UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Tecnología de Contenedores Docker

Tesis de Grado

Presentada a la Facultad de la

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

por

Pol Ponsico Martín

En cumplimiento parcial

de los requisitos para el grado en

INGENIERÍA TELEMÁTICA

Tutor: Jose Luis Muñoz Tapia

Barcelona, Octubre 2017

Resumen

Cuando empiezas a investigar sobre la tecnología de virtualización de Docker lo primero que te llama la atención son las expectativas de futuro que ha levantado por todo el mundo.

Docker es un proyecto que permite crear aplicaciones en contenedores de software que son ligeros, portátiles y autosuficientes

La principal diferencia de Docker con los modelos tradicionales de virtualización es que utiliza los contenedores en vez de las máquinas virtuales.

Los contenedores son un paquete de elementos que te permite crear un entorno donde correr aplicaciones independientemente del sistema operativo.

El propósito de este proyecto es investigar el uso de la virtualización de servidores basada en Docker Containers, entender la tecnología que se ejecuta detrás de ella, conocer las posibilidades que tiene y construir un clúster de contenedores que ejecuten una aplicación.

Ya sea como complemento de las máquinas virtuales o sustituyéndolas completamente, todo indica que Docker es el siguiente paso en las tecnologías de virtualización.

Resum

Quan comences a investigar sobre la tecnologia de virtualització de Docker el primer que et crida l'atenció són les expectatives de futur que ha aixecat per tot el món.

Docker és un projecte que permet crear aplicacions en contenidors de programari que siguin lleugeres, portàtils i autosuficients

La principal diferència de Docker amb els models tradicionals de virtualització és que utilitza els contenidors en comptes de màquines virtuals.

Els contenidors són paquets d'elements que et permeten crear un entorn on córrer aplicacions independentment del sistema operatiu del host.

El propòsit d'aquest projecte és investigar l'ús de la virtualització de servidors basada en Docker Containers, entendre la tecnologia que s'executa darrere d'ella, conèixer les possibilitats que té i construir un clúster de contenidors que executin una aplicació.

Ja sigui com a complement de les màquines virtuals o substituint-les completament, tot indica que Docker és el següent pas en les tecnologies de virtualització.

Abstract

When you start a research on Docker's virtualization technology, the first thing that catches your eye is the future expectations it has raised all over the world.

Docker is a project that allows you to create applications in software containers that are light, portable and self-sufficient

The main difference between Docker and traditional virtualization models is that it uses containers instead of virtual machines.

Containers are a package of items that allow you to create an environment in which applications run independently of the operating system.

The purpose of this project is to investigate the use of server virtualization based on Docker Containers, understand the technology running behind it, learn about the possibilities that it has and build a container cluster.

Whether as a complement to the virtual machines or completely replacing them, Docker is presumably the next step in virtualization technologies.

Reconocimientos

Me gustaría agradecer a mi tutor, Jose Luis Muñoz Tapia, por la supervisión del trabajo y la ayuda prestada.

Historial de revisiones y registro de aprobación

Revisión	Fecha	Propósito
0	18/09/2017	Creación del documento
1	06/10/2017	Revisión del documento

LISTA DE DISTRIBUCIÓN DEL DOCUMENTO

Nombre	e-mail
Pol Ponsico Martín	pol.ppm@gmail.com
Jose Luis Muñoz Tapia	jose.munoz@entel.com

Escrito por:		Revisado y aprobado por:	
Fecha	08/10/2017	Fecha	08/10/2017
Nombre	Pol Ponsico Martín	Nombre	Jose Luis Muñoz
Posición	Project Author	Posición	Project Supervisor

Tabla de contenidos

Lista de Figuras

La lista de figuras que aparecen en el documento es la siguiente:

1. Introducción

1.1. Statement of purpose

El proyecto se ha llevado a cabo en el departamento de ingeniería telemática de la Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona, ETSETB.

El propósito de este proyecto es investigar el uso de la virtualización de servidores basada en Docker Containers, entender la tecnología que se está ejecutando detrás de ella y conocer las posibilidades que tiene.

También me concentraré en cómo funciona el clúster de contenedores y por eso investigaré los dos mayores programas que hacen eso posible hoy en día, Docker Swarm y Kubernetes.

Con el conocimiento adquirido crearé una serie de ejercicios para las personas que se inician en la virtualización de contenedores y la creación de clústeres, para ayudarles a entender cómo funciona y darles herramientas para utilizarlo correctamente.

1.2. Requerimientos y especificaciones

Requerimientos del proyecto:

- Los ejercicios deben ser capaces de enseñar cómo crear un clúster de contenedores virtualizados y debería ser explicado de forma que la persona que lo realiza pueda relacionar el ejercicio realizado con la teoría previamente expuesta.
- La investigación debe abarcar diferentes maneras de ejecutar un clúster de contenedores Docker, diferentes funcionalidades de Docker, así como los procesos internos que sigue el motor de Docker.

1.3. Métodos y procedimientos

Para realizar todas las pruebas con Docker he utilizado una máquina virtual con una imagen de Windows 10 corriendo en Virtualbox, las máquinas virtuales creadas para los otros nodos fueron provistas de sistemas operativos boot2docker.

En el caso de las pruebas con Kubernetes utilicé una máquina virtual, con el sistema operativo Ubuntu, dónde instalé minikube, una herramienta que facilita la ejecución de Kubernetes en local.

1.4. Work plan

Project: Tecnología de Contenedores Docker	WP ref: 1	
Constituyente principal: Documentación inicial	Cuadro 1 of 6	
Breve descripción: Estudio de la tecnología de virtualización basada en Docker y los diferentes proyectos existentes.	Fecha inicial planeada: feb 20 Fecha final planeada: mar 17	
	Inicio del evento: feb 20 Final del evento: mar 17	
	Entregables: State of the art: informe con explicaciones conceptuales, definiciones y proyectos existentes basados en la virtualización con contenedores.	Fecha: mar 17

Project: Tecnología de Contenedores Docker	WP ref: 2	
Constituyente principal: Estudiar la gestión de Docker Engine	Cuadro 2 of 6	
Breve descripción: Instalación de Docker Engine, adquirir conocimientos sobre la gestión de contenedores y familiarizarse con los comandos y archivos.	Fecha inicial planeada: mar 20 Fecha final planeada: abr 7	
	Inicio del evento: mar 20 Final del evento: abr 14	
	Entregables: Manual de operaciones de Docker	Fecha: abr 7

Project: Tecnología de Contenedores Docker	WP ref: 3	
Constituyente principal: Creación y configuración de un clúster con Docker Swarm	Cuadro 3 of 6	
Breve descripción: Crear y configurar un clúster con Docker Swarm i investigar las diferentes opciones de configuración y rendimiento.	Fecha inicial planeada: abr 10 Fecha final planeada: abr 28	
	Inicio del evento: abr 17 Final del evento: may 19	
	Entregables: Manual de creación y configuración de un manual de un clúster de Docker Swarm.	Fecha: may 19

Project: Tecnología de Contenedores Docker	WP ref: 4	
Constituyente principal: Creación y configuración de un clúster con Kubernetes	Cuadro 4 of 6	
Breve descripción: Crear y configurar un clúster con Kubernetes i investigar las diferentes opciones de configuración y rendimiento.	Fecha inicial planeada: may 22 Fecha final planeada: jun 20	
	Inicio del evento: may 22 Final del evento: jun 20	
	Entregables: Manual de creación y configuración de un manual de un clúster de Docker Swarm.	Fecha: jun 20

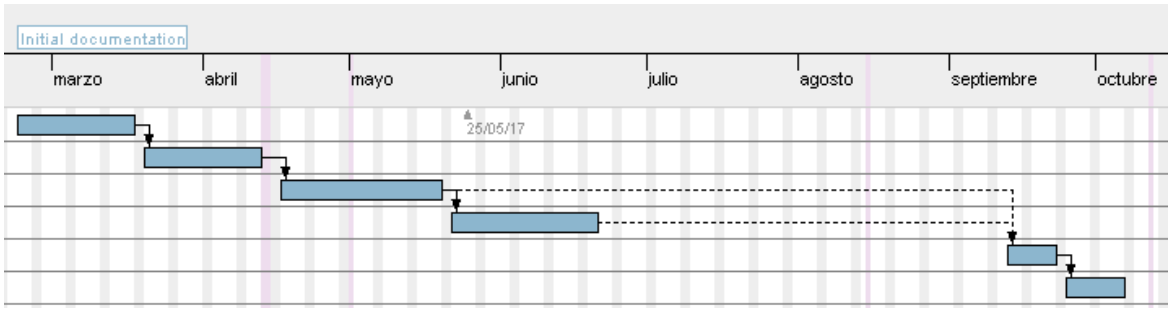
Project: Tecnología de Contenedores Docker	WP ref: 5	
Constituyente principal: Investigar la creación de entornos de contenedores a partir de ficheros	Cuadro 5 of 6	
Breve descripción: Crear e investigar las diferentes posibilidades que ofrecen los ficheros de configuración de Docker.	Fecha inicial planeada: set 13 Fecha final planeada: set 22	
	Inicio del evento: set 13 Final del evento: set 22	
	Entregables: Manual de creación y configuración de una imagen de un contenedor y del despliegue de un clúster a partir de un fichero.	Fecha: set 22

Project: Tecnología de Contenedores Docker	WP ref: 6	
Constituyente principal: Informe final	Cuadro 6 of 6	
Breve descripción: Recopilar toda la información y establecer una conclusión.	Fecha inicial planeada: set 25 Fecha final planeada: oct 06	
	Inicio del evento: set 25 Final del evento: oct 06	
	Entregables: Informe final.	Fecha: oct 06

1.4.1. Milestones

#WP	# Tarea	Título	Milestone / Entregable	Semanas
1	1	Documentación inicial	State of the art.	4
2	2	Gestión Docker Engine	Manual de operaciones Docker.	4
3	3	Creación de un clúster con Swarm	Manual de Docker Swarm.	4
4	4	Creación de un clúster con Kubernetes	Manual de Kubernetes.	5
5	5	Despliegue a partir de archivos	Manual de archivos de entornos.	2
6	6	Informe final	Informe final.	2

1.5. Diagrama Gantt



1.6. Modificaciones del Work Plan

Se hicieron cambios en el Work Plan debido a que se modificó el plazo de finalización del proyecto.

