una ventana solicitando que se habilite la *API* de *Kubernetes Engine*. Una vez habilitada la *API* hay que pulsar en el botón de "*CREAR*". Solicitará que se elija el tipo de *cluster* que se quiere crear,

presentándonos dos opciones, el modo *estándar* y el modo *autopilot*. La diferencia de ambos tipos de *clusters* la tenemos en [21]. En este caso se elegirá el modo estándar pulsando en el botón "*CONFIGURAR*" correspondiente.

Los parámetros más destacados que se deben introducir son los siguientes (el resto se deben mantener a sus valores por defecto):

- Un nombre del *cluster*, por ejemplo, el que propone: "*cluster-1*".
- Se deja marcado el botón de opción "*Zonal*" en lugar de "*Regional*". La diferencia es que en un *cluster* zonal todos los nodos estarán en una misma zona, mientras que en un *cluster* regional los nodos estarán replicados en distintas zonas con el objetivo de ofrecer una mayor disponibilidad (preferible para entornos de producción).
- Se mantiene la zona "*us-central1-c*", aunque como norma general se debe seleccionar el *Data Center* más cercano.
- En el apartado de "*default-pool*", campo "*Cantidad de nodos*" hay que establecer el número de nodos *worker* del *cluster*, por ejemplo, 3, que es el valor por defecto.
- Si se marca la casilla "*Habilitar el escalador automático de clústeres*", el número de nodos del *cluster* se incrementará o reducirá automáticamente en función de la carga de trabajo. En este caso, no lo activaremos.
- En el apartado *default-pool*/*Nodos* hay que configurar los nodos del *cluster*: En el campo "*Tipo de imagen*" se selecciona el sistema operativo y motor de contenedores, por ejemplo, "*Ubuntu con containerd*" (*Kubernetes* ya no aconseja el tipo de imagen "*Ubuntu con Docker*"). En los campos "*Serie*" y "*Tipo de máquina*" se debe elegir la máquina más adecuada para los nodos, por ejemplo, la serie "*N1*" y el tipo "*n1-stantard-1*".
- En el apartado *default-pool*/*Seguridad* se recomienda "*Permitir el acceso total a todas las APIs de Cloud*".

Finalmente se pulsa en "*CREAR*". Trascurridos unos 5 minutos quedará disponible el *cluster* de *Kubernetes* con los 3 nodos *worker* y un plano de control que hace funciones de nodo *master*, tal como se aprecia en la siguiente imagen:



4.3.2 Instalación de *kubectl* en la máquina local

Una vez disponible nuestro *cluster* de *Kubernetes* es necesario poder comunicarse con él desde la máquina local *Ubuntu*. Existen tres formas de hacerlo:

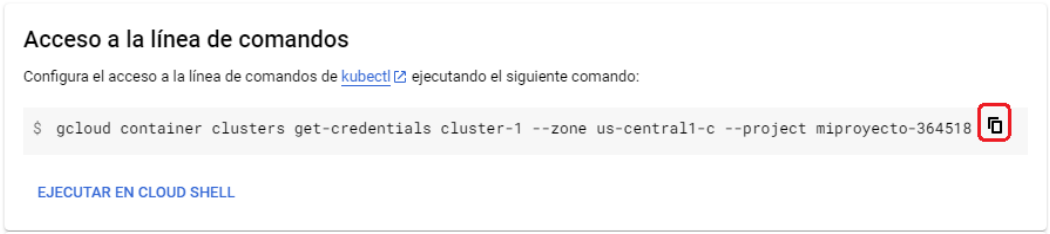
- A través de la consola de *Google Cloud*.

- Directamente, mediante una llamada a la *API* de *Kubernetes* a través del protocolo *HTTP/gRPC*.
- Indirectamente, mediante el terminal de comandos (*CLI*) proporcionado por la herramienta *kubectl*.

Con el comando de Linux "*sudo snap install kubectl --classic*" se puede descargar e instalar el software de *kubectl* en la máquina local.

Ahora hay que identificar el *cluster* al que conectarse mediante sus credenciales. Estas credenciales se pueden obtener en la página de *Google Cloud*, en el apartado *Kubernetes Engine/Clústeres* pulsando en ⋮ y seleccionando "*Conectar*", tal como aparece en la siguiente imagen:

Se copia el comando *gcloud* propuesto:



, y finalmente se pega el siguiente comando en el terminal *Ubuntu* de la máquina local para establecer la conexión con el *cluster* de *Kubernetes*:

gcloud container clusters get-credentials cluster-1 --zone us-central1-c --project miproyecto-364518

Para verificar la correcta conexión de la máquina local con el *cluster*, se puede solicitar que se muestren todos los objetos de *Kubernetes* de nuestro proyecto, con el comando *kubectl get all*:

```
araceli@ubuntu:~$ kubectl get all
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.8.0.1     <none>        443/TCP   25d
```

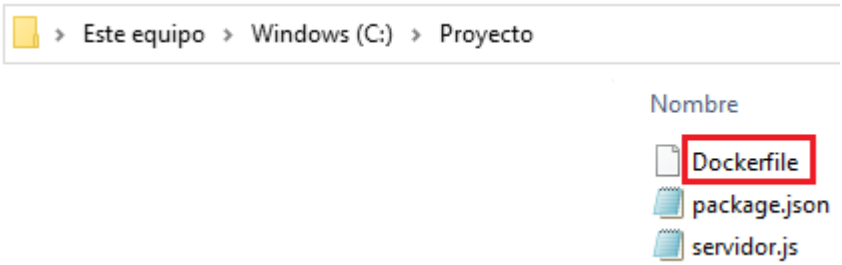
Se observa que únicamente está disponible el servicio correspondiente a la *API* de *Kubernetes*, responsable de controlar las solicitudes de los comandos *kubectl*.

4.3.3 Construcción de la imagen con *Kaniko*

Kubernetes, desde la versión 1.20, ya no utiliza el motor de *Docker* sino *Containerd*, y aunque las imágenes de *Docker* son compatibles en *Kubernetes*,

no las puede construir al no disponer de *Docker*. La herramienta *Kaniko* permite a *Kubernetes* crear imágenes de *Docker* sin tener *Docker*, usando el *Dockerfile*. En este apartado se mostrará cómo utilizar la herramienta *Kaniko*.

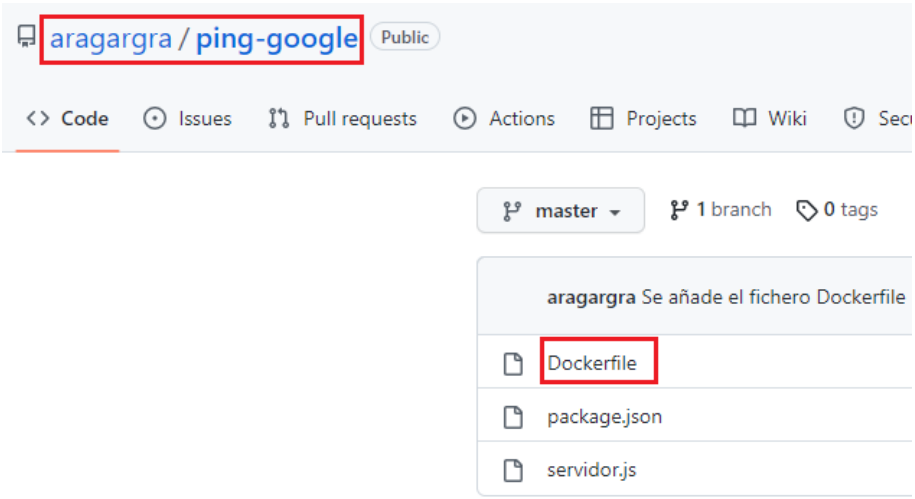
Lo primero que se va a hacer es copiar el fichero *Dockerfile* que se creó en el apartado 4.2.2 en la carpeta de *Windows* donde se encuentran los ficheros de la aplicación construida en el apartado 4.1.4 (en este caso, "*C:\Proyecto*").



Ahora hay que añadir el *Dockerfile* al repositorio de *GitHub* de la aplicación. Para ello, desde la carpeta anterior se pulsa con el botón derecho del ratón, se selecciona "*Git Bash Here*" e introducen los siguientes comandos *Git*:

```
juanl@DESKTOP-I6J9IEC MINGW64 /c/proyecto (master)
$ git add .

juanl@DESKTOP-I6J9IEC MINGW64 /c/proyecto (master)
$ git commit -m "Se añade el fichero Dockerfile"
[master 9796ef6] Se añade el fichero Dockerfile
 1 file changed, 7 insertions(+)
 create mode 100644 Dockerfile

juanl@DESKTOP-I6J9IEC MINGW64 /c/proyecto (master)
$ git push -u origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 449 bytes | 449.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/juanluisgh/ping-google.git
   b160931..9796ef6  master -> master
branch 'master' set up to track 'origin/master'.

juanl@DESKTOP-I6J9IEC MINGW64 /c/proyecto (master)
$
```

En el repositorio *GitHub* ya debe aparecer el *Dockerfile* tal como se aprecia en la siguiente imagen:



Kubernetes va a necesitar acceder a *Docker Hub* para guardar la imagen que se va a construir, y para ello requiere autenticarse mediante un *token*. Para crear este *token* hay que entrar y validarse en el portal de *Docker Hub*, pulsar luego sobre el desplegable con el nombre del usuario (en este caso, "*araceli96*") y seleccionar *Account Setting*/*Security*. Luego hay que pulsar en el botón azul "*New Access Token*", en "*Access Token Description*" dar un nombre al *token*, por ejemplo, "*mi-token*", pulsar en "*Generate*" para crear el *token*, y finalmente copiar el *token* (p.ej., "*dckr_pat_LfqC6afABkAGm-HwbkkV4x0gtUc*"). Este *token* debe ir cifrado mediante un objeto de *Kubernetes* llamado "*secret*".

El siguiente comando crea el objeto *secret*, por ejemplo, con el nombre "*secreto*" utilizando el usuario de *Docker Hub* y el *token* anterior:

> *kubectl create secret docker-registry* ***secreto*** *--docker-username=****araceli96*** *--docker-password=****dckr_pat_LfqC6afABkAGm-HwbkkV4x0gtUc***

Las credenciales (*username* y *password*) para registrarse en *Docker* se han incluido en el fichero "*.dockerconfigjson*" del *secret*, tal como se aprecia en la siguiente imagen:

```
araceli@ubuntu:~$ kubectl describe secret secreto
Name:         secreto
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  kubernetes.io/dockerconfigjson

Data
====
.dockerconfigjson:  190 bytes
```

Este fichero "*.dockerconfigjson*" debe guardarse en el directorio "*/kaniko/.docker/*" del contenedor con el nombre de "*config.json*", que es de donde *Docker* va a tomar las credenciales para validarse en *Docker Hub*.