2. State of the art de la tecnología utilizada o aplicada en esta tesis:

Cuando empiezas a investigar sobre la tecnología de virtualización de contenedores Docker lo primero que te llama la atención son las expectativas de futuro que ha levantado por todo el mundo.

Muchos analistas prevén que Docker es el siguiente paso en las tecnologías de virtualización y que en un futuro los contenedores reemplazarán a las máquinas virtuales.

Docker fue lanzado al mercado como un proyecto de código abierto por “dotCloud” en 2013, pero la idea de los contenedores no es nueva, ya que utiliza conceptos que han estado presentes desde los primeros días de Unix, como pueden ser los namespaces y cgroups, para asegurar el aislamiento de los recursos y el empaquetado de una aplicación, así como de sus dependencias.

Este empaquetado de dependencias permite al desarrollador escribir una aplicación en cualquier lenguaje y poder ejecutarla en diferentes entornos muy diferentes, independientemente de su sistema operativo.

Los contenedores de Linux (LXC) son la tecnología en la que se basa el software de Docker, que salió al mercado el 6 de agosto de 2008.

Docker, Inc es la compañía que está detrás del desarrollo de este software. Tiene su base en San Francisco (California) y en 2015 tenía más de 120 empleados trabajando.

Después de que Docker saliera al mercado los desarrolladores empezaron a darse cuenta de cómo este nuevo enfoque podía solucionar muchos de sus problemas.

En agosto de 2013 Docker lanzó su tutorial interactivo, que fue probado por más de 10000 desarrolladores. En un año compañías como Red Hat o Amazon incorporaron soporte comercial para Docker y cuando Docker anunció su versión 1.0 en junio de 2014, el Docker Engine ya tenía 2,75 millones de descargas. Hoy en día ya tiene más de 100 millones.

Docker cuenta con grandes nombres entre sus clientes, como pueden ser PayPal, Spotify o Yelp.

Este gran éxito de Docker ha llamado la atención de otras compañías, que intentan darle otro enfoque a la virtualización de contenedores o incluso crear los suyos propios.

Como respuesta a Docker el CEO de la compañía CoreOS Alex Polvi lanzó al mercado Rocket, ya que según él Docker no es seguro, ya que utiliza un daemon de Docker centralizado, en cambio Rocket utiliza un daemon de systemd para crear los contenedores.

Hay un gran debate sobre si los contenedores sustituirán a las máquinas virtuales, ya que los contenedores son más eficientes, por lo tanto, es una forma de hacer lo mismo que se hacía con los hipervisores, pero a un menor coste.

Como bien sabemos la industria de los hipervisores ha estado liderada por VMware y su más acérrimo competidor, Microsoft, para los cuales los contendores suponen tanto una amenaza como una oportunidad de negocio.

3. Metodología / desarrollo del proyecto:

Para realizar todas las pruebas con Docker he utilizado una máquina virtual con una imagen de Windows 10 corriendo en Virtualbox, las máquinas virtuales creadas para los otros nodos fueron provistas de sistemas operativos boot2docker.

En el caso de las pruebas con Kubernetes utilicé una máquina virtual con Sistema operativo Ubuntu dónde instalé minikube, una herramienta que facilita la ejecución de Kubernetes en local.

3.1. Docker

Docker es un proyecto “open source” que permite crear aplicaciones en contenedores de software que son ligeros, portátiles y autosuficientes

La principal diferencia de Docker con los modelos tradicionales de virtualización es que utiliza los contenedores en vez de las máquinas virtuales.

Los contenedores son un paquete de elementos que te permite crear un entorno donde correr aplicaciones independientemente del sistema operativo.

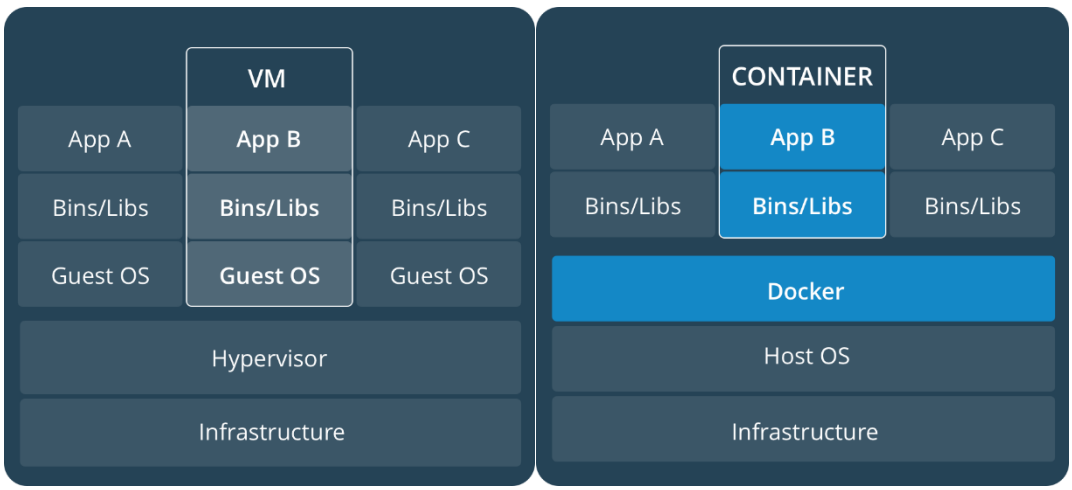


Ilustración 1; Arquitectura contenedores

El código corre en estos contenedores totalmente aislado de otros contenedores y todos ellos comparten los recursos de la máquina sin la sobrecarga que aporta la capa del hipervisor.

Los contenedores son mucho más ligeros que las máquinas virtuales ya que, mientras que las máquinas virtuales requieren de la instalación de un sistema operativo, asignación de disco, CPU y memoria RAM para funcionar, un contenedor de Docker sólo necesita el sistema operativo que corre en la máquina en la que está el contenedor.

Docker se compone de las siguientes partes:

- Docker daemon
- Docker Client CLI
- Docker Hub
- Imágenes
- Contenedores

Los pasos que sigue Docker para correr un contenedor son los siguientes:

El Docker Client CLI (Command Line Interface) contacta con el Docker daemon, este demonio se encarga de descargar la imagen deseada del repositorio de Docker, llamado Docker Hub, y despliega un contenedor a partir de esa imagen. Si se ejecuta un programa en ese contenedor, la salida será enviada al Docker Client y el cliente lo enviará al terminal.

La capa en la que se ejecuta Docker se llama Docker Engine que fue escrito en el lenguaje Golang y corre en sistemas nativos Linux.

Docker actualmente no soporta checkpointing, restoring o migraciones en caliente entre hosts, pero es una funcionalidad que podría llegar en un futuro.

3.2. Docker Network

Hay varias formas en las que los contenedores se pueden conectar entre ellos y con el host. Las principales redes son las siguientes:

- None
- Bridge
- Host
- Overlay

3.2.1. None

None es sencillamente que el contenedor no tiene interfaz de red, aunque sí que tiene interfaz de loopback. Este modo se utiliza para contenedores que no utilizan la red, como pueden ser contenedores para pruebas que no necesiten comunicación externa o contenedores que pueden dejarse preparados para ser conectados a una red posteriormente.

3.2.2. Bridge

Un bridge de linux proporciona una red interna en el host a través de la cual los contenedores se pueden comunicar. Las direcciones IP asignadas en esta red no son accesibles desde fuera del host. La red “bridge” aprovecha las iptables para hacer NAT y mapeado de puertos. La red bridge es la red por defecto de Docker.

La creación del contenedor incluye los siguientes pasos respecto a la red:

1. Se proporciona al host una red bridge.
2. Un namespace para cada contenedor es proporcionado en ese bridge.
3. Las interfaces de red de los contenedores se mapean a las interfaces privadas del bridge.
4. Las iptables con NAT se usan para mapear entre cada contenedor y la interfaz pública del host.

3.2.3. Host

En este caso el contenedor comparte su namespace de red con el host, lo que se traduce en un mejor rendimiento ya que elimina la necesidad de NAT, pero esto también provoca conflictos de puertos. Por lo tanto, el contenedor tiene acceso a todas las interfaces de red del host. La red “host” es la que se utiliza por defecto en “Mesos”.

3.2.4. Overlay

Overlay utiliza los túneles de red para la comunicación de contenedores en diferentes hosts. Esto permite que dos contenedores que estén en hosts diferentes se comporten como si se encontraran en el mismo host, ya que hacen túneles de subredes de un host a otro.

La tecnología de tunneling que utiliza Docker es VXLAN (Virtual Extensible Local Area Network), que se incluye de forma nativa desde el lanzamiento de la versión 1.9.

Las redes multi-host requieren de parámetros adicionales cuando lanzamos el demonio de Docker, así como un registro key-value.

Este tipo de red es la utilizada en la orquestación de contenedores distribuidos en diferentes hosts, ya que no se podrían comunicar entre sí por la red “bridge” comentada anteriormente.

3.3. Docker Swarm

Docker Swarm nos permite gestionar un grupo de hosts de Docker como un solo host de Docker virtual. Swarm utiliza la API estándar de Docker, por lo que cualquier herramienta que se comunique con el Docker Daemon puede utilizar Docker Swarm, que permite la escalabilidad a varios hosts.

Para desplegar una imagen de una aplicación cuando Docker Engine está en modo Swarm hay que crear un servicio. Normalmente el servicio será la imagen de un microservicio dentro de una aplicación más grande. Por ejemplo, un servicio puede ser un servidor HTTP, una base de datos, o cualquier otro tipo de programa ejecutable que se desee correr en un entorno distribuido.

Cuando se crea un servicio hay que especificar qué imagen de contendor usar, así como que comandos se ejecutarán en esos contenedores. También se definen las diferentes opciones de configuración del servicio como pueden ser:

- El puerto por el que Docker Swarm hará el servicio accesible desde el exterior.
- La red Overlay que permitirá conectar al servicio con otros servicios del clúster.
- Límites y reservas de memoria y CPU.
- Políticas de actualizaciones.
- Número de réplicas de la imagen que corren en el clúster.

En el momento de despliegue del servicio en Swarm, el gestor de Swarm acepta la definición como el estado deseado del servicio y distribuye el servicio en los nodos del clúster (dependiendo del número de réplicas). Estos procesos se ejecutan independientemente en cada uno de los nodos del clúster y se llaman task.

Por ejemplo, en la figura que podemos ver a continuación tenemos un servidor web nginx que reparte la tarea de escucha HTTP en tres réplicas. Cada una de estas instancias es un proceso diferente en el clúster.

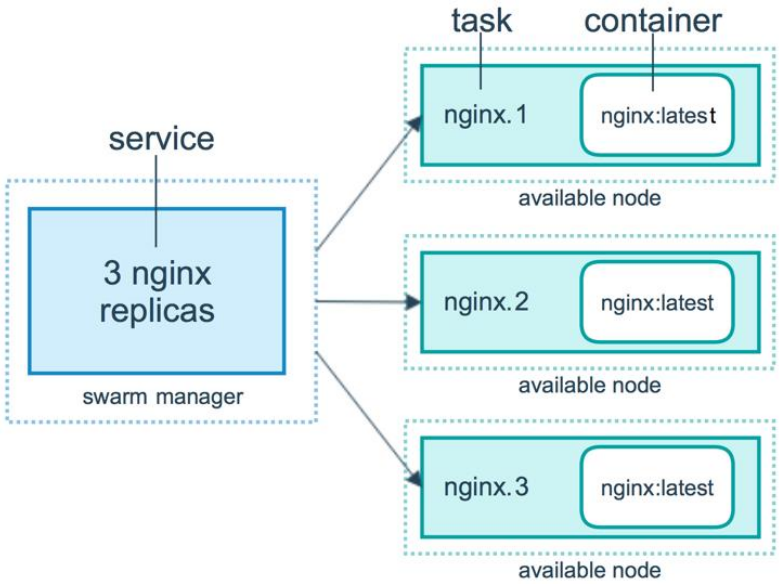


Ilustración 2: Diagrama de un servicio replicado en Swarm

En el modelo de Swarm cada task se corresponde con un contenedor.

Una task es la unidad básica de scheduling en un clúster de Swarm. Cuando declaras el estado deseado de un servicio mediante la creación de un servicio, el orquestador obtiene ese estado mediante el scheduling de tasks en los diferentes nodos.

Una task sigue un mecanismo unidireccional, ya que progresa siempre por una serie de estados (assigned, prepared, running, …). Si la task falla, el orquestador elimina la tarea y su contenedor y crea una nueva que la reemplaza para asumir el estado especificado en la creación del servicio.

El principal propósito de Docker Swarm es el de ser programador y orquestador de contenedores, por lo que los servicios y tasks tienen otro nivel de abstracción que les permite no ser conscientes de las aplicaciones que corren en los contenedores que implementan.

La siguiente figura muestra la manera en la que el gestor de Swarm acepta la creación de un servicio y envía la orden de ejecutar la task a los nodos.

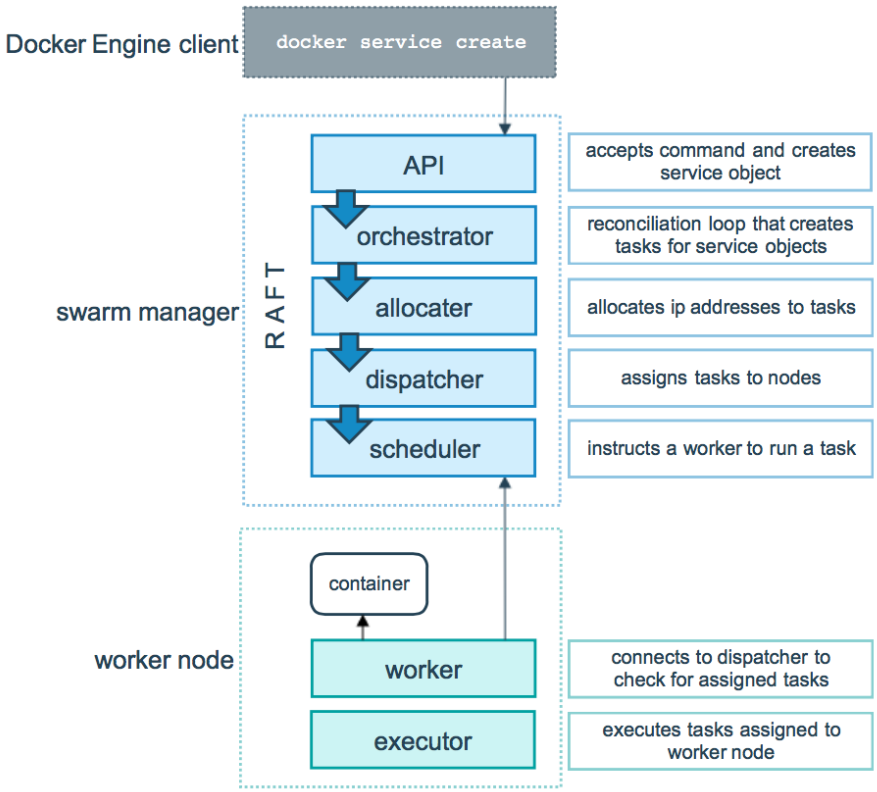


Ilustración 3: Flujo de tareas del gestor de Swarm

Un servicio puede ser configurado de manera que ningún nodo del clúster pueda ejecutar sus tasks, en este caso el servicio se quedará en “pending”.

Hay dos tipos de implementaciones de servicios: replicados y globales.

En el caso de los servicios replicados se especifica el número de tasks idénticas que se quieran ejecutar.

Un servicio global es un servicio que ejecuta una task en cada nodo. No hay un número especificado de tareas. Cada vez que añades un nodo al clúster, el orquestador de Swarm crea una task y el scheduler asigna la tarea a un nuevo nodo.

En el siguiente diagrama podemos observar en amarillo un servicio replicado de tres replicas en amarillo y uno global en gris.

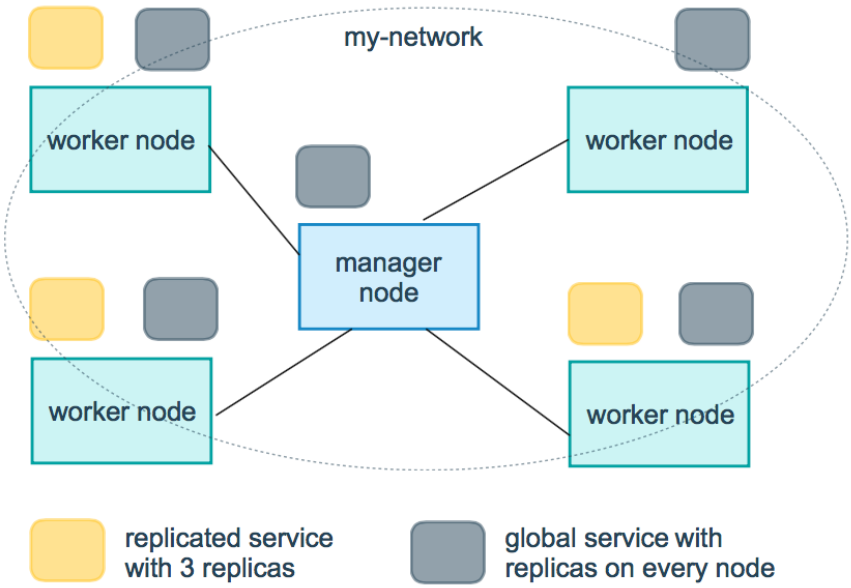


Ilustración 4: Diagrama de un servicio replicado i uno global

Existen dos modos de disponibilidad en los nodos de Docker Swarm: “active” y “drain”.

El modo de disponibilidad “drain” permite que el nodo no reciba nuevas tasks de Swarm, también implica que las tareas que se están ejecutando en ese nodo se paren y se relancen en otros nodos que estén en modo de disponibilidad activa.

Actualmente Docker Swarm tiene tres estrategias para decidir en qué nodos del clúster se deben ejecutar los contenedores:

- Spread: es el utilizado por defecto. Intenta balancear la carga de contenedores por igual en todos los nodos del clúster. Para ello tiene en cuenta la CPU y memoria RAM disponible, y el número de contenedores corriendo en cada nodo.
- BinPack: es todo lo contrario a Spread, este ejecuta todos los contenedores en el siguiente nodo disponible. El objetivo de ello es utilizar el menor número de nodos posible.
- Random: por último, como sugiere su nombre, esta estrategia utiliza un patrón aleatorio para la colocación de contendores en los nodos.

Swarm también permite el uso de cinco filtros que nos ayudan a gestionar los contenedores:

- Restriction: también llamados etiquetas de nodos, las restricciones son pares key/value asociados a un nodo en particular.
- Affinity: Este filtro se utiliza para hacer que diferentes contenedores se ejecuten en una misma red.
- Dependency: Cuando dos contenedores dependen uno del otro, este filtro permite situarlos en el mismo nodo.