Una vez creado el *secret*, hay que levantar en *Kubernetes* un *pod* con un único contenedor que aloje la herramienta *kaniko* con el objetivo de construir la imagen de la aplicación y guardarla en *Docker Hub*. La validación en *Docker Hub* se realiza montando un volumen en el contenedor y depositando el *secret* en dicho volumen, tal como se verá más adelante.

En *Kubernetes*, la creación de un *pod*, como el resto de los objetos, se realiza mediante un fichero de texto llamado "*manifiesto*", con extensión "*yml*" o "*yaml*". En este fichero se han de definir los distintos componentes del objeto. En este caso al manifiesto se le ha denominado "*kaniko.yml*" y contiene lo siguiente (se explica más abajo):

```
apiVersion: v1
kind: Pod
metadata:
  name: kaniko
spec:
  containers:
  - name: kaniko
    image: gcr.io/kaniko-project/executor:latest #v1.7.0
    args:
    - "--dockerfile=./Dockerfile"
    - "--context=git://github.com/aragargra/ping-google"
    - "--destination=araceli96/ping-google:latest"
    volumeMounts:
    - name: docker-config
      mountPath: /kaniko/.docker/
  restartPolicy: Never
  volumes:
  - name: docker-config
    secret:
      secretName: secreto
      items:
          - key: .dockerconfigjson
            path: config.json
```

Todo manifiesto comienza con el campo "*apiVersion*" que identifica la versión de la *API* de *Kubernetes* del recurso que se quiere crear, en este caso, de un *pod*. La versión de la *API* que tiene asignado cada recurso se puede obtener con el comando *kubectl-api-resources*.

En el campo "*kind*" hay que indicar el tipo de objeto que se desea, en este caso, un *pod*.

Los metadatos, "*metadata*", son los valores que identifican y configuran al objeto, como por ejemplo el nombre, "*name*", del pod, p. ej., "*kaniko*".

El apartado de especificaciones, "*spec*", describe las características que debe tener el objeto, en este caso, el *pod*. Dentro de las especificaciones, en el apartado "*containers*", se definen los contenedores que va a tener el *pod*, en

este caso será un solo contenedor al que se le ha llamado "*kaniko*". El contenedor *kaniko* se levantará con la imagen de *kaniko*, "*gcr.io/kaniko-project/executor:latest #v1.7.0*", y se le pasan tres argumentos,"*args*", que usará la imagen de *kaniko*:

- "*dockerfile*": es el propio fichero *Dockerfile*.
- "*context*": es el repositorio de *GitHub* donde se encuentra los ficheros de la aplicación ("*servidor.js*", "*package.jon*") y el "*Dockerfile*".
- "*destination*": es el repositorio de *Docker Hub* donde se desea guardar la imagen de la aplicación.

Lo siguiente es montar un volumen en el contenedor ("*volumeMounts*") con el nombre "*docker-config*" en el *path* "*/kaniko/.docker/*" del contenedor.

La política "*restartPolicy: Never*" del *pod* hace que sus contenedores nunca se reinicien automáticamente.

Finalmente se define el volumen ("*volumes*") que se quiere montar en el contenedor, con el mismo nombre de antes, "*docker-config*", y con el secreto que se generó anteriormente ("*secreto*"). El fichero "*.dockerconfigjson*" se monta como indica el "*path*" en la ruta "*mountPath*", es decir, en "*/kaniko/.docker/config.json*" del contenedor, que es donde *Docker* va a tomar las credenciales para la validación.

Finalmente, se arranca el *pod* con el comando *kubectl apply -f kaniko.yml*:

```
araceli@ubuntu:~/mi-app$ kubectl apply -f kaniko.yml
pod/kaniko created
araceli@ubuntu:~/mi-app$ kubectl get all
NAME         READY   STATUS      RESTARTS   AGE
pod/kaniko   0/1     Completed   0          77s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.8.0.1     <none>        443/TCP   2d
araceli@ubuntu:~/mi-app$
```

En la imagen anterior se puede observar que el *pod* "*kaniko*" ha terminado su trabajo ("*STATUS*" igual a "*Completed*") una vez construida la imagen. Se puede comprobar que, en efecto, se ha subido la imagen de la aplicación al repositorio de *Docker Hub*:

4.3.4 Despliegue de la aplicación

En *Kubernetes* existen varias formas de crear un *pod* con el que desplegar una aplicación:

- Directamente, mediante el objeto *Pod*, al igual que se hizo en el apartado anterior. Las réplicas de los *pods* no estarán orquestadas.
- Mediante un objeto *Deployment*. Los *pods* quedarán orquestados, es decir, se mantendrá siempre el número de réplicas especificado en el *deployment*. Esta es la forma recomendada para desplegar aplicaciones.
- Mediante un objeto *DaemonSet*. En este caso se creará tantos *pods* orquestados como nodos tenga el *cluster*. Uso típico en monitorización.
- Mediante un objeto *StatefulSet*. Es similar al *Deployment*, pero en este caso se crea un volumen montado en los *pods* de manera que los datos de los *pods* nunca se pierdan, aunque fallen.

Por las características de la aplicación del caso, se va a crear un *pod* mediante un objeto "*deployment*", así como un objeto *service* para exponer la aplicación en el puerto 8080. Al manifiesto se le ha llamado "*mi-pod.yml*":

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mi-despliegue
spec:
  replicas: 1
  selector:
    matchLabels:
      role: identificador
  template:
    metadata:
      labels:
        role: identificador
    spec:
      containers:
      - name: mi-contenedor
        image: araceli96/ping-google:latest
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: mi-servicio
spec:
  type: NodePort
  ports:
  - port: 8085
```

```
    targetPort: 8080
    nodePort: 30000
  selector:62
    role: identificador
```

Como se aprecia, para crear un objeto *deployments* se debe usar la *apiVersion* "*apps/v1*". El nombre que se le ha dado al *deployments* es "*mi-despliegue*". En las especificaciones del despliegue ("*spec*") se indica el número de réplicas ("*replicas: 1*"). En el campo "*spec.selector.matchLabels*" se identifican los *pods* que el *deployments* va a administrar, en este caso, los *pods* que tengan la clave-valor "*role: identificador*". El campo "*spec.template*" es la plantilla que define las características de los *pods* que se van a crear. En el campo "*spec.template.metadata.labels*" se asigna una clave-valor a los *pods*, que debe coincidir con el campo "*spec.selector.matchLabels*". En el campo "*spec.template.spec.containers*" se definen los contenedores que van a tener los *pods*, en este caso, solo uno, con el nombre "*mi-contenedor*", con la imagen "*araceli96/ping-google:latest*", y que escuchará en el puerto indicado en "*containerPort*", es decir, 8080.

En el manifiesto también se ha definido un objeto *service* con la *apiVersion* "*v1*", y el nombre "*mi-servicio*". El tipo de servicio es "*NodePort*", el cual permite el acceso desde el exterior del *cluster* a través de un puerto (*30000*) y con la *IP* pública de cualquiera de los nodos del *cluster* (independientemente donde se alojen los pods que atienden el servicio). El campo *spec.ports.port* es el puerto del servicio que se expone internamente en el *cluster*. El campo "*targetPort*" es puerto por el que escuchan los *pods* (y que debe coincidir con el campo "*containerPort*"), y el "*nodePort*" es el puerto que se expone externamente al *cluster* (es decir, con el que se accederá al servicio desde fuera del *cluster*). En la siguiente figura se aprecia el mapeo de estos puertos:

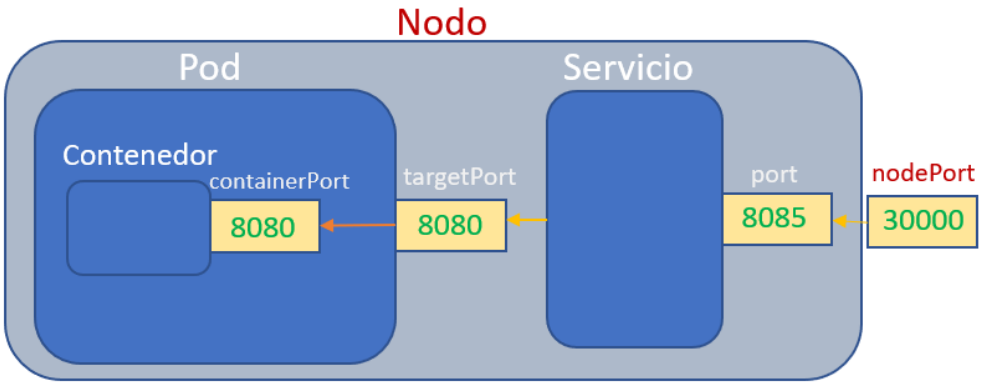


Figura 19. Estructura de puertos de un servicio de *Kubernetes* (elaboración propia)

Con el comando *kubectl apply -f mi-pod.yml* se realiza el despliegue de los *pods* y del servicio:

```
araceli@ubuntu:~$ kubectl apply -f mi-pod.yml
deployment.apps/mi-despliegue created
service/mi-servicio created
```

Los objetos que se ahora están disponibles en el *cluster* son los siguientes:

```
araceli@ubuntu:~$ kubectl get all
NAME                                    READY   STATUS      RESTARTS   AGE
pod/kaniko                              0/1     Completed   0          52s
pod/mi-despliegue-56f8db89f7-7m8qw      1/1     Running     0          41s

NAME                    TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)          AGE
service/kubernetes      ClusterIP   10.8.0.1      <none>        443/TCP          73m
service/mi-servicio     NodePort    10.8.9.57     <none>        8085:30000/TCP   41s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mi-despliegue       1/1     1            1           42s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/mi-despliegue-56f8db89f7       1         1         1       42s
```

El comando *kubectl get pods -o wide* permite visualizar dónde se ha alojado el *pod*:

```
araceli@ubuntu:~$ kubectl get pods -o wide
NAME                              READY   STATUS      NODE
kaniko                            0/1     Completed   gke-cluster-1-default-pool-6a14c54b-7rcl
mi-despliegue-56f8db89f7-7m8qw    1/1     Running     gke-cluster-1-default-pool-6a14c54b-7rcl
```

Ya se puede probar la aplicación desde un navegador, con la *IP* de cualquiera de los nodos del *cluster* y el puerto 30000:

Como se aprecia en la imagen anterior, el servicio se ha atendido por el único *pod* del despliegue.

4.3.5 Pruebas de orquestación

Aquí se van a realizar las mismas pruebas de escalabilidad, balanceo de carga y disponibilidad que se hicieron en la plataforma *Docker*.

Prueba 1. Escalado horizontal de la aplicación

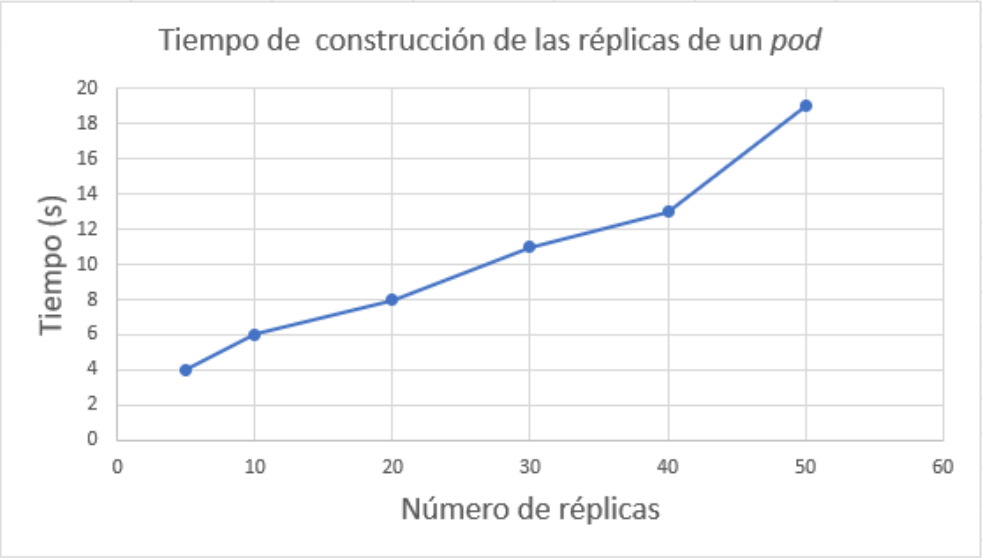
El escalado (horizontal) de la aplicación, por ejemplo, con 3 *pods*, se realiza con el siguiente comando:

```
kubectl scale --replicas=3 deployment/mi-despliegue
```

En efecto, ahora tenemos tres *pods* para el servicio, más el *pod* de *kanico*. Este último está ya terminado (estado "*completed*") ya que su trabajo ha finalizado:

```
araceli@ubuntu:~$ kubectl get pods
NAME                              READY   STATUS      RESTARTS   AGE
kaniko                            0/1     Completed   0          12m
mi-despliegue-56f8db89f7-2z9qw    1/1     Running     0          15s
mi-despliegue-56f8db89f7-7m8qw    1/1     Running     0          12m
mi-despliegue-56f8db89f7-b9z9n    1/1     Running     0          15s
```

De los distintos ensayos que se han realizado, se ha obtenido una gráfica en la observa el tiempo que *Kubernetes* ha destinado para crear las réplicas:



Se observa que el tiempo que *Kubernetes* tarda en levantar las réplicas es bastante lineal con respecto al número de ellas.

Prueba 2. Balanceo de carga entre pods

Con esta prueba se demostrará que el orquestador balancea el tráfico entre los distintos *pods* que atienden el servicio. En este caso, el tráfico entra al *cluster* por el puerto 30000 y debe repartirse uniformemente entre los tres *pods*.

Se realizan varias solicitudes para ver qué contenedor lo atiende:

Ping a Google desde el servidor mi-despliegue-56f8db89f7-b9z9n

nombre del *pod* que atiende el servicio

Ping a Google desde el servidor mi-despliegue-56f8db89f7-7m8qw

Ping a Google desde el servidor mi-despliegue-56f8db89f7-2z9qw

Se aprecia como, en efecto, las distintas solicitudes se van atendiendo uniformemente en un orden determinado, entre los distintos *pods*, método denominado "***round-robin***", que es el que está configurado en *Kubernetes* por defecto. Existen otros métodos de balanceo de carga [42], como el "***consistent hash***" que utiliza un algoritmo *hash* para enviar todas las solicitudes de un cliente (o sesión) a un mismo *pod*, el "***resource based/least load***" que

encamina las peticiones al *pod* menos cargado, o el "***least connections***" que distribuye las peticiones de los clientes al *pod* que tenga menos conexiones activas.

Prueba 3. Reparto uniforme de *pods* entre los nodos

En esta prueba se comprobará que los *pods* que atienden el servicio se han ubicado uniformemente entre los distintos nodos del *cluster*. El comando *kubectl get pods -o wide* proporciona la ubicación de los distintos *pods*, en efecto:

```
araceli@ubuntu:~$ kubectl get pods -o wide
NAME                              READY   NODE
kaniko                            0/1     gke-cluster-1-default-pool-6a14c54b-7rcl
mi-despliegue-56f8db89f7-2z9qw    1/1     gke-cluster-1-default-pool-6a14c54b-64nk
mi-despliegue-56f8db89f7-7m8qw    1/1     gke-cluster-1-default-pool-6a14c54b-7rcl
mi-despliegue-56f8db89f7-b9z9n    1/1     gke-cluster-1-default-pool-6a14c54b-61vw
```

Se observa que cada *pod* del servicio se ha alojado en un nodo distinto.

Prueba 4. Desescalado de la aplicación

El desescalado de la aplicación se realiza con el mismo comando que el escalado, por ejemplo, si se quiere dejar ahora únicamente dos *pods*:

kubectl scale --replicas=2 deployment/mi-despliegue

Se comprueba que, de los tres *pods* iniciales, se ha eliminado uno de ellos, quedando ahora únicamente dos *pods*, y que estos se han distribuido uniformemente entre los nodos del *cluster*:

```
araceli@ubuntu:~$ kubectl get pods -o wide
NAME                              READY   NODE
kaniko                            0/1     gke-cluster-1-default-pool-6a14c54b-7rcl
mi-despliegue-56f8db89f7-2z9qw    1/1     gke-cluster-1-default-pool-6a14c54b-64nk
mi-despliegue-56f8db89f7-7m8qw    1/1     gke-cluster-1-default-pool-6a14c54b-7rcl
```

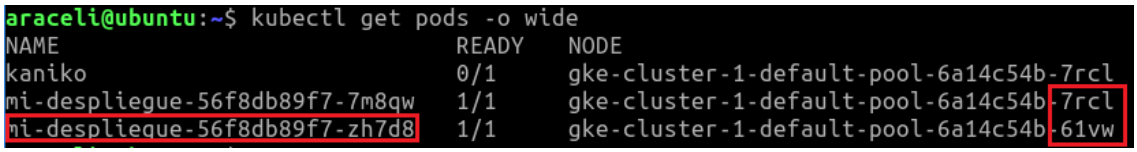
El servicio sigue funcionando perfectamente:

Prueba 5. Mantenimiento de *pods* o fallo de un *pod*

En esta prueba se va a simular el fallo del primer *pod* de la imagen anterior borrando dicho *pod*, con el siguiente comando:

kubectl delete pod mi-despliegue-56f8db89f7-2z9qw

Los *pods* que ahora tiene el *cluster* son los siguientes:



Se observa que el orquestador ha levantado automáticamente un nuevo *pod* con el fin de mantener constante el número de *pods* del servicio. También se aprecia que los dos *pods* se han alojan en nodos distintos balanceando así la carga en el *cluster*.

Prueba 6. Escalado horizontal automático (*HPA*)

Kubernetes soporta el escalado automático ("*HPA*") de la aplicación con el fin de adaptar el número de *pods* a la carga que debe soportar el servicio. Con el siguiente comando se va a incrementar el número de *pods* cuando éstos comiencen a usar más de un 50% de su *CPU*, manteniendo siempre un mínimo de 1 *pod* y un máximo de 6 *pods*.

kubectl autoscale deployment mi-despliegue --cpu-percent=50 --min=1 --max=6

La desactivación del escalado horizontal se realiza de la siguiente forma:

kubectl delete hpa <id del despliegue>

A diferencia del escalado horizontal (*HPA*), el escalado vertical (*VPA*) modifica los recursos de *CPU* y memoria reservados en los *pods* para adecuarse a la carga soportada por el servicio. Cuando la carga cambia repentinamente es preferible utilizar *HPA*, mientras que si es sostenida en el tiempo es conveniente usar *VPA*. En este trabajo no han realizado pruebas de escalado automático por la dificultad de simular peticiones simultáneas.

4.4 Despliegue en la plataforma *OpenShift*

Como ya se comentó, se va a trabajar con la versión de *OpenShift* denominada *MiniShift* ya que no se dispone de un *cluster* propietario para usar *OKD*, y el resto de las versiones, o no son gratuitas *(OpenShift Online*, etc.) o su instalación no es compatible con *Windows 10 Home* (*CRC*). *MiniShift* crea un *cluster* virtualizado *Linux* de un solo nodo en una máquina utilizando un hipervisor (como *VirtualBox*). No se instalará en la máquina local de *Ubuntu* para evitar una virtualización anidada, haciéndolo directamente sobre el sistema operativo *Windows*.

Los procesos que se van a describir en este apartado aparecen destacados (fondo anaranjado) en la siguiente imagen:

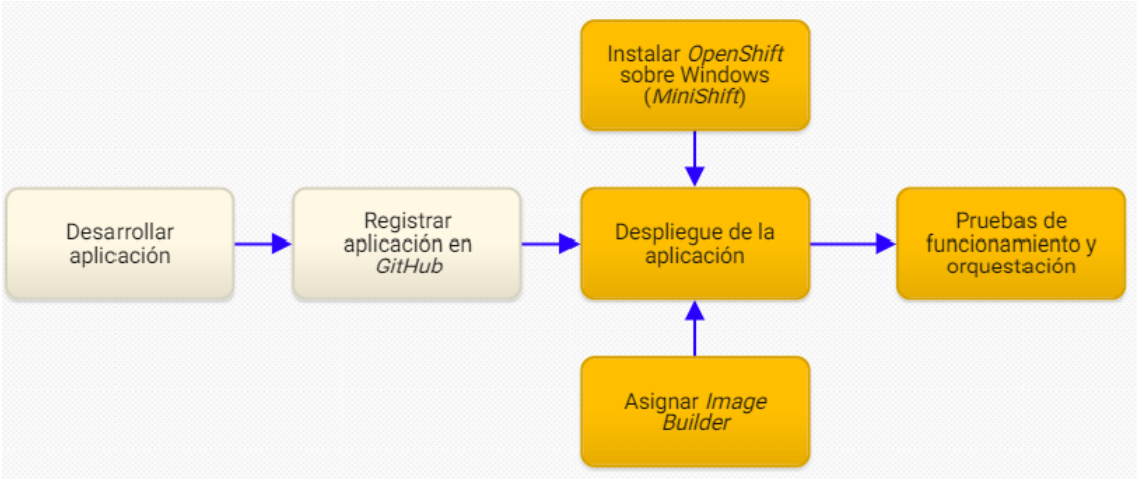


Figura 20. Proceso de despliegue en un *cluster* de *OpenShift* (*MiniShift*)

4.4.1 Instalación de *MiniShift* en la máquina local *Windows*

La descarga de *MiniShift* se puede realizar desde el repositorio de *GitHub* [26]. La última versión de *MiniShift* para un equipo *Windows* de 64 bits es la *v1.34.3*, y corresponde con el fichero comprimido "*minishift-1.34.3-windows-amd64.zip*". Este fichero se debe descomprimir en cualquier directorio, por ejemplo, "*C:/minishift*".

Para poder acceder a *MiniShift* desde cualquier carpeta de *Windows* hay que agregar el directorio anterior a la variable de entorno *PATH* siguiendo los pasos indicados en [39]. Tras ello, hay que abrir un terminal (*CMD*) de *Windows* como administrador, e introducir el siguiente comando para asignar a *VirtualBox* como hipervisor para virtualizar el nodo del *cluster OpenShift*:

minishift config set vm-driver virtualbox

Ahora hay que arrancar *MiniShift*, con lo que se levantará un nodo en el *cluster* con 4GB de RAM, 2 CPUs y 20GB de disco:

minishift start

La salida del comando anterior proporciona la *URL* de acceso a la consola de *Minishift*, p. ej., *https://192.168.99.127:8443*, así como información de acceso para un usuario "*developer*" y un usuario administrador:

```
The server is accessible via web console at:
    https://192.168.99.127:8443/console

You are logged in as:
    User:     developer
    Password: <any value>

To login as administrator:
    oc login -u system:admin
```

Para que el *cluster* permita a cualquier usuario ("*anyuid*") ejecutar *pods* es necesario introducir el siguiente comando:

minishift addon apply anyuid

Si se desea ejecutar los comandos "*oc*" de *OpenShift* desde cualquier directorio de *Windows* es necesario conocer la ruta en la que se ha instalado la herramienta "*oc*", con el comando *minishift oc-env*:

```
C:\WINDOWS\system32>minishift oc-env
SET PATH=C:\Users\juanl\.minishift\cache\oc\v3.11.0\windows;%PATH%
REM Run this command to configure your shell:
REM     @FOR /f "tokens=*" %i IN ('minishift oc-env') DO @call %i
```

, y finalmente, se añade esta ruta a la variable de entorno *PATH*:

SET PATH=C:\Users\juanl\.minishift\cache\oc\v3.11.0\windows;%PATH%

Para parar el *cluster* se introduce el comando *"minishift stop*", para volverla a arrancar, "*minishift start*", y para borrar el *cluster*, "*minishift delete*".

4.4.2 Despliegue mediante la consola web de *MiniShift*

Desde la consola web de *OpenShift* se pueden realizar entre el 80% y el 90% de las tareas [43], sin necesidad de entrar en la línea de comandos ("*oc*"), mejorando muchísimo la experiencia de usuario.

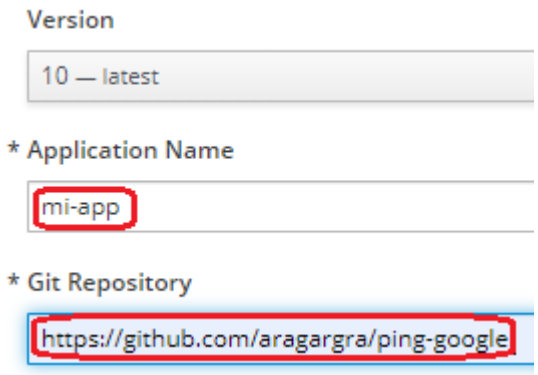
Para entrar en la consola de *MiniShift* hay acceder con un navegador a la *URL* que se obtuvo en la instalación, tal como *https://192.168.99.127:8443/console*, tras lo cual hay que validarse. La instalación proporciona un usuario "*developer*" (y cualquier *password*), y un usuario *admin* (y contraseña *admin*):

Tras la validación aparecerá la consola de trabajo de *OpenShift*. Ahora hay que crear un proyecto ("*project*"). Un proyecto de *OpenShift* es prácticamente lo mismo que un "*namespace*" de *Kubernetes*, es decir, una forma de agrupar todos los objetos que se necesiten, como imágenes, *pods*, etc. Para crear un nuevo proyecto se ha de pulsar en "*Create Project*", y luego introducir un nombre para el proyecto, por ejemplo, "*mi-proyecto*". Una vez creado el proyecto hay que seleccionarlo pulsando sobre él.

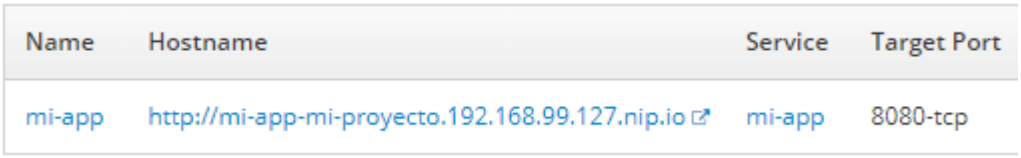
A diferencia de *Kubernetes* (y de *Docker*), *OpenShift* puede construir una imagen directamente del código fuente de la aplicación mediante una herramienta denominada *S2I* ("*Source to Image*"). *S2I* utiliza una imagen constructora llamada "*image builder*" como base para crear la imagen, y un objeto "*buildconfig*" para crea la imagen. Existen distintos tipos de "*image builder*" dependiendo del lenguaje en que se haya escrito la aplicación.

Para crear una aplicación *OpenShift*, se debe seleccionar previamente la imagen constructora desde el repositorio "*Browse Catalog*", seleccionando la

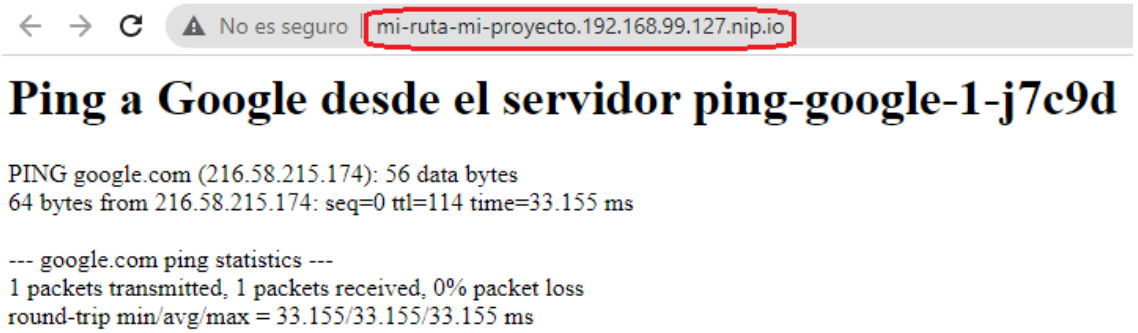
"*image builder*" de la plataforma de desarrollo de la aplicación correspondiente, en este caso, *Nodejs*. Tras ello, *OpenShift* solicitará un nombre para la aplicación y la *URL* del repositorio donde se encuentra el código de la aplicación, en este caso, *GitHub*, tal como aparece en la siguiente imagen:



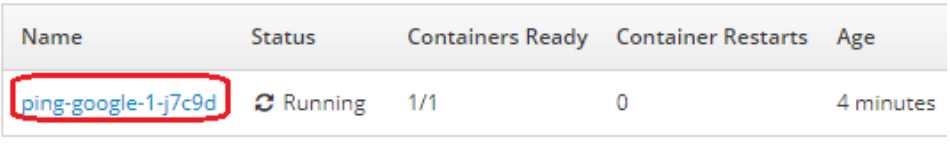
La *URL* asignada a la aplicación se puede obtener seleccionando *Applications/Routes*, tal como se muestra:



Ya se puede probar la aplicación introduciendo esta *URL* en un navegador, o pulsando sobre el propio enlace anterior. El resultado es el siguiente:

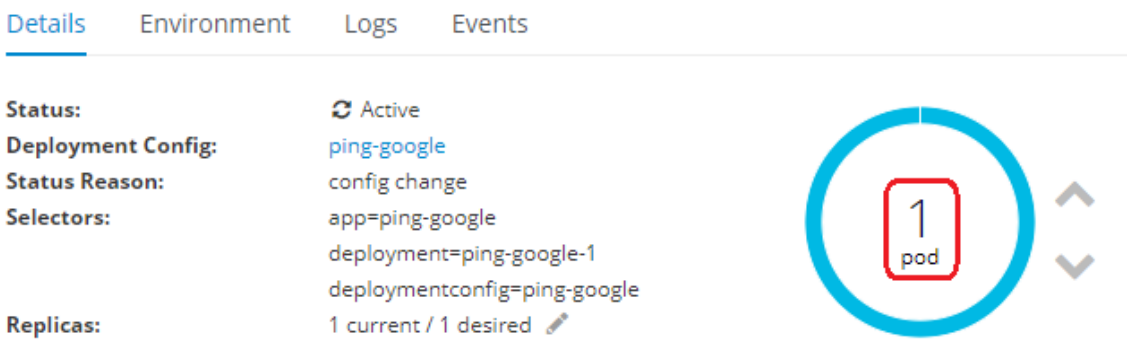


Los *pods* de la aplicación se puede visualizar en el apartado *Applications/Pods*, y luego pulsando en el enlace del *pod*:



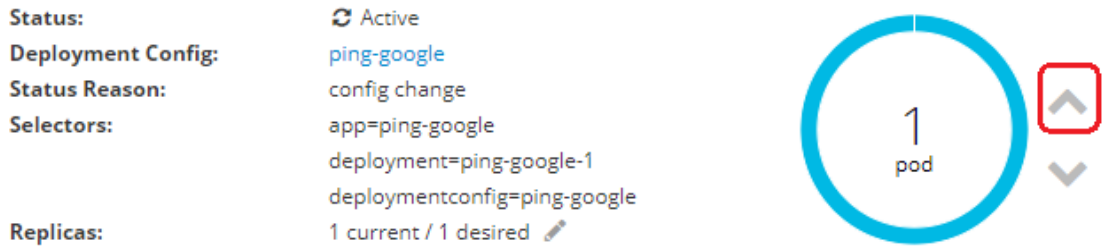
Para ir a los detalles de un *pod* se pulsa en "*#1*":

, y entre otros datos, se puede observar que el *pod* tiene una única réplica:

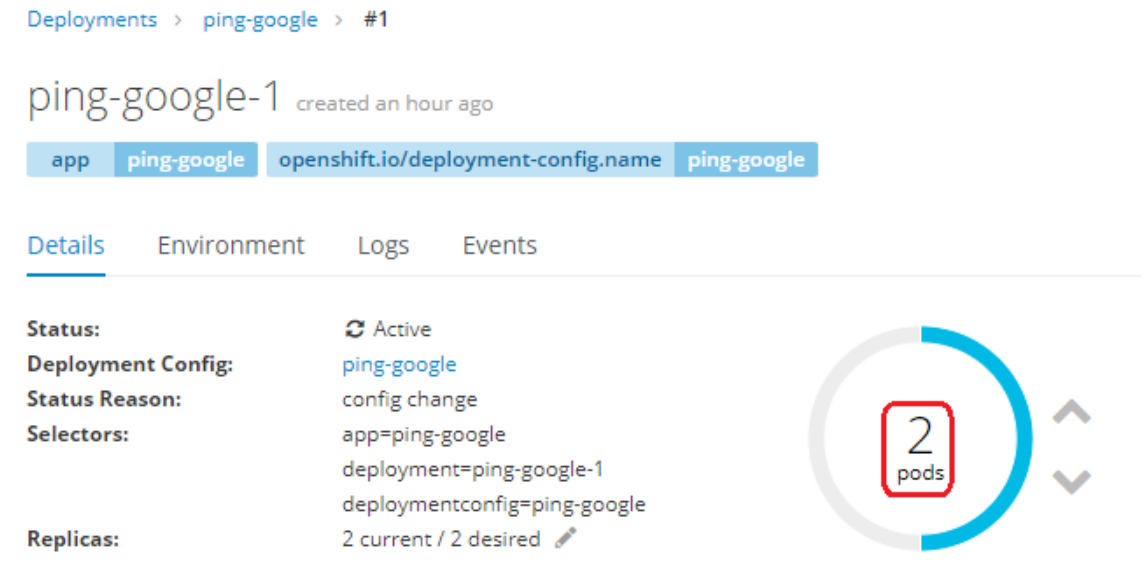


Prueba 1 de orquestación. Escalado horizontal de la aplicación

Para incrementar el número de *pod* simplemente hay que pulsar en la flecha "*Scale up*":



, y de forma casi instantánea se genera una nueva réplica del *pod*, en efecto:



En el apartado *Applications/Pods* aparecerán ya las dos réplicas del *pod*:

Name	Status	Containers Ready	Container Restarts	Age
ping-google-1-vgf2f	Running	1/1	0	4 minutes
ping-google-1-j7c9d	Running	1/1	0	15 minutes

Prueba 2 de orquestación. Mantenimiento de *pods* o fallo de *pods*

Se va a simular el fallo de la réplica del *pod* "*ping-google-1-j7c9d*" eliminándolo manualmente. Se pulsa sobre el enlace de la réplica que se desea borrar:

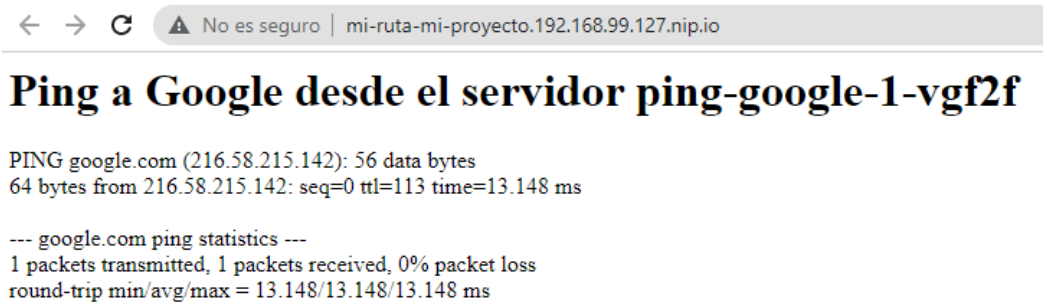
Name	Status	Containers Ready	Container Restarts	Age
ping-google-1-vgf2f	Running	1/1	0	4 minutes
ping-google-1-j7c9d	Running	1/1	0	15 minutes

, y en la esquina superior derecha se selecciona *Actions*/*Delete*:

Se puede comprobar que automáticamente se levanta una nueva réplica del *pod* con el objetivo de mantener el número de réplicas asignado en el despliegue:

Name	Status	Containers Ready	Container Restarts	Age
ping-google-1-wss2m	Running	1/1	0	a minute
ping-google-1-vgf2f	Running	1/1	0	9 minutes

La aplicación sigue funcionando perfectamente:



4.4.3 Despliegue mediante comandos *oc*

En este apartado se desplegará la aplicación en *OpenShift* utilizando comandos "*oc*" (*OpenShift Command Line Interface*) en lugar de la consola. Lo primero es validarse en el *cluster* de *OpenShift* con el comando "*oc login*":