- Health: Si un nodo no funciona correctamente, este filtro previene el despliegue de contendores en ese nodo.

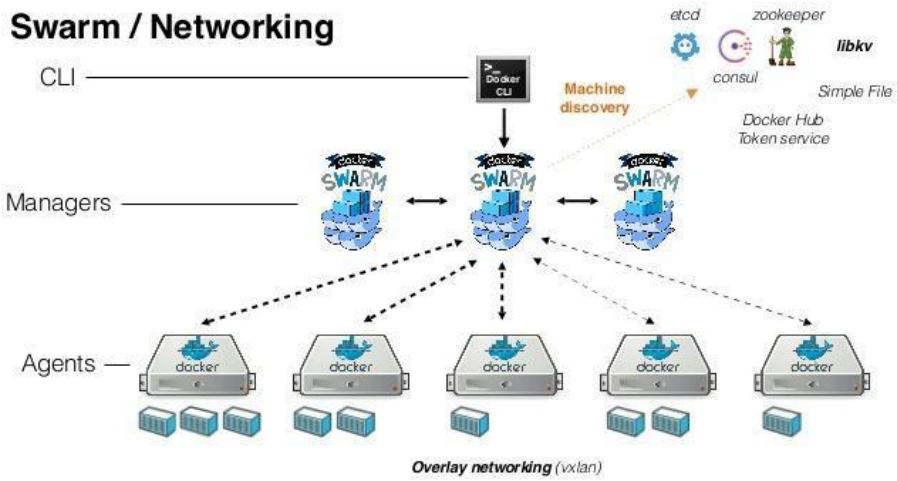


Ilustración 5: Cluster de Swarm

3.4. Docker Hub

Docker Hub provee una fuente de recursos centralizada para el descubrimiento y distribución de imágenes de contenedores que fomenta la colaboración entre usuarios.

Es el repositorio en la nube de Docker donde se encuentran todas las imágenes de los contendedores. Permite a los usuarios subir sus propias imágenes y descargar tanto imágenes subidas por la comunidad como las imágenes oficiales o de diferentes proyectos como pueden ser PostgreSQL, Ubuntu, Alpine, BusyBox, etc.

3.5. Kubernetes

Kubernetes es una herramienta de código abierto desarrollada por Google para gestionar aplicaciones en el entorno de un clúster. En otras palabras, Kubernetes te permite tratar grupos de contenedores de Docker como unidades con su dirección IP propia y escalarlos a tu antojo.

Kubernetes tiene terminología básica que deberemos conocer antes que nada:

- **Pods:** grupos de uno o más contenedores
- **Controllers:** entidades que conducen el estado del clúster al estado deseado.
- **Service:** un grupo de pods que trabajan juntos
- **Label:** Un simple par nombre-valor
- **Hyperkube:** es el binario que ejecuta el servidor

Las “label” son el eje central de Kubernetes. Labelando las entidades de Kubernetes podremos gestionar todos los pods de nuestro clúster.

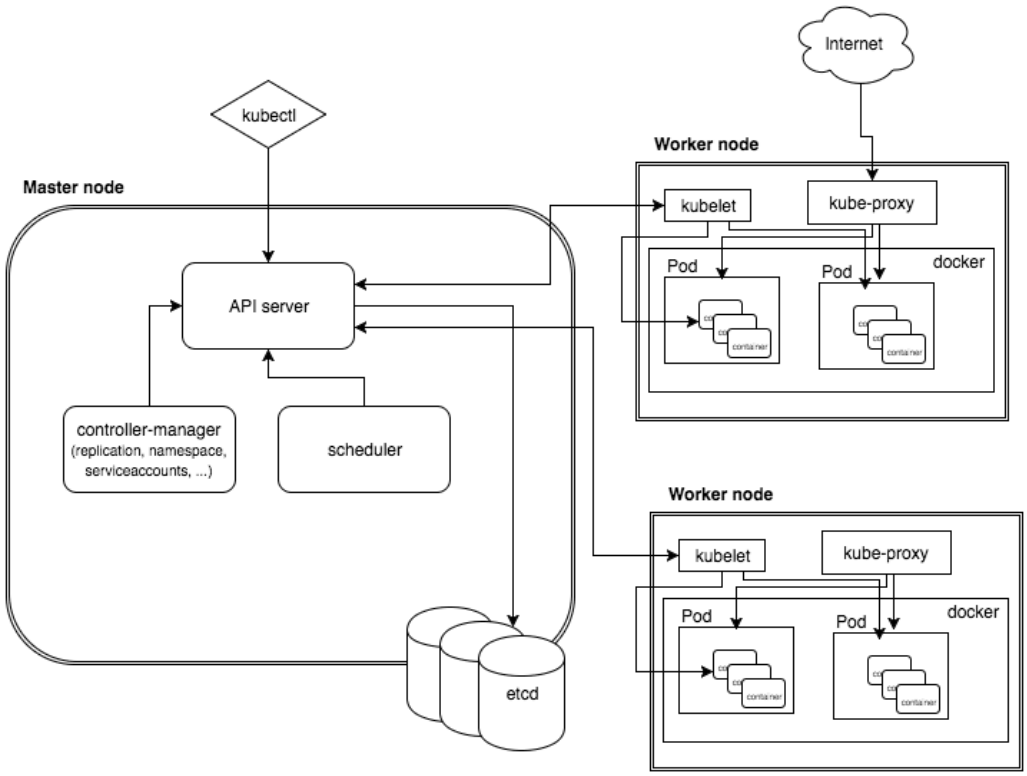


Ilustración 6: Diagrama de un clúster con Kubernetes

3.5.1. Componentes Master

Los servicios de control en el clúster de Kubernetes se llaman componentes master. Éstos funcionan como los principales puntos gestión para administradores. Pueden instalarse en una sola máquina o estar distribuidos por diferentes máquinas.

Los servidores que ejecutan estos componentes tienen una serie de servicios de gestión únicos, que se utilizan para administrar la carga de trabajo del clúster y dirigir las comunicaciones por todo el sistema.

3.5.2. Nodo

En Kubernetes los servidores que realizan algún trabajo se llaman Nodos. Los nodos deben cumplir ciertos requerimientos que son necesarios para que se comuniquen con los componentes master, configurar las redes de los contendores, y ejecutar las cargas de trabajo que se les asignan.

3.5.3. Etcd

Uno de los componentes fundamentales que Kubernetes necesita es un almacén de configuraciones disponible globalmente. El proyecto etcd, desarrollado por el equipo de CoreOS, es un almacén de par clave/valor que se distribuye a través de múltiples nodos.

Kubernetes utiliza etcd para almacenar la información de la configuración, que puede ser utilizada por todos los nodos del clúster.

Etcd se puede configurar como un solo servidor master o estar distribuido por unas cuantas máquinas.

3.5.4. Servidor API

Este es el principal punto de gestión de todo el clúster, ya que permite al usuario configurar las cargas de trabajo de Kubernetes.

3.5.5. Servicio Controller Manager

El servicio Controller Manager es un servicio general que aporta muchas posibilidades. Es el responsable de una serie de controladores que regula el estado del clúster y realiza tareas rutinarias.

Por ejemplo, el controlador de replicación garantiza que el número de réplicas definidas para un servicio coincide con el número que actualmente hay desplegado en el clúster.

Los detalles de estas operaciones están escritos en etcd, que es donde el Controller Manager consulta los cambios a través del servidor API.

3.5.6. Servicio Scheduler

El proceso que realmente asigna las cargas de trabajo a nodos específicos en el clúster es el scheduler. Este se utiliza para leer los requerimientos de un servicio, analizar la infraestructura del entorno, y colocar la tarea en un nodo que cumpla los requisitos.

3.5.7. Servicio Kubelet

Kubelet es un agente que se ejecuta en los nodos y monitoriza los contenedores, reiniciándolos si es necesario.

Este servicio es el responsable de comunicarse con los componentes master para recibir las ordenes que debe ejecutar. También interactúa con etcd para leer detalles de la configuración y escribir nuevos valores.

3.5.8. Servicio Proxy

Un pequeño proxy se ejecuta en cada nodo para hacer accesibles los servicios al exterior. Este proceso envía las solicitudes a los contenedores adecuados.

3.6. Modelo de Redes de Kubernetes

Kubernetes asume que los pods se pueden comunicar con otros pods, independientemente del host en el que se encuentren. Cada pod tiene su propia dirección IP, por lo que no hay que crear enlaces específicos entre pods ni mapear puertos de contenedores con puertos de hosts.

Todo esto permite tener un modelo limpio donde los pods pueden ser tratados como máquinas virtuales o hosts físicos desde la perspectiva de la asignación de puertos, nomenclatura, descubrimiento de servicios, balance de carga, configuración de aplicaciones y migración.

Para lograr esto debemos imponer algunos requisitos en cuanto a la configuración de la red del clúster.

Como hemos visto antes, Docker utiliza por defecto la red Bridge, que es una red interna del host, para cada contenedor que Docker crea, se le asigna una tarjeta de red virtual llamada veth, la cual se conecta a la red del bridge. Esta tarjeta “veth” se mapea dentro

del contenedor como “eth0” utilizando los namespaces de Linux. A esta interfaz eth0 se le da una dirección IP del rango que maneja la red Bridge.

Como resultado de esto los contenedores de Docker sólo se pueden comunicar con los que están en su mismo nodo.

Kubernetes impone los siguientes requisitos en cualquier implementación de redes:

- Todos los contenedores deben poder comunicarse todos con todos sin NAT.
- Todos los nodos deben poder comunicarse con todos los contenedores sin NAT.
- La dirección IP que un contenedor ve que él tiene, tiene que ser la misma que los otros contenedores ven.