```
C:\Users\Araceli>oc login
Authentication required for https://192.168.99.127:8443 (openshift)
Username: developer
Password:
Login successful.

You have access to the following projects and can switch between them with 'oc
 project <projectname>':

  mi-proyecto
* myproject

Using project "myproject".
```

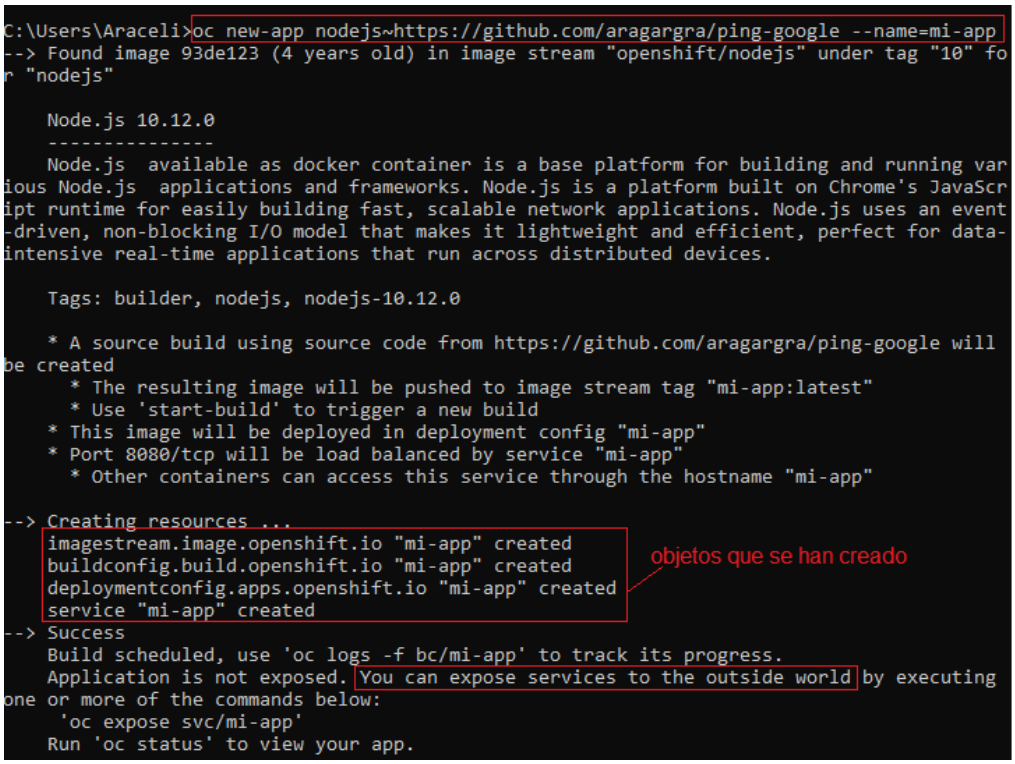
La salida del comando anterior devuelve los proyectos que están disponibles para el usuario. En este caso únicamente se dispone del proyecto por defecto "*myproject*", y el que se creó en el apartado anterior. El que aparece con un asterisco es el proyecto de trabajo. Se puede cambiar de proyecto de trabajo con el comando "*oc project <proyecto>*".

Para crear un nuevo proyecto se introduce el comando "*oc new-project <proyecto>*", por ejemplo, *oc new-project mi-proyecto2*:

```
C:\Users\Araceli>oc new-project mi-proyecto2
Already on project "mi-proyecto2" on server "https://192.168.99.127:8443".

You can add applications to this project with the 'new-app' command. For example, try:

    oc new-app centos/ruby-25-centos7~https://github.com/sclorg/ruby-ex.git

to build a new example application in Ruby.
```

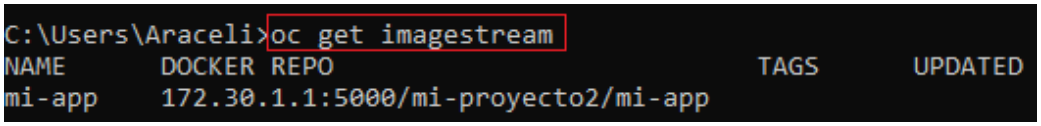
Como indica la salida anterior, para crear la aplicación en *OpenShift* hay que introducir el comando "*oc new-app*", indicando la imagen constructora (en este caso "*nodejs*"), seguido del carácter virgulilla, "~", y del repositorio *GitHub* donde se encuentra el código de la aplicación. Se puede asignar un nombre a la aplicación, con la opción "*--name*". Es posible que no sea necesario indicar el "*image builder*" ya que *OpenShift* lo detecta automáticamente del tipo de ficheros de la aplicación, pero si no se indica y en el repositorio *GitHub* se encuentra un *dockerfile*, entonces se construirá la imagen utilizando este *dockerfile*, en lugar de usar un *image builder*. Una forma de forzar a que se utilice un *dockerfile* para crear la imagen es utilizar la opción "*--strategy=docker*", o "*--strategy=source*" si se quiere usar un *image builder*:

oc new-app nodejs~https://github.com/aragargra/ping-google --name=mi-app

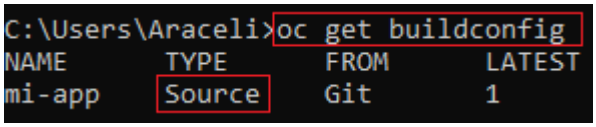


Como se observa en la imagen anterior, se han generado los siguientes objetos:

- Un *imagestream*. A diferencia de *Kubernetes*, *OpenShift* maneja las imágenes mediante un objeto llamado *ImageStream*, que es una referencia o puntero a la imagen almacenada en un repositorio, lo que hace mucho más flexible la gestión de imágenes.



- Un *buildconfig*. Como ya se dijo, es el objeto encargado de construir la imagen a partir del código de la aplicación y de un *image builder*.



  En la salida anterior, el campo "*TYPE*" indica el tipo de construcción de la imagen. En este caso es "*Source*" ya que se ha realizado a partir del de un "*image builder*" con la herramienta *S2I*. También se indica que el código de la aplicación se ha obtenido de un repositorio *Git*.

- Un *deploymentconfig*. Es un objeto similar al *deployment* de *Kubernetes*, es decir, encargado de crear los *pods* orquestados.