Kubernetes aplica direcciones IP a los Pod, los contenedores en un mismo Pod comparten los namespaces de red, incluida la dirección IP. Lo cual implica que los contenedores que estén en un mismo Pod pueden alcanzarse entre ellos utilizando la dirección localhost.

3.7. Comparación Swarm vs Kubernetes

La instalación de Docker es tan simple como la de cualquier aplicación disponible en el gestor de paquetes de un sistema operativo. Con Swarm desplegar un nodo y darles las órdenes para unirse a un clúster es todo lo que hace falta para tener tu clúster funcionando. Además de estas facilidades en su uso, Swarm también aporta flexibilidad ya que permite que cualquier nuevo nodo se una a un clúster ya existente como manager o worker y cambiar los roles de los nodos ya existentes.

En cambio, Kubernetes requiere de más configuraciones manuales para que componentes como Etcd, Flannel y Docker Engine funcionen juntos correctamente. Kubernetes también necesita tener información de la configuración del clúster por adelantado, como pueden ser las direcciones IP, el rol y el número total de nodos del clúster.

3.8. Docker Compose

Docker Compose es una herramienta que permite definir y correr aplicaciones de Docker multi-container.

Esta herramienta te permite configurar los servicios de tu aplicación en un archivo “Compose” y después desplegarlos con un sólo comando.

Para utilizar Docker Compose sólo hay que seguir los siguientes pasos:

1. Definir el entorno de tu aplicación en un archivo “Dockerfile”.
2. Definir los servicios de los que se compone tu aplicación en un archivo “.yml”.
3. Ejecutar el comando “docker-compose up”.

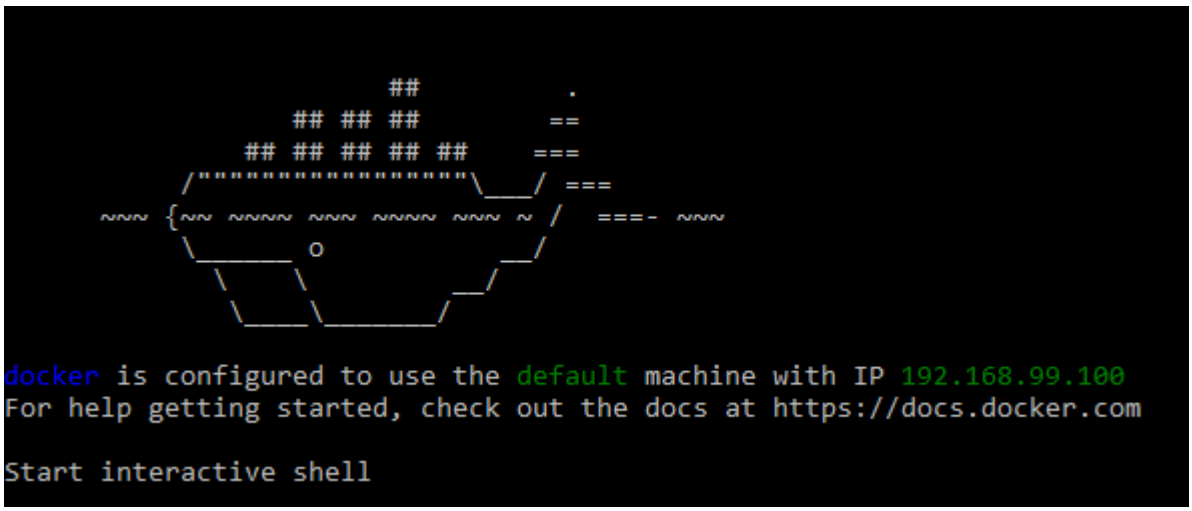
Docker Compose también tiene otros comandos que te permiten iniciar y parar servicios, así como ver el estado de estos o correr un comando en uno de ellos.

4. Resultados

Después de obtener las bases teóricas del funiconamiento de las dos herramientas más importantes de virtualización basada en contenedores actualmente, como resultado de los conocimientos adquiridos realizo un manual a través del cual aprender a utilizar las estas herramientas y como resultado final desplegar un cluster de contenedores virtuales, que funcione correctamente, con ambas tecnologías (Docker y Kubernetes).

4.1. Docker basics

La primera pantalla que veremos al empezar a utilizar Docker es la siguiente:



Para descargarnos una imagen es tan sencillo como ejecutar el siguiente comando

$ docker pull *ubuntu*

El demonio de docker buscará la imagen en Docker Hub y se la descargará.

```
$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
9fb6c798fa41: Pull complete
3b61febd4aef: Pull complete
9d99b9777eb0: Pull complete
d010c8cf75d7: Pull complete
7fac07fb303e: Pull complete
Digest: sha256:31371c117d65387be2640b8254464102c36c4e23d2abe1f6f4667e47716483f1
Status: Downloaded newer image for ubuntu:latest
```

Así como podemos descargarnos una imagen también podemos subirla al Docker Hub con el comando “push”.

$ docker push username/repository:tag

A continuación, ejecutamos el contenedor “hello-world”, el cual nos muestra una explicación del proceso que sigue docker para desplegar este contenedor.

$ docker run *hello-world*

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
5b0f327be733: Pull complete
Digest: sha256:1f19634d26995c320618d94e6f29c09c6589d5df3c063287a00e6de8458f8242
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

Ahora ejecutaremos el siguiente comando que nos permite desplegar un contenedor BusyBox y meternos en la Shell donde podremos ejecutar todos los comandos BusyBox que deseemos.

$ docker run -it --rm *busybox*

```
$ docker run -it --rm busybox
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
03b1be98f3f9: Pull complete
Digest: sha256:99ccecf3da28a93c063d5dddcdf69aeed44826d0db219aabc3d5178d47649dfa
Status: Downloaded newer image for busybox:latest
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ #
```

Para realizar el mapeo entre los puertos del host y del contenedor utilizaremos el siguiente comando. Hay que tener en cuenta que el puerto del host siempre va a la izquierda de los dos puntos y el puerto del contenedor a la derecha.

El comando que vemos a continuación es el que nos dice todos detalles de la versión de docker que estamos utilizando, tanto la del cliente como la del servidor.

$ docker version

```
$ docker version
Client:
 Version:      17.07.0-ce
 API version:  1.24 (downgraded from 1.31)
 Go version:   go1.8.3
 Git commit:   8784753
 Built:        Tue Aug 29 17:41:05 2017
 OS/Arch:      windows/amd64

Server:
 Version:      1.12.3
 API version:  1.24 (minimum version )
 Go version:   go1.6.3
 Git commit:   6b644ec
 Built:        Wed Oct 26 23:26:11 2016
 OS/Arch:      linux/amd64
 Experimental: false
```

El comando "$ docker build *Dockerfile*" sirve para construir imágenes a partir de archivos Dockerfile. La creación de estos archivos es tratada más adelante en el apartado "Dockerfile".

El comando "search" nos permite hacer búsquedas en Docker Hub.

$ docker search *alpine*

Este comando busca todas las versiones de imágenes alpine en Docker Hub, ya sean oficiales o creadas por la comunidad de usuarios y las ordena por popularidad con un sistema de estrellas.

```
$ docker search alpine
NAME                                DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
alpine                              A minimal Docker image based on Alpine Lin...   2578      [OK]
mhart/alpine-node                   Minimal Node.js built on Alpine Linux           312
anapsix/alpine-java                 Oracle Java 8 (and 7) with GLIBC 2.23 over...   238                  [OK]
gliderlabs/alpine                   Image based on Alpine Linux will help you ...   166
frolvlad/alpine-glibc               Alpine Docker image with glibc (~12MB)          104                  [OK]
kiasaki/alpine-postgres             PostgreSQL docker image based on Alpine Linux   34                   [OK]
zzrot/alpine-caddy                  Caddy Server Docker Container running on A...   32                   [OK]
davidcaste/alpine-tomcat            Apache Tomcat 7/8 using Oracle Java 7/8 wi...   20                   [OK]
easypi/alpine-arm                   AlpineLinux for RaspberryPi                     19
janes/alpine-lamp                   lamp base on alpine linux                       19                   [OK]
zzrot/alpine-ghost                  Docker Image running Ghost Blogging Servic...   14                   [OK]
byrnedo/alpine-curl                 Alpine linux with curl installed and set a...   6                    [OK]
troyfontaine/armhf-alpinelinux      Alpine Linux for ARMHF (ARM v7l)                5
zzrot/alpine-node                   Alpine Node Base Image Running On Alpine L...   5                    [OK]
davidcaste/alpine-java-unlimited-jce Oracle Java 8 (and 7) with GLIBC 2.21 over...  5                    [OK]
tenstartups/alpine                  Alpine linux base docker image with useful...   3                    [OK]
spotify/alpine                      Alpine image with `bash` and `curl`.            3                    [OK]
timhaak/sonarr-alpine               Alpine image for sonarr - (https://sonarr....   3                    [OK]
graze/php-alpine                    Smallish php7 alpine image with some commo...   3                    [OK]
colstrom/alpine                     Alpine Linux is a security-oriented, light...   2                    [OK]
functions/alpine                    Alpine Linux / BusyBox with watchdog baked...   1
17media/fluentd-alpine              A minimal docker image based on Alpine Linux    1
hivesolutions/alpine_dev            Development oriented version of alpine Lin...   0                    [OK]
smartentry/alpine                   alpine with smartentry                          0                    [OK]
casept/alpine-amd64                 A basic alpine linux image.                     0
```

$ docker search --filter=stars=15 ubuntu

Con este comando podemos filtrar las búsquedas de imágenes en Docker Hub por estrellas, así solo nos aparecerán las imágenes que tengan más estrellas que el número que hayamos indicado.

```
$ docker search --filter=stars=15 ubuntu
NAME                                               DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
ubuntu                                             Ubuntu is a Debian-based Linux operating s...   6577      [OK]
dorowu/ubuntu-desktop-lxde-vnc                     Ubuntu with openssh-server and NoVNC            129                  [OK]
rastasheep/ubuntu-sshd                             Dockerized SSH service, built on top of of...   101                  [OK]
ansible/ubuntu14.04-ansible                        Ubuntu 14.04 LTS with ansible                   86                   [OK]
ubuntu-upstart                                     Upstart is an event-based replacement for ...   77        [OK]
neurodebian                                        NeuroDebian provides neuroscience research...   39        [OK]
ubuntu-debootstrap                                 debootstrap --variant=minbase --components...   30        [OK]
nuagebec/ubuntu                                    Simple always updated Ubuntu docker images...   22                   [OK]
tutum/ubuntu                                       Simple Ubuntu docker images with SSH access     18
1and1internet/ubuntu-16-nginx-php-phpmyadmin-mysql-5   ubuntu-16-nginx-php-phpmyadmin-mysql-5      16                   [OK]
```

El comando que vemos a continuación nos permite ver las imágenes que tenemos descargadas y disponibles en nuestra máquina.

$ docker images

```
$ docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
ubuntu                        latest              2d696327ab2e        5 days ago          122MB
alpine                        latest              76da55c8019d        11 days ago         3.97MB
busybox                       latest              54511612f1c4        11 days ago         1.13MB
```

Para eliminar cualquier imagen solo hay que ejecutar el comando “$ docker image rm <IMAGE ID>”.

Para ver los contenedores que hay desplegados actualmente en el nodo utilizamos estos dos comandos:

$ docker ps

Lista todos los contenedores que están corriendo ahora mismo.

$ docker ps –a

Lista todos los contenedores, incluidos los que están parados.

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                     PORTS               NAMES
1f2ebd0de7b8        hello-world         "/hello"            7 minutes ago       Exited (0) 7 minutes ago                       lonely_bohr
```

Podemos observar que cada contenedor está identificado por un CONTAINER ID (1f2ebd0de7b8) y por un NOMBRE (lonely_hour).

Esto nos permite gestionar el contenedor a través de cualquiera de los dos identificadores con los siguientes comandos:

$ docker logs container_id # Muestra la salida del contenedor

$ docker container start <ID ó nombre> # Arranca un contenedor parado

$ docker container stop <ID ó nombre> # Permite parar un contendor de forma ordenada.

$ docker container kill <ID ó nombre> # Permite parar de manera forzosa un contenedor

$ docker container rm <ID ó nombre> # Elimina el contenedor del nodo

A continuación veremos los comandos relacionados con Docker Machine, que es una herramienta que nos permite instalar Docker Engine en cualquier máquina virtual y gestionar los hosts con sus comandos.

Estos comandos nos permiten arrancar, inspeccionar, parar y reiniciar los hosts gestionados, actualizar el cliente y el demonio de Docker y configurar el cliente para que hable con el host.

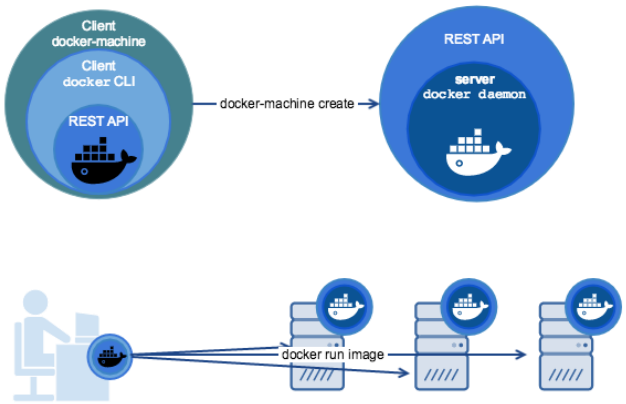


Ilustración 7: Funcionamiento de Docker Machine

$ docker-machine create --driver *virtualbox name*

Con este comando crearemos una nueva máquina virtual con Virtualbox dotada con un sistema operativo mínimo llamado boot2docker.

```
$ docker-machine create --driver virtualbox node1
Running pre-create checks...
(node1) Default Boot2Docker ISO is out-of-date, downloading the latest release...
(node1) Latest release for github.com/boot2docker/boot2docker is v17.06.2-ce
(node1) Downloading C:\Users\Pol\.docker\machine\cache\boot2docker.iso from https://github.com/boot2docker/boot2docker/releases/download/v17.06.2-ce/boot2docker.iso...
(node1) 0%....10%....20%....30%....40%....50%....60%....70%....80%....90%....100%
Creating machine...
(node1) Copying C:\Users\Pol\.docker\machine\cache\boot2docker.iso to C:\Users\Pol\.docker\machine\machines\node1\boot2docker.iso...
(node1) Creating VirtualBox VM...
(node1) Creating SSH key...
(node1) Starting the VM...
(node1) Check network to re-create if needed...
(node1) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: C:\Program Files\Docker Toolbox\docker-machine.exe env node1
```

El comando “$ docker-machine ls” nos permite ver las máquinas virtuales creadas, tanto activas como paradas. También nos da información del driver utilizado y de la versión de Docker de cada una de ellas.

```
$ docker-machine ls
NAME        ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER        ERRORS
default     *        virtualbox   Running   tcp://192.168.99.100:2376           v1.12.3
host1       -        virtualbox   Stopped                                       Unknown
node1       -        virtualbox   Running   tcp://192.168.99.101:2376           v17.06.2-ce
node-1      -        virtualbox   Stopped                                       Unknown
node-2      -        virtualbox   Stopped                                       Unknown
node-3      -        virtualbox   Stopped                                       Unknown
nodekub-1   -        virtualbox   Stopped                                       Unknown
```

Los siguientes comandos nos permiten gestionar las máquinas creadas con Docker Machine.

$ docker-machine kill

$ docker-machine start

$ docker-machine status

$ docker-machine rm

4.2. Docker Network

En este apartado veremos el uso de las redes en Docker, como sabemos Docker tiene 4 redes por defecto, para verlas podemos utilizar el siguiente comando:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
1e726e6c6f07        bridge              bridge              local
21daf639c924        host                host                local
bbeb2300b2dd        none                null                local
```

La red Bridge es la red por defecto de Docker, a no ser que especifiques el tipo de red Docker siempre desplegará el contendor en la red Bridge.

Ahora ejecutaremos un contenedor llamado test:

```
$ docker run -itd --name=test ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
9fb6c798fa41: Pull complete
3b61febd4aef: Pull complete
9d99b9777eb0: Pull complete
d010c8cf75d7: Pull complete
7fac07fb303e: Pull complete
Digest: sha256:d45655633486615d164808b724b29406cb88e23d9c40ac3aaaa2d69e79e3bd5d
Status: Downloaded newer image for ubuntu:latest
c872bb1e954393276063d89143eef837ed2ce5b1e7be4249b8faf83c28c33366
```

Para averiguar la IP asignada a nuestro contenedor tan solo tenemos que ejecutar el comando inspect que nos da información de la red.

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "1e726e6c6f076a3456092e9cb8056980add527113e6eea5514e93e8566b6ef0b",
        "Created": "2017-10-05T11:20:35.29894898Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "c872bb1e954393276063d89143eef837ed2ce5b1e7be4249b8faf83c28c33366": {
                "Name": "test",
                "EndpointID": "5245832069f69ee3f173ad8a06363e39c72e4609bd2f1423c5f95a6b6d7f9511",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
```

Para quitar el contenedor de la red tan solo tenemos que desconectarlo. El en comando “disconnect” tenemos que especificar el tipo de red y el nombre del contenedor o en su defecto su container ID.

```
$ docker network disconnect bridge test
```

Así mismo no se puede eliminar la red Bridge. Las redes son maneras naturales de aislar o conectar a los contenedores de otros contenedores, por lo que se convierte una herramienta muy útil en Docker.

Como hemos visto en la parte teórica, Docker Engine soporta de forma nativa redes bridge y overlay. La red bridge está limitada a un solo host mientras que la overlay incluye varios hosts. En este caso crearemos una red bridge:

```
$ docker network create -d bridge bridge_test
fb5b9546a20606613bd59ef5856d9a1a01b92f4636d974580b99db0b5d1118c8
```

El parámetro -d nos permite especificar el uso del driver bridge para nuestra nueva red. Si no ponemos este parámetro Docker siempre utilizará la red bridge.

Comprobamos que se ha creado la red correctamente con el siguiente comando:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
1e726e6c6f07        bridge              bridge              local
fb5b9546a206        bridge_test         bridge              local
21daf639c924        host                host                local
bbeb2300b2dd        none                null                local
```

Si inspeccionamos la red veremos que está vacía.

```
$ docker network inspect bridge_test
[
    {
        "Name": "bridge_test",
        "Id": "fb5b9546a20606613bd59ef5856d9a1a01b92f4636d974580b99db0b5d1118c8",
        "Created": "2017-10-05T11:34:16.669383305Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Para crear aplicaciones web que funcione de forma segura debemos utilizar las redes, que son por definición, proporciona aislamiento para los contenedores.

Ahora añadiremos un contendor que ejecuta una base de datos PostgreSQL a la red bridge_test:

$ docker run -d --net=bridge_test --name db training/postgres

```
$ docker run -d --net=bridge_test --name db training/postgres
Unable to find image 'training/postgres:latest' locally
latest: Pulling from training/postgres
a3ed95caeb02: Pull complete
6e71c809542e: Pull complete
2978d9af87ba: Pull complete
e1bca35b062f: Pull complete
500b6decf741: Pull complete
74b14ef2151f: Pull complete
7afd5ed3826e: Pull complete
3c69bb244f5e: Pull complete
d86f9ec5aedf: Pull complete
010fabf20157: Pull complete
Digest: sha256:a945dc6dcfbc8d009c3d972931608344b76c2870ce796da00a827bd50791907e
Status: Downloaded newer image for training/postgres:latest
1987dcc89a126334e8cc741091db735a748e1310a036ce85ccf4e340523913bf
```

Si inspeccionamos la red bridge_test veremos que tiene el contenedor db añadido, otra forma de verlo es inspeccionar el contenedor db para ver a qué red está conectado.

$ docker inspect db

```
"Networks": {
    "bridge_test": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": [
            "1987dcc89a12"
        ],
        "NetworkID": "fb5b9546a20606613bd59ef
        "EndpointID": "efcf2983ca6115e80aede5
        "Gateway": "172.18.0.1",
        "IPAddress": "172.18.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:12:00:02",
        "DriverOpts": null
    }
```

Ahora arrancaremos otro contenedor que ejecuta un servidor web Nginx.

$ docker run -d --name web nginx

¿En qué red está corriendo la aplicación Nginx? Lo podremos saber inspeccionando el contenedor, si especificamos el parámetro "--format" podremos encontrarlo más fácilmente, en este caso:

$ docker inspect --format='{{json .NetworkSettings.Networks}}' web

```
"bridge":
```

```
"IPAddress":"172.17.0.2"
```

Podemos observar que la red es bridge por defecto y la dirección IP en este caso es 172.17.0.2.

Ahora accedemos al contenedor que ejecuta la base de datos e intentamos llegar al contendor web mediante un ping.