```
C:\Users\Araceli>oc get deploymentconfig
NAME      REVISION   DESIRED   CURRENT   TRIGGERED BY
mi-app    1          1         1         config,image(mi-app:latest)
```

- Un *service*. Es un objeto que expone la aplicación internamente al *cluster*. Para exponerla al exterior es necesario crear una ruta ("*route*").

```
C:\Users\Araceli>oc get service
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
mi-app    ClusterIP   172.30.217.90   <none>        8080/TCP   3m
```

Se observa en la salida del comando anterior que se ha creado un servicio tipo "*ClusterIP*", que es el mismo que utiliza *Kubernetes* para exponer una aplicación internamente en el *cluster*.

Los *pods* que se han generado para esta aplicación son los siguientes:

```
C:\Users\Araceli>oc get pods
NAME              READY     STATUS      RESTARTS   AGE
mi-app-1-build    0/1       Completed   0          6m
mi-app-1-qn64l    1/1       Running     0          6m
```

Como se aprecia, aparecen dos *pods*. Uno corresponde a la única réplica de la aplicación, y que se encuentra en estado de "*running*", y el otro es el *pod* que *OpenShift* ha utilizado para construir la imagen. El nombre de este último *pod* termina en "*build*" y se encuentra en estado "*completed*" al haber completado su trabajo con la construcción de la imagen. Si en el despliegue se parte de una imagen de *Docker* (en lugar del código de la aplicación) no se creará un *pod* del tipo "*build*".

Una vez arrancada la aplicación es necesario acceder a ella desde el exterior, y para ello hay que crear una ruta que exponga el servicio. Esto se realiza con el comando "*oc expose service*" seguido del nombre de la aplicación:

```
oc expose service mi-app
```

La ruta que *OpenShift* ha creado se puede obtener con el siguiente comando *oc get routes*:

```
C:\Users\Araceli>oc get routes
NAME      HOST/PORT
mi-app    mi-app-mi-proyecto2.192.168.99.127.nip.io
```

Esta ruta o *URL* se puede introducir en un navegador para probar la aplicación:

El comando para desplegar una aplicación partiendo de su imagen (p. ej., del repositorio de *Docker Hub*) es el siguiente:

oc new-app <nombre_imagen> --name=<nombre_app>

Prueba 1 de orquestación. Escalado horizontal de la aplicación

El siguiente comando escalará la aplicación con 5 *pods*:

oc scale --replicas=5 deploymentconfig mi-app

Los *pods* de la aplicación más el *pod* de construcción de la imagen ("*build*") son los siguientes:

```
C:\Users\Araceli>oc get pods
NAME              READY     STATUS      RESTARTS   AGE
mi-app-1-bk59g    1/1       Running     0          42s
mi-app-1-build    0/1       Completed   0          12m
mi-app-1-h9lgj    1/1       Running     0          42s
mi-app-1-kr277    1/1       Running     0          42s
mi-app-1-qn64l    1/1       Running     0          12m
mi-app-1-zlp5d    1/1       Running     0          42s
```

Como se aprecia, el servicio que ofrece la aplicación está ahora atendido por 5 *pods*, o como también se suele decir, un *pod* con 5 réplicas.

Prueba 2 de orquestación. Balanceo de carga entre pods

Para realizar esta prueba es necesario eliminar las *cookies* del navegador en cada nueva petición ya que *OpenShift* utiliza estas *cookies* para mantener sesiones persistentes y, por tanto, las peticiones desde el mismo *host* siempre serán atendida el mismo *pod*. Con *Docker* y *Kubernetes* esto no ocurre.

El resultado de las peticiones realizadas son las siguientes (se indica únicamente la primera línea con el nombre del *pod* que atiende la solicitud):

Ping a Google desde el servidor mi-app-1-zlp5d
Ping a Google desde el servidor mi-app-1-qn64l
Ping a Google desde el servidor mi-app-1-bk59g
Ping a Google desde el servidor mi-app-1-h9lgj
Ping a Google desde el servidor mi-app-1-kr277
Ping a Google desde el servidor mi-app-1-zlp5d
Ping a Google desde el servidor mi-app-1-qn64l

En efecto, se observa que las peticiones se van distribuyendo uniformemente entre los pods que atienten el servicio, método llamado *round-robin*.

Prueba 3 de orquestación . Desescalado de la aplicación

El comando para desescalar es el mismo que el del escalado, por ejemplo, si se quiere dejar únicamente tres *pods* se introduce el siguiente comando:

oc scale --replicas=3 deploymentconfig mi-app

Se puede comprobar que ahora solo hay 3 *pods* que atienden el servicio:

```
C:\Users\Araceli>oc get pods
NAME             READY     STATUS      RESTARTS   AGE
mi-app-1-bk59g   1/1       Running     0          17m
mi-app-1-build   0/1       Completed   0          28m
mi-app-1-qn64l   1/1       Running     0          28m
mi-app-1-zlp5d   1/1       Running     0          17m
```

Prueba 4 de orquestación. Mantenimiento de *pods* o fallo de un *pod*

En este apartado se simulará el fallo del primer *pod* de la lista anterior, procediendo a su eliminación, con el siguiente comando:

oc delete pod mi-app-1-bk59g

Los *pods* que ahora están disponibles para el servicio son los siguientes:

```
C:\Users\Araceli>oc get pods
NAME             READY     STATUS      RESTARTS   AGE
mi-app-1-build   0/1       Completed   0          30m
mi-app-1-kgd82   1/1       Running     0          29s
mi-app-1-qn64l   1/1       Running     0          30m
mi-app-1-zlp5d   1/1       Running     0          18m
```

Como se aprecia, el orquestador de *OpenShift* ha levantado automáticamente el *pod* "*mi-app-1-kgd82*" para mantener las tres réplicas, como indica su despliegue.

La aplicación sigue funcionando perfectamente:

Prueba 6 de orquestación. Escalado horizontal automático

OpenShift, igual que *Kubernetes*, permite realizar un escalado automático ("*hpa*") de la aplicación con el comando "*oc autoscale dc*", por ejemplo:

```
oc autoscale dc mi-app --cpu-percent=50 --min=1 --max=6
```

El comando anterior indica que se debe escalar el número de *pods* cuando éstos usen más de un 50% de su *CPU*, manteniendo siempre un mínimo de 1 *pod* y un máximo de 6 *pods*.

Este escalado horizontal se puede eliminar en cualquier momento con el comando *oc delete hpa mi-app*.

4.5 Resumen

En este capítulo se ha realizado el despliegue de una aplicación en tres sistemas de contenedores, además de las pertinentes pruebas de escalabilidad, balanceo de carga y disponibilidad, que han puesto de manifiesto los beneficios de la orquestación.

Tal como se vio en la figura 15, y que aquí se vuelve a incluir, en este capítulo se han desarrollado todos los procesos que intervienen en el alojamiento de aplicaciones en una plataforma de contenedores de la nube pública de *Google*.

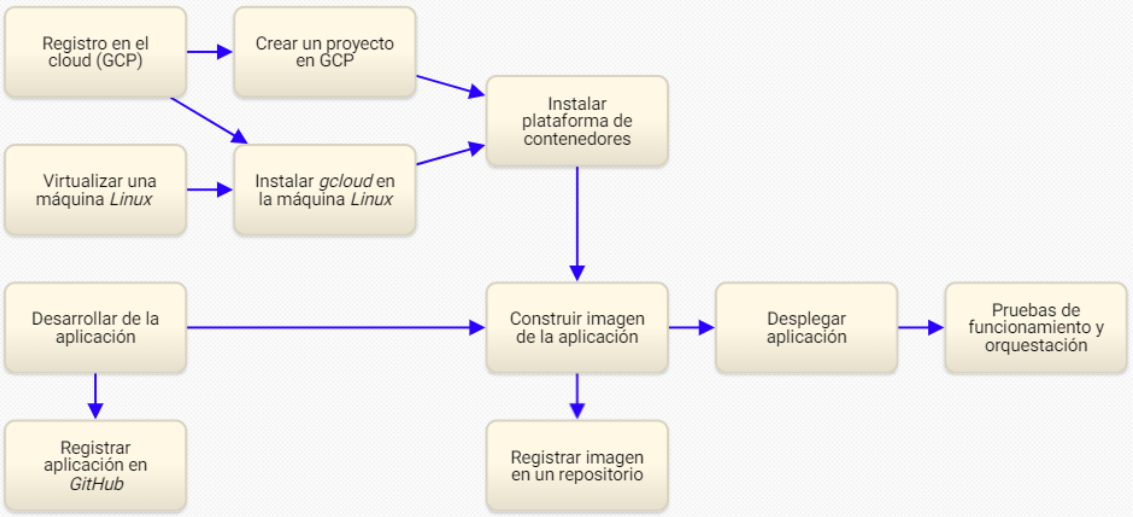


Figura 15. Procesos despliegue en un entorno *cloud* (elaboración propia)

A continuación, se realiza un breve resumen de cada uno de los procesos:

Registro en una nube pública. En este punto se ha procedido al registro en una plataforma *cloud*, concretamente en la nube de *Google* desde su portal *web* denominado *Google Cloud Platform*. Este proveedor proporciona un acceso gratuito durante 90 días con un crédito máximo de 300€ en consumo de recursos. En el caso de *OpenShift* no ha sido necesario este registro pues se ha utilizado un *cluster* alojado en la máquina local con la distribución *MiniShift*.

Crear un proyecto en la plataforma *cloud*. Un proyecto de *Google Cloud* es la entidad sobre la que se posteriormente se han creado, habilitado y usado los recursos necesarios de *Google Cloud* como, por ejemplo, las máquinas virtuales que van a componer los nodos del *cluster*, la agrupación de estos nodos en el propio *cluster*, el firewall para la apertura de puertos, etc. Como se ha indicado en el punto anterior, en el caso de *OpenShift*, al utilizar *MiniShift*, no ha sido necesario consumir recursos *cloud*.

Virtualizar una máquina *Linux*. En este punto se ha virtualizado una máquina *Linux* (concretamente de *Lubuntu*) en el host local de *Windows* desde la que se han realizado los despliegues en las plataformas *Docker* y *Kubernetes*. No se ha efectuado directamente desde la máquina local *Windows* dado que el sistema operativo *Linux* está mejor soportado por estas plataformas de contenerización. Con *OpenShift* no ha sido necesario realizar este apartado, pues *MiniShift* virtualiza automáticamente un *cluster* de un nodo en la propia máquina local, y si se hubiera instalado *MiniShift* sobre *Linux* se estaría realizando una doble virtualización, que no es recomendable.

Instalar *gcloud* en la máquina *Linux*. *Gcloud* es la herramienta de *Google* que ha permitido la conexión de la máquina local con el *cluster* de *Google Cloud* para poder consumir los recursos del *cloud*, tanto en la plataforma *Docker* como en *Kubernetes*. Con *OpenShift* no ha sido necesario por lo ya expuesto en los puntos anteriores.

Instalar plataforma de contenedores. Aquí se describe la instalación y configuración de la plataforma de contenerización tanto en la propia máquina local como en la infraestructura *cloud*. Con *Docker* se ha instalado el cliente "*docker*", desde el que se ha configurado los nodos con *Docker Machine*, y el propio *cluster* con *Docker Swarm*. Con *Kubernetes* se ha han configurado los nodos y el *cluster* directamente desde el portal del proveedor *cloud*, y el cliente "*kubectl*" en la propia máquina local. Con *OpenShift* únicamente ha sido necesario instalar *MiniShift* ya que esta distribución implementa internamente un *cluster* de un solo nodo en la propia máquina local.

Desarrollar la aplicación. En este punto se ha codificado una aplicación *Nodejs*, que es la que posteriormente se ha utilizado para desplegarla en los contenedores de las distintas plataformas. La aplicación *Nodejs* se compone del fichero "*servidor.js*" que contiene el código *JavaScript* para levantar un servidor en un *host*, y el fichero "*package.json*" donde se establecen las dependencias de la aplicación.

Registrar aplicaciones en *GitHub*. Este es el repositorio donde se ha almacenado el código de la aplicación, es decir, los ficheros del punto anterior. Este repositorio se utilizará tanto en *Kubernetes* como en *OpenShift* para construir la imagen. En *Docker* no es necesario este registro.

Construir la imagen de la aplicación. En este punto se ha creado una imagen de la aplicación a partir de su código y de sus dependencias. Estos trabajos han sido diferentes en función de la plataforma de contenedores. Con *Docker* se ha realizado a través de un fichero denominado *Dockerfile*. Con *Kubernetes* ha sido necesario utilizar una herramienta externa denominada *Kaniko* para poder construir la imagen *Docker* desde un *cluster* cuyos nodos no disponen del motor de *Docker*. Con *OpenShift* se ha construido la imagen directamente a partir del código de la aplicación utilizando una herramienta interna denominada *S2I* ("*Source to Image*").

Registrar imagen en un repositorio. Una vez creada la imagen de la aplicación ésta se ha registrado en el repositorio *Docker Hub*, desde el que posteriormente se han efectuado los distintos despliegues. En *Docker* se realiza mediante el comando "*docker push*". En *Kubernetes*, como se indicaba en el punto anterior, es *Kaniko* quien además de generar la imagen, la registra en *Docker Hub*. Con *OpenShift* no ha sido necesario este registro ya que levanta directamente un *pod* desde el propio código de la aplicación.

Desplegar la aplicación. Este punto incluye los trabajos que han sido necesarios para poder levantar uno o más contenedores orquestados con la imagen de la aplicación.

Pruebas de funcionamiento y orquestación. Aquí se han realizado las pertinentes pruebas de funcionamiento y orquestación que han evidenciado tanto la simplicidad de administración de las aplicaciones, como la disponibilidad, escalabilidad, y balanceo de carga que ofrecen estas tecnologías.

5. Cumplimiento de objetivos

En el apartado 1.2 se identificaron seis objetivos para este trabajo, cuyo cumplimiento se verifican aquí.

El primer punto se ha desarrollado fundamentalmente en el apartado 3.1.2, "*ventajas e inconvenientes*", donde se han expuesto los beneficios que se obtienen con un despliegue en el *cloud*, destacando la agilidad y sencillez con la que se prestan los servicios, la reducción de costes que evitan elevadas inversiones en infraestructura, la alta disponibilidad y escalabilidad de las aplicaciones, y el pago únicamente por el uso de los recursos del *cloud*.

Respecto al segundo objetivo, en los apartados 2 y 3, "*estado del arte*" y "*conceptos previos*", se ha justificado por qué se están desplegando actualmente las aplicaciones contenerizadas en un *cloud*. Se ha expuesto la base teórica sobre la que se sustenta este paradigma, definiendo qué es un *cloud*, un *cluster*, un nodo, un contenedor, un *pod*, un *runtime*, un orquestador, etc. Finalmente, se ha realizado una breve introducción a las tres plataformas de orquestación más utilizadas en la actualidad.

En cuanto a los siguientes tres objetivos, eminentemente prácticos, en el apartado 4, "*puesta en práctica*", se ha expuesto detalladamente todos los procesos para poner en producción una aplicación en este entorno, desde su desarrollo, hasta la realización de pruebas de escalado, balanceo de carga y disponibilidad, en cada una de las tres plataformas.

Respecto al tercer objetivo, en el apartado 4.2 se ha creado una imagen *Docker* a partir de una aplicación *Node*, y posteriormente se han arrancado contenedores orquestados con dicha imagen sobre un *cluster* de *Docker* alojado en *Google Cloud*.

En cuanto al cuarto objetivo, en el apartado 4.3, se ha construido una imagen *Docker* de la aplicación sobre un *cluster* de *Kubernetes* cuyos nodos no disponen de un motor de *Docker* y, por lo tanto, no lo puede construir directamente. La solución se ha llevado a cabo levantando en el *cluster* de *Kubernetes* un contenedor con una imagen de una herramienta llamada *Kaniko*.

Respecto al quinto objetivo, en el apartado 4.4, se ha desplegado la aplicación con *MiniShift*, versión de laboratorio de *OpenShift*, utilizando tanto la consola *web* como el terminal de comandos "*oc*". Se ha creado la imagen de la aplicación directamente desde su código fuente utilizando la herramienta *S2I* de *OpenShift*. Se ha evidenciado la buena experiencia de usuario que ofrece *OpenShift* a través de su consola.

En cuanto al último objetivo, en el apartado 6, "*conclusiones y trabajos futuros*", se realiza una comparativa de estas tres tecnologías.

6. Conclusiones y trabajos futuros

A medida que la tecnología ha ido avanzando, se ha incrementado el número de organizaciones que esperan que sus departamentos de *TI* les ayuden a identificar los procesos que pueden ser transformados digitalmente. El paradigma tradicional de aplicaciones monolíticas no se adapta al ritmo de despliegue que demandan actualmente estas empresas. En cambio, las nuevas plataformas de aplicaciones alojadas en la nube y basadas en contenedores se están convirtiendo en un catalizador para la transformación digital.

En la literatura de contenerización se pueden encontrar multitud de escenarios en los que se desarrollan aplicaciones soportadas en la nube, incluso algunos que lo hacen bajo el paradigma de la contenerización, pero la mayoría de ellos se enfocan en el diseño y desarrollo del servicio que ofrecen, sin entrar en detalle en los procesos para alojarlos en la nube. Este trabajo es relevante al poner el foco tanto en el despliegue de las aplicaciones como en su orquestación, además, se realiza una comparativa que puede servir de referencia para la elección de la plataforma de contenerización más adecuada.

El proceso de alojamiento de una *app* contenerizada en un entorno *cloud* se puede resumir en tres fases: la configuración de la plataforma, la construcción de la imagen de la aplicación, y finalmente, el propio despliegue. La primera fase es la que representa mayor complejidad debido a la variedad de escenarios: tipos de nubes (públicas, privadas, híbridas), tipos plataformas de contenerización, etc., aunque se trata de una tarea que se realiza una única vez con el primer despliegue.

Respecto a la fase de la construcción de una imagen a partir del código de la aplicación, el proceso es muy diferente en función del tipo de plataforma utilizada:

- *Docker* construye las imágenes a partir de un fichero de texto llamado *Dockerfile* donde se deben incluir una serie de instrucciones.
- *Kubernetes*, sin embargo, al no utilizar actualmente el *daemon* de *Docker*, no puede construir las imágenes directamente a partir de un *Dockerfile*. Una solución a este problema la dio *Google* con el desarrollo de la herramienta *Kaniko*, arrancando un *pod* con una imagen constructora que es capaz de generar la imagen de la aplicación a partir de fichero *Dockerfile*.