```
$ docker exec -it db bash
root@1987dcc89a12:/# ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
^C
--- 172.17.0.2 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5006ms
```

Al estar en diferentes redes vemos que no hay conectividad entre sí.

Para solucionar este problema simplemente tenemos que conectar el contenedor "web" a la red que hemos creado anteriormente "bridge_test", que es donde está el contenedor "db".

```
$ docker network connect bridge_test web
```

Ahora intentaremos hacer ping otra vez desde el contenedor "db", esta vez especificando el nombre en vez de la dirección IP, al contenedor web y vemos que esta vez funciona correctamente.

```
$ docker exec -it db bash
root@1987dcc89a12:/# ping web
PING web (172.18.0.3) 56(84) bytes of data.
64 bytes from web.bridge_test (172.18.0.3): icmp_seq=1 ttl=64 time=0.137 ms
64 bytes from web.bridge_test (172.18.0.3): icmp_seq=2 ttl=64 time=0.071 ms
64 bytes from web.bridge_test (172.18.0.3): icmp_seq=3 ttl=64 time=0.063 ms
64 bytes from web.bridge_test (172.18.0.3): icmp_seq=4 ttl=64 time=0.063 ms
64 bytes from web.bridge_test (172.18.0.3): icmp_seq=5 ttl=64 time=0.066 ms
^C
--- web ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4000ms
rtt min/avg/max/mdev = 0.063/0.080/0.137/0.028 ms
```

En este caso también podemos observar que la dirección IP del contenedor ha cambiado, ya que ahora no se encuentra en la red bridge, si no en la red bridge_test.

La creación de redes overlay la trataremos en el apartado de "Creación de Cluster Swarm" donde nos será de utilidad para dotar de conectividad a todos los contenedores del clúster, independientemente del nodo en el que estén situados.

4.3. Ejercicio de Creación de un Clúster con Docker Swarm

Antes de empezar definiremos el entorno que utilizaremos en la creación del de un clúster Swarm de contenedores Docker.

Trabajaremos con 3 nodos, de los cuales el node-1 tendrá el rol de “master” y los nodos node-2 y node-3 serán “workers”.

Crearemos los nodos a través de Docker Machine con los siguientes comandos:

$ docker-machine create –d virtualbox node-1

$ docker-machine create –d virtualbox node-2

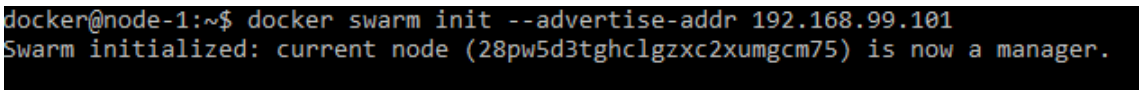
$ docker-machine create –d virtualbox node-3

A continuación nos conectaremos al node-1 por SSH.

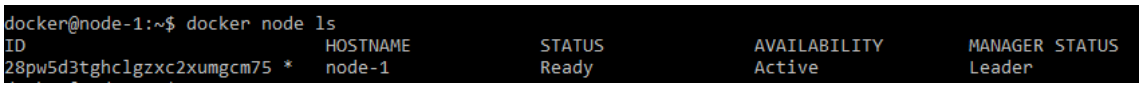
Con el siguiente comando creamos el swarm y le añadimos el nodo 1 con el rol de "master":

$ docker swarm init –advertise-addr 192.168.99.101

La dirección IP corresponde a la del node-1.

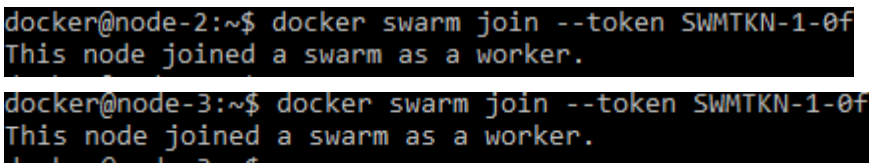


Con el comando "$ docker node ls" podemos ver la información de todos los nodos.



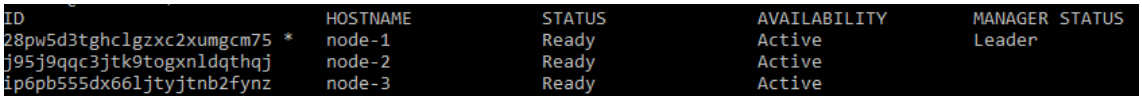
Para añadir los nodos al clúster como workers nos conectamos a cada uno de ellos por SSH y ejecutamos el siguiente comando:

$ docker swarm join --token SWMTKN-1-0fv35ch2t4j7lt7jbyahmvhzld1r02femqoo6b0btrqtkx12yh-317kcrd3f9s64i0bb39uqv09z 192.168.99.101:2377



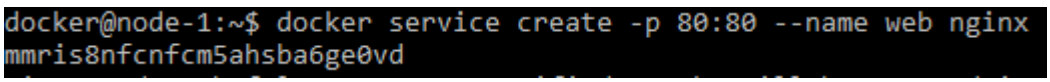
Ahora si nos conectamos al nodo master podremos ver la información de los nodos adjuntos al clúster.

$ docker node ls



A continuación procedemos a la creación de un servicio, Nginx en nuestro caso.

En el node-1 ejecutamos el comando $docker service create nginx



Este comando nos creará un servicio Nginx con una sola réplica.

Con $ docker service ls podremos ver los servicios que se están ejecutando y cuantas instancias hay de cada uno.

```
docker@node-1:~$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
mmris8nfcnfc        web                 replicated          1/1                 nginx:latest        *:80->80/tcp
```

El comando "$docker service ps web" nos da información de en qué nodo está corriendo el servicio especificado.

```
docker@node-1:~$ docker service ps web
ID                  NAME                IMAGE               NODE
pozqykmp6l91        web.1               nginx:latest        node-2
```

Si accedemos a un navegador y ponemos la dirección IP de cualquiera de los nodos deberíamos ser capaces de acceder a una web como esta.

Esto se debe a que un servicio ejecutándose en un nodo es accesible por todos los nodos del clúster.

Otra característica de Docker Swarm es la escalabilidad, con la que podemos conseguir instancias adicionales del mismo servicio. Si quisiéramos escalar este servicio web a 4 réplicas tan solo tendríamos que utilizar el siguiente comando.

$ docker service scale web=4

```
docker@node-1:~$ docker service scale web=4
web scaled to 4
```

Podemos observar que hay cuatro tasks corriendo, distribuidas por los tres nodos.

```
docker@node-1:~$ docker service ps web
ID                  NAME                IMAGE               NODE
pozqykmp6l91        web.1               nginx:latest        node-2
jg2bd72nthhg        web.2               nginx:latest        node-3
wdjyospm8wr0        web.3               nginx:latest        node-1
t02s4erz5new        web.4               nginx:latest        node-3
```

Si quisiéramos que un nodo no tuviera tareas asignadas haríamos que su estado pasara de ACTIVE a DRAIN.

$ docker node update –availability drain node-2

```
docker@node-1:~$ docker node update --availability drain node-2
node-2
```

Podemos ver como se ha parado la réplica del node-2 y se ha reiniciado inmediatamente en el node-1.

```
docker@node-1:~$ docker service ps web
ID                  NAME                IMAGE               NODE                DESIRED STATE
uocfx4l60xps        web.1               nginx:latest        node-1              Running
pozqykmp6l91         \_ web.1           nginx:latest        node-2              Shutdown
jg2bd72nthhg        web.2               nginx:latest        node-3              Running
wdjyospm8wr0        web.3               nginx:latest        node-1              Running
t02s4erz5new        web.4               nginx:latest        node-3              Running
```

Para finalizar podemos ver el estado del servicio con el siguiente comando:

$ docker service inspect --pretty web

```
docker@node-1:~$ docker service inspect --pretty web

ID:             mmris8nfcnfcm5ahsba6ge0vd
Name:           web
Service Mode:   Replicated
 Replicas:      4
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:         nginx:latest@sha256:af32e714a9cc31571573
Resources:
Endpoint Mode:  vip
Ports:
 PublishedPort = 80
  Protocol = tcp
  TargetPort = 80
  PublishMode = ingress
```

4.4. Dockerfile

Dockerfile es un archivo que recoge todas las instrucciones necesarias para crear la imagen de un contenedor de Docker. En el contiene todas las acciones que realizará automáticamente en la imagen base para crear la nueva imagen.

Docker utiliza este archivo para crear una imagen descrita en el archivo Dockerfile.

Las instrucciones que más se utilizan en Dockerfile son las siguientes:

- FROM: define la imagen base a partir de la cual crearemos la imagen que construirá el Dockerfile.
- MAINTAINER: aquí escribiremos el nombre del autor, este comando no se ejecuta.
- ENV HOME: establece el directorio HOME que usarán los comandos RUN.
- RUN: permite ejecutar una instrucción en el contenedor.

- ADD: Este comando permite copiar archivos de un origen a un destino, en muchas ocasiones se utiliza para proporcionar la configuración de servicios como ssh, mysql, ….
- VOLUME: permite el acceso desde el contenedor a un directorio de la máquina host como punto de montaje.
- EXPOSE: indica los puertos TCP/IP por los que se pueden acceder a los servicios del contenedor
- CMD: establece el comando del proceso de inicio que se usará si no se indica uno al iniciar un contenedor con la imagen. Es similar al comando RUN, pero mientras que RUN se ejecuta durante la creación de la imagen, CMD se ejecuta una vez que se ha creado el contenedor.

Un ejemplo de Dockerfile podría ser un servidor Nginx.

```
FROM Ubuntu:17.04
MAINTAINER Pol Ponsico
RUN apt-get update && apt-get –y install nginx
EXPOSE 80
```

Después construimos la imagen ejecutando $ docker build -t pol/web, que podremos ejecutar en forma de contenedor a continuación para arrancar el servidor Nginx.

4.5. Docker Compose

Como hemos visto anteriormente Docker Compose es una herramienta que permite definir y ejecutar aplicaciones Docker con múltiples contenedores. Para ello utiliza un fichero YAML para configurar los servicios de la aplicación.

Un ejemplo de fichero YAML es el siguiente, donde encontramos la definición de un servicio web Nginx que consta de 3 instancias y que tiene la CPU limitada al 10% y la memoria RAM a 30MB. Estas limitaciones están referidas al host, no a cada uno de los nodos.

```yaml
version: '3'
services:
 web:
  image: nginx:latest
  deploy:
   replicas: 3
   resources:
    limits:
     cpus: '0.1'
     memory: 50M
   restart_policy:
    condition: on-failure
  ports:
   - '80:80'
  networks:
   - webnet
networks:
 webnet:
```

A la hora de escribir este tipo de ficheros hay que ir con mucho cuidado ya que las tabulaciones están prohibidas y la jerarquía, que se marca con espacios, es muy importante que esté bien definida.

Este fichero lo guardaremos con el nombre compose-test.yml. Antes de hacer el "deploy" de este comando iniciaremos el clúster de Docker Swarm en el nodo master con el comando visto anteriormente "$ docker swarm init".

Ahora podremos desplegar nuestra aplicación.

$ docker stack deploy -c compose-test.yml web

```
$ docker stack deploy -c compose-test.yml web
Creating network web_webnet
Creating service web_web
```

Para obtener más información de la aplicación ejecutamos: $ docker service ls

Para ver las tasks del servicio ejecutamos el comando: $ docker service ps web

```
$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
qt9xnhkvaqz9        web_web             replicated          0/3                 nginx:latest        *:80->80/tcp
```

Como hemos visto antes podemos inspeccionar el servicio, así como ver los contenedores corriendo en este servicio.

También es posible escalar la aplicación mediante la modificación del fichero YML, tan solo tendremos que ejecutar otra vez el comando "deploy" para que el cambio se haga efectivo.

```
$ docker service ps web_web
ID                  NAME                IMAGE               NODE                DESIRED STATE
          PORTS
m7g7zv972gzq        web_web.1           nginx:latest        default             Running
lx207xftc5dx        web_web.2           nginx:latest        default             Running
j9n1gmmamjav        web_web.3           nginx:latest        default             Running
```

Por último, para parar la aplicación ejecutaremos el siguiente comando.

$ docker stack rm web

```
$ docker stack rm web
Removing service web_web
Removing network web_webnet
```

4.6. Ejercicio de Creación de un Clúster con Kubernetes

En este ejercicio aplicaremos los conocimientos obtenidos en los apartados teóricos referentes a Kubernetes.

Kubectl es la interfaz de línea de comandas de los clústers de Kubernetes, es el programa que interactúa con la API.

A continuación, ejecutaremos nuestra apliación Nginx a partir de la imagen de Nginx, que la descarga de Docker Hub.

$ kubectl run my-nginx --image=nginx --port=80

```
  > kubectl run my-nginx --image=nginx --port=80
deployment "my-nginx" created
```

Con la ejecución de este comando Kubernetes ha buscado un nodo donde pudiera correr la aplicación, la ha situado en ese nodo y ha configurado el clúster para buscar un nuevo nodo para la apliación cuando sea necesario.

El siguiente comando nos permite ver los pods que se están ejecutando en el nodo.

$ kubectl get pods

```
  > kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
my-nginx-379829228-6n9fp    1/1       Running   0          1m
```

Los pods que se ejecutan en Kubernetes están en una red aislada, son visibles para los pods y servicios en el mismo cluster, pero no para el exterior.

En cuanto a la escalabilidad de las aplicaciones, podemos crear varias instancias de un deployment facilmente con el comando:

$ kubectl scale deployments/my-nginx –replicas=3

```
deployment "my-nginx" scaled
```

Si ejecutamos el comando “$ kubectl get deployments” nos mostrará todas las instancias que están corriendo de nuestra aplicación.

```
  > kubectl get deployments
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
my-nginx   3         3         3            3           1m
```

También podemos comprobar que el número de pods ha cambiado con el comando:

$ kubectl get pods -o wide

5. Presupuesto

Todos los softwares utilizados para realizar esta tesis, ya sea tanto Docker, Kubernetes como Virtualbox son softwares open source, por lo tanto, no requieren de la compra de licencias para su uso.

Teniendo esto en cuenta solo hay que contabilizar los costes de las horas dedicadas a la realización del trabajo. En mi caso, calculando que el tiempo requerido para realizar el trabajo es de aproximadamente 600 horas y con una remuneración de 8 euros por hora, obtenemos que el coste total del proyecto asciende a 4800 euros.

6. Conclusiones y futuros desarrollos:

Docker es un proyecto que permite crear aplicaciones en contenedores.

El código corre en estos contenedores totalmente aislado de otros contenedores y todos ellos comparten los recursos de la máquina sin la sobrecarga que aporta la capa del hipervisor.

Los contenedores son mucho más ligeros que las máquinas virtuales ya que, mientras que las máquinas virtuales requieren de la instalación de un sistema operativo, asignación de disco, CPU y memoria RAM para funcionar, un contenedor de Docker sólo necesita el sistema operativo que corre en la máquina en la que está el contenedor.

Además, Docker tiene muchas herramientas que permiten explotar diferentes vertientes de los contenedores como pueden ser Docker Swarm, Kubernetes, Docker Compose, Docker Machine, y Docker Hub.

La tecnología de los contenedores virtuales lleva funcionando mucho tiempo ya, pero con el aumento de popularidad de Docker, la virtualización de contenedores se normalizó. La industria ahora se beneficia de un flujo de trabajo de contenedores estandarizado y un amplio ecosistema de herramientas y servicios.

Si bien los contenedores funcionan muy bien en muchos escenarios, sigue habiendo casos en los que servidores tradicionales o máquinas virtuales tendrán más sentido.

Si hablamos de entornos de desarrollo, Docker es la herramienta ideal ya que es un sistema aislado que cuenta únicamente con librerías que utilizaremos para nuestro proyecto, por lo tanto, tendremos un entorno óptimo para desarrollar nuestra aplicación, que podremos ejecutar en cualquier máquina, independientemente del sistema operativo

Y con la ayuda de Docker Swarm o Kubernetes podremos crear un clúster fácilmente escalable para ejecutar los diferentes servicios de los que esté compuesta nuestra aplicación.

En cuanto a entornos de producción, y aunque se ha hablado mucho de que en un futuro los contenedores reemplazarán a las máquinas virtuales, yo creo que lo que realmente sucederá será la integración de esta tecnología con la tecnología de virtualización clásica, que aportará mejoras en el rendimiento de muchas aplicaciones.

Bibliografía:

[1] Aaron Grattafiori "Understanding and Hardening Linux Containers". *NCC Group*, April 20, 2016 – Version 1.0.

[2] Ian Miell, Aidan Hobson Sayers. "*Docker in Practice*", 1st ed. Manning Publications, 2016

[3] T. Tarun, P. Viswanathan, S. Suman. "Wireless Sensor Network White Paper". *Tetcos Engineering*, 2012. [Online] Available: http://www.tetcos.com/Enhancing_Throughput_of_WSNs.pdf. [Accessed: 23 October 2012].

[4] Anónimo, Docker Official Documentation, [Online] Available: https://docs.docker.com

[5] Anónimo, Kubernetes Official Documentation, [Online] Available: https://kubernetes.io/docs/concepts/

[6] Jeff Nickoloff. "Evaluating Container Platforms at Scale, [Online] Available: https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c

[7] Milind Lele, Saroj Pradhan. "Powering Microservices with Docker" *Cognizant 20-20 Insights*, August 2017

Glosario

Los acrónimos que se encuentran en este documento pertenecen a las siguientes palabras:

- **LXC:** Linux Containers
- **CLI:** Command Line Interface
- **NAT:** Network address translation
- **CEO:** Chief Executive Officer
- **VXLAN:** Virtual Extensible Local Area Network
- **API:** Application Programming Interface
- **HTTP:** Hypertext Transfer Protocol
- **CPU:** Central Processing Unit
- **RAM:** Random Access Memory
- **TCP:** Transmission Control Protocol
- **IP:** Internet Protocol
- **YAML:** YAML Ain't Another Markup Language