- *OpenShift*, sin embargo, construye las imágenes a partir del código fuente de la aplicación utilizando la herramienta *S2I* (“*Source to Image*”), proceso totalmente transparente al usuario. Para ello únicamente se debe seleccionar una imagen constructora (“*Image Builder*”) según el tipo de aplicación (*Node*, *Ruby*, *Phyton*, *Java*, *PHP*, etc.), aunque en la mayoría de los casos *OpenShift* es capaz de detectarla automáticamente a partir del tipo de ficheros la aplicación.

La última fase, el propio despliegue de la imagen en una plataforma de contenedores, es muy simple comparada con el trabajo que requiere un despliegue tradicional en un *mainframe*. En *Docker* únicamente se necesita introducir un único comando ("*docker service create*"). En *Kubernetes* hay que construir un fichero de configuración "*yaml*" para definir un "*Deployment*" con el que se crean los *pods*, un "*Service*" para exponer el servicio al exterior, y finalmente, introducir un único comando ("*kubectl apply*"). El *OpenShift* todo el proceso (imagen incluida) se realiza mediante un único comando ("*oc new-app*").

La complejidad de la aplicación y el tipo de usuario son los factores que más condicionan la elección de la plataforma. *Docker* está más orientado a usuarios que necesiten desplegar aplicaciones de unos pocos contenedores sin orquestación, o con una orquestación muy limitada, de hecho, el desarrollo *open source* de su orquestador (*Docker Swarm*) está actualmente paralizado. Es muy raro actualmente encontrar una empresa que utilice esta plataforma de contenerización. La mayor ventaja de *Docker* es que sus imágenes representan un estándar y se utilizan en muchas otras plataformas de contenedores. *Kubernetes*, sin embargo, es la plataforma ideal para aquellas pymes que utilizan aplicaciones con bastantes contenedores (o *pods*) y, por tanto, cuya orquestación manualmente sería compleja. Casi todos los proveedores de la nube ofrecen servicios administrados de *Kubernetes* como, *Kubernetes Engine* (*GKE*) de *Google Cloud*, el *EKS* de *AWS*, o *AKS* de *Azure*. Además, *Kubernetes* dispone de una comunidad de usuarios mayor que *Docker* por lo es más fácil encontrar una solución a cualquier problema. *OpenShift*, sin embargo, está orientado a aplicaciones corporativas para grandes organizaciones donde se prime la seguridad y automatización de los despliegues. *OpenShift* ofrece una abstracción muy alta de las tecnologías de contenerización con una mejor experiencia de usuario, especialmente trabajando con su consola *web*, lo que redunda en una gestión muy sencilla de los proyectos, y aunque es una distribución de *Kubernetes*, tiene muchas más funcionalidades que éste.

Se han realizado varias pruebas en cada una de las tres plataformas, habiéndose puesto de manifiesto las ventajas de utilizar un orquestador. Con un único comando se han escalado las aplicaciones en los tres sistemas, adaptándola a las demandas de tráfico de los usuarios finales. Las tres plataformas implementan, por defecto, el balanceo automático de carga mediante el algoritmo de *round-robin*, lo que minimiza los tiempos de respuesta del servicio, mejora su desempeño y limita la saturación del sistema. También se ha puesto a prueba la disponibilidad simulando el fallo de contenedores y *pods*, así como el mantenimiento del número de réplicas. Finalmente, se ha puesto de manifiesto la facilidad y rapidez con que se implementa el escalado automático de los despliegues, mediante un único comando, tanto en *Kubernetes* como en *OpenShift* (*Docker* no lo soporta).

Se deja para futuros estudios el despliegue de aplicaciones con *OpenShift* en un entorno de nube *híbrida* pues este tipo de *cloud* es considerado por muchos autores [38] como el entorno ideal para el *cloud*. No se ha podido tratar aquí al no disponer de un *cluster* gratuito de *OpenShift* además de que habría excedido la extensión recomendada para este trabajo.

7. Glosario

App: abreviatura de la palabra inglesa *Application*, es decir, una aplicación *software*.
API (*Application Programming Interface*): interfaz de programación de aplicaciones o conjunto de reglas que permiten la comunicación entre dos aplicaciones.
Balanceador: dispositivo que distribuye la carga entre distintos servidores
CLI (*Command Line Interface*): interfaz de usuarios con el fin de introducir comandos en un sistema.
Cluster: plataforma compuesta por varios nodos o servidores que funcionan como una única entidad.
Contenedor: instancia de ejecución de una imagen.
CRI (*Container Runtime Interface*): estándar de *runtime* de contenedores.
CRI-O: *runtime* de contenedores que cumple con el estándar *CRI*.
Daemon: proceso que ejecuta un motor de contenedores.
DDoS (*Distributed Denial of Service*): ataque de denegación de servicio distribuido.
Dependencias: software adicional que requiere una aplicación para poder ejecutarse en un contenedor.
Dockerfile: fichero utilizado por *Docker* para la construcción de imágenes.
Firewall: sistema de seguridad para monitorizar el tráfico de una red y decidir si lo permite o lo bloquea.
Hipervisor: software para crear y ejecutar máquinas virtuales en un *host*.
In-house: *on-premise*.
Mainframe: computadora física de alto rendimiento.
Nodejs: entorno de tiempo de ejecución de *JavaSacript*.
ODS: Objetivos de Desarrollo Sostenible de la Agenda 2030.
On-premise: sistema alojado en la propia infraestructura de una empresa.
Off-premise: sistema cuya infraestructura es externa a la empresa.
Open source: modelo de desarrollo de software basado en la colaboración abierta.
Pod: agrupación lógica de contenedores.
Router: dispositivo que permite interconectar redes.
Runtime o Container runtime: software para ejecutar contenedores.
S2I (*Source to Image*): herramienta de *OpenShift* para la generación de imágenes a partir del código de las aplicaciones.
Switch: equipo de red que permite conectar dispositivos en una red *LAN*.
TI (Tecnologías de la Información): Conjunto de tecnologías desarrolladas para obtener una comunicación eficiente, facilitando la emisión, el acceso, y el tratamiento de la información.
URL (*Uniform Resource Locator*): localizador de recursos en internet.
VM (*Virtual Machine*): software que simula un sistema de computación.
VPN (*Virtual Private Network*): sistema que permite conectar equipos como si estuvieran en la misma red de área local.
Workloads: cualquier carga de un *cluster* (*pods*, servicios, etc.).

Bibliografía

[1] Asad Ali (2022). *Kubernetes vs Docker: ¿Cuál elegir en 2022*?. https://geekflare.com/es/docker-vs-kubernetes/

[2] Lacy P., Daugherty P., Durg K., Ponomarev P. (2020). *La sostenibilidad del Cloud.* https://www.accenture.com/es-es/insights/strategy/green-behind-cloud

[3] Wickramasinghe S. (2021). *AWS vs Azure vs GCP: Comparing The Big Cloud Platforms* https://www.bmc.com/blogs/aws-vs-azure-vs-google-cloud-platforms/

[4] Kumar Singh R. (2020). *Introduction to Container Technology*. https://onionlinux.com/container-technology/

[5] *Adoptando la nube: ¿Nube híbrida, privada, IaaS, PaaS, o SaaS?.* https://www.bitec.es/servicios-cloud/adoptando-la-nube-nube-hibrida-privada-iaas-paas-o-saas/

[6] Understanding pods. https://wangwei1237.github.io/Kubernetes-in-Action-Second-Edition/docs/Understanding_pods.html

[7] Tomas E. (2019). *Contenedores: cómo es el ciclo de vida de una aplicación en Kubernetes.* https://www.campusmvp.es/recursos/post/contenedores-como-es-el-ciclo-de-vida-de-una-aplicacion-en-kubernetes.aspx

[8] Blanch A., Fuentes F., Léon M., y otros (2022). *¿Qué es Kubernetes y cómo funciona* https://www.arsys.es/blog/kubernetes-comofunciona

[9] *Kubernetes Componets.* https://kubernetes.io/docs/concepts/overview/components/

[10] Urtiaga G. (2022). *Kubernetes desde cero*. Aprendeit

[11] *Google Cloud*. https://cloud.google.com/

[12] *Ubuntu*. https://lubuntu.me/downloads/

[13] *VirtualBox*. https://www.virtualbox.org/

[14] *VirtualBox - Instalar las Guest Additions de Lubuntu.* https://www.youtube.com/watch?v=uMoO58tPc5c

[15] *Instala la CLI de gcloud*. https://cloud.google.com/sdk/docs/install?hl=es-419#deb

[16] *Node*. https://nodejs.org/en/

[17] *GitHub*. https://github.com/

[18] *Git*. *https://git-scm.com/download/win*

[19] *Docker Hub*. *https://hub.docker.com/*

[20] *Consola de Google Cloud*. https://console.cloud.google.com/welcome?project=miproyecto-364518

[21] *Compara las características de GKE Autopilot y Standard*. https://cloud.google.com/kubernetes-engine/docs/resources/autopilot-standard-feature-comparison?hl=es-419

[22] *GCP Infraestructure.* https://www.datacenterknowledge.com/google-data-center-faq-part-2

[23] *Install Docker Engine on Ubuntu*. https://docs.docker.com/engine/install/ubuntu/`

[24] Winton Winton. *OpenShift 4 infra deep live*. Slide 10. https://www.slideshare.net/wintonjkt/open-shift-4-infra-deep-dive

[25] *Compara las características de GKE Autopilot y Standard*. https://cloud.google.com/kubernetes-engine/docs/resources/autopilot-standard-feature-comparison?hl=es-419
[26] *MiniShift. Releases*. https://github.com/minishift/minishift/releases
[27] *Certified kubernetes software* https://www.cncf.io/certification/software-conformance/
[28] *Docker Swarm* https://docs.docker.com/engine/reference/commandline/swarm/
[29] Evan J. (2017). *Reasons Kubernetes is cool*. https://jvns.ca/blog/2017/10/05/reasons-kubernetes-is-cool/?ref=refind
[30] Luksa M. (2018). *Kubernets in action*. Manning publications.
[31] https://livebook.manning.com/book/kubernetes-in-action-second-edition/chapter-6/v-6/14
[32] Gouigoux J. (2018). *Docker. Primeros pasos y puesta en práctica de una arquitectura basada en micro-servicios*. ENI.
[33] Iradier A., Martinez I. (2021). *Docker para DevOps*. Tecnología espublico.
[34] Wood J., Tannous B. (2021). *OpenShift for Developers*. O'Reilly
[35] Duncan J., Osborne J. (2018). *OpenShift in Action*. Manning Publications
[36] Carvalho L., Marden M. (2016). *El valor empresarial de negocio de Red Hat OpenShift*. IDC. https://go.54cuatro.com/wp-content/uploads/US41845816_WP_ES.pdf
[37] Sanchez Hernandez J. J. (2022). *Aprende Docker, un enfoque práctico*. Marcombo.
[38] Carrillo G. (2022). *¿Es la nube híbrida el futuro de la TI corporativa?* https://www.datacenterdynamics.com/es/features/es-la-nube-h%C3%ADbrida-el-futuro-de-la-ti-corporativa/
[39] Lázaro E., (2020). *Cómo agregar un directorio al PATH en Windows*. https://www.neoguias.com/agregar-directorio-path-windows/
[40] Prat J., (2022). *Building container images on Kubernetes with Kaniko*. https://pet2cattle.com/2022/04/kaniko-build-image-on-kubernetes
[41] https://docker-docs.netlify.app/machine/install-machine/#install-machine-directly
[42] Team Copado (2021). *Kubernetes Load Balancer Strategies for Maximun Availability and Scalability*. https://docker-docs.netlify.app/machine/install-machine/#install-machine-directly
[43] Chelowa T. (2019). *10 most important differences betwen OpenShift and Kubernetes*. https://blog.cloudowski.com/articles/10-differences-between-openshift-and-kubernetes